Scalable Omniscient Debugging

Guillaume Pothier [*] Éric Tanter [†] José Piquer

DCC – University of Chile
Avenida Blanco Encalada 2120 – Santiago, Chile
{gpothier,etanter,jpiquer}@dcc.uchile.cl

Abstract

Omniscient debuggers make it possible to navigate backwards in time within a program execution trace, drastically improving the task of debugging complex applications. Still, they are mostly ignored in practice due to the challenges raised by the potentially huge size of the execution traces. This paper shows that omniscient debugging can be realistically realized through the use of different techniques addressing efficiency, scalability and usability. We present TOD, a portable Trace-Oriented Debugger for Java, which combines an efficient instrumentation for event generation, a specialized distributed database for scalable storage and efficient querying, support for partial traces in order to reduce the trace volume to relevant events, and innovative interface components for interactive trace navigation and analysis in the development environment. Provided a reasonable infrastructure, the performance of TOD allows a responsive debugging experience in the face of large programs.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Debugging aids; D.2.6 [*Programming Environments*]: Integrated environments; D.3.4 [*Processors*]: Debuggers; H.2.3 [*Languages*]: Query languages; H.2.4 [*Systems*]: Distributed databases; H.2.4 [*Systems*]: Query processing

General Terms Algorithms, Design, Performance

Keywords Omniscient debugging, scalability, execution traces, specialized distributed database, partial traces, interface components

1. Introduction

Debugging software is a major task of the software development process, both in terms of time and cost. Unfortunately debuggers in most development environments only provide very minimal assistance and debugging remains a tedious and time-consuming task.

There are two traditional approaches to debugging: log-based debugging and breakpoint-based debugging. The first approach consists in inserting logging statements within the source code, in order to produce an ad-hoc trace during program execution. This technique exposes the actual history of execution but *(a)* it requires cumbersome and widespread modifications to the source code, and *(b)* it does not scale because manual analysis of huge traces is hard. The second approach consists in running the program under a dedicated *debugger* which allows the programmer to pause the execution at determined points, inspect memory contents, and then continue execution step-by-step. Although not subject to the two issues of log-based debugging, breakpoint-based debugging is limited: when execution is paused, the information about the previous state and activity of the program is limited to introspection of the *current* call stack.

Omniscient debuggers, also known as *back-in-time* or *post-mortem* debuggers, overcome all these issues [11, 15, 17]. An omniscient debugger records the events that occur during the execution of the debugged program, and then lets the user conveniently navigate through the obtained execution trace. This approach combines the advantages of log-based debugging –past activity is never lost– and those of breakpoint-based debugging –easy navigation, step-by-step execution, complete stack inspection. An omniscient debugger can simulate step-by-step execution forward *and backward*, and makes it possible to immediately answer questions that would otherwise require a significant effort, like *"At what point was variable* `x` *assigned value* `y` *?"* or *"What was the state of object* `o` *when it was passed as an argument to the method* `foo` *?"*.

While the advantages of omniscient debugging over traditional approaches are incredibly clear, it has had a very limited impact in production environments, and is still mostly seen as an unrealistic approach. It is true that omniscient debugging raises important issues. First, except when us-

[*] Guillaume Pothier is financed by a PhD grant from NIC Chile

[†] É. Tanter is partially financed by the Millenium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile, and by FONDECYT Project 11060493.

OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.
Copyright © 2007 ACM 978-1-59593-786-5/07/0010. . . $5.00

ing specialized hardware probing ports [10], the emission of events causes a significant overhead to the debugged application. Second, as emphasized in [9, 20, 28], for CPU-intensive applications, the execution trace can rapidly become huge (hundreds of million events), implying that *(a)* trace data must be stored very quickly and requires scalable storage; and *(b)* the user interface of the debugger must be responsive enough –this requires fast query execution on huge traces–, and must assist the user in overcoming the cognitive burden of dealing with a large amount of information in order to rapidly locate the points of interest.

The contribution of this paper is to show that omniscient debugging *can be realistically realized* through the use of different techniques enhancing efficiency, scalability, and usability. This claim is validated by TOD[1], a portable Trace-Oriented Debugger for Java integrated into the Eclipse IDE [5]. TOD features:

- **Efficient event generation** based on a compact trace model, a custom binary encoding of events, and a fast, portable low-level weaver.
- **Specialized distributed database engine** for scalable and fast storing and querying of events, which leverages the highly-constrained nature of execution traces. On a dedicated 10-node cluster TOD handles a sustained input rate of approx. 470kEv/s (thousands events per second), and hundreds of queries per second.
- **Support for partial traces** by offering static and dynamic mechanisms for selective trace generation, and adequate reporting of incomplete information.
- **Responsive GUI** thanks to efficient query processing; TOD was used to debug an application as complex as Eclipse while preserving interactivity.
- **Specialized GUI components** providing high-level views on huge event traces for more effective navigation, such as *thread murals*.

Section 2 details the features and challenges of omniscient debugging. Section 3 overviews the architecture of TOD, the event model, and the GUI components. Section 4 describes the efficient indexing scheme of TOD for storing and querying events, and Section 5 shows how it is parallelized. Benchmarks are provided in Section 6. We explain the advantage of partial traces and how they are dealt with in Section 7. Related work is discussed in details in Section 8. Section 9 concludes, and identifies opportunities for further enhancements in the field.

2. Challenges of Omniscient Debugging

We now present the main features of an omniscient debugger compared to traditional debuggers, and outline the scalability challenges of omniscient debugging.

2.1 Features of an omniscient debugger

An omniscient debugger (OD) provides four major features: stepping, state reconstitution, control flow reconstitution, and root cause finding. The latter is unique to omniscient debuggers, while others are typical debugger features.

In breakpoint-based debuggers, *Stepping* consists in executing the target program one instruction at a time. There are two variants of stepping: *step over* executes behavior[2] call statements without halting inside the called behavior, while *step into* halts at the beginning of the called behavior. *State reconstitution* consists in letting the programmer *inspect* object state when the target program is halted. *Control flow reconstitution* permits to obtain a view on the current call stack of the program, with bound variables and objects. ODs extend these three features with complete freedom with respect to time: stepping can be done both forward and backward in time, programmers can inspect the state of objects as they were at any given point in time, and can freely browse the control flow tree.

Finally, one of the most useful features of ODs is their ability to find *when* and *in which context* a particular field or variable was given a certain value. Indeed, bugs often manifest long after their root cause occurs. For instance, trying to dereference a `null` reference obtained from a given field causes a crash, which is the *symptom* of the bug. The information the programmer needs is *when* was the field set to `null`. With breakpoint-based debuggers, even if execution is halted just before the faulty dereference, the *root cause* of the bug can be already lost, *e.g.*because the code that caused it is not in the call stack anymore.

2.2 Scalability challenges

Underlying the features presented above lies the necessity to generate and record execution traces. The potentially huge size of these traces poses several scalability challenges, which are the main reason for the lack of production-quality ODs.

- Events must be recorded quickly, preferably in real time, so that *(a)* debugging can begin immediately after the target program terminates or crashes, and *(b)* runtime overhead is minimized to preserve overall performance of the debugged program, and interactivity when needed (*e.g.* debugging Eclipse).
- The debugger should cause minimal interference to the target program in order to not affect its behavior. In particular, the address space and memory management of the target process should not be altered.

[1] `http://reflex.dcc.uchile.cl/TOD` for download and small illustrative videos.

[2] We give methods and constructors the collective name of *behaviors*.

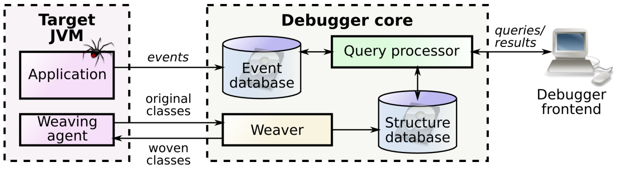


Figure 1. High-level architecture of TOD

- The event storage capacity of an omniscient debugger must be aligned with the expected number of events in a useful trace: with GHz CPUs, hundreds of millions events can be generated in only a few minutes of execution.
- Queries on the execution trace must execute at a speed compatible with user interaction, *e.g.* in tenths of seconds for operations like stepping.
- Information must be presented in a way that addresses the cognitive burden of navigating through huge event traces, enabling rapid bug identification.

This work addresses the above issues via optimized event representations and aggressive indexing, a simple query model, a distributed database backend, support for partial traces, and specialized presentation and interaction components.

3. Overview of TOD

TOD is a Trace-Oriented Debugger for Java that addresses the scalability issues identified above. The objective is to address these issues in order to obtain an omniscient debugger that is practically applicable. This section gives an overview of TOD via its architecture, the event model, and the GUI components.

3.1 Architecture

TOD is designed around two central ideas: to decouple the core of the debugger from the target program execution, and to be portable. It is made up of three components (Fig. 1): the target Java Virtual Machine (JVM) in which the debugged program runs and emits events, the debugger core that implements the main functionalities of TOD, and the debugger frontend through which the user interactively queries and navigates in the execution trace.

The rationale for storing events in a database rather than in memory as done in other omniscient debuggers [11, 15, 17] is precisely to address some of the challenges discussed in Sect. 2.2: storing events in the address space of the target application is not scalable past a few hundred megabytes of trace data, and interferes with memory management, in particular with the garbage collector. The increased capture cost incurred by the use of a database is compensated by a better scalability and non intrusiveness. As a side effect, the ability to serialize execution traces allows for *post-mortem debugging*, which opens interesting perspectives for software companies willing to offer software with high-quality support: overlooking the storage cost, a navigatable execution trace is a far more relevant input for a bug report than an ad-hoc text description.

During execution, the target application emits events that are sent to the debugger core, where they are recorded and indexed in an *event database*. The way events are emitted is discussed later. The event database leverages the peculiarities of the event stream and the restricted set of possible queries to provide both high recording throughput and good query performance (see Sect. 4 and 5). The debugger core contains another database, the *structure database*, which contains static information about the target application. In particular it keeps track of the 32-bit integer identifiers that are assigned to structural elements of the target program (*i.e.* classes, methods, and fields). Queries performed by the user rely on both the event and the structure databases.

3.2 Representation and emission of events

We now introduce the representation of events and event traces, as well as how events are emitted by a debugged application in TOD.

Event and trace model. An *event* is a structure characterized by a number of attributes chosen among the set $A = \{a_0, ..., a_k\}$. We note $e.a_j$ the value of attribute a_j of event e. For each $j \in [0..k]$, let D_j be the *domain* of a_j, *i.e.* the set of all distinct values that can be taken by a_j for any event in the trace. An *event trace* $T = \langle e_1, ..., e_n \rangle$ is an ordered sequence of n heterogeneous events.

The a_0 attribute corresponds to the timestamp of the event; it is characterized by the fact that *(a)* all events have a value for a_0, *(b)* there exists a complete order on D_0 and *(c)* the events in T are ordered by their value of a_0. Table 1 shows which concrete events are captured and what are their attributes.

Emission of events. The debugger core of TOD captures events emitted by the target application (Fig. 1). There are three ways in which events can be emitted: specialized hardware trace ports [10], virtual machine or interpreter instrumentation [16], and application code instrumentation [11, 15]. TOD uses the last one: although not as fast as hardware probes and significantly more space-consuming than VM-level instrumentation in terms of code size, application instrumentation is also much more portable and easier to implement.

In TOD, the JVM that hosts the target application is set up to use a JVMTI[3] native agent. The agent intercepts class load events and replaces the original class definitions by instrumented versions. Instrumentation itself is performed by the weaver in the debugger core: the agent sends the original bytecode to the core, the weaver instruments the class and

[3] JVMTI: Java Virtual Machine Tool Interface, part of the Java 5 platform.

	kind	ts	tid	depth	pev	loc	fid	bid	vid	idx	val	ret	tgt	exc	args
Field write (FW)	√	√	√	√	√	√	√				√		√		
Local var. write (VW)	√	√	√	√	√	√			√		√				
Array write (AW)	√	√	√	√	√	√				√	√		√		
Exception (Ex)	√	√	√	√	√	√		√						√	
Behavior call (BC)	√	√	√	√	√	√		√					√		√
Behavior enter (Bn)	√	√	√	√	√	√		√					√		√
Behavior exit (Bx)	√	√	√	√	√	√		√				√	√		

Row headers are event kinds and column headers are the possible attributes (ts: timestamp, tid: thread id, depth: call stack depth, pev: pointer to parent event, loc: source code location, fid: field id, bid: behavior id, vid: local variable id, idx: array index, val: value, ret: return value, tgt: target, exc: exception, args: arguments).

Table 1. Events and their attributes.

stores structural information in the structure database, and the modified class is sent back to the target JVM where it is eventually loaded (Fig. 1). The agent caches instrumented classes on the hard disk to reduce the number of inter-process round trips. This is particularly useful for frequently-used classes such as those in the JDK.

Instrumentation is done using the ASM bytecode manipulation library [3]: event emission code is added before and/or after specific bytecode patterns in the original code, such as a field write or a method call. When the instrumented code is executed, events are constructed along with their attributes, serialized in a custom binary format, and sent through a socket to the event database.

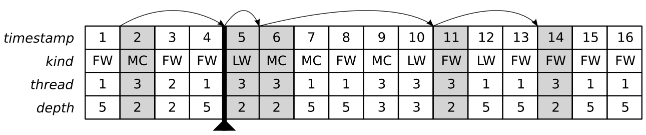
Non-ambiguous event timing. Although event timestamps are obtained through the nanosecond-precision time service of Java, its potential lack of accuracy makes it is possible for several events of the same thread to share the same timestamp value. As this is incompatible with the event indexing scheme used by TOD (Sect. 4), we shift original timestamp values a few bits to the left and use the free bits to differentiate events of the same thread that share the same timestamp. When comparing the timestamps of events of different threads, we use the original timestamps to preserve inter-thread event ordering.

Scoped trace capture. The instrumentation scheme described above is *selective*, that is, it is possible to supply user-defined filters that limit the number of emitted events. This feature is described in Section 7.

Object identification. The JVMTI agent of TOD assigns a unique identifier to each object in the target application; whenever an event needs to refer to an object it uses this identifier. Additionally objects whose state cannot be reconstituted, like `String` and `Exception`, are sent in a serialized form the first time they are referenced. As an exception to this mechanism objects that represent primitive values (*e.g.* `Integer`, `Float`, etc.) are passed by value.

3.3 Low-level queries: cursors and counts

All the features presented in Section 2.1 (stepping, state reconstitution, control flow reconstitution and root cause finding) can be expressed in terms of two low-level queries: *cursors* and *counts*, which we introduce below. Both are based on filtering events in the trace according to some conditions on their attributes. Conditions can be any boolean combination of simple predicates of the form $attribute = value$, where $value$ is a constant. For instance $(kind = FW \vee kind = BC) \wedge target = obj145$. If Q is such a condition and e is an event, we define the predicate function $Q(e)$ whose value is true iff e verifies condition Q.



timestamp	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
kind	FW	MC	FW	FW	LW	MC	MC	FW	MC	LW	FW	LW	FW	FW	FW	FW
thread	1	3	2	1	3	3	1	1	3	3	3	1	1	3	1	1
depth	5	2	2	5	2	2	5	5	3	3	2	5	5	2	5	5

The current position of the cursor is depicted by the bold line between events 4 and 5. Events that match the cursor's predicate are grayed. Successive calls to $next()$ return events 5, 6, 11 and 14; calling $posNext(11)$ positions the cursor between events 10 and 11; calling $posPrev(11)$ positions it between events 11 and 12.

Figure 2. Navigation among events matching a cursor predicate.

operation	*semantics*
$next()$/ $prev()$	Returns the next/previous matching event and moves the cursor forward/backward.
$posNext(t)$/ $posPrev(t)$	Moves so that the next call to next()/prev() returns the first/last event whose timestamp is greater/lesser than or equal to t.
$posNext(ev)$/ $posPrev(ev)$	Moves so that the next call to next()/prev() returns the given event.

Table 2. Cursor operations.

Cursors. We define $cursor(Q)$ as an iterator over events that match condition Q (Fig. 2). Cursors have a current position that is situated between two consecutive events (or at the beginning or end of the trace). A cursor supports a number of navigation operations, as shown in Table 2.

Counts. Given a time interval $[t_1, t_2]$ divided in s slices of length $\delta t = (t_2 - t_1)/s$ each, and a condition Q on event attributes, a count query returns an array of s integers such that $s[i] = |\{e \in T : within(e, t_1 + i \cdot \delta t) \wedge Q(e)\}|$ where $within(e,t) \Leftrightarrow e.ts \geq t \wedge e.ts < t + \delta t$. Each slot of the array contains the number of events matching Q that occur during the corresponding time slice.

3.4 High-level queries

We now explain how cursors and counts are algorithmically combined to implement the high-level features described in Sect. 2.1. Section 4 discusses the aggressive database optimization enabled by using only filtering-based queries.

Stepping. We define $stepper$ as an object that has a current event ev and supports forward and backward *step into* and *step over* operations. For instance, forward step into is defined as follows:

```
$c \leftarrow cursor(thread = ev.thread)$
$c.posPrev(ev)$; $ev \leftarrow c.next()$
```

Forward step over changes the cursor condition to: $thread = ev.thread \wedge depth = ev.depth$. Backward stepping is symmetric to forward stepping.

State reconstitution. The value v of a field f of a particular object o at time t can be retrieved as follows:

```
$c \leftarrow cursor(kind = FW \wedge fid = f \wedge target = o)$
$c.posPrev(t)$; $v \leftarrow c.prev().val$
```

The state of an object can be retrieved by performing the same operation for each field. Stack frames are reconstituted in a similar way, using variable write events instead of field write events.

Control flow reconstitution. Events that occurred in the top-level control flow of a given method call event e are retrieved as follows:

```
$c \leftarrow cursor(thread = e.thread \wedge depth = e.depth + 1)$
$c.posPrev(e.ts)$; $cflow = \langle\rangle$
repeat
  $ev = c.next()$; $cflow \leftarrow cflow \cup \langle ev \rangle$
until $ev.kind = BEx$
```

Root cause finder. Determining how a field has been assigned an undesired value is as direct as the state reconstruction query above: instead of obtaining the value of the field write event that assigned the value to the field, the event itself is made current, giving access to the context at that time. Backward-in-time exploration of the cause can go on like this, up to the root cause.

3.5 User interface components

The frontend of TOD can be used standalone or as a plugin for the Eclipse Java IDE (Fig. 3). The user navigates between different *views* using widely-understood web browser metaphors (hyperlinks, back button). The available views are: object inspector, control flow, and murals. The object inspector view shows reconstitutions of objects, and allows root cause finding for field values through a convenient *why?* link next to each field. The control flow view shows a reconstitution of the control flow and allows stepping operations as well as root cause finding for local variable values.

Murals. High-level overviews are useful for spotting abnormal behavior patterns. However representing a huge number of events in a limited number of pixels is difficult. Jerding and Stasko introduced the *information mural* [12] as a *"reduced representation of an entire information space that fits entirely within a display window"*. TOD features *event murals*, which are graphs that show the evolution of *event density*, or number of events per unit of time, in a given period:

- Thread murals show the event density of each thread for the whole execution of the target application (Fig. 4).
- Object activity murals show the density of calls to methods of a particular object.
- Method murals show the density of calls to a particular method on any object.

In all cases densities are obtained through counts (Sect. 3.3), where the length of the time slices corresponds to the space occupied by a single pixel bar in the mural. The user can zoom and pan the murals; when the zoom level permits to distinguish individual events the user can select an event and see its context in a stepper view. Thread murals have a variety of applications, *e.g.* to understand the interplay between threads, or spotting dead- and livelocks.

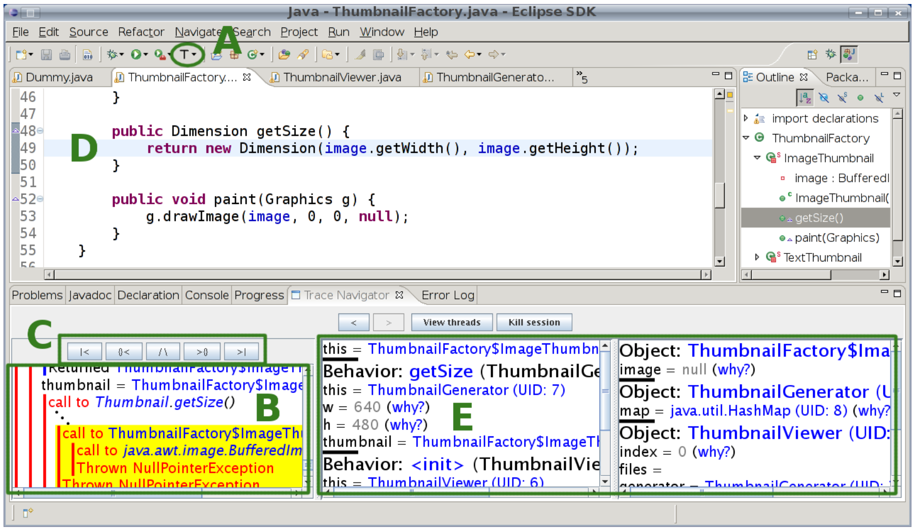
4. High-Speed Database Backend

We now describe and analyze the database backend of TOD, which allows for efficient query execution while being fast enough to allow a high recording throughput. Section 5 shows how our solution is amenable to parallelization, and Section 6 reports on actual performance measurements.

The need to develop a specialized database backend for TOD was motivated by the poor performance of widely-used database management systems for our purposes: for instance Postgres and Oracle only support storing events at a rate of 50 and 500 events per second respectively, while we rather aim at rates in the order of hundreds of thousands events per second [22]. Our high-throughput specialized database backend leverages the following specificities of the event stream of an execution trace: *(a)* the event stream is read-only, *(b)* events arrive ordered by timestamp[4] and *(c)* queries are limited to filtering.

Sect. 4.1 describes the indexing scheme used by the database. Sections 4.2, 4.3 and 4.4 analyze the cost of executing the queries described in Sect. 3.3. Finally Sect. 4.5 analyzes the recording throughput that can be achieved by the

[4] Events of different threads might arrive out of order because of the way serialized events are buffered; as reordering them is cheap (only the last few events must be considered) we assume events have been reordered before they reach the backend.



```
        }

        public Dimension getSize() {
            return new Dimension(image.getWidth(), image.getHeight());
        }

        public void paint(Graphics g) {
            g.drawImage(image, 0, 0, null);
        }
    }
```

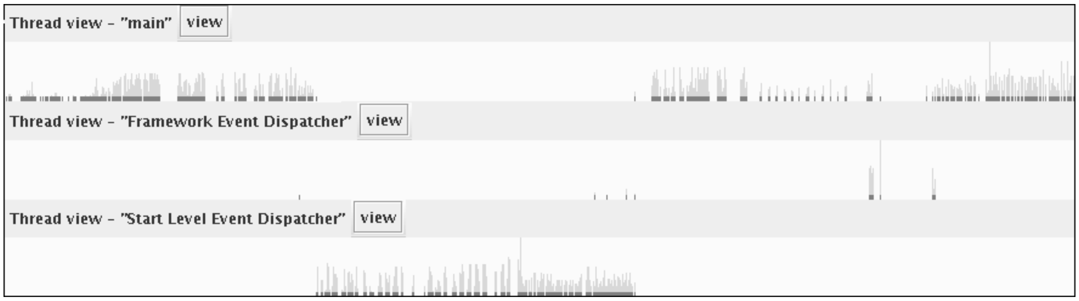
```
Returned ThumbnailFactory$ImageTh
thumbnail = ThumbnailFactory$Image
call to Thumbnail.getSize()
  call to ThumbnailFactory$ImageThi
    call to java.awt.image.BufferedIm
    Thrown NullPointerException
  Thrown NullPointerException
```

```
this = ThumbnailFactory$ImageThumbn
Behavior: getSize (ThumbnailGe
this = ThumbnailGenerator (UID: 7)
w = 640 (why?)
h = 480 (why?)
thumbnail = ThumbnailFactory$ImageTh
Behavior: <init> (ThumbnailVie
this = ThumbnailViewer (UID: 6)
```

```
Object: ThumbnailFactory$Ima
image = null (why?)
Object: ThumbnailGenerator (U
map = java.util.HashMap (UID: 8) (why?
Object: ThumbnailViewer (UID:
index = 0 (why?)
files =
generator = ThumbnailGenerator (UID:
```

Button (A) launches the program with trace recording. The user navigates in the control flow (B) using stepping buttons (C), or by clicking on an event. The line corresponding to the current event is highlighted in the source window (D). The state of the stack frames and current object is shown in window (E). The user can jump to the instruction that set a variable or field to its current value by clicking the *why?* link next to it.

Figure 3. Stepping with TOD in Eclipse.



The graphs shows the density of events of each thread along a time axis.

Figure 4. Thread murals.

system and presents an important trade-off between memory requirements and efficiency.

4.1 Aggressive indexing of events

In most database management systems the indexing scheme consists in maintaining one index on attribute value for selected attributes. Such an index permits to quickly retrieve the records that have a specific value for the indexed attribute. TOD adopts a more aggressive indexing scheme in which there is a separate index on timestamp for each distinct value of each attribute. This enables a highly-efficient processing of the *cursors* and *counts* queries defined in Sect. 3.3, and at the same time permits to sustain a high recording throughput.

Using the notation defined in Section 3.2 we define the *index set* of trace T on attribute a_j as, in a first approximation, a function $IS_j : D_j \mapsto \mathbb{N}^*$ for $j \in [1..k]$ so that IS_j maps any possible value v of a_j to an *index*, which is a sequence of event pointers. A pointer i appears in index $IS_j(v)$ if and only if $e_i.a_j = v$, where e_i is the i^{th} event of T. Those indexes can be used directly to retrieve all events that match a simple query of the form $a_j = v$; compound conditions are discussed in Sect. 4.2.

However, TOD queries consist not only in finding matching events, but also in finding matching events that occurred at, after or before a particular point in time. Therefore indexes contain *timestamps* in addition to event pointers. Hence in a second approximation, an index $IS_j(v)$ is a sequence of (ts, i) entries, ordered by their value of ts. In such an index an event near a particular timestamp can be retrieved using a binary search.

It is nevertheless much more efficient to use a B+Tree structure (Fig. 5). Refining the above definition, the index $IS_j(v)$ becomes a hierarchical index comprising several levels. The index sequence described above constitutes level 0. The (ts, i) entries of that level-0 index are stored on the hard disk in small pages, where each page contains a number of entries pertaining to the same index. When such a page is

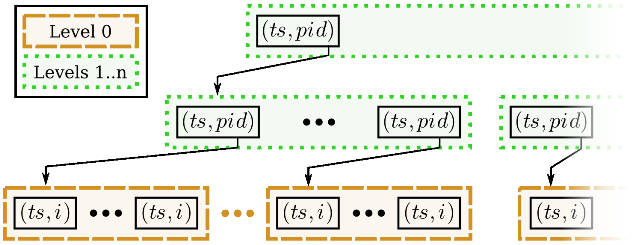


Figure 5. B+Tree index.

full, an entry of the form (ts, pid) is created in the level-1 index: ts is taken from the first (ts, i) entry of the recently-filled page, and pid is a pointer to that page. Level-1 entries are in turn accumulated in pages; when a level-1 page is filled, a level-2 entry is created, and so on. The top level always contains a single page, called the *root* page. The number of levels above level 0 of an index is called the *height* of the index. In such a structure the number of page accesses necessary to retrieve an event near a given timestamp is at most the height of the index.

Storage requirements Experiments show that the average size of an event is $\|e\| = 38$ bytes. The size of a level-0 entry is $\|(ts, i)\| = 16$ bytes (two 64-bits integers). The size of upper-level entries is $\|(ts, pid)\| = 12$ bytes (pid is only 32 bits). The experimentally-determined optimal page size is $P = 4096$ bytes, therefore level-0 index pages contain 256 entries, upper-level index pages contain 341 entries and event pages contain 108 events in average. The height h of indexes is logarithmic with respect to the number of entries and in practice never exceeds 5 (an index of height 5 allows for $341^5 \approx 4 \cdot 10^{12}$ entries).

The amount of index data generated for each event is actually greater than the event itself. In our experiments we found that in average an event plus the associated index data occupies 190 bytes of storage, although the event itself occupies only 38 bytes.

4.2 Cost of event retrieval

We now present the algorithms that permit to retrieve events matching an arbitrary predicate in *linear time* with respect to the *size of the involved indexes*. The algorithms are for timestamp-order retrieval; reverse-timestamp retrieval has the same cost.

Single-term conditions. For a simple condition of the form $a_j = C$ where C is a constant, we can retrieve matching events ordered by timestamp simply by obtaining the (ts, i) entries from $I_j(C)$. If the actual event is needed (*i.e.* for cursors), it is directly retrieved from the trace as e_i; otherwise (*i.e.* for counts) the event does not need to be accessed. In any case, all entries can be retrieved in linear time, as the index is simply scanned once.

Conjunctive conditions. For a boolean conjunction of simple conditions of the form $a_{j_1} = C_1 \wedge \ldots \wedge a_{j_m} = C_m$, we use a variant of the *sort merge join* algorithm [1], widely-used in database management systems, to identify matching events without accessing them (Algorithm 1): we obtain the $I_{j_l}(C_l)$ for every simple condition, and for each we maintain a pointer to a current (ts_l, i_l) entry. Then we loop: at every step we check if all of the i_l are equal, in which case we add any of the current entries to the result: the fact that they all point to the same event means that the event matches all conditions. Then we advance the pointer of the index whose current entry has the minimum value of ts. As each index is scanned only once and there is no nested loop, *merge join* runs in linear time with respect to the sum of the sizes of the considered indexes.

Algorithm 1 MERGE-JOIN

```
merge-join($S$, $j_1,\ldots,j_m$, $C_1,\ldots,C_m$):
  $result \leftarrow \emptyset$
  for $l = 1$ to $m$ do
    $index[l] \leftarrow I_{j_l}(C_l)$, $pos[l] \leftarrow 1$
  while there are more elements do
    $match \leftarrow true$, $refI \leftarrow -1$
    $minL \leftarrow -1$, $minTS \leftarrow +\infty$
    for $l = 1$ to $m$ do
      $(curTS, curI) \leftarrow index[l][pos[l]]$
      if $refI = -1$ then
        $refI \leftarrow curI$
      else if $curI \neq refI$ then
        $match \leftarrow false$
      if $curTS < minTS$ then
        $minTS \leftarrow curTS$, $minL \leftarrow l$
    if $match$ then
      $result \leftarrow result \cup \{s_{refI}\}$
    $pos[minL] \leftarrow pos[minL] + 1$
```

Generic boolean conditions. The above can be generalized to any compound boolean condition, by performing a *merge join* for each conjunction and a regular merge (the merging step of *merge sort*) for each disjunction. The cost thus remains linear with respect to the sum of the sizes of the considered indexes. Because both *merge join* and regular merge are stream operators (*i.e.* they produce an output tuple as soon as they have received enough input tuples, without needing past or future input tuples), it is possible to pipeline them so that no intermediate results have to be stored.

4.3 Cost of cursors

Cursors support retrieving matching events in forward or backward timestamp order, and absolute positioning by timestamp. Given a compound filtering condition, one index is used for each simple condition component. A pointer to a current entry is associated to each index and the merging algorithms described above are applied, incrementing or decrementing the pointer of each index as dictated by the desired retrieval order. The cost of retrieving successive matching events is extremely variable depending on the number of components of the condition and the density of matching events.

Algorithm 2 FIND-POSITION

```
find-position($I = I_j(v), t$):
  $page \leftarrow root(I)$
  $level \leftarrow height(I)$
  while $level > 0$ do
    $(ts, pid) = binarySearch(page, t)$
    $page \leftarrow getPage(I, pid)$
    $level \leftarrow level - 1$
  $(ts, i) = binarySearch(page, t)$
  return $i$
```

For absolute positioning, we reposition the pointer of each index so that the next timestamp of the entry is immediately before or after the specified timestamp. This is achieved by performing a binary search of the given timestamp at each level of the index, starting by the root (Algorithm 2). The number of page accesses needed by this operation is at most equal to the height of the index, and can be less if some pages are found in the page buffer.

4.4 Cost of counts

The counts queries retrieve the number of matching events in every time slice of length δt of a given interval. There are two ways these counts can be obtained.

Merge counts. The simplest way is to use the merging algorithms described previously: whenever a (ts, i) index entry corresponding to a matching event is found, the count of the time slice containing ts is incremented, without needing to fetch the actual event. This method works for arbitrary compound conditions but can be very costly if counts are required over a large interval.

Fast counts. In some cases we can leverage our hierarchical index structure to obtain counts at a much lower cost. Although this optimization applies only to simple conditions, it is useful *e.g.* to compute thread murals of the whole execution trace. Its scope can be extended if indexes of compound conditions are materialized (*i.e.* a new index is generated that references events that match the compound condition), a topic we do not address here.

The number of time slices requested in a counting query usually does not depend on the size of the interval but rather on the number of pixels of the debugger frontend window (Sect. 3.4). Therefore when counts are requested over a large interval, each time slice is also large. Because a higher-level index entry is created when a lower-level page is full (Sect. 4.1), we can know the number n of level-0 entries that are between two level-l entries for $l > 0$:

$$n = \frac{\|(ts, i)\|}{P} \cdot \left(\frac{\|(ts, pid)\|}{P}\right)^{l-1}$$

Given two consecutive level-l entries (ts_1, pid_1) and (ts_2, pid_2) of index $I_j(C)$ we know that n events matching $a_j = C$ occurred between ts_1 and ts_2. This information can then be used to provide average counts at a reduced cost. The index levels to use are determined by the ratio between the requested time slice length δt and the interval $ts_2 - ts_1$ between successive entries in each level. Note that various levels can be used during the execution of the same request, taking into account variations in the distribution of matching events: if the time between successive entries in level l is larger than δt we drill down into level $l - 1$, and conversely we roll up to level $l + 1$ if the time interval is too short.

4.5 Cost of indexing

The above sections show that the indexing scheme of the database allows for efficient query execution. It remains to show that indexes can be created efficiently so as to allow a high recording throughput. This section shows that this can be achieved by carefully tuning memory requirements.

For each event that enters the database there are at most $k = |A| - 1$ indexes to update (as there is no separate index on a_0). Experiments indicate that on average $k = 10$. Given that events arrive in order with respect to a_0, it is not necessary to use the costly B+Tree *insert* method for updating an index. Instead, the much cheaper *bulk load* method is used, which consists in appending an entry at the end of the current level-0 page, and at the end of higher-level pages whenever a lower-level page is filled. The I/O and memory costs of this operation are as follows:

- If the current page of each level of the index can be kept in memory, an I/O cost is incurred only when a page is filled. The average number of page accesses per incoming event is:

  $$\frac{\|e\| + k \cdot (\|(ts, i)\| + A)}{P} \simeq \mathbf{0.05}$$

  where A is the contribution of level 1 and above:

  $$A = \frac{\|(ts, i)\|}{P} \cdot \|(ts, pid)\| \cdot \sum_{i=0}^{h-1} \left(\frac{\|(ts, pid)\|}{P}\right)^i$$

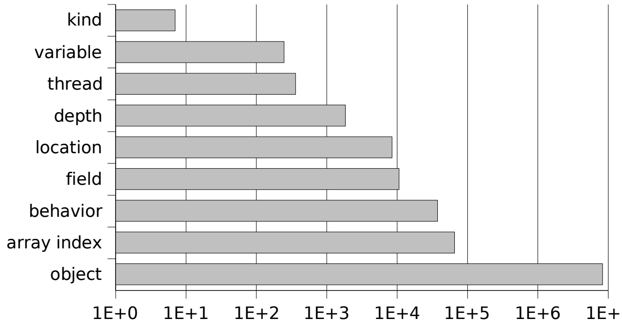
- If only the current level-0 page of the index can be kept in memory, when a page is filled it must be written, the current level-1 page read, updated and written back to disk. The contribution of higher levels become:

  $$A = \frac{\|(ts, i)\|}{P} \cdot 2P \cdot \sum_{i=0}^{h-1} \left(\frac{\|(ts, pid)\|}{P}\right)^i$$

  The average number of page access per incoming event is then $\mathbf{0.13}$.

- If no index page can be kept in memory, every update implies the three operations above, giving $2 \cdot k = \mathbf{20}$ accesses per event.

In order to achieve a high recording throughput it is therefore crucial to minimize the number of page accesses per incoming event: at least one page per index should be kept in memory so as to avoid the last situation, which is 150 times more costly than the second situation above.



The number of indexes varies from 7 for the *kind* index set to more than 8 millions for the *object* index set in an execution trace of 720 million events.

Figure 6. Number of indexes for each index set in an Eclipse execution trace.

Index set splitting	*Number of indexes*	*Entries per index*	*Query cost*
No	▭ d	B/d	B/d
Yes	▭ $\sqrt[N]{d}$	$B/\sqrt[N]{d}$	$\frac{N \cdot B}{\sqrt[N]{d}}$
	⋮ ⋮	⋮	
	▭ $\sqrt[N]{d}$	$B/\sqrt[N]{d}$	

Effect of index set splitting on the number of indexes, number of entries per index, and query cost. N is the number of split components and B is the total number of entries in the index set.

Table 3. Index set splitting.

Memory requirements The memory requirements of the system depend on the number of indexes to maintain, which in turn depends on the size of the domain of each attribute. Figure 6 shows the domain size of each attribute as observed with a large execution trace of an Eclipse session (720 millions events). The domain of object ids largely dominates all other domains, reaching almost 10 millions distinct values. Maintaining the corresponding indexes would require $P \cdot 10^7 = 40$GB of buffer space, which is not a reasonable figure. A solution to this problem is to split the index sets of large attributes, as explained below.

Index set splitting As maintaining millions of indexes is not practically feasible, we devised a strategy that permits to trade memory requirements for recording throughput and querying efficiency: attributes with large domains are *split* into components that are indexed separately.

Let a_j be an attribute and $d = |D_j|$ the number of distinct values it can take (hence d is also the number of indexes in the corresponding index set). Assuming that all the distinct values are the first d positive integers –which is always the case in practice–, any value v of D_j can be represented in binary by $n = log_2(d)$ bits. Such a value can be split into N components of n/N bits each, and instead of having a single index set on attribute a_j there are now N index sets, one for each component. The number of indexes to maintain becomes $N \cdot 2^{n/N} = N \cdot \sqrt[N]{d}$ instead of d, yielding a dramatic reduction of memory requirements, even for N as low as 2. For instance with $d = 10^7$, corresponding to the size of the *object id* domain, the memory requirements using index set splitting with $N = 2$ would be reduced from 40GB to 25MB.

Index set splitting therefore implies huge reduction of memory requirements. Let us now assess the impact of this technique on efficiency. For recording, the number of index updates is multiplied at most by N. Given that not all events have values for split attributes, the actual slowdown is lower.

For querying, boolean expressions involving split indexes are replaced by a conjunction of N conditions, one for each value component. As shown in Sect. 4.2, the efficiency of queries is proportional to the size of the involved indexes. Table 3 summarizes the slowdown incurred by splitting an index set containing d indexes and totalling B entries. Each index in the set contains on average B/d entries, thus the cost of a query on one of those indexes is proportional to B/d. If the index set is split the number of indexes per index set becomes $d' = \sqrt[N]{d}$ and there are on average B/d' entries per index. The cost of the query becomes proportional to $N \cdot B/d'$, yielding a slowdown of $N \cdot d/d' = N \cdot d^{1-\frac{1}{N}}$. For instance, with $d = 10^7$ and $N = 2$, the slowdown would be approx. $6,300$. Although this might seem prohibitive, it is important to note that the index sets that are subject to splitting have domains orders of magnitude larger than other index sets (Fig. 6), thus each individual index is small compared to the indexes of non-split index sets. As in practice most queries are compound and involve both split and non-split index sets, the contribution of split index sets to the *total cost* of the query is reasonable.

To illustrate this, let us consider the *state reconstitution* query of Sect. 3.4, which is based on a conjunction of conditions on the *field id* and *object id* attributes[5]. In the Eclipse trace previously mentioned there are about $10,000$ distinct *field id* values and $10,000,000$ distinct *object id* values (Fig. 6). With a trace containing B events, assuming a uniform distribution of *field id* and *object id* values, and assuming that every event has a value for both attributes, each index on *field id* would contain $B/10^4$ entries; each index on *object id* would contain $B/10^7$ entries without index set splitting, and approx. $B/\sqrt{10^7} = B/3,160$ with index set splitting and $N = 2$. Therefore the actual slowdown of index set splitting for the compound query is:

$$\frac{cost\,with\,splitting}{cost\,without\,splitting} = \frac{1/10^4 + 2/3160}{1/10^4 + 1/10^7} \approx 7$$

Despite this slowdown, *state reconstitution* queries execute fast enough to be used interactively in the debugger frontend, as will be shown in Sect. 6.2.

[5] The $kind = FW$ part of the query is omitted in practice because only Field Write events have a value for the *field id* attribute.

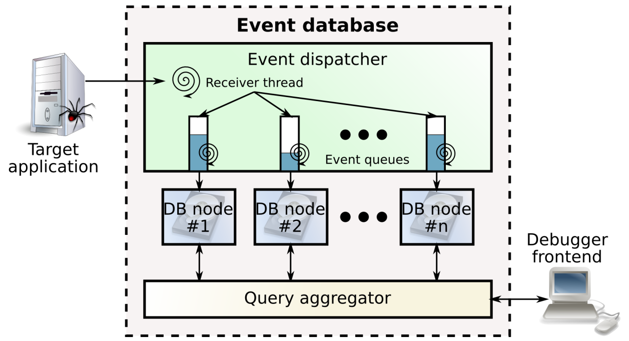


Figure 7. Architecture of the distributed database backend.

5. Scaling Up with a Debugging Cluster

The efficient indexing and retrieval techniques used in the event database of TOD can benefit from parallelization. This section shows how TOD supports a distributed database backend, allowing its efficiency to increase linearly in terms of the number of nodes, within certain limits.

5.1 Distributed Architecture

The architecture of the distributed backend of TOD consists of three layers (Fig. 7):

- A *dispatcher* that receives the events from the target program and distributes them to a number of database nodes. The dispatcher maintains a local sending queue for each connected database node. A receiving thread reads each incoming event and forwards it to the smallest queue, so as to achieve proper load balancing between database nodes.
- A number of *database nodes*, each of which receives a subset of all generated events. They are individually able to index events and process queries in the same way as the non-distributed backend described in Section 4. No change to the indexing structure is necessary.
- A *query aggregator* that receives queries from the debugger frontend, passes them to each database node and aggregates the results before returning them to the frontend.

Note that neither the structure database nor the weaver mentioned in 3.1 need to be parallelized as their processing and storage requirements are modest.

5.2 Scalability

Parallelization Both event recording and query processing are embarrassingly parallel problems, that is to say, their parallelization is straightforward because no special coordination is required between the parallel tasks. In particular, queries do not need to perform any kind of joins between events. All database nodes can perform the same query independently and then send their results to the aggregator, which is able to merge them efficiently.

Furthermore, cursors and counts have very light processing and bandwidth requirements on the aggregator, enabling excellent scalability properties:

Parallel cursors. When the aggregator receives a cursor query with filtering condition Q it requests a similar cursor to each database node and returns an *aggregating cursor* to the client. In the same way regular cursors merge entries from various indexes, the aggregating cursor merges events from each of its base cursors using the regular merge algorithm from merge sort.

Parallel counts. The aggregator obtains partial count results from each node and simply returns a new counts array where the value of each slot is the sum of the values of the corresponding slot in each partial result array.

Scalability limits The throughput of this architecture is theoretically linear in terms of the number of database nodes. However the scalability is in practice limited by one of these factors: the dispatcher (resp. aggregator) can act as a bottleneck for recording throughput (resp. query processing), or the network link bandwidth can be saturated. In our current implementation, the actual bottleneck is the dispatcher, as reported in details in the following benchmarks.

6. Benchmarks

This section reports on a first set of benchmarks evaluating different aspects of TOD. In particular, we first measure the overhead imposed on a running application debugged with TOD, and then report on the efficiency and scalability of the distributed database backend, both in terms of recording throughput and query evaluation.

6.1 Trace capture overhead

Capturing the execution trace of a debugged program causes a significant runtime overhead. We measured it in two different scenarios:

- A fully-instrumented, CPU-intensive toy program designed to represent a *worst-case situation*, in which the debugged applications emits events at a rate as high as the CPU can handle;
- An interactive Eclipse session reflecting a *real-world situation*, in which the JDK classes are *not* instrumented (partial traces are further discussed in Sect. 7), and in which the interaction between the user and the debugged application entails that the event emission rate is less sustained in time than in the worst case above.

In these experiments only the event emission overhead caused by TOD is measured, not its database performance. Therefore events are simply written to disk, without any indexing. Both benchmarks were conducted on a Pentium M 2GHz notebook with 1GB of RAM running Linux kernel 2.6.17 and the Sun 1.5.0_08 JVM.

Setup	*RAM*	*time*	*emit.*	*rec.*	*rate*	*ovh.*
None	16	1.53	-	-	-	1
ODB1	500	179	110m	5m	614	116
ODB2	64	188	110m	530k	585	122
TOD	16	173	90m	90m	520	113

The *RAM* column is the JVM heap size in MB. The *time* column is the execution time in seconds. The *emit.* and *rec.* columns indicate the number of events emitted, and recorded (available to the debugger). The *rate* column is the recording throughput, in kEv/s. The *ovh.* column is the overhead compared to the standalone execution.

Table 4. Overhead of event emission.

Worst-case scenario We use a CPU-intensive program that creates 100 `Object` instances and then iterates 10 million times in a loop taking one of these objects at random and passing it to a method that performs a simple arithmetic operation on its hash code. The program does not call any non-instrumented method. Therefore, every execution step emit events, so the event emission rate is bounded only by the CPU speed.

We compare the execution time of this program running *(a)* standalone, *(b)* with TOD and *(c)* with the ODB [15] omniscient debugger for Java. Results are presented in Table 4. With ODB, events are stored *in the JVM heap* of the target program; old events are discarded when the heap is full. We therefore conducted two ODB tests, varying the JVM heap size: with 500MB of heap we were able to record 5 million events out of the 110 million emitted during program execution, while with 64MB we could record only 500,000 events. On the other hand with TOD we were able to record all emitted events[6] without interfering with the JVM heap. In spite of the heavier processing in the case of TOD –where events are serialized and written to disk rather than simply kept in RAM– the overhead imposed on the application execution time is similar in TOD and ODB: around 115 times the cost of standalone execution. The execution trace generated by TOD weighs in at 3.6GB.

Eclipse session This experiment consists in performing a sequence of actions in the Eclipse IDE, with and without trace capture. Note that only the classes of the Eclipse IDE are instrumented, not those of the JDK (Sect. 7).

The actions performed are: creation of a new project, creation of a few classes, edition of their source code using auto-completion and other productivity features, execution of a rename refactoring, and step-by-step execution of the created program under the Eclipse integrated debugger.
The following *quantitative* observations can be made:

- The Eclipse session is 10 times slower with trace capture enabled: it takes 244s (4 min.) without trace capture and 2324s (38 min.) with trace capture.
- The recorded execution trace comprises around 720 million events and weighs in at 33GB. The average event emission rate is 313kEv/s, 40% less than the worst-case scenario presented above.

On the *qualitative* side, this experiment shows that:

- The start-up time of Eclipse is greatly augmented when trace capture is enabled, due to the loading of instrumented classes (which are roughly 3 times bigger than non-instrumented classes).
- The Java source editor remains interactive for typing, although there is a noticeable slowdown.
- Some operations, such as invoking auto-completion, generating constructors or getters, or stepping with the debugger, are significantly slower with trace capture, but at a tolerable level.

Even though using TOD implies a perceptible slowdown of the debugged program, we believe that the benefits of omniscient debugging in quickly pinpointing hard-to-find bugs far outweigh this inconvenience.

6.2 Database performance

To evaluate the performance of the distributed database of TOD we conducted measurements of recording throughput and query performance against the captured Eclipse trace of Sect. 6.1. Recall that the database performance is crucial to the debugging experience: *(a)* trace recording should ideally be in real time so that it is possible to start a debugging session as soon as the debugged program terminates (or reaches some determined state), and *(b)* the database must process queries in times compatible with human interaction so that the navigation interface is responsive.

Cluster setup. The TOD distributed trace database was deployed for these experiments on a dedicated cluster consisting of 10 database nodes (3GHz Intel Pentium 4 with Hyper-Threading disabled, 1GB of RAM, Sun JDK 1.5.0_08) and one dispatcher node (2.13GHz Intel Core2, 2GB of RAM, Sun JDK 1.6.0). The nodes are connected through a Gigabit Ethernet switch, but only the dispatcher node has a Gigabit link; the database nodes have a 100Mbit/s link. Each database node has a partition with 38GB of free space on a 80GB 7200RPM SATA hard drive.

Recording. The first experiment consists in determining the maximum throughput achievable by the dispatcher, with event storage and indexing disabled in the database nodes. As shown in Fig. 8, in the setup with only one database node, the recording throughput is limited to 250kEv/s, which corresponds to the limitation imposed by the 100Mbit/s network link. With more than one node, the dispatcher is able to handle up to 470kEv/s, regardless of the number of nodes. This represents around 20MB/s of outgoing network traffic on the dispatcher, which is lower than what is achievable with a Gigabit link: surprisingly the bottleneck of the dispatcher is the

[6] The different numbers of emitted events between TOD and ODB are apparently due to differences in the trace model.

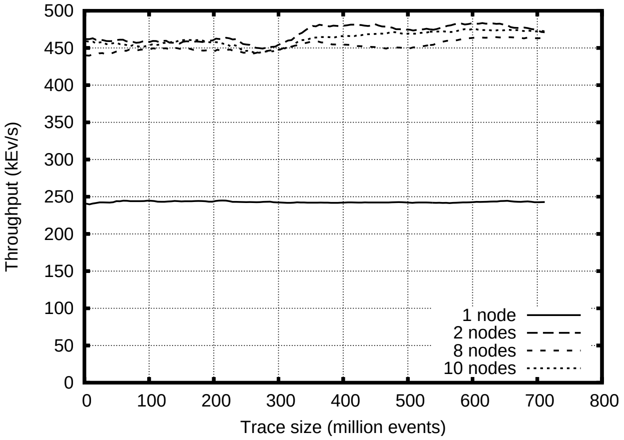


Figure 8. Dispatcher throughput.

CPU. Profiling shows that most of the time is spent copying buffers in methods of the `java.io` framework. We assume that this issue would disappear in an optimized C version of the dispatcher.

Fig. 9 shows the evolution of recording throughput as events are added to the database. In average, a single database node is able to handle around 54kEv/s, and with 10 nodes we reach the dispatcher limit with 470kEv/s. The throughput of 54kEv/s of a single node translates to around 10MB/s of disk writes. This is lower than the maximum throughput of the disks that were used (around 40MB/s), and again the bottleneck is the CPU: most of the time is spent in methods of `DataInputStream` marshalling and unmarshalling primitive values. Note that with less than 4 nodes it is impossible to record the whole trace due to disk space constraints; therefore the following benchmarks consider scalability starting at 4 nodes.

Stepping queries. Figure 10 shows the efficiency of the *step into* and *step over* queries (Sect. 3.4). These results are obtained by starting a stepper at a random timestamp on each of the 350 threads of the recorded Eclipse session and performing 100 step operations. It is clear that step into queries are faster than step over queries, due to the fact that the former translate to a cursor query on thread id while the latter additionally use the call stack depth. The efficiency of both step queries surprisingly decreases as more database nodes are used; we are currently investigating this issue more thoroughly. In any case, step queries are fast enough to be used interactively, since they execute in less than a hundred milliseconds in the worst case.

Object reconstitution queries. The efficiency of object reconstitution queries is measured as the time taken to reconstitute the state of random objects of the Eclipse execution trace at different points in time. Figure 11 shows that these queries scale well with the number of nodes. On average, individual field values are retrieved in 120ms to 350ms. The time to reconstitute a full object is directly proportional to the number of its fields, thus the time to reconstitute an object of 7 fields (an average number) is comprised between 0.8s and 2.4s. Note that the object inspector window of TOD updates asynchronously, so that the user is not blocked until the state of the current object is fully reconstituted.

nodes	*merge (ms)*	*fast (ms)*	*speedup*	*dist. (%)*
1	7444	233	31.9x	1.54
2	4062	222	18.3x	1.11
8	1206	120	10.1x	0.18
10	1114	117	9.5x	0.21

The *merge* and *fast* columns indicate the average query execution time using two counting methods. The *speedup* column indicates how much faster is the fast method. The *dist.* column is the distortion of the fast method compared to the exact merge method.

Table 5. Comparison of merge and fast count queries.

Count queries. We measured the execution speed of count queries and compared the two counting methods described in Sect. 4.4: merge counts and fast counts. We requested the event counts for each of the 350 threads of the Eclipse execution trace, on the entire time span of the trace and divided in $n = 1,000$ subintervals.

A comparison of the two count techniques is shown in Table 5. Fast counts perform 10 to 30 time faster than merge counts while providing a very precise approximation, with a distortion[7] below 2%. Figure 12 shows that merge counts scale very well but fast counts less so, because as each node records less events, the fast count algorithm must more frequently resort to lower-level indexes.

6.3 Summary

Figure 13 summarizes our experimental results regarding trace capture and recording: the rate of event emission varies from 313kEv/s for a partially-instrumented interactive Eclipse session to 520kEv/s for a fully-instrumented CPU-intensive program; the recording throughput boasts a perfect scalability up to 8 nodes, and reaches 470kEv/s with 10 database nodes, where it is limited by the dispatcher bottleneck. It is therefore possible to record execution traces almost in real time. Count queries display good scalability, while step queries scale poorly. Still, the database is able to execute queries at a speed compatible with interactive navigation.

As a bottom line, although the results presented in this section could with no doubt be further enhanced through various optimizations, they already represent a consequent improvement over other existing implementations of omniscient debuggers. TOD is practically usable today, even on large traces produced by complex applications.

[7] Calculated as: $(\sum_{i=1}^{n} |merge[i] - fast[i]|) / \sum_{i=1}^{n} merge[i]$

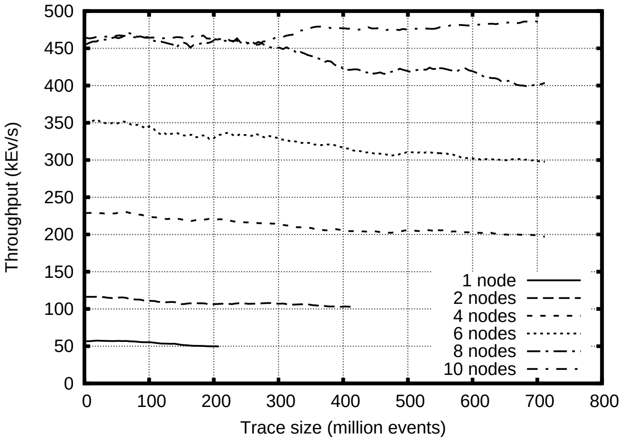


Figure 9. Recording throughput.

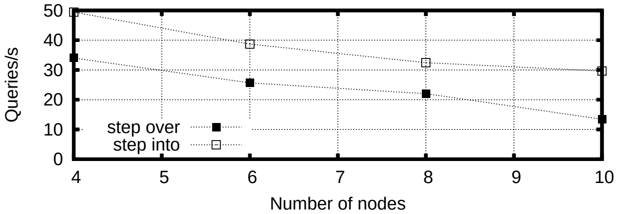
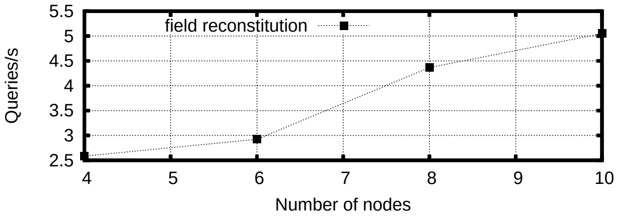
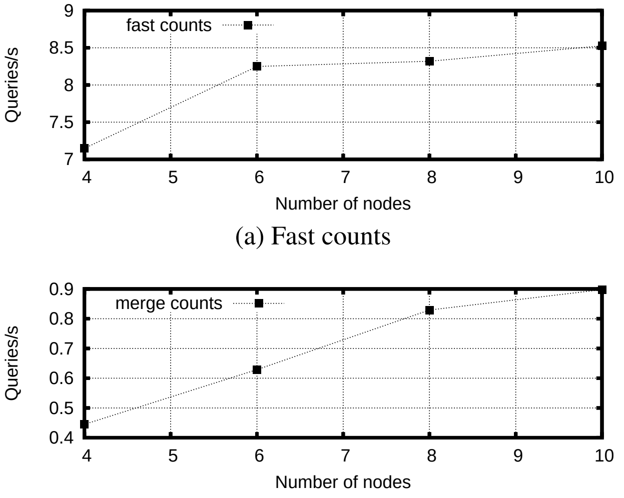


Figure 10. Efficiency of stepping queries.



The graph shows the number of field values that can be reconstituted per second.

Figure 11. Efficiency of object reconstitution queries.



(a) Fast counts

(b) Merge counts

Figure 12. Efficiency of counts queries.

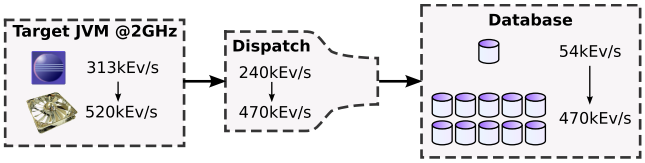


Figure 13. Experimental results for trace emission, dispatching, and recording.

7. Working with Partial Traces

Although TOD is designed to support huge execution traces, it is not always practical to record each and every event: the runtime overhead of event capture is important (Sect. 6.1), and so is the storage requirement. The idea of *partial traces* is to leverage the fact that during the development of a piece of software, some components are *trusted*, i.e. mature and well-tested, and it may not be necessary to generate and store events for the inner activities of these components. This section shows how scoped trace capture can facilitate debugging and how TOD makes it possible to work with partial traces.

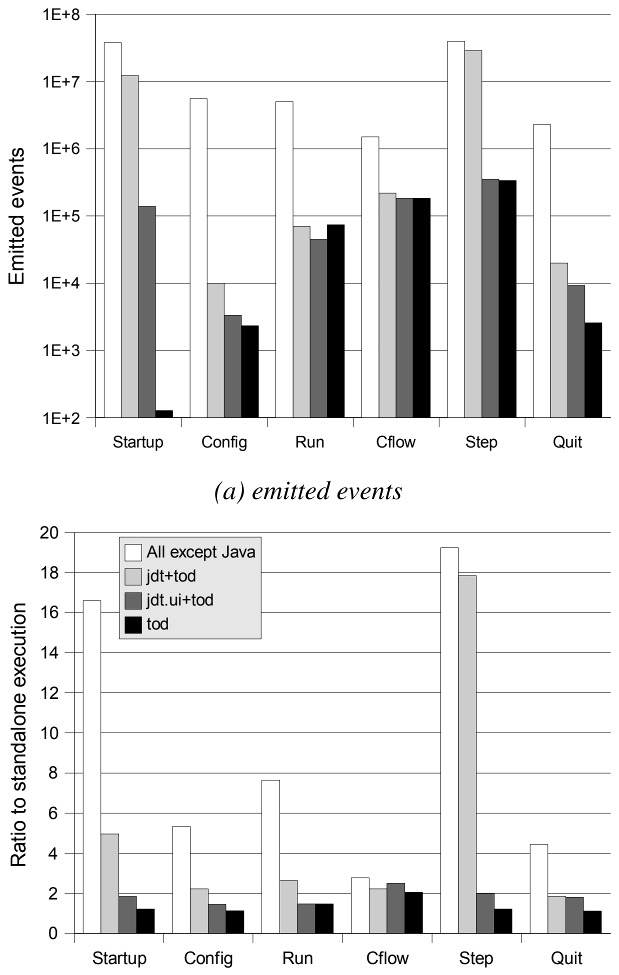
7.1 Motivating example: debugging the TOD Eclipse plugin

Let us consider as an example the debugging of the TOD Eclipse plugin itself. This example is fairly representative of component development for existing, trusted, frameworks or plugin architectures. Here, we might be interested in two types of bugs: those that are internal to the plugin and those that relate to the interaction between the plugin and the platform. In the first case, we do not need to capture events that occur within the Eclipse platform because it is considered trusted. In the second case, we have to record events that occur within the platform, but not necessarily all of them: it might be enough to record events of the Java tooling (JDT), or only of some part of it, for instance the UI.

Figure 14 shows the impact of different trace scoping strategies on both the number of emitted events and the runtime overhead of trace capture, during different phases of the execution of the TOD plugin. In this small experiment we see that by appropriately scoping the trace capture, there are up to five orders of magnitude of difference in the number of emitted events (Fig. 14a), and that the gains in runtime overhead can be up to 20 times (Fig. 14b).

7.2 Dealing with missing information

Working with partial traces greatly enhances the applicability of TOD, but it implies that some information is lacking to reconstruct the whole history of the debugged program. It is therefore important that TOD *systematically reports* on missing information so that the user can soundly reason about the presented information. Missing information manifests in control flow and state reconstitution.



(a) emitted events

(b) runtime overhead

The measures are taken after the following execution phases are passed: the IDE starts up; the TOD launch configuration dialog is opened; the target program is run; the control flow view is opened; events are navigated step by step; and the IDE exits.

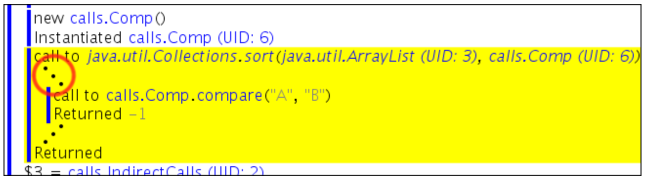
Figure 14. Emitted events and runtime overhead using scoped capture.

Control flow reconstitution. When non-instrumented code is called from instrumented code, and in turn calls instrumented code, some control flow information is lost. Such a case is illustrated in Fig. 15. The code in Fig. 15a calls a non-instrumented JDK method (`Collections.sort`) from an instrumented one (the `main` method). The `sort` method in turn calls the instrumented `Comp.compare` method, but indirectly (through the `sort` and `mergeSort` methods of `Arrays`). In Fig. 15b the small dots indicate that control flow information is missing. In the absence of such an indication the user might think that `Comp.compare` was called directly and was the only method called by `sort`, which is not the case.

State reconstitution. If a class has a non-private field that is written to by non-instrumented code, the value of this field at a given point in time cannot be determined accurately. TOD represents these fields in a distinctive color in the corresponding views. Again, without such a warning the user might not be able to reason accurately about the program.

```
public class Comp
            implements Comparator<String> {
 public int compare(String s1, String s2) {
  return s1.compareToIgnoreCase(s2);
 }
 public static void main(String[] args) {
  List l = new ArrayList();
  l.add("A"); l.add("B");
  Collections.sort(l, new Comp());
}}
```

(a) Code excerpt


```
new calls.Comp()
Instantiated calls.Comp (UID: 6)
call to java.util.Collections.sort(java.util.ArrayList (UID: 3), calls.Comp (UID: 6))
  call to calls.Comp.compare("A", "B")
  Returned -1
Returned
$3 = calls.IndirectCalls (UID: 2)
```

(b) Control flow view

Figure 15. Materialization of incomplete control flow information.

7.3 Specifying partial traces

Partial traces are supported by means of mechanisms similar to those of partial behavioral reflection [27]: both spatial and temporal selection of event generation. For spatial scoping, TOD supports class selectors, which are predicates on classes that should generate events (*e.g.* classes of a certain set of packages).[8] For temporal scoping, TOD supports dynamic activation of event generation, either globally or per thread, through a simple API. This is particularly useful in situations where a bug occurs after a long running time, or under specific dynamic conditions (which may for instance be related to control or data flow properties).

Implementation In our current prototype event emission code is woven with the original application code at load time. As a consequence, the spatial scope of event emission is fixed for the whole debugging session.[9] Temporal scoping is achieved by a flag check in event emission code, so there is still very light runtime overhead when event emission is disabled at runtime, compared to non-instrumented code.

[8] It would be possible to refine spatial scoping using operation selectors [27], enabling expression-level selection to further reduce the size of the execution trace. This feature has not yet been integrated in TOD.

[9] Although recent JVMs allow classes to be modified at runtime, we do not yet use this feature.

8. Related work

The work on TOD relates to different areas. Omniscient debugging of course, but execution trace recording is also used in a broader range of program understanding approaches, in particular query-based debugging and profiling. Techniques improving the efficiency of program understanding have been proposed in several areas like profiling and debugging of distributed application. We also discuss how our work on the database of TOD relates to general database techniques.

Omniscient debugging. Three proposals of omniscient debuggers are related to TOD. ZStep 95 is a *reversible stepper* for Lisp. In addition to the standard features of omniscient debuggers, ZStep 95 provides animated views of data structures of the debugged program. It provides excellent solutions to the cognitive issues of debugging but does not address performance and scalability. More recently, Bil Lewis proposed an omniscient debugger for Java, ODB [15], and Hofer *et al.* implemented a similar system for Smalltalk, called Unstuck [11]. The work on TOD was actually inspired by the omniscient debugger of Bil Lewis. It has the ability to not only navigate the execution history but also to *restore* the state of the program as it was at any given point in time, so that its execution can be resumed at that point. Events are stored in the target program's address space, which has serious limitations in terms of scalability and potential influence of the debugger on the behavior of the debugged application. For scalability, ODB makes it possible to set a fixed limit on the number of events that can be kept in the execution trace; older events are discarded. TOD provides much better scalability, as demonstrated in this paper. Furthermore, both ODB and Unstuck lack the high-level overviews that are provided by TOD in murals.

CodeGuide [21] is a commercial development environment for Java that features a back-in-time debugger. Breakpoint-based debugging can be combined with trace-based, bi-directional stepping. The trace is however limited to the last few thousands events, and the important feature of root cause finding is not available. High-level overviews are also not provided.

Query-based debugging. Query-based debugging consists in identifying events that match a query expressed in a high-level language. Queries can be formulated *a priori* (before running the program) or *a posteriori* (after the program has been executed). In Hy+ [4] a-posteriori queries are expressed in a graphical language and deal with distributed computations. PQL [19] provides a very high-level and powerful a-priori query model. Whyline [13] guides the programmer by proposing a set of possible a-posteriori queries. The TQuel language allows programmers to express a-priori queries declaratively, providing explicit support for temporal queries [25]. LeDoux and Parker [14] formulate a-posteriori Prolog queries on the execution of concurrent Ada programs. Opium [7] uses Prolog queries to debug Prolog programs and seamlessly supports breakpoint-based debugging and trace-based debugging. Coca [6] uses a-priori Prolog queries to debug C programs. In an upside down approach the Mercury Declarative Debugger [18] *asks* the user questions about the correctness of computations performed by the program so as to quickly locate incorrect ones.

The limited class of queries supported by TOD is sufficient for the features of omniscient debugging, but scalability and efficiency come at the expense of a much less expressive query model than those provided in the above approaches. Although in TOD basic queries can be combined algorithmically, queries that relate several events cannot be executed efficiently.

Trace reduction. One of the priorities of profiling is to reduce the performance impact of the tool on the target application. One technique consists in reducing the trace via clustering [20, 28], which also reduces the overhead of capture, although at the price of a loss in precision. Debugging distributed applications benefits from high-level trace recording, as only message sends between nodes need be recorded [20, 4]: useful views of the computations can be provided with much less information than that used in omniscient debugging. Opium [7] lets the user specify prefiltering predicates, that by filtering out uninteresting events permit to reduce the number of context switches between the debugged process and the debugger. Mercury [18] by default only records events up to a certain call depth; if more detail is needed relevant goals are automatically re-executed.

With TOD we took the opposite approach, providing a scalable event database that copes with huge execution traces. However TOD also provides a mean for reducing the size of the execution traces by letting the user select which parts of the program emit events.

Replay-based debugging. Back-in-time debugging can be achieved by *replaying* the debugged program until some determined point before the current execution point. Igor [8] and Bdb [2] make use of periodic state saving, or *checkpoints*, to reduce the time needed to reach a particular past execution point: execution is resumed at the last checkpoint preceding the desired execution point. The main advantage of replay-based debugging compared to trace capture is the lower runtime overhead (around 2x for Bdb and 4x for Igor, versus a maximum of 115x for TOD). However, backward debugging moves can be slow, especially when going far away from the current execution point. Furthermore, if the execution point is moved backwards, moving it again forwards means re-executing the program, which is not practical for long-running programs. With TOD the entire history of the program is available and freely navigatable.

A crucial issue of replay-based debugging is that of deterministic replay: system calls that rely on external resources such as network connections might return different results at different times. Bdb [2] and Jockey [23] address this by recording the results of non-deterministic system calls and reinjecting them into the program when it is replayed. However this is a brittle solution as many system calls must be handled in different ways.

Database techniques. Finally, the importance of physical data layout in the efficiency of several relational data indexing techniques has been shown in [1]. Seshadri *et al.* [24] present query plan optimization techniques for sequential databases, a superset of execution trace databases like ours. Stonebraker *et al.* [26] make a strong point in favor of specialized database management systems for specific applications. Our work on TOD applies classical indexing and paging techniques in a domain-specific manner, leveraging the very specificities of execution traces.

9. Conclusion

Assuming the great potential of omniscient debuggers in alleviating one of the most tedious and costly part of software development, this work shows that it is realistic to provide omniscient debuggers in modern development environments if appropriate measures are taken to address the associated efficiency, scalability, and usability issues.

We have presented TOD, a Trace-Oriented Debugger for Java, which contributes to the scalability of omniscient debugging at three levels: *(a)* at the trace generation level, by relying on an efficient ad-hoc weaver providing selective emission of events encoded in a concise binary format; *(b)* at the storage and query level, by proposing a specialized distributed database with an optimized indexing scheme; and *(c)* at the user interface level, by providing specialized interface components, in particular murals, which ease the interactive analysis of huge event traces, and visual feedback supporting the use of partial traces. The scalability of TOD has been shown by giving both a complexity analysis of the indexing and querying algorithms, and by reporting on benchmarks of the actual prototype.

There are several promising directions for experimenting with other techniques enhancing the applicability of omniscient debuggers. Full-scale experiments of using TOD in large real-world development projects would be invaluable for empirically assessing the benefits of omniscient debugging. At the database level, indexes for frequently-used compound conditions could be materialized so as to improve query efficiency, and the overhead caused by event emission could be strongly reduced by not reifying redundant information. For dealing with partial traces, it would be interesting to leverage the hot swap feature of modern JVMs for adding or removing instrumentation at runtime. It would also be worthy to refine spatial scoping to the expression level, and to explore the use of static analysis to reduce even more the set of generated events. Finally, specific behavior simulations could be provided for trusted and widely-used classes (*e.g.* `ArrayList`), so that their state could be reconstituted without needing to fully instrument their internals.

Acknowledgments

We thank Bil Lewis for his inspiring work on the Omniscient Debugger, Johan Fabry and Jacques Noyé for their invaluable feedback on this manuscript, and Sergio Aguilera and Hernan Cuevas for their support with the cluster used for the benchmarks. We also thank the anonymous OOPSLA reviewers for their insightful comments. This work is partially financed by the EU NoE CoreGRID.

References

[1] M. Blasgen and K. Eswaran. Storage and access in relational databases. *IBM Systems Journal*, 16(4):363, 1977.

[2] Bob Boothe. Efficient algorithms for bidirectional debugging. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 299–310, New York, NY, USA, 2000. ACM Press.

[3] Éric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: a code manipulation tool to implement adaptable systems. In *Proceedings of the ASF (ACM SIGOPS France) Journées Composants 2002: Adaptable and extensible component systems*, November 2002.

[4] Mariano P. Consens, Masum Z. Hasan, and Alberto O. Mendelzon. Visualizing and querying distributed event traces with Hy+. In *Proceedings of the International Conference on Application of Databases*, volume 819, pages 123–141. LNCS, 1994.

[5] Jim des Rivières and John Wiegand. Eclipse: A platform for integrating development tools. *IBM Systems Journal*, 43(2):371–383, 2004.

[6] Mireille Ducassé. Coca: an automated debugger for c. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 504–513, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[7] Mireille Ducassé. Opium: An extendable trace analyzer for prolog. *Journal of Logic Programming*, 39(1-3):177–223, 1999.

[8] Stuart I. Feldman and Channing B. Brown. Igor: a system for program debugging via reversible execution. In *PADD '88: Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging*, pages 112–123, New York, NY, USA, 1988. ACM Press.

[9] Abdelwahab Hamou-Lhadj and Timothy C. Lethbridge. Compression techniques to simplify the analysis of large execution traces. In *IWPC '02: Proceedings of the 10th International Workshop on Program Comprehension*, page 159, Washington, DC, USA, 2002. IEEE Computer Society.

[10] Charles R. Hill. A real-time microprocessor debugging technique. In *SIGSOFT '83: Proceedings of the symposium on High-level debugging*, pages 145–148, New York, NY, USA, 1983. ACM Press.

[11] Christoph Hofer, Marcus Denker, and Stéphane Ducasse. Implementing a backward-in-time debugger. In *Proceedings of NODe'06*, volume P-88, pages 17–32. Lecture Notes in Informatics, 2006.

[12] Dean F. Jerding and John T. Stasko. The information mural: A technique for displaying and navigating large information spaces. *IEEE Trans. Vis. Comput. Graph.*, 4(3):257–271, 1998.

[13] Andrew J. Ko and Brad A. Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In Elizabeth Dykstra-Erickson and Manfred Tscheligi, editors, *CHI*, pages 151–158. ACM, 2004.

[14] Carol H. LeDoux and D. Stott Parker, Jr. Saving traces for ada debugging. In *Ada in Use, Proceedings of the Ada International Conference*, pages 97–108, September 1985. Published as ACM Ada Letters, volume 5, number 2.

[15] Bil Lewis. Debugging backwards in time. In M. Ronsse and K. De Bosschere, editors, *Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003)*, Ghent, Belgium, 2003.

[16] Henry Lieberman. Reversible object-oriented interpreters. In Jean Bézivin, Jean-Marie Hullot, Pierre Cointe, and Henry Lieberman, editors, *ECOOP*, volume 276 of *Lecture Notes in Computer Science*, pages 11–19. Springer, 1987.

[17] Henry Lieberman and Christoper Fry. ZStep 95: A reversible, animated source code stepper. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualization — Programming as a Multimedia Experience*, pages 277–292, Cambridge, MA-London, 1998. The MIT Press.

[18] Ian MacLarty, Zoltan Somogyi, and Mark Brown. Divide-and-query and subterm dependency tracking in the mercury declarative debugger. In *AADEBUG'05: Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 59–68, New York, NY, USA, 2005. ACM Press.

[19] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: a program query language. *ACM SIGPLAN Notices*, 40(10):365–383, October 2005.

[20] O. Y. Nickolayev, P. C. Roth, and D. A. Reed. Real-time statistical clustering for event trace reduction. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):144–159, Summer 1997.

[21] OmniCore. Codeguide back-in-time debugger.

[22] Guillaume Pothier. Benchmarks of COTS database management systems. Technical Report TR/DCC-2006-16, University of Chile, October 2006.

[23] Yasushi Saito. Jockey: a user-space library for record-replay debugging. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging (AADEBUG 2005)*, pages 69–76, New York, NY, USA, 2005. ACM Press.

[24] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. Sequence query processing. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 23(2):430–441, June 1994.

[25] Richard Snodgrass. Monitoring in a software development environment: A relational approach. *SIGPLAN Not.*, 19(5):124–131, 1984.

[26] Michael Stonebraker, Chuck Bear, Ugur Çetintemel, Mitch Cherniack, Tingjian Ge, Nabil Hachem, Stavros Harizopoulos, John Lifter, Jennie Rogers, and Stanley B. Zdonik. One size fits all? part 2: Benchmarking studies. In *Conference on Innovative Data Systems Research (CIDR 2007)*, pages 173–184. www.crdrdb.org, 2007.

[27] Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In Ron Crocker and Guy L. Steele, Jr., editors, *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2003)*, pages 27–46, Anaheim, CA, USA, October 2003. ACM Press. ACM SIGPLAN Notices, 38(11).

[28] Andy Zaidman and Serge Demeyer. Managing trace data volume through a heuristical clustering process based on event execution frequency. In *CSMR '04: Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'04)*, page 329, Washington, DC, USA, 2004. IEEE Computer Society.