TOWARDS PRACTICAL OMNISCIENT DEBUGGING

TESIS PARA OPTAR AL GRADO DE DOCTOR EN CIENCIAS MENCIÓN COMPUTACIÓN

GUILLAUME POTHIER

SANTIAGO DE CHILE
JUNIO 2011
TOWARDS PRACTICAL OMNISCIENT DEBUGGING

TESIS PARA OPTAR AL GRADO DE DOCTOR EN CIENCIAS MENCIÓN COMPUTACIÓN

GUILLAUME POTHIER

PROFESOR GUÍA: Éric Tanter

MIEMBROS DE LA COMISION:
Shigeru Chiba
Johan Fabry
Luis Mateu Brule
Romain Robbes

SANTIAGO DE CHILE
JUNIO 2011
Towards Practical Omniscient Debugging

June 6, 2011
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Resumen</strong></td>
<td>4</td>
</tr>
<tr>
<td><strong>Acknowledgments</strong></td>
<td>5</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>6</td>
</tr>
<tr>
<td>1.1 A simple omniscient debugging session</td>
<td>7</td>
</tr>
<tr>
<td>1.2 Problem statement</td>
<td>9</td>
</tr>
<tr>
<td>1.3 Thesis and contributions</td>
<td>10</td>
</tr>
<tr>
<td>1.4 Outline</td>
<td>11</td>
</tr>
<tr>
<td><strong>2 Survey of omniscient debugging and related systems</strong></td>
<td>13</td>
</tr>
<tr>
<td>2.1 Omniscient debugging</td>
<td>16</td>
</tr>
<tr>
<td>2.2 Query-based and automated debugging</td>
<td>19</td>
</tr>
<tr>
<td>2.3 Profiling and trace analysis</td>
<td>20</td>
</tr>
<tr>
<td>2.4 Replay-based debugging</td>
<td>21</td>
</tr>
<tr>
<td>2.5 Database techniques</td>
<td>23</td>
</tr>
<tr>
<td>2.6 Conclusion</td>
<td>24</td>
</tr>
<tr>
<td><strong>3 A first step: exhaustive capture and indexing</strong></td>
<td>26</td>
</tr>
<tr>
<td>3.1 Overview of TOD</td>
<td>29</td>
</tr>
<tr>
<td>3.1.1 Architecture</td>
<td>29</td>
</tr>
<tr>
<td>3.1.2 Representation and emission of events</td>
<td>30</td>
</tr>
<tr>
<td>3.1.3 Low-level queries: cursors and counts</td>
<td>31</td>
</tr>
<tr>
<td>3.1.4 High-level queries</td>
<td>33</td>
</tr>
<tr>
<td>3.1.5 User interface</td>
<td>33</td>
</tr>
<tr>
<td>3.2 High-speed database backend</td>
<td>36</td>
</tr>
<tr>
<td>3.2.1 Aggressive indexing of events</td>
<td>37</td>
</tr>
<tr>
<td>3.2.2 Cost of event retrieval</td>
<td>38</td>
</tr>
<tr>
<td>3.2.3 Cost of cursors</td>
<td>40</td>
</tr>
<tr>
<td>3.2.4 Cost of counts</td>
<td>41</td>
</tr>
<tr>
<td>3.2.5 Cost of indexing</td>
<td>41</td>
</tr>
</tbody>
</table>
## CONTENTS

3.3 Scaling up with a debugging cluster ........................................ 45  
3.3.1 Distributed architecture ............................................. 45  
3.3.2 Scalability .......................................................... 46  
3.4 Benchmarks ............................................................. 46  
3.4.1 Trace capture overhead ............................................. 47  
3.4.2 Database performance ............................................... 49  
3.4.3 Summary ............................................................ 53  
3.5 Working with partial traces ............................................. 53  
3.5.1 Motivating example: debugging the TOD Eclipse plugin .......... 54  
3.5.2 Dealing with missing information .................................. 54  
3.5.3 Specifying partial traces ........................................... 56  
3.6 Related work ........................................................... 57  
3.7 Conclusion .................................................................. 58  

4 Strong response time guarantees with summarized trace indexing and querying ................................................. 59  
4.1 Overview ................................................................. 60  
4.2 Summarized trace indexing and querying ............................... 62  
4.3 Trace capture and partial deterministic replay ....................... 64  
4.3.1 Capture ............................................................... 64  
4.3.2 Initial replay ......................................................... 66  
4.3.3 Snapshots ............................................................ 66  
4.4 Indexing of control flow ................................................ 68  
4.4.1 Range Min-Max Tree ................................................ 69  
4.4.2 Indexing and querying ............................................. 70  
4.5 Indexing of memory accesses ......................................... 72  
4.5.1 Index structure and querying ..................................... 72  
4.5.2 Pipelined index construction .................................... 74  
4.5.3 Local variable handling .......................................... 76  
4.6 Benchmarks ............................................................. 76  
4.6.1 Global results ....................................................... 77  
4.6.2 Cost of individual features ....................................... 79  
4.7 Related work ........................................................... 80  
4.8 Conclusion .................................................................. 81  

5 Omniscient debugging of aspect-oriented programs .................. 83  
5.1 AOP and debugging ..................................................... 85  
5.1.1 AspectJ ............................................................... 85  
5.1.2 Debug model for AOP ............................................ 85  
5.2 State of the practice ..................................................... 86  
5.2.1 The AJDT debugger ................................................ 87
Resumen

La depuración representa una parte importante del costo del desarrollo de software. Un estudio del NIST (National Institute of Standards and Technology) del 2002 muestra que los errores de software tienen un costo enorme sobre la economía de E.E.U.U [39] y menciona que “los desarrolladores ya gastan aproximadamente el 80 por ciento de los costos en identificar y corregir defectos”. En un estudio empírico de “hazañas” de depuración, Marc Eisenstadt determinó que la principal causa de la dificultad de encontrar los errores es la distancia temporal y espacial entre la causa raíz y el síntoma del error [19]; una vez que un error está precisamente localizado, arreglarlo es a menudo trivial. Desafortunadamente, la mayoría de los depuradores en uso hoy en día otorgan una ayuda muy limitada con respecto a la navegación temporal; los programadores deben frecuentemente simular mentalmente la ejecución de sus programas.

Los depuradores omniscientes mejoran de sobra manera esa situación, permitiendo a los programadores navegar fácilmente hacia adelante y hacia atrás en el historial de ejecución de un programa, así como inmediatamente encontrar la causa raíz de los errores gracias a vínculos causales que pueden ser atravesados hacia atrás en el tiempo [31]. Por lo tanto, un depurador omnisciente puede tener un gran impacto sobre la eficiencia del proceso de desarrollo.

La depuración omnisciente no es, desde luego, una idea nueva: el primer depurador omnisciente, EXDAMS [3], fue creado en el 1969. Sin embargo, a pesar de las numerosas propuestas que se han hecho desde entonces, los depuradores omniscientes todavía no forman parte del típico ambiente de desarrollo. Esto se debe a que existen desafíos prácticos importantes para implementar depuradores omniscientes: la cantidad de información que se debe manejar para poder reconstituir el historial completo de ejecución de un programa es enorme, y la captura misma de este historial, o huella de ejecución, hace más lenta la ejecución del programa. Existe también un desafío adicional, aunque no es propio de la depuración omnisciente: el uso de nuevos paradigmas de programación (ej. la programación por aspectos) dificulta más la depuración ya que hacen más borrosa la correspondencia estructural entre el código fuente y lo que finalmente se ejecuta.

En esta tesis presentamos soluciones que permiten acercarse a la realización de un depurador omnisciente utilizable en la práctica. Luego de una revisión del estado del arte, exponemos las tres direcciones que hemos explorado:

• Creación de dos procesos de indexación y consultas. El primer proceso es basado en la indexación exhaustiva de la huella de ejecución y es altamente escalable. El segundo proceso es basado en la obtención de resúmenes de bloques de la huella de ejecución, y aprovechando el determinismo de las mayoría de las computaciones.

• Construcción de un prototipo utilizable en la práctica, junto a un motor de base de datos escalable especializado para la depuración omnisciente.

• Extensión del prototipo para soportar la depuración de programas por aspectos.

Los resultados obtenidos fueron alentadores, por lo que consideramos que este trabajo es una contribución significativa al estado del arte en la materia.
Acknowledgments

First of all I want to thank the Computer Science Department of the University of Chile for making this work possible through financial support, top notch facilities and an exceptionally rich academic environment. I am particularly grateful to José Pino, who welcomed me back after a volatile start, and Claudio Gutierrez and Gonzalo Navarro, who backed my scholarship grants and patiently provided all kinds of support. I am also grateful to NIC Chile for providing me with a scholarship during the last two years of my PhD.

I am most grateful to my advisor and long-time friend Éric Tanter. He always sets the bar very high and gives the necessary tools to reach one’s goals. His mastery of the art of writing and willingness to transmit it, his patience and his tolerance made working under his guidance an enlightening experience.

Although our interaction was sparse, I want to thank Gonzalo Navarro for being a great inspiration for this work. I find his field of research amazingly interesting.

I am very grateful to Shigeru Chiba, Johan Fabry, Luis Mateu and Romain Robbes for accepting to be part of my jury, reviewing my dissertation, and providing constructive comments. I am particularly grateful to Shigeru for coming to my defense from so far away.

Those five years would definitely not have been the same without the company of my office mates Oscar Callaú, Paul Leger and Rodolfo Toledo, and of my colleagues Milton Inostroza and Victor Ramiro. So many good laughs we had! It was a pleasure spending this time with them.

I am also very grateful to Sergio Aguilera and Hernan Cuevas of the IT support team for their truly stellar service. Their help was crucial in many phases of my work.

Words cannot express my heartfelt gratitude to my wife Karla and to my family for their support during all these years. It has been a long and sometimes uncertain journey, which I would not have survived without them.
Chapter 1

Introduction

From then on, when anything went wrong with a computer, we said it had bugs in it.

---

RADM Grace Hopper, on the removal of a 2-inch-long moth from the Harvard Mark I experimental computer at Harvard in August 1945

Debugging represents a major cost in the software development process. A 2002 NIST study establishes that software errors have an enormous cost on the U.S. economy [39] and mentions that “software developers already spend approximately 80 percent of development costs on identifying and correcting defects”. In an empirical study of debugging stories, Marc Eisenstadt found that the major reason why bugs are difficult to track down is the large temporal or spatial chasm between the root cause and the actual symptom of a bug [19]; once a bug is precisely located, fixing it is often trivial.

There are two traditional approaches to debugging (Fig. 1.1): 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 a determined point, inspect the contents of memory at that point, 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. Developers using breakpoint-based debuggers are familiar with having to re-run the whole program many times with different sets of breakpoints to progressively home in on the bug.

Omniscient debuggers, also known as back-in-time or post-mortem debuggers, overcome
CHAPTER 1. INTRODUCTION

all these issues [26, 31, 34]. 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 $foo$?”.

1.1 A simple omniscient debugging session

To illustrate the features of omniscient debuggers, we present a very simple debugging session using the Trace-Oriented Debugger (TOD), our prototype omniscient debugger for Java (described in details in Chapter 3). TOD supports temporal navigation via stepping both forward and backward in time. In addition, it supports fast causal navigation via a why? link displayed next to the value of inspected variables, which permits to directly jump to the event that assigned the variable its current value. We describe a bug hunting session that makes use of this feature, as illustrated in Fig. 1.2a.

After launching the buggy program with the TOD launch button (1), we can easily locate the exception event in the execution trace. Once the exception event is selected in the main control flow view (2), the corresponding source code line is automatically highlighted (3). Here we notice that the thumbnail field of the current ThumbnailPanel object is indeed null (4), which is why the exception was thrown. Clicking the why? link (4) immediately brings us to not only the source code line where the value of the field is set (5), but also to the precise event that caused this particular assignment (6). Note that the assignment occurred
CHAPTER 1. INTRODUCTION

(a) Hunting for the root cause of a NullPointerException using the *why?* link.

(b) Navigation history.

Figure 1.2: TOD user interface.
in a different thread than the one that threw the exception (7). Inspecting the program state at the newly selected event shows that we tried to create a thumbnail of a .sh file (8), which failed.

In this simple example, TOD allowed us to jump in just a few steps directly from the symptom of a bug (the exception) to its cause (the mishandling of non-image files). The same bug hunting with a breakpoint-based debugger would have been tedious because there are potentially many places in the program where the thumbnail field is set apart from the constructor, and many correct instantiations of ThumbnailPanel to step through.

Such a toy example does not show the full potential of TOD however. Although it would be too lengthy to relate here, we have been able to use TOD to quickly solve difficult problems, such as bugs in the TOD database itself, and to understand issues that arose in our use of highly complex software such as the abc1 AspectJ compiler.

1.2 Problem statement

While the advantages of omniscient debugging over traditional approaches are 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, which we describe below.

Trace capture efficiency Except when using specialized hardware probing ports [25], the emission of events causes a significant overhead to the debugged application. This is problematic for two reasons: (a) it makes the tool cumbersome to use as it greatly increases the length of edit-compile-debug cycles (although the amount of necessary cycles is usually reduced), and (b) it can disrupt time-sensitive behaviors, thus hiding the very bugs the user tries to fix, or adding new bugs that would not appear in normal conditions.

Processing and querying scalability As emphasized in [24, 40, 59], for CPU-intensive applications, the execution trace can rapidly become huge, and thus requires specialized processing. In our experience, a program running under an omniscient debugger (and thus slowed down by trace capture) generates half a million events per second on current hardware. Execution traces of several gigabytes or dozens of gigabytes are the norm rather than the exception. An interactive debugging experience requires extremely fast queries on the recorded execution trace; executing queries quickly requires the execution trace to be properly indexed.

Cognitive scalability As execution traces are potentially huge, the user can get lost in an information deluge. Omniscient debuggers must thus assist the user in overcoming the

1http://abc.comlab.ox.ac.uk/introduction
cognitive burden of dealing with a large amount of information in order to rapidly locate the points of interest [27].

Making omniscient debugging practical requires advances in the three above points. Moreover, another issue should be taken into account, although it is applicable to debugging in general, not only to omniscient debugging: execution platforms such as the Java Virtual Machine and the .NET CLR are increasingly used to host applications written in languages different than the ones they were initially designed for. In particular, several Aspect-Oriented Programming (AOP) [21] languages rely on transforming the original program by inserting code that performs the aspect-specific logic at determined points. Such transformations blur the structural correspondence [7] between source code constructs and actual computations, and require special debugging support.

1.3 Thesis and contributions

Thesis statement

Although costlier than traditional debugging, omniscient debugging can be made practical by optimizing the execution trace capture and processing mechanisms, and reducing the capture scope. We back this claim with benchmarks of two different omniscient debugging engines and a prototype omniscient debugger for Java.

Because of its data storage and processing requirements, omniscient debugging is necessarily costlier than traditional breakpoint-based debugging. We claim, however, that it can still be made practical by applying two complementary measures:

- Optimizing trace capture and processing. We show that a high processing and querying throughput can be achieved by indexing only a summary of trace data, and relying on deterministic replay for getting the final responses.

- Reducing the amount of information to capture and process by working with partial traces. Partial traces are obtained by selectively enabling or disabling trace capture for certain parts of the computation, either lexically (i.e. by disabling trace generation in some parts of the program), or temporally (i.e. by enabling or disabling capture during the execution of the program). We show in particular that judicious lexical scoping yields orders of magnitude of trace size reduction (Section 3.5).

Our work provides three main contributions to the field:

- Two different processes for implementing an omniscient debugger. A first-generation process adopts a brute-force approach in which each event of an execution trace is individually indexed. This process is embarrassingly parallel in nature and we show that it is highly scalable on a computing cluster. A second-generation process adopts
CHAPTER 1. INTRODUCTION

A subtler approach in which only a summary of the trace is indexed and deterministic replay is used to reconstitute portions of the original trace when needed. The latter process features a high storage throughput and guarantees a fast query response time, thus ensuring a highly interactive debugging experience.

- A working prototype omniscient debugger for Java, implemented as a plugin of the Eclipse development environment. This tool could be used to conduct studies on the actual usefulness of omniscient debugging, a subject we do not tackle in this work.

- An extension of omniscient debugging for supporting aspect-oriented programming. We present a simple user interface that allows to select the level of detail of aspect-specific computations that have no correspondence in the source code, from full intimacy (all the details of the computation are shown) to full obliviousness (only the events related to the base code are shown).

1.4 Outline

After a survey of related work in Chapter 2, this document contains three main parts:

- Chapter 3 presents TOD (Trace-Oriented Debugger), our prototype omniscient debugger for Java. TOD represents a brute-force approach to execution trace indexing, in which each attribute of each event is individually indexed. Although this exhaustive indexing enables relatively flexible queries, it is costly both in the overhead caused by trace capture and indexing, and in query execution. We overcome this cost by using a distributed trace database, whose throughput is linear in the number of nodes, up to a point where certain bottlenecks are reached.

- Chapter 4 presents a more refined process named STIQ, for Summarized Trace Indexing and Querying. The execution trace is divided into bounded-size execution blocks about which summary information is indexed. Blocks themselves are discarded, and retrieved as needed through partial deterministic replay. For querying, the index provides coarse answers at the level of execution blocks, which are then replayed to find the exact answer. Query processing requires \( O(\log n) \) disk accesses and \( O(1) \) CPU time for traces of arbitrary size \( n \), and never exceeds a few hundred milliseconds in practice, guaranteeing a highly interactive debugging experience. Benchmarks on a prototype for Java show that the system is fast in practice, and outperforms existing omniscient debuggers.

- Chapter 5 shows how omniscient debugging can be extended to effectively support Aspect-Oriented Programming (AOP). Debugging AOP programs is difficult because to implement the semantics of aspects, a number of implicit activities are performed
at runtime, whose relation to source code is not direct to grasp. We present AspectJ-specific extensions to TOD that let the user select the desired trade-off between obliviousness and intimacy, i.e. the level of detail on the internal operation of the AOP framework.

Finally, Chapter 6 concludes the dissertation.

The work described in this document was partially presented in the following conference and journal papers:

• OOPSLA 2007: “Scalable Omniscient Debugging” [45]
• SAC 2008: “Extending Omniscient Debugging to Support Aspect-Oriented Programming” [42].
• IEEE Software: “Back to the future with omniscient debugging” [43].
• ECOOP 2011: “Indexing large execution traces for interactive back-in-time debugging” [44].
Chapter 2

Survey of omniscient debugging and related systems

Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Omniscient debugging</td>
<td>16</td>
</tr>
<tr>
<td>2.2</td>
<td>Query-based and automated debugging</td>
<td>19</td>
</tr>
<tr>
<td>2.3</td>
<td>Profiling and trace analysis</td>
<td>20</td>
</tr>
<tr>
<td>2.4</td>
<td>Replay-based debugging</td>
<td>21</td>
</tr>
<tr>
<td>2.5</td>
<td>Database techniques</td>
<td>23</td>
</tr>
<tr>
<td>2.6</td>
<td>Conclusion</td>
<td>24</td>
</tr>
</tbody>
</table>
Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

Brian Kernighan

Most modern IDEs provide a breakpoint-based debugger out of the box. They all roughly have the same set of features:

**Breakpoints and watchpoints.** The central feature of breakpoint-based debuggers is the definition of breakpoints (resp. watchpoints), which permit to suspend execution when a given source code line is reached (resp. when a given memory location is accessed). Once the execution is suspended, the user can inspect, and in some cases alter, the current state of the program. Breakpoints are said to be conditional when they require the verification of an additional condition to activate (for instance, a check on the value of a variable).

**Stepping.** When the execution is suspended at a breakpoint, the user can perform stepping operations, i.e. executing the next operation and suspending again as soon as it is completed. Common variants of stepping are *step into* (if the operation to execute is a method call, the debugger suspends execution at the beginning of the called method), *step over* (if the operation to execute is a method call, the debugger executes the called method and suspends execution when the method returns), and *step out* (the debugger continues execution until the current method terminates).

**Edit and continue.** On platforms that support it, debuggers can let the user modify the program while it is running so that fixes can be tested immediately, without needing to re-execute the program entirely.

All of the above features are actually decades old: a 1966 survey [22] already described how they were being adapted from assembly to higher-level languages such as LISP and Fortran. The usefulness of execution traces for debugging has also been recognized a long time ago: the first omniscient debugger, EXDAMS [3], dates back to 1969, and richer query-based systems started to appear during the early 80s [50, 30].

1A notable exception being the OmniCore CodeGuide IDE for Java, which provides a (limited) omniscient debugger. The product seems to have been discontinued, though.
Trace-based debugging, however, is mostly absent from the mainstream toolkits. We conjecture that this is due to two factors:

- While breakpoint-based debugging can be implemented efficiently, trace-based debugging incurs a substantial cost on trace capture and processing. Although the probe effect (the alteration of program behavior caused by the very fact of running it under a debugger) can be observed under both debugging paradigms, the higher overhead of trace-based debugging makes it more prominent under this paradigm.

- The learning curve of breakpoint-based debugging is much lower than that of advanced query-based systems that require that the user learns a new query language.

We argue that omniscient debuggers are as easy to learn as breakpoint based debuggers, as they present similar navigation metaphors; the aim of our work is to make them more usable by tackling the efficiency and scalability issues.

In this section we present four areas of research that are closely related to our work:

- Omniscient debuggers are interactive tools that rely on capturing and querying execution traces. The set of queries that can be performed is limited to those needed to provide basic navigation features such as forward and backward stepping, state inspection, and automatic traversal of causality links.

- Query-based debuggers identify 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). Although they are more flexible than omniscient debuggers, they are also more complex, as the user must learn a query language.

- Trace analysis is usually not directly concerned with debugging, but rather with obtaining summarized views of program execution, such as time or memory profiles, or correlations between several executions of the same program. The techniques employed are however related to those needed for omniscient debugging.

- Replay-based debuggers let users go back in time by replaying the program up to certain points instead of recording execution traces. They have a much lower overhead than omniscient debuggers, at the cost of a more limited feature set.

Finally, as the trace storage and processing dimension of omniscient debugging is essentially a data management issue, we also give an overview of a range of database techniques that can be used as a point of reference.
2.1 Omniscient debugging

Omniscient debuggers (also known as back-in-time or post-mortem debuggers) capture the execution trace of the debugged program and let the user navigate in the trace by using the familiar metaphors of breakpoint-based debuggers (step-by-step execution and state inspection), but with the additional ability to step backward in time. Most omniscient debuggers also support the automatic traversal of causality links, i.e. the ability the jump from the point a value is observed in a given variable to the point in the past when the value is assigned to that variable, which is an extremely valuable tool to resolve the chain of causes and effects that lead to a bug.

The precursor. EXDAMS [3] is the first omniscient debugger we know of. It works by instrumenting algebraic programs (PL/I, Algol, Fortran) at the source level and recording execution traces on tape. EXDAMS contains two main components: a language-dependent trace capture engine, and a language-independent debugging engine that relies on the captured trace. It supports forward and backward execution, state reconstitution and causality links, although these operations are not backed by indexes to speed them up.

Advanced features. In 1984, ZStep [32] provided a reversible stepper for Lisp: it permitted to step forward and backward, and to see the result of evaluated expressions in parallel to the corresponding source code. Its sequel ZStep95 [34] added the possibility to relate graphical output to the event that caused it, as well as tape recorder-like controls for easier navigation. Although they provide excellent solutions to the cognitive issues of debugging, these systems do not handle side effects and causal links (except for graphical output), and do not address efficiency and scalability issues.

Towards production-ready tools. ODB [31] is one of the first omniscient debuggers for Java. It obtains execution traces by instrumenting classes as they are loaded by the JVM. ODB stores the captured trace data inside the target JVM, which raises some issues: the amount of trace data is limited by the available heap space, and references to objects that are no longer in use are kept, preventing proper garbage collection. To reduce the overhead of trace capture as well as the amount of information to store, ODB supports partial traces through lexical scoping, i.e. the ability to exclude certain trusted classes from the instrumentation process so that they do not contribute to the execution trace. A unique feature of ODB is the ability to resume execution from any point in time with a modified state.

Unstuck [26] is an omniscient debugger for Smalltalk, which is very similar in architecture and operation to ODB and thus suffers from the same scalability limitations.

Alternative scoping strategies for partial traces. Lienhard et al. [36] proposed another omniscient debugger for Smalltalk that handles the scalability issue by using partial traces,
but in a very different way than ODB (or TOD). They postulate that information about objects that are not reachable anymore at a certain point in time (i.e. objects eligible for garbage collection) can be discarded. Although discarding that information indeed boosts efficiency, the root cause of a bug can have occurred in the context of objects that have been discarded long before the symptoms of the bug manifest themselves, thus rendering this approach ineffective in some cases.

**Flow-centric user interface.** While most debuggers, omniscient or otherwise, are control-flow-centric, further work by Lienhard *et al.* [35] introduce object-flow-centric debugging: their system permits to visualize the lifecycle of individual objects as they are created and passed around through method calls.

**Interactive visualizations.** JIVE [23] is an interactive visualization environment for Java programs. It provides UML-like sequence diagrams as well as object diagrams extended with information about the current method call. The level of detail of the diagrams can be reduced so as to fit more information on the screen, but it is not clear that this mechanism scales past a few hundred elements. JIVE supports forward and backward stepping, but not quick causal navigation. The execution trace is captured using JPDA, the debugging interface of the JVM, and processed in memory, thus limiting the efficiency and scalability of the system.

**Debugging Firefox.** Chronicle\(^2\) is an open-source omniscient debugger for native x86 Linux programs. It instruments binaries so that they send trace data to an out-of-process, disk-based database. A key characteristic of Chronicle is the aggressive compression and indexing of the events, that permits to record very large traces and to process queries efficiently. In particular, it has been used to debug the well-known Firefox web browser. It has, however, a significant runtime overhead on the debugged program (around 300x).

**Commercial offerings.** TimeMachine by Green Hills Software\(^3\) is an omniscient debugger for embedded systems (PowerPC, ARM and similar architectures). On some platforms, a specialized hardware probe permits to capture trace data without incurring any runtime overhead; otherwise traditional software instrumentation is used. In addition to the traditional features of omniscient debuggers, TimeMachine can also be used as a profiling tool. Unfortunately, there is no precise and publicly available information about the architecture, runtime overhead and scalability of the system.

OmniCore CodeGuide is a (now discontinued) commercial development environment for Java that features an omniscient 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 causal link traversal is not available.

---

\(^2\)http://code.google.com/p/chronicle-recorder/

\(^3\)http://www.ghs.com/products/timemachine.html
Summary.

Figure 2.1 gives a summary of the characteristics of several of the systems presented in this section and compares them with those of our proposals, TOD (Chapter 3) and STIQ (Chapter 4). The History size column gives the order of magnitude of the number of events that can reasonably be collected and processed by the system. Runtime overhead gives an idea of the slowdown caused by the debugger. For systems on which we performed our own benchmarks there are two figures (X/Y), where X is the runtime overhead in the worst case (i.e. for a fully-instrumented, CPU-intensive program), and Y corresponds to a more typical situation (a run of the jTidy HTML beautifier program). For the other systems, the unique figure is the one provided by the authors of the system. In the Partial traces column, lexical scoping means that it is possible to select the classes or packages to instrument, and dynamic scoping means that it is possible to activate or deactivate trace capture at runtime. Causal navigation indicates if the debugger permits to directly jump to the past event that set a variable to its current value. High-level overviews indicate if the system can provide summary views of the debugged program.

---

4The Whyline is a hybrid between omniscient debuggers and the query-based debuggers presented in the next section.

5In the case of TOD, the worst-case figure is higher than the one published in the OOPSLA’07 paper, as further tests showed that the test program used was not the absolute worst case. In the case of STIQ, we did not repeat the jTidy test, but the the runtime overhead varies between 12x and 30x (see Sect. 4.6).
CHAPTER 2. SURVEY

Most existing systems have a limited scalability, with the exception of Chronicle and TimeMachine. Thanks to the use of a hardware probe, trace capture in TimeMachine does not incur any runtime overhead, but unfortunately little more is known about the tool. Chronicle is highly scalable but suffers from a rather high runtime overhead. The aim of our work is to provide a system that has both a low runtime overhead and a high scalability.

2.2 Query-based and automated debugging

Query-based debugging consists in identifying events that match a query expressed in a high-level language, where queries can be formulated \textit{a priori} (before running the program) or \textit{a posteriori} (after the program has been executed). In automated debugging, the queries are inherent to the tool instead of user-defined.

Relational queries. TQuel \cite{50} is a declarative relational querying language with native support for temporal queries through the availability of the \textit{when} clause in addition to the classical \textit{where} clause. In the cited paper, TQuel is used to perform queries over execution traces of possibly distributed programs. Basic events such as procedure calls and returns are captured by \textit{sensors} that are inserted into the original program. To improve the efficiency of the system, the TQuel queries must be specified a priori; the system compiles them into an \textit{update network} that is used both to determine which sensors should be enabled, and to incrementally update query results as sensors are reached.

Logic queries. LeDoux and Parker \cite{30} present YODA, a system in which the user formulates \textit{a-posteriori} Prolog queries on the execution of concurrent Ada programs. Temporal queries can be expressed without needing a language extension, as timestamps are explicitly available in the event model.

Opium \cite{16} uses Prolog queries to debug Prolog programs and seamlessly supports breakpoint-based debugging and trace-based debugging. Coca \cite{15} uses a-priori Prolog queries to debug C programs.

Visual querying. Hy+/GraphLog is a generic querying and visualization system that can be applied to the debugging of distributed programs \cite{12}. Information about the exchange of messages between hosts is first stored into a database in a raw form. Declarative queries are then expressed in the GraphLog visual language to construct custom visualizations of the trace. The user can specify patterns of normal and abnormal behaviors so that these behaviors can be quickly pinpointed in the visualization. In addition to static queries that encompass the whole execution trace, it is also possible to express dynamic queries that are animated as new events are registered by the system.
Automatically generated queries. The Java Whyline [29] lets the user select questions about why some behavior did or did not occur. These questions are automatically generated based on a combination of static and dynamic analysis, and can deal not only with internal state of the program \((e.g. \text{“why does variable } x \text{ have value } y”)\), but also with its textual and graphical output, down to individual pixels. Questions can also be negative: it is possible to ask why a given behavior did not occur, which is a frequent question during debugging sessions. Although the captured trace is stored on disk, it must be completely loaded into memory for analysis, which limits the scalability of the system.

Advanced stateful event matching. PQL [38] provides a very high-level and powerful a-priori query model. A stateful matcher permits to evaluate complex queries such as detecting all method invocations on objects that were deserialized from a socket, or tracking strings that were potentially provided by the user (to avoid SQL injection attacks for instance).

The user as an oracle. In an upside down approach the Mercury Declarative Debugger [37] asks the user questions about the correctness of computations performed by the program so as to quickly locate incorrect ones through a divide-and-conquer approach.

Delta debugging. When the user is able to isolate an input that makes the program fail and another one that makes it pass, it is possible to apply delta debugging [60] to automatically determine the chain of cause and effects that produce the failure through a divide-and-conquer approach. Delta debugging obtains the execution state of the failing and passing execution, and gradually injects the state of the failing execution into the passing execution until it causes it to fail. The process is repeated until a minimal set of failure-inducing alterations is found.

Summary. The tools presented in this section represent another alternative to traditional debugging approaches such as breakpoint-based debuggers and ad-hoc logging. We argue that most of these tools, although very powerful, have a steeper learning curve than omniscient debuggers, as they require the user to learn a new query language. A notable exception is the Whyline, which assists the user in a very intuitive way. It suffers, however, from a limited scalability.

2.3 Profiling and trace analysis

Capture of execution traces for automatic offline analysis is a well studied topic. Various optimizations permit to execute relatively complex queries \((e.g. \text{calculating dynamic slices, detecting hot paths for later optimization, or matching instruction flows in different versions of the same program})\) in seconds or minutes instead of the hours or days it would take using a naive approach. In contrast, the challenge of omniscient debugging is to execute simple
queries in a few milliseconds instead of the seconds or minutes it would take using a naive approach.

**Compression of whole execution traces.** Zhang et al. [61] present several lossless compression techniques used to record whole execution traces of native programs. These compression algorithms support fast bidirectional navigation in the compressed traces.

One of the priorities of profiling is to reduce the performance impact of the tool on the target application. One technique that can be used in distributed applications is clustering [40, 59], which consists in grouping the processors that perform similar tasks at any given time. This technique reduces both the amount of profile data to process and the capture overhead, although at the price of a loss in precision. Debugging distributed applications can also benefit from high-level trace recording, as only message sends between nodes need be recorded [40, 12]: useful views of the computations can be provided with much less information than that used in omniscient debugging.

**Indirect capture of data dependencies.** Tallam et al. [54] show that it is possible to extend control flow traces to indirectly capture runtime data dependencies, yielding a 25-fold reduction in combined trace size while incurring only a 20% additional overhead. The key technique is to add synthetic control flow paths that are taken or not according to the runtime value of accessed memory addresses.

**Detailed capture of control flow.** Xin et al. [57] present a technique to efficiently capture control flow at a level of granularity finer than procedure calls, and provide a numbering scheme of executed statements that can be used, for instance, to correlate several executions of the same program.

**Summary.** The approaches presented in this section tackle the issue of program analysis at a global scale rather than the more local scale omniscient debuggers deal with. However, the ideas used in trace analysis tools are an important inspiration for the optimization of trace capture and storage of omniscient debugging.

### 2.4 Replay-based debugging

Deterministic replay consists in re-executing a program so that it behaves exactly in the same way as during the original execution. As program execution can be influenced by external factors such as user input, data read from files or network connections, thread scheduling, etc., it is necessary to record the outcome of these non-deterministic operations during the original execution and reinject them into the program during replay.

The back-in-time navigation features of omniscient debugging can be simulated by performing a deterministic replay of the debugged program until some determined point before
the current execution point. Most deterministic replay systems implement a periodic snapshot mechanism that make it unnecessary to replay the whole program to get to any given point.

The main advantage of deterministic replay compared to the trace capture mechanism used in most omniscient debuggers is the lower runtime overhead, as only non-deterministic events need to be recorded. Navigation operations are slower, however, as significant portions of the program might have to be replayed to get to the desired point.

**Single-threaded replay.** Flashback [51] and Jockey [48] are deterministic replay systems for native Linux programs. Flashback relies on a modified kernel while Jockey relies on program instrumentation. They both take periodic snapshots of the state of the debugged process and record the interactions between the program and its environment. Snapshots are based on a fork of the process and take advantage of the copy-on-write mechanism of the kernel to avoid having to explicitly copy the entire address space. However, the fact that snapshots have to stay in memory make it necessary to discard older ones. Both systems have a runtime overhead lower than that of most omniscient debuggers (2x-4x for Flashback, and only up to 30% for Jockey), but they do not properly handle multithreaded programs.

**Multithreaded replay.** Concurrency affects the order in which data is written to memory in a way that is not controlled by the program, thus making memory reads non-deterministic operations.

Nirvana [4] is a deterministic replay system for native programs that properly supports multithreading. It records the results of memory reads to account for scheduling-induced non determinism. Its runtime overhead is between 5x and 17x.

DejaVu [9] is a deterministic replay system for Java based on modifications of the JVM. It supports multithreaded programs by explicitly recording the order in which threads are scheduled and enforcing the same ordering during replay. It has a rather low runtime overhead (usually less than 100%), but the JVM used does not have a JIT compiler and thus only runs in interpreted mode, which has very different performance characteristics compared to production JVMs.

UndoDB is an omniscient debugger for native x86 Linux programs by Undo Ltd[6]. It is based on a checkpoint/replay mechanism: it periodically obtains a checkpoint, or snapshot, of the process memory, and uses a replay technique to reconstruct the state of the program between checkpoints. This mechanism yields a relatively low runtime overhead, but does not allow causal links traversal.

**VM-based replay.** Retrace [58] is a deterministic replay system for uniprocessor VMWare virtual machines. It has an extremely low runtime overhead (around 5%) and produces very compact traces. Such a low runtime overhead is possible because the recorded system is the

---

entire (virtual) machine, therefore the amount of interaction with the environment is limited to mostly IO operations; in particular, thread scheduling and the associated non-determinism on memory accesses need not be captured, as the scheduling itself is a deterministic part of the recorded system. Retrace does not provide backward stepping or causality links, but simply allows the user to deterministically re-execute her program as many times as needed under the control of a traditional debugger.

Summary. While deterministic replay systems indeed have a much lower impact on the debugged program, they do not provide the seamless navigation features of omniscient debuggers. One of our contributions is to show that deterministic replay can be combined with an indexing scheme for implementing an omniscient debugger (Chapter 4).

2.5 Database techniques

Storing and querying execution traces is basically a data management problem. The challenges of managing data in an efficient and scalable fashion has been studied widely even before Codd invented the relational model in the early 1970s [11]. Although it is possible to use general purpose database management systems to store execution traces, we determined that such systems do not have a sufficient throughput to process the volume of data produced by omniscient debuggers [41]. It has been argued that systems that have very specific data management needs should use specialized databases [53]. This section presents specialized database management approaches that should be considered when designing a scalable omniscient debugger.

Stream databases. Data sources such as physical sensor networks, financial markets or network monitoring equipments produce a massive and continuous flow of data. Processing such data streams requires systems with a very high throughput, well beyond the capabilities of traditional relational databases systems. Stream databases have been proposed as a solution to this issue.

Gigascope [13] is a stream database for network monitoring applications. It features a pure stream query language similar to SQL, where queries on input streams produce output streams, which can in turn be used as the input for other queries. Queries can use both non-blocking operators, such as filtering input tuples or selecting particular fields of the input tuples, and blocking operators, such as aggregation and joins, that rely on the availability of sliding windows of the most recently received tuples of each input stream. The amount of tuples that must be stored is automatically determined by the system according to the queries that are executed, but there is no information about the scalability of the system with respect to window size. Benchmarks on realistic use cases demonstrate a very high throughput (around 70MB/s) for non-trivial queries.
Aurora [1] is another stream database with a richer feature set: it supports triggers, relaxed result accuracy for improved efficiency, and has a scheduling mechanism to handle real-time requirements. Tuple storage for sliding windows, whose size must be explicitly specified, is scalable: the default is to store them in main memory, but they can be stored on disk if necessary.

Despite their high processing throughput, stream databases are not directly suitable for implementing an omniscient debugger: they do not store data beyond limited sliding windows, and thus cannot be used to perform queries about long-past history.

**Sequence databases.** Traditional relational databases do not provide a native abstraction for ordered collections, or sequences, of tuples. The ORDER BY clause of SQL only affects the order in which results are output and does not usually have an effect on the query plan. SEQ [49] is a database management system with native support for sequential data. Similarly to stream databases, it supports queries on sliding windows, but such queries can be performed on the complete sequence instead of only on the most recent tuples. The query language of SEQ has special keywords to express sequence-aware queries, and such queries can be much better optimized than what a traditional relational database could achieve. SEQ does not address, however, the issue of high-throughput storage of the sequences.

**Summary.** Stream databases can be used to process online queries on high-throughput data streams, whereas sequence databases are designed to optimize the processing of complex sequence-aware queries on already-stored data. A database management system for omniscient debugging should combine the features of both kinds of databases: high processing throughput for storing the execution trace, and the ability to efficiently process queries on the stored trace.

### 2.6 Conclusion

Efficiency and scalability are the key issues to resolve to see a widespread adoption of omniscient debugging. Although the idea of omniscient debugging has been around for a long time, these issues have not been studied thoroughly. Most of the works cited in Section 2.1 describe proof-of-concept prototypes for different platforms, with different feature sets, but address the efficiency issue through the capture of partial traces only, if at all. Chronicle and TimeMachine are exceptions: Chronicle uses an optimized on-disk indexing mechanism (but has a rather high runtime overhead and does not guarantee fast query response times), and TimeMachine can use a hardware-based probe mechanism that eliminates the runtime overhead (but little is known about the way the trace is stored and queried).

Another issue not dealt with in the omniscient debugging literature is the support for debugging programs written in languages or paradigms more abstract than those originally designed for the execution platform. For example, in the case of Java, Aspect-Oriented
Programming (AOP) [21] is a paradigm that is gaining widespread acceptance in the industry through the AspectJ [28] language. Aspect-oriented programs are more difficult to debug because the structural correspondence [7] between the source code and the actual execution is less evident. Although the debugging of AOP programs is not an issue specific to omniscient debugging but rather to debugging in general, it is interesting to study how omniscient debugging can facilitate the debugging of such programs.

Query-based debugging allows richer queries, but we have not seen it adopted in mainstream environments. Although efficiency is also an issue here, we assume that the relatively low popularity is due mainly to the difficulty of learning the query language. Omniscient debugging, in contrast, only use the familiar debugging metaphors of stepping, and the easily understood mechanism of causality links.

Profiling and trace analysis systems do place an emphasis on efficiency, but at a different scale than that of omniscient debugging: they implement optimizations that permit to run complex analyses in seconds or minutes instead of the hours or days it would take without optimizations. Omniscient debugging, on the other hand, require that (simple) queries run in a few hundred milliseconds so that they can be used interactively.

Replay-based systems have been use extensively for debugging, some of them being commercial products. Their big advantage over omniscient debuggers is that their runtime overhead is much lower. On the other hand, they do not efficiently support all the queries needed for omniscient debugging.

Our goal is to make omniscient debugging practical. In this work we explore two approaches that tackle the efficiency and scalability issues, and we experiment with the debugging of aspect-oriented programs. On the efficiency side, we first present a prototype that offers a very high scalability, at the cost of rather high hardware requirements (Chapter 3). We then present a more subtle approach, drawing from both the omniscient debugging and replay-based debugging literature, and using state-of-the-art compact data structures (Chapter 4). On the applicability side, we present an extension of our prototype to support AOP that lets the user choose the desired level of detail regarding the internal operation of the AOP framework (Chapter 5).
Chapter 3

A first step: exhaustive capture and indexing

Contents

3.1 Overview of TOD .................................................. 29
   3.1.1 Architecture ............................................... 29
   3.1.2 Representation and emission of events .................. 30
   3.1.3 Low-level queries: cursors and counts ................. 31
   3.1.4 High-level queries ........................................ 33
   3.1.5 User interface ............................................. 33
3.2 High-speed database backend ................................. 36
   3.2.1 Aggressive indexing of events ......................... 37
   3.2.2 Cost of event retrieval .................................. 38
   3.2.3 Cost of cursors ........................................... 40
   3.2.4 Cost of counts .......................................... 41
   3.2.5 Cost of indexing ........................................ 41
3.3 Scaling up with a debugging cluster ....................... 45
   3.3.1 Distributed architecture ............................... 45
   3.3.2 Scalability ............................................. 46
3.4 Benchmarks .................................................... 46
   3.4.1 Trace capture overhead .................................. 47
   3.4.2 Database performance .................................... 49

1The contents of this chapter are based on our OOPSLA 2007 paper “Scalable Omniscient Debugging” [45], with some additions from our IEEE Software paper “Back to the future with omniscient debugging” [43].
3.4.3 Summary ......................................................... 53

3.5 Working with partial traces ..................................... 53
  3.5.1 Motivating example: debugging the TOD Eclipse plugin ... 54
  3.5.2 Dealing with missing information .......................... 54
  3.5.3 Specifying partial traces ................................... 56

3.6 Related work .................................................... 57

3.7 Conclusion ....................................................... 58
If debugging is the process of removing bugs, then programming must be the process of putting them in. 

F. W. Dijkstra

The aim of this chapter 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\(^2\), a portable Trace-Oriented Debugger for Java integrated into the Eclipse IDE [14]. 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 3.1 overviews the architecture of TOD, the event model, and the GUI components. Section 3.2 describes the efficient indexing scheme of TOD for storing and querying events, and Section 3.3 shows how it is parallelized. Benchmarks are provided in Section 3.4. We explain the advantage of partial traces and how they are dealt with in Section 3.5. Related work is discussed in Section 3.6. Section 3.7 concludes, and identifies opportunities for further enhancements in the field.

\(^2\)http://pleiad.cl/tod for download and a small illustrative video.
3.1 Overview of TOD

TOD is a Trace-Oriented Debugger for Java that addresses the scalability issues identified in Chapter 1. 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.

3.1.1 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. 3.1): 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 [26, 31, 34] is that 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 navigable 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. 3.2 and 3.3). The debugger core contains another database, the struc-
CHAPTER 3. EXHAUSTIVE CAPTURE AND INDEXING

<table>
<thead>
<tr>
<th>kind</th>
<th>kind</th>
<th>ts</th>
<th>tid</th>
<th>depth</th>
<th>pev</th>
<th>loc</th>
<th>fid</th>
<th>bid</th>
<th>vid</th>
<th>idx</th>
<th>val</th>
<th>ret</th>
<th>tgt</th>
<th>exc</th>
<th>args</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field write (FW)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Local var. write (VW)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Array write (AW)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Exception (Ex)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Behavior call (BC)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Behavior enter (Bu)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Behavior exit (Bx)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: Events and their attributes.

ture 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.

3.1.2 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 3.1 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. 3.1). There are three ways in which events can be emitted: specialized hardware trace ports [25], virtual machine or interpreter instrumentation [33], and application code instrumentation [26, 31]. 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 that 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 stores structural information in the structure database, and the modified class is sent back to the target JVM where it is eventually loaded (Fig. 3.1). 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 [8]: 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. 3.2), we shift original timestamp values 8 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 3.5.

**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.

### 3.1.3 Low-level queries: cursors and counts

All the navigation operations of an omniscient debugger (stepping, state reconstitution and causality links) 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

---

\(^3\)JVMTI: Java Virtual Machine Tool Interface, part of the Java 5 platform.
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 3.2: Navigation among events matching a cursor predicate.](image)

<table>
<thead>
<tr>
<th><code>operation</code></th>
<th><code>semantics</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>next()</code></td>
<td>Returns the next/previous matching event and moves the cursor forward/backward.</td>
</tr>
<tr>
<td><code>prev()</code></td>
<td></td>
</tr>
<tr>
<td><code>posNext(t)</code></td>
<td>Moves so that the next call to <code>next()</code>/<code>prev()</code> returns the first/last event whose timestamp is greater/less than or equal to <code>t</code>.</td>
</tr>
<tr>
<td><code>posPrev(t)</code></td>
<td></td>
</tr>
<tr>
<td><code>posNext(ev)</code></td>
<td>Moves so that the next call to <code>next()</code>/<code>prev()</code> returns the given event.</td>
</tr>
<tr>
<td><code>posPrev(ev)</code></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2: Cursor operations.

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, the condition `(kind = FW ∨ kind = BC) ∧ target = obj145` yields all the field write and behavior call events whose target is the object `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`.

**Cursors.** We define `cursor(Q)` as an iterator over events that match condition `Q` (Fig. 3.2). 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 3.2.

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

We now explain how cursors and counts are algorithmically combined to implement the high-level features of TOD. Section 3.2 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:

\[
\begin{align*}
& c \leftarrow \text{cursor}(\text{thread} = ev.\text{thread}) \\
& \text{c.posPrev}(ev); ev \leftarrow \text{c.next}()
\end{align*}
\]

Forward step over changes the cursor condition to: \( \text{thread} = ev.\text{thread} \land \text{depth} = ev.\text{depth} \). 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:

\[
\begin{align*}
& c \leftarrow \text{cursor}(\text{kind} = \text{FW} \land \text{fid} = f \land \text{target} = o) \\
& \text{c.posPrev}(t); v \leftarrow \text{c.prev}().\text{val}
\end{align*}
\]

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:

\[
\begin{align*}
& c \leftarrow \text{cursor}(\text{thread} = e.\text{thread} \land \text{depth} = e.\text{depth} + 1) \\
& \text{c.posPrev}(e.\text{ts}); cflow = \langle \rangle \\
& \text{repeat} \\
& \quad \text{ev} = \text{c.next}(); cflow \leftarrow cflow \cup \langle \text{ev} \rangle \\
& \text{until} \quad \text{ev.kind} = \text{Bx}
\end{align*}
\]

**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 then continue recursively, up to the root cause.

3.1.5 User interface

The frontend of TOD can be used standalone or as a plugin for the Eclipse Java IDE (Fig. 3.3). The user navigates between different *views* using widely-understood web browser metaphors.
CHAPTER 3. EXHAUSTIVE CAPTURE AND INDEXING

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 3.3: Stepping with TOD in Eclipse.](image)

The available views are: object inspector, control flow, and murals. The object inspector view shows reconstructions 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* [27] 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. 3.4).
- 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. 3.1.3), where the length of the time slices corresponds to the space occupied by a single pixel bar in the mural. The user can
CHAPTER 3. EXHAUSTIVE CAPTURE AND INDEXING

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

Figure 3.4: Thread murals.

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.

Not getting lost: bookmarking to the rescue Given the huge amount of events that can be recorded by TOD, it is crucial to help the user not to get lost while navigating in an execution trace. To this end, TOD lets the user bookmark events and objects, and permits to quickly access already visited locations.

Bookmarked events are displayed in a timeline above the main TOD view. The event currently selected in the main view is also marked in the timeline, so that the user can immediately find her way around in the execution trace relative to known landmarks. This is particularly useful when using the why? link described above, which can lead to events that occurred very far in the past, in completely different contexts.

The green balloons in Figure 1.2a illustrate the process of event bookmarking. The position of the current event is always marked in the timeline (a, b). When the user feels the current event is an important landmark, or a starting point for exploring several program paths, she can bookmark it by pressing the bookmark button (c), which also permits to choose a name and a color for the event (d).

Objects can also be bookmarked: individual objects can be assigned a color and/or a name, which they will bear whenever they are displayed. It is therefore possible, for instance, to mark an object that is involved in a failure so that previous usage of that object is easy to spot during navigation.

In addition to bookmarking, the user interface provides back and forward buttons similar to those of a web browser (Figure 1.2b), that also permit to access the entire navigation history.

Custom formatters By default, TOD displays reconstituted objects as a list of field = value pairs, and lets the user explore the object graph by “opening” objects of interest, as do
Most breakpoint-based debuggers. Although this is in some cases sufficient, there are many cases where a higher-level representation is more useful. Consider for instance a linked list (see Figure 3.5): the user is usually more interested in the sequence of elements of the list than in the details of the next/previous pointers of each node.

Like modern IDEs such as Eclipse, TOD lets the user define custom formatters for classes of interest. In TOD these formatters are small scripts\(^4\) that can easily access the fields of reconstituted objects to produce rich (HTML) textual representations. Figure 3.5 shows an example of such a formatter.

### 3.2 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 3.3 shows how our solution is amenable to parallelization, and Section 3.4 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, and achieve, rates in the order of hundreds of thousands of events per second [41]. Our high-throughput specialized database backend leverages the following specificities of the event stream of an execution trace: (a) the event stream is

---

\(^4\)Currently in Python, but support for other languages can be easily added.
(b) events arrive ordered by timestamp\(^5\) and (c) queries are limited to filtering.

Sect. 3.2.1 describes the indexing scheme used by the database. Sections 3.2.2, 3.2.3 and 3.2.4 analyze the cost of executing the queries described in Sect. 3.1.3. Finally Sect. 3.2.5 analyzes the recording throughput that can be achieved by the system and presents an important trade-off between memory requirements and efficiency.

### 3.2.1 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. 3.1.3, and at the same time permits to sustain a high recording throughput.

Using the notation defined in Section 3.1.2 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. 3.2.2.

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.

---

\(^5\)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.
It is nevertheless much more efficient to use a B+Tree, a hierarchical index structure comprising several levels (Fig. 3.6). Refining the above definition, the index $I_{j}(v)$ becomes a B+Tree. 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 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.

3.2.2 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 $a_{j_1} = C_1 \land \ldots \land a_{j_m} = C_m$, we use a variant of the sort merge join algorithm [6], widely-used in database management systems, to identify matching events without accessing them (Algorithm 3.1): we obtain the $I_{j_i}(C_i)$ for every simple condition, and for each we maintain a pointer to a current $(ts_i, i_i)$ entry. Then we loop: at every step we check if all of the $i_i$
Algorithm 3.1 MERGE-JOIN

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

**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.

### 3.2.3 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, as it depends on the number of components of the condition and the density of matching events. This is actually a key limitation of our approach, as in practice it can make some queries too slow.

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 3.2). 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.
3.2.4 Cost of counts

The counts queries retrieve the number of matching events in every time slice of length $\delta t$ 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. 3.1.4). 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. 3.2.1), 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 $\delta t$ 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 $\delta t$ we drill down into level $l - 1$, and conversely we roll up to level $l + 1$ if the time interval is too short.

3.2.5 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:

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

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:

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

The average number of page access per incoming event is then 0.13.

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

**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 3.7 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.
CHAPTER 3. EXHAUSTIVE CAPTURE AND INDEXING  

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 3.7: Number of indexes for each index set in an Eclipse execution trace.](image)

### 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 $N \cdot 2^{n/N} = N \cdot \sqrt[d]{d}$ 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. 3.2.2, the efficiency of queries is proportional to the size of the involved indexes. Table 3.3 summarizes the slowdown incurred by splitting an index set containing $d$ indexes and totaling $B$ entries. Each index in the set contains on average $B/d$ entries, thus the cost of a query on one of those indexes
CHAPTER 3. EXHAUSTIVE CAPTURE AND INDEXING

<table>
<thead>
<tr>
<th>Index set splitting</th>
<th>Number of indexes</th>
<th>Entries per index</th>
<th>Query cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>$d$</td>
<td>$B/d$</td>
<td>$B/d$</td>
</tr>
<tr>
<td>Yes</td>
<td>$\sqrt[d]{N}$</td>
<td>$B/\sqrt[d]{N}$</td>
<td>$N \cdot B/\sqrt[d]{N}$</td>
</tr>
</tbody>
</table>

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 3.3: Index set splitting.

is proportional to $B/d$. If the index set is split the number of indexes per index set becomes $d' = \sqrt[d]{N}$ 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 $N \cdot d/d' = N \cdot d^{1-\frac{1}{d}}$. 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. 3.7), 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. 3.1.4, which is based on a conjunction of conditions on the field id and object id attributes. In the Eclipse trace previously mentioned there are about $10,000$ distinct field id values and $10,000,000$ distinct object id values (Fig. 3.7). 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. $B/\sqrt[10]{B} = B/3,160$ with index set splitting and $N = 2$. Therefore the actual slowdown of index set splitting for the compound query is:

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

Despite this slowdown, state reconstitution queries execute fast enough to be used interactively in the debugger frontend, as will be shown in Sect. 3.4.2.
3.3 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.

3.3.1 Distributed architecture

The architecture of the distributed backend of TOD consists of three layers (Fig. 3.8):

- 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 3.2. No change to the indexing structure is necessary.

The kind = FW part of the query is omitted in practice because only Field Write events have a value for the field id attribute.
• 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 3.1.1 need to be parallelized as their processing and storage requirements are modest.

### 3.3.2 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.

### 3.4 Benchmarks\(^7\)

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.

---

\(^7\)This section presents the benchmark results as published in our OOPSLA’07 paper [45]. We later realized that what we thought were worst cases, for trace capture as well as for queries, were not actually worst cases, as detailed in Section 4.6.
3.4.1 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. 3.5), 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.

**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 [31] omniscient debugger for Java. Results are presented in Table 3.4. 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\(^8\) 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. 3.5).

---

\(^8\)The different numbers of emitted events between TOD and ODB are apparently due to differences in the trace model.
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>
<thead>
<tr>
<th>Setup</th>
<th>RAM</th>
<th>time</th>
<th>emit.</th>
<th>rec.</th>
<th>rate</th>
<th>ovh.</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>16</td>
<td>1.53</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>ODB1</td>
<td>500</td>
<td>179</td>
<td>110m</td>
<td>5m</td>
<td>614</td>
<td>116</td>
</tr>
<tr>
<td>ODB2</td>
<td>64</td>
<td>188</td>
<td>110m</td>
<td>530k</td>
<td>585</td>
<td>122</td>
</tr>
<tr>
<td>TOD</td>
<td>16</td>
<td>173</td>
<td>90m</td>
<td>90m</td>
<td>520</td>
<td>113</td>
</tr>
</tbody>
</table>

Table 3.4: Overhead of event emission.

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.
3.4.2 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. 3.4.1. 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. 3.9, 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 CPU. Profiling shows that most of the time is spent copying buffers in methods of the \texttt{java.io} framework. We assume that this issue would disappear in an optimized C version of the dispatcher.

Fig. 3.10 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 \texttt{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 3.11 shows the efficiency of the \textit{step into} and \textit{step over} queries (Sect. 3.1.4). 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 \textit{step into} queries are faster than \textit{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 \textit{step} queries surprisingly decreases as more database nodes are used, but in any case, they 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 3.12 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.

**Count queries.** We measured the execution speed of count queries and compared the two counting methods described in Sect. 3.2.4: 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 3.5. Fast counts perform 10 to 30 time faster than merge counts while providing a very precise approximation, with a
CHAPTER 3. EXHAUSTIVE CAPTURE AND INDEXING

Figure 3.10: Recording throughput.

Figure 3.11: Efficiency of stepping queries.

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

Figure 3.12: Efficiency of object reconstitution queries.
Figure 3.13: Efficiency of counts queries.

Table 3.5: Comparison of merge and fast count queries.
distortion\(^9\) below 2\%. Figure 3.13 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.

### 3.4.3 Summary

Figure 3.14 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.

### 3.5 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. 3.4.1), 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.

\(^9\)Calculated as: \(\frac{\sum_{i=1}^{n} |\text{merge}[i] - \text{fast}[i]|}{\sum_{i=1}^{n} \text{merge}[i]}\)
3.5.1 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 3.15 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. 3.15a), and that the gains in runtime overhead can be up to 20 times (Fig. 3.15b).

3.5.2 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.

**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. 3.16. The code in Fig. 3.16a 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. 3.16b 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.
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 3.15: Emitted events and runtime overhead using scoped capture.
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

(b) Control flow view

Figure 3.16: Materialization of incomplete control flow information.

3.5.3 Specifying partial traces

Partial traces are supported by means of mechanisms similar to those of partial behavioral reflection [55]: 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).\(^{10}\) 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.\(^{11}\) 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.

\(^{10}\)It would be possible to refine spatial scoping using operation selectors [55], enabling expression-level selection to further reduce the size of the execution trace. This feature has not yet been integrated in TOD.

\(^{11}\)Although recent JVMs allow classes to be modified at runtime, we do not yet use this feature.
3.6 Related work

As an omniscient debugger, TOD is in the same category as the tools presented in 2.1. These are the characteristics of TOD that make it a competitive alternative:

- The **scalable database engine** enables fast storing and querying of events. Moreover, it can be distributed over a cluster of machines to further improve its scalability. It makes TOD the most scalable omniscient debugger for the Java platform.

- The **support for partial traces** dramatically enhances the applicability of TOD by offering expressive means to specify selective trace generation, and adequately reporting of incomplete information.

- The **responsiveness of the GUI**, achieved thanks to efficient query processing, permits to interactively navigate huge execution traces.

- The **specialized GUI metaphors**, such as the *why?* link, bookmarks, and time lines, allow for effective navigation and program understanding.

- The tight **Eclipse integration** permits to smoothly integrate TOD in the development process.

On the other hand, some of the features provided by other systems are missing from TOD:

- The Whyline [29] permits to ask **negative questions** like “why did method x not execute?”; those questions frequently occur during the debugging process.

- The Whyline permits to relate the **textual and graphical output** of the program to the event that caused them. Support for textual output in TOD would be easy to add, but support for graphical output would require considerable work.

- TOD has a **runtime overhead** similar to that of ODB [31] and the Whyline, while providing a much greater scalability. However, systems like UndoDB and the one by Lienhard [36] have a much lower overhead, while TimeMachine can have no overhead at all using a hardware probe. The next chapter shows how we reduce the runtime overhead by using deterministic replay.

- JIVE [23] provides graphical **visualizations of the object graph**, which can be very useful for program understanding. Although TOD supports custom formatters, these are only textual.
3.7 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.

Afterword  This chapter represents the culmination of our work regarding the features and user interface of the debugger. The performance of the system, on the other hand, has been much improved since. In the months following the completion of the prototype and the submission of the corresponding papers, we have used TOD extensively, in particular to debug new iterations of the very database engine TOD is based on. This usage uncovered efficiency issues that were not apparent in the benchmarks presented in this chapter: although the response time of queries is fast enough to be compatible with interactive navigation most of the time, there are cases where simple queries like stepping or object reconstitution take a very long time (up to several minutes). This is due to the fact that these queries rely on a conjunction of two or more indexes, which can be inefficient under adverse conditions. For instance, step over queries use both the thread id and depth indexes. If two threads in the debugged program perform a large amount of computation at the same control flow depths, finding the next or previous event at a given depth for a given thread can require the scanning of a significant portion of each involved index. It is thus desirable to devise a querying scheme that guarantees a bounded response time. Another point that should be improved is the runtime overhead on the debugged program. The next chapter presents a completely different capture and indexing process that enables very significant gains in both query response time and capture overhead.
Chapter 4

Strong response time guarantees with summarized trace indexing and querying\textsuperscript{1}

Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Overview</td>
<td>60</td>
</tr>
<tr>
<td>4.2</td>
<td>Summarized trace indexing and querying</td>
<td>62</td>
</tr>
<tr>
<td>4.3</td>
<td>Trace capture and partial deterministic replay</td>
<td>64</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Capture</td>
<td>64</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Initial replay</td>
<td>66</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Snapshots</td>
<td>66</td>
</tr>
<tr>
<td>4.4</td>
<td>Indexing of control flow</td>
<td>68</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Range Min-Max Tree</td>
<td>69</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Indexing and querying</td>
<td>70</td>
</tr>
<tr>
<td>4.5</td>
<td>Indexing of memory accesses</td>
<td>72</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Index structure and querying</td>
<td>72</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Pipelined index construction</td>
<td>74</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Local variable handling</td>
<td>76</td>
</tr>
<tr>
<td>4.6</td>
<td>Benchmarks</td>
<td>76</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Global results</td>
<td>77</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Cost of individual features</td>
<td>79</td>
</tr>
<tr>
<td>4.7</td>
<td>Related work</td>
<td>80</td>
</tr>
<tr>
<td>4.8</td>
<td>Conclusion</td>
<td>81</td>
</tr>
</tbody>
</table>

\textsuperscript{1}The contents of this chapter are based on our ECOOP 2011 paper “Indexing large execution traces for interactive back-in-time debugging” [44].
The previous chapter showed that omniscient debugging can be made highly scalable by carefully engineering the trace capture, and the indexing and querying processes. However, the proposed approach is rather brute force and suffers from two important inefficiencies:

- The execution trace is captured exhaustively: every operation is recorded and written to the trace. This is inefficient because a lot of operations sequences are deterministic: knowing the state of the program at the beginning of such a sequence permits to determine with full accuracy the state of the program at the end of the sequence. Recording the outcome of non-deterministic operations is therefore enough to be able to reconstitute the complete execution.

- The indexing of events is also exhaustive: all attributes of all events are individually indexed. However, many queries require the joining of several indexes: for instance, the step over queries combine the control flow depth and thread id indexes to retrieve the next event at the same depth in the same thread. Unfortunately, joining indexes is costly because it can require a scan of an unbounded portion of all involved indexes. This causes some queries to take a very long time to execute.

This chapter presents an approach that significantly improves both issues, in particular providing a strong guarantee regarding the execution time of queries.

4.1 Overview

Interactive navigation in large execution traces is challenging: memory-based approaches allow fast navigation, but do not scale past a few hundred megabytes of trace data and therefore must discard older data [31, 36]. To handle larger traces without losing information, a disk-based solution is mandatory, but this typically reduces the efficiency of the system. In addition, most omniscient debuggers, including TOD (Chapter 3), rely on directly capturing exhaustive executions traces [26, 31, 36]. Unfortunately, this incurs a significant runtime overhead on the debugged program, which is problematic for two reasons: (a) it makes the
system less practical to use because of long execution times, and (b) the probe effect can perturb the execution enough that the behaviors to examine do not occur.

An alternative technique to avoid capturing exhaustive traces that alleviates the above issues is **deterministic replay** \[4, 9, 48, 51, 58\]. It consists in capturing only a *minimal trace* of non-deterministic events during the initial execution of a program. This is much cheaper than capturing an exhaustive trace, and thus greatly reduces the probe effect. The minimal trace can be deterministically replayed to obtain the exhaustive trace without affecting the execution of the debugged program. Non-deterministic events are typically related to external inputs and system calls. However, another source of non-determinism is thread scheduling, something that is not properly supported in several deterministic replay systems.

Some deterministic replay systems support restarting the replay in the middle of the trace through **snapshots** that capture the state of the program at given points in time \[48, 51\]. However these snapshots are *heavyweight* because they represent the full state of the heap. These snapshots can be produced efficiently by combining process forks and OS-level copy-on-write mechanisms, but they are not easily serializable to disk. Therefore, snapshots remain in memory and older ones must be discarded, limiting the scalability or precision of the approach.

**Contributions** This chapter presents a novel scalable disk-based approach that supports efficient capture and interactive navigation of arbitrarily large execution traces. This approach relies on dividing the execution trace into bounded-size **execution blocks**, about which summary information is efficiently indexed. Execution blocks themselves are not stored on disk; rather, we support **partial deterministic replay**: the ability to quickly start replaying arbitrary execution blocks as needed. For querying, **summarized indexes** provide coarse answers at the level of execution blocks, which are then replayed and scanned to find the exact answer. More precisely:

- We describe the general approach and its instantiation as a new Java omniscient debugging engine called STIQ, for Summarized Trace Indexing and Querying (Section 4.2). The approach is based on capturing non-deterministic events during the execution of the debugged program, followed by an initial replay phase during which snapshots are taken and indexes are constructed. The high-level architecture of the system is the same as that of TOD (Sect. 3.1.1).

- We present an efficient deterministic replay system for Java (Section 4.3). This system supports partial deterministic replay through **lightweight snapshots** that are both fast to obtain and easy to serialize. We explain how lightweight snapshots make it unnecessary to capture the heap.

- We propose indexing techniques for both control flow and memory accesses. The techniques leverage a recent succinct data structure \[47\] for efficient control flow indexing
(Section 4.4), as well as the principle of temporal locality of memory accesses to reduce the amount of information to index (Section 4.5).

- We demonstrate through benchmarks that the approach enables a highly interactive omniscient debugging experience (Section 4.6). Specifically, the proposed technique allows very fast index construction and query processing. Index construction takes 4 to 25 times the original, non-captured program execution time on realistic workloads. Query processing requires $\mathcal{O}(\log n)$ disk accesses and $\mathcal{O}(1)$ CPU time for traces of arbitrary size $n$, and never exceeds a few hundred milliseconds in practice. We are not aware of any omniscient debugging system that provides either such efficient index building, or such strong guarantees in query response time.

4.2 Summarized trace indexing and querying

Interactive omniscient debugging requires that queries are processed fast enough to give the user a feeling of immediacy. For large execution trace, this mandates the use of indexing techniques: otherwise, arbitrarily large portions of the trace would have to be linearly scanned for each query. The system described in this chapter, dubbed STIQ, provides an indexing scheme that is fast to build and yet processes queries very efficiently. The key insight is to divide the execution trace into bounded-size execution blocks and to index only summarized information about each block; queries are then processed in two phases: the indexes first provide a coarse-grained answer at the level of execution blocks, which are then scanned to find the exact answer. This section gives an overview of the complete process, whose steps are detailed in subsequent sections.

**Trace capture** The debugged program is transparently instrumented so that whenever a non-deterministic operation (such as a system call or a memory read) is executed, its outcome is recorded into a minimal execution trace that is stored on disk. The trace is interspersed with regular synchronization points that give a rough timestamping of events, so that an approximate ordering of events of different threads can be obtained.

**Initial replay** The minimal trace produced by the capture phase is not directly useful for our indexing process. An initial replay is thus performed to obtain a semi-exhaustive trace consisting of memory write events and cursory method call information (only the fact that a method is entered/Exited is needed). This is achieved by feeding the minimal trace to a replayer that re-executes the original program, but with non-deterministic operations replaced by stubs that read the recorded outcome from the trace. The program is also instrumented so that it generates the needed semi-exhaustive trace. Additionally, when a synchronization point is encountered, a lightweight snapshot is generated so that replay can be restarted from that point later on. Snapshots thus define the boundaries of individually replayable execution blocks.
Chapter 4. Summarized Trace Indexing and Querying

The semi-exhaustive trace produced in the initial replay is not stored but rather consumed on the fly by an indexer that efficiently builds the indexes. The indexer summarizes the information of each execution block, as depicted in Figure 4.1. For method entry and exit events, the indexer builds a control flow tree and represents it as a range min max tree (RMM Tree) [47], a state-of-the-art succinct data structure that allows very fast navigation operations. Auxiliary structures map the beginning of execution blocks to positions in the RMM Tree. Together, these structures allow efficient stepping operations while using only slightly more than one bit per event. For memory writes, the indexer coalesces all the writes to a given location that occur within an execution block into a single index entry. In practice, this reduces the number of entries to index by 95%: because of the principle of temporal locality, if a memory location is accessed at some point in time, it is very likely that it will be accessed again in the near future, i.e. in the same execution block. Finally, snapshots are simply stored in an on-disk dictionary structure.

Querying The indexes constructed in the previous step can determine the execution block that contains the answer to a given query very quickly: they only require $O(\log n)$ disk accesses and $O(1)$ CPU time (with very favorable hidden constants—in practice they take 1-10ms). Once the execution block has been determined, the corresponding snapshot is retrieved (again in $O(\log n)$ disk accesses) and the block is replayed and then scanned to find the exact event of interest in $O(1)$ CPU time (as the size of execution blocks is bounded and thus does not depend on the size of the trace). In practice, queries take a dozen milliseconds on average, and never take more than a few hundred milliseconds (see Sect. 4.6).
4.3 Trace capture and partial deterministic replay

This section describes the key features of our deterministic replay system. Many implementation details are omitted or only glossed over. Section 4.3.1 describes how the trace is captured: which events are captured, how we avoid having to simulate the heap, and how memory locations are identified. It also describes the scoping abilities of our system. Section 4.3.2 discusses the replayer, and Section 4.3.3 details how and when snapshots are taken.

4.3.1 Capture

Trace capture is achieved by transforming the original program through bytecode instrumentation so that non-deterministic events are serialized and stored when the program is executed.

Non-deterministic events

Non-deterministic operations are those whose outcome can vary from one program execution to another, namely:

- **Native operations.** The outcome of native operations such as disk or network reads cannot be predicted. In addition, as discussed later in this section, our system supports user-defined scoping. Out-of-scope methods are considered non-deterministic.

- **Heap memory reads.** Thread scheduling can affect the order in which memory write operations are executed, and as scheduling is outside the control of the debugged program, the contents of memory is non-deterministic.

Avoiding heap simulation

A strategy to deal with the non-determinism of memory reads due to multi-threading consists in recording the order in which threads are scheduled, and forcing the same order during replay [9]. This is of limited usefulness, however, on multicore architectures. Another strategy, which we use in our system, consists in recording the value obtained by every memory read [4].

Although capturing memory reads is enough to allow a fully accurate replay, it does not allow to avoid the simulation of the heap during replay because some control-flow-altering operations (polymorphic method dispatch and `instanceof`) rely on the content of the heap, as the type of objects must be known. Although these operations are deterministic, we capture their outcome and include it in the minimal trace. This has a very small impact on the trace capture overhead, but makes the simulation of the heap during replay unnecessary, reducing the memory requirements of the system to just the buffers used for the indexing process.
Moreover, the only information needed to start replaying at execution block boundaries can thus be represented in *lightweight snapshots* that only contain the values of the local variables of the current stack frame and the identifier of the current method. Such snapshots are cheap to obtain and take up a reasonable space (0.8% to 14% of total trace size in practice).

**Identification of memory locations**

To be properly indexed, memory locations must be uniquely identifiable. Two distinct types of locations must be considered: heap locations (object fields and array slots), and stack locations (local variables).

For heap locations, we regard both objects and arrays as *structures* that contain a fixed number of slots. Structures are assigned a unique id at creation time, and the id of a particular location within a structure is obtained by adding the index of the slot to the id of the structure. For objects, the index of the accessed slot is determined statically (each field of a given class can be assigned an index statically). For arrays, the index of the accessed slot is explicitly specified at runtime. To ensure the uniqueness of memory location ids, the sequence that is used to give a new structure its unique id is incremented by the number of slots of the structure.

In Java, the ideal way to store the id of structures would be to add a field to the `Object` class. However, adding fields to certain core classes such as `Object`, `String` or array classes is problematic in most Java implementations (*e.g.* doing so makes the HotSpot JVM crash). To solve this issue, we add the id field to all the other subclasses of `Object`, and we use a global weak identity hash map for the problematic classes; this unfortunately incurs a significant runtime overhead (as shown in Sect. 4.6).

For stack locations, we use a compound id consisting of the id of the current thread, the current call stack depth, and the index of the variable within the stack frame. This scheme does not uniquely identify each location, because local variables in subsequent invocations of different methods by the same thread at the same depth will share the same id. However, this is not a problem because the temporal boundaries of method invocations are known. We come back to this issue in Section 4.5.3.

**Scoping**

In many cases some parts of the debugged program might be trusted to be free of bugs (for instance, the JDK classes in the case of Java), or the bug can be known to manifest only under certain runtime conditions (Sect. 3.5). Trace scoping reduces the runtime overhead on the debugged program, the size of the execution trace, and the indexing and querying cost, by limiting the set of events that are captured. Static scoping consists in limiting capture to a set of classes, while dynamic scoping consists in activating or deactivating capture dynamically at runtime. Our system currently supports only static scoping; dynamic scoping would however be relatively easy to add.
The user configures the static scope by specifying a set of classes or packages to include or exclude from the trace. We define the set of out-of-scope methods as all the regular bytecode-based methods that belong to out-of-scope classes, as well as all native methods.

By definition, the execution of out-of-scope code cannot be replayed. It is therefore necessary to capture additional information at the runtime boundaries between in-scope and out-of-scope code. In particular, the return values of out-of-scope methods called by in-scope methods, as well as the arguments of in-scope methods called by out-of-scope code must be captured.

Because of polymorphism, however, it is not possible to statically determine whether a particular call site will result in the execution of an in-scope or of an out-of-scope method; similarly, it is not possible to determine if a given method will be called by in-scope or out-of-scope code. Therefore, in the trace capture phase we instrument the envelope (ie. entry and exit) of all out-of-scope methods in order to maintain a thread-local scope stack of booleans that indicates whether the thread is currently executing in-scope or out-of-scope code.

4.3.2 Initial replay

The main task of the replayer is to inject the recorded outcomes of non-deterministic operations into the replayed program. To that end, we transform the program through bytecode instrumentation so that non-deterministic operations are replaced by proxies that read their outcome from the trace.

As explained above, the heap is never explicitly reconstituted; therefore, the replayer never needs to instantiate any class of the original program: instances are instead represented by a generic ObjectId class that is simply a container for the identifier of the object\(^2\). All the non-static in-scope methods of the program are replaced by static ones that take an additional ObjectId parameter that represents the target of the method.

On the other hand, as out-of-scope methods do not record any information in the trace (except the envelope as explained above), they all behave exactly in the same way as far the replayer is concerned: a black box that swallows parameters and generates a return value. Therefore the original out-of-scope methods are not used at all in the replayer, and are collectively replaced by a single, generic method provided by the replayer infrastructure.

Also, note that thanks to avoiding the reconstitution of the heap, the memory requirements of the replayer do not depend on the memory requirements of the debugged application. This frees up valuable memory for the indexing process.

4.3.3 Snapshots

Snapshots define the boundaries of execution blocks. Recall that snapshots are taken during the initial complete replay of the program, and not during capture, so as to reduce the

\(^2\)We use a container instead of a scalar because the actual value of the id is mutable in the case of instantiations, but this is beyond the scope of this document.
runtime overhead of capture as much as possible. We now describe how and when snapshots are taken.

**Snapshot probes**

The ability to take a snapshot at a given program point requires the insertion of a piece of code, called a *snapshot probe*, that performs the following tasks:

1. Check if a snapshot is actually requested at this moment, by reading a thread-local flag (detailed below).
2. Store the necessary information in the snapshot: identification of the snapshot probe, current position in the minimal execution trace, and the values of local variables and operand stack slots.

Recalling that the heap is not reconstituted during replay, the information mentioned above is sufficient for replaying the current method and all the method called from there, recursively. It is not sufficient, however, to return to the caller of the current method: the stack frame of the caller is not recorded in the snapshot. This problem is addressed by always inserting snapshot probes after method calls, and forcing the creation of a snapshot at those probes if a snapshot was taken during the execution of the method. Thus, although the partial replay cannot directly continue after the current method returns, there is always another snapshot at the right point in the caller method so that another partial replay can be started right where the previous one finished.

**Snapshot intervals**

The size of execution blocks must be chosen considering a trade off between indexing efficiency and querying efficiency:

- Larger blocks make it possible to coalesce more object accesses into one index entry, thus increasing indexing throughput.
- Shorter blocks can be replayed faster and thus queries can be answered faster.

It is also important to take into account the involved magnitudes:

- Indexing is performed on the fly during the initial complete replay, and preemptively considers all of the objects that existed during the execution of the program: all object accesses in the trace incur an indexing cost. Therefore, small variations in indexing throughput can noticeably affect the global efficiency of the system.
- Queries deal with individual objects and are performed by a human being, who cannot differentiate between a one microsecond or a one millisecond response time. Therefore, important variations in querying efficiency can go largely unnoticed up to a certain point.
The time interval between snapshots define the maximum size of execution blocks. This interval is configurable by the user, controlling the tradeoff between indexing efficiency and query response time.

**Probe density**

Probes should be inserted densely enough in the program so that a snapshot can be taken quickly once it is requested. However, snapshot probes are costly both in code size and in speed (because of the runtime check) so it is preferable to limit their number. As we must insert snapshot probes after every method call anyway (as explained above), the density is usually already sufficient with just those probes. Nevertheless, it is possible for the program to contain a loop with no method calls at all, like a complex calculation on a large array; in this case, an additional probe would be needed in the loop. For the sake of simplicity, and because this kind of program is rather infrequent, we currently do not insert these additional probes.

### 4.4 Indexing of control flow

We now turn to the indexing techniques. This section describes indexing of control flow, and Section 4.5 describes indexing of memory accesses. Indexing control flow is necessary to efficiently perform step over and step out queries. Without an index, it would be necessary to linearly scan the execution trace to skip the events that occurred in the control flow of the stepped over call, or between the current event and the beginning of the current method.

The control flow can be represented as a tree whose nodes correspond to method calls. Stepping operations then simply correspond to moving from a node to its next/previous sibling (for step over), or to its parent (for step out). We store the control flow tree using a range min-max tree (RMM Tree) [47], a recent succinct data structure that is disk-friendly, fast to build and supports fast navigation operations. Auxiliary data structures maintain a correspondence between execution blocks and their initial node in the control flow tree. This approach uses only slightly more than 1 bit per method call or return event, while requiring only a few milliseconds to answer arbitrary stepping queries, making them seem instantaneous to the user.

This section first briefly describes the RMM Tree structure and then explains our control flow indexing and querying mechanism.

3We also set a minimum size for execution blocks, so that a thread that spends most of its time sleeping does not generate plenty of useless snapshots.

4Step into queries do not need an index as they do not require to skip events.
### 4.4.1 Range Min-Max Tree

A succinct data structure is one that stores objects using space close to the information-theoretic lower bound, and at the same time supports fast queries on the stored objects. In the case of a tree\(^5\) with \(n\) nodes, the lower bound is \(2n - \Theta(\log n)\) bits [47]. A classical way to represent trees using \(2n\) bits is the balanced parentheses sequence (see Figure 4.2): each node is represented by a pair of matched parentheses that enclose the representation of its children. A node in the tree is identified by the position of the corresponding opening parenthesis.

Although such a structure is compact (as only one bit is needed for each parenthesis), it does not allow per se to quickly answer queries like finding the next sibling, previous sibling or parent of a given node. The range min-max tree (RMM Tree) [47] adds an indexing layer on top of the balanced parentheses representation that incurs very little space overhead while allowing extremely fast querying. In theory, the RMM Tree supports queries in constant time \(O(c^2)\) with a data structure using \(2n + O(n/\log^c n)\) bits, for any constant \(c > 0\). In practice, we trade the constant time for logarithmic time with a very big base.

The essential idea of the RMM Tree is to compute a running sum of the bits that represent the parentheses sequence: opening parentheses increment the sum by 1, and closing parentheses decrement the sum by 1. For each fixed-size block of parentheses, a summary indicating the minimum and maximum value that the sum takes within the block is stored separately. Fixed-size blocks of summaries are then recursively summarized (the minimum and maximum of a whole block of summaries are separately stored at a higher level). This results in a tree structure of height \(H\) in which the leaves are the bits that represent the balanced parentheses sequence, and the nodes contain the minimum and maximum value of the running sum in their subtree. Subtle observations about the relationship between the running sum and the primitive tree navigation operations make it possible to guarantee that queries can be answered by accessing at most \(2H\) blocks (going up to the root and then down to the correct leaf) [47].

In practice, blocks correspond to disk pages (usually 4KB). The summary information to store for each block (minimum and maximum values plus some ancillary data) occupies only 10 bytes. As a consequence the tree is quite flat: for instance, an RMM Tree of height 4 can store up to \(\left\lfloor \frac{4096}{10} \right\rfloor^3 \cdot 4096 \cdot 8 \approx 2 \cdot 10^{12}\) bits in its leaves and occupies around \(4096 \cdot \sum_{i=0}^{3} \left\lfloor \frac{4096}{10} \right\rfloor^i \approx 280\)GB, thus requiring roughly 2.005 bits per original tree node (slightly more than 1 bit per

\(^5\)Specifically, ordinal trees, where a node can have any number of ordered children.
CHAPTER 4. SUMMARIZED TRACE INDEXING AND QUERYING

Algorithm 4.1 Find return event.

Finds the return event corresponding to the call event denoted by \((t, b, i)\).

1: function FindReturn\((t, b, i)\)
2: \(tree \leftarrow \text{getCFlowTree}(t)\)
3: \(p_{\text{call}} \leftarrow \text{eventToPosition}(t, b, i)\)
4: \(p_{\text{ret}} \leftarrow tree.getClose(p_{\text{call}})\)
5: \((t_{\text{ret}}, b_{\text{ret}}, i_{\text{ret}}) \leftarrow \text{positionToFEvent}(t, p_{\text{ret}})\)
6: \(\text{return } (t, b_{\text{ret}}, i_{\text{ret}})\)
7: end function

\[\text{By construction } t = t_{\text{ret}}\]

Algorithm 4.2 Event to position.

Returns the RMM Tree position corresponding to the given event reference.

1: function EventToPosition\((t, b, i)\)
2: \((\text{tree, map}) \leftarrow \text{getCFlowIndex}(t)\)
3: \(p \leftarrow \text{map.getPos}(b)\)
4: \(\text{block} \leftarrow \text{getBlock}(t, b)\)
5: for \(k \in 1, i\) do
6: if \(\text{block}[k]\) is a call or return event then
7: \(p \leftarrow p + 1\)
8: end if
9: end for
10: return \(p\)
11: end function

parenthesis).

4.4.2 Indexing and querying

The indexing process for control flow is straightforward: each execution thread has its own RMM Tree that stores all the method call (resp. return) events as one opening (resp. closing) parenthesis as they occur. Also, execution blocks are identified by a thread-local block id, equal to the timestamp of the initial snapshot of the block. Blocks ids are unique within a thread, but not globally. Whenever a new execution block starts, a pair (block id, current RMM position) is stored in a bidirectional map, which makes it possible to either retrieve the block id given a RMM position, or the RMM position given a block id. More precisely, this bidirectional map consists of two BTrees, one where the block ids are the keys and the RMM positions are the values, and another one with the opposite relationship. As BTrees use binary search for keys, the keys used for lookup do not need to be exact values. We take advantage of this feature when looking up a block id given a position: there is usually no record for the exact position, but we can instead return the id of the block that contains this position.

To perform a step over operation\(^6\), it is necessary to determine the return event corresponding to the call event that is being stepped over. The result of the step over operation is simply the event following the return event. The \textit{findReturn} function (Algorithm 4.1) is thus the basis of the step over operation.

\(^6\)We describe forward step over; backward step over and step out are similar.
**Algorithm 4.3** Position to event.

Returns the event reference corresponding to the given RMM Tree position.

```plaintext
1: function PositionToEvent(t, p)
2:   (tree, map) ← getCFlowIndex(t)
3:   b ← map.getBlockId(p)                       ▷ p₀ is the position of the RMMTree corresponding to the beginning of block b
4:   p₀ ← map.getPos(b)
5:   block ← getBlock(t, b)
6:   i ← 1
7:   while p₀ < p do
8:     if block[i] is call or return event then
9:       p₀ ← p₀ + 1
10:    end if
11:   i ← i + 1
12: end while
13: return (t, b, i)
14: end function
```

Events are identified by a (t, b, i) tuple where t is the thread id, b is the block id, and i is the index of the event within the block. The algorithm consists of three steps: (a) determining the position of the bit (or opening parenthesis) of the RMM Tree that corresponds to the given method call event, (b) determining the corresponding closing parenthesis, that corresponds to the return event, and finally (c) translating the RMM Tree position back to an event reference. Translating back and forth between event references and RMM Tree positions is implemented in the subroutines specified in Algorithms 4.2 and 4.3.

The algorithms use the following auxiliary procedures:

- _getBlock(t, b)_ replays block b of thread t and returns the exhaustive list of events for that block.

- _getCFlowIndex(t)_ returns the RMM Tree and bidirectional map corresponding to thread t; _getCFlowTree(t)_ returns only the RMM Tree. These are constant-time operations.

There are three components to the cost of the algorithm:

- The replaying of the initial and final execution blocks (although the initial execution block is usually available in a cache, as it corresponds to the events the user was currently observing). These operations take a time proportional to the size of the blocks, which is a constant that can be tuned by the user.

- The obtention of block ids and positions through the bidirectional map.\(^7\) These operations are BTree lookups that require \(\mathcal{O}(\log n)\) disk accesses.

- The navigation to the closing parenthesis in the RMM Tree. This operation also requires \(\mathcal{O}(\log n)\) disk accesses.

\(^7\)In Algorithm 4.3, lines 3 and 4 are actually a single operation, as the binary search for the given position gives both the registered position and the corresponding block id.
In practice, arbitrary stepping queries only take a dozen milliseconds on average, and never take more than a few hundred milliseconds, allowing highly interactive stepping (see Sect. 4.6).

4.5 Indexing of memory accesses

Two of the essential features of omniscient debuggers are the ability to inspect the state of memory locations at any point in time, and the ability to instantly navigate to the event that assigned its value to a location. Both features rely on the same basic query: finding the last write to the location that occurred before a reference event (the point of observation). The write event indicates both the value that was written and the moment it was written.

The key to being able to answer such queries efficiently is to have a separate index for each memory location; if a single index is shared between several locations, a linear scan (which can take a time proportional to the size of the trace) is necessary. This said, constructing an exhaustive index of all write accesses for each location is prohibitively costly (see Sect. 3.3). Instead, we index only a summary of the write accesses: we coalesce all accesses to a given location that occur within an execution block to a single index entry. We thereby exploit the principle of temporal locality: if a given location is accessed at a point in time it is very likely to be accessed again in the near future, i.e. in the same execution block. In practice, this approach allows us to discard around 95% of memory accesses. This compression ratio, along with the pipelined index construction process described later, makes it possible to maintain a separate index for each memory location.

To answer queries, the index is used to determine, in logarithmic time, the execution block that contains the access of interest; the block is then replayed and linearly scanned to retrieve the exact event. As block size is bounded, this linear scan does not depend on the size of the trace, and is very fast in practice, as will be shown in Sect. 4.6.

In the following we first present the general structure of the index and the way it is queried, before explaining how to build it efficiently using a multicore-friendly pipelined process. This section deals mostly with heap memory locations (object fields and array slots). The capture system assigns a unique identifier to each heap location, as explained in Sect. 4.3.1. We explain how stack locations (local variables) are handled in Sect. 4.5.3.

4.5.1 Index structure and querying

Memory inspection queries consist in finding the last write to a given location that occurred before a certain reference event. As explained above, there is one individual index for each

---

8Although to simplify the presentation we consider a single result for memory inspection queries, there is actually a set of write events that might have written the current value of the location at the time the reference event occurred. The reason the query produces a set and not a single event is that the resolution of data races is limited by the accuracy of the timestamping of events.
Algorithm 4.4 Memory location reconstitution.

1: function GetLastWrite(loc, (t, b, i))
2:     index ← getLocationIndex(loc)
3:     (b2, threads) ← index.getAtOrBefore(b)
4:     for t2 in threads do
5:         block ← getBlock(t2, b2)
6:         if b2 = b and t2 = t then
7:             limit ← i − 1
8:         else
9:             limit ← length(block)
10:        end if
11:     for k in limit, 1 do
12:         if block[k] is write to loc then
13:             yield (t2, b2, k)
14:         break
15:     end if
16:     end for
17: end for
18: end function

memory location. As there are many such location indexes, there is also a master index used to retrieve particular location indexes.

The process of answering a query is sketched in Algorithm 4.4. It consists of three main steps:

1. Retrieve the index for the particular location using the master index (line 2). This is implemented as a BTree lookup, and thus requires $O(\log n)$ disk accesses.

2. Search the location index for the execution block(s) that occurred at the same time or just before the block $b$ that contains the reference event (line 3). This search can produce as many blocks as there are threads writing to the location in the same time span as block $b$. As we explain later, there are different implementation of the location indexes, according to the number of entries in the index, but in the worst case the search requires $O(\log n)$ disk accesses.

3. Replay the blocks of the previous step to find the last write(s) to the inspected location. Although there can be any number of blocks to replay, the size of blocks decreases with the number of concurrent threads. This is because blocks are delimited by elapsed time (see Sect. 4.3.3): the more threads execute concurrently at a given time, the less events there are in the corresponding blocks.\(^9\) The time required to replay those blocks is therefore bounded and does not depend on the size of the trace.

As shown in Sect. 4.6, such queries in practice only take two dozen milliseconds on average, and never more than a few hundred milliseconds, allowing very fast reconstitution of memory locations.

\(^9\)Modulo the number of available CPU cores, but this is also a constant.
4.5.2 Pipelined index construction

The previous section showed that it is possible to query the memory locations index in logarithmic time. We now show that the index can also be efficiently built. As explained in Sect. 4.2, an initial replay of the minimal trace is performed so as to obtain a semi-exhaustive execution trace that contains memory writes events. The semi-exhaustive trace is consumed on the fly by the indexer.

The indexing process is divided into 5 pipelined stages (see Fig. 4.3), and can thus take advantage of multicore systems, as the different stages can run in parallel (although the CPU utilization is not evenly distributed between all stages). The first three stages operate in main memory, while the latter two deal with storing data on disk. By conveniently ordering the data, the first stages help reduce the amount of disk seeks needed at the later stages.

Summarizing  This stage (Fig. 4.3a) is instantiated for each thread of the debugged program. It scans incoming execution blocks, and for each memory write, it adds the identifier of the written location to a hash set. Using a set is key to our indexing approach, ensuring that each written location appears only once per execution block. Once an execution block is finished, the set is transformed into a \((t, b, a)\) tuple where \(t\) and \(b\) are the thread and block id, and \(a\) is a sorted array of the location identifiers that have been written to within the block. The tuple is then passed on to the next stage.

Because execution blocks are relatively small in practice, all the operation of this stage can be performed in memory.
Reordering  During trace capture, events are stored in thread-local buffers before being stored in the minimal trace so as to reduce contention on shared data structures. Busy threads emit many events, so they quickly fill their event buffers, while threads that spend a lot of time waiting might take a long time to fill a single buffer. It is therefore possible that execution blocks of different threads are stored in the trace out of order. The later stages of the pipeline can cope with this situation, but at the cost of a significant loss of throughput. The goal of the reordering stage (Fig. 4.3b) is thus to avoid as much as possible the costly reordering by downstream stages.

This stage accumulates the tuples in a buffer, and when their total size exceeds a certain threshold (32 MB in practice), they are sorted by block id, and the oldest ones (the oldest 60% in practice) are passed on to the next stage in a bundle for processing. The remaining ones stay in the buffer and will be sorted again, along with newer ones and possible later comers, in the next round. The aforementioned threshold size is chosen to be small enough so that the data sets of this stage and the following one can fit in main memory, but large enough to impede most out-of-order blocks from going through.

Inversion  This stage (Fig. 4.3c) receives bundles of \((t, b, a)\) tuples and operates in two phases:

1. Each \((t, b, a)\) tuple is expanded into a list of \((t, b, l)\) tuples, one for each memory location \(l \in a\). The threshold size chosen in the previous stage has to be small enough that the expanded tuples of this stage can fit in main memory.

2. The concatenated list of all the \((t, b, l)\) tuples is then sorted by location id \(l\), then by block id \(b\) and finally by thread id \(t\).

As a consequence of the sorting, the tuples produced in this stage are grouped by location, which reduces the amount of disk seeks needed to build the on-disk index in the following stages. Additionally, having the tuples within each group sorted by block id and thread id enables the use of compact encodings, thus reducing the size of the indexes, as explained below.

Allocation  For each location group in the tuple list produced by the previous stage, an entry is allocated in the master index (or retrieved, if it already existed). An entry is simply a pointer that references the page where the individual index corresponding to the location is stored. The tuple list of the previous stage is passed on to the next stage, along with a list of allocated entries, so that the next stage can perform the actual storage of the tuples of each group without having to access the master index anymore.

Storage  This final stage performs the actual storage of \((t, b)\) tuples in the individual indexes corresponding to each location \(l\). According to the number of tuples to store in each index, three different index formats are used:
Because most objects are short lived and therefore are accessed in only one execution block, most indexes (around 80%) contain a single tuple. We store these indexes in shared pages, which we call *singles pages*. Thanks to the ordering performed in the previous stage and the use of gamma codes\textsuperscript{10} to store the difference between successive tuple components, a 4KB singles page contains around 800 indexes on average.

For indexes that contain more than one tuple but less than the number of tuples that can fit in half a disk page, we use another type of shared pages, which we call *n-shared pages*, with $n \in \{2^m\}$ for $m \in [1..7]$. In these pages, space is evenly distributed between $n$ indexes.

For bigger indexes, we use BTrees where keys are block ids and values are thread ids. Again, we use gamma codes to store the tuples in these trees.

As indexes are built on the fly, we do not know beforehand what the size of each index will be. They thus migrate from singles page to $n$-shared pages to BTrees as more tuples are added.

### 4.5.3 Local variable handling

Having a separate index for each memory location implies that each location can be uniquely identified. As explained in Section 4.3.1, our trace capture system assigns a unique id to each heap location (object fields and array items), but this uniqueness constraint is relaxed for stack locations (local variables). Stack locations are assigned a compound id that is made of the thread id, the local variable index, and the call stack depth. This entails that there cannot be a separate index for each stack location, as the stack frames of subsequent method executions at the same level will share some local variable indexes. However, queries can still be processed efficiently: we already know the temporal boundaries during which particular stack locations exist (these boundaries are defined by method entry and exit, which are indexed). To process a stack location inspection query, we query the corresponding index as if it was not shared. If the answer is outside the temporal boundaries of the current method invocation, it means there is no write to the variable before the reference event.

### 4.6 Benchmarks

In this section we present the experimental results we obtained with our STIQ system, and we compare them with those obtained with TOD (Sect. 3.4). (We compare to other related systems in Section 4.7.) All the benchmarks were performed on a Quad-core 2.40GHz Xeon X3220 machine with 4GB RAM and two 160GB SCSI hard drives in a RAID-0 configuration.

\textsuperscript{10}Gamma codes [20] represent an integers $x$ in (roughly) $2\log_2 x$ bits. Small numbers are thus encoded in very few bits.
CHAPTER 4. SUMMARIZED TRACE INDEXING AND QUERYING

<table>
<thead>
<tr>
<th>Workload</th>
<th>$t_0$</th>
<th>$t_{STIQ}$</th>
<th>$\rho_{STIQ}$</th>
<th>$t_{TOD}$</th>
<th>$\rho_{TOD}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>avrora</td>
<td>5.5s</td>
<td>163s</td>
<td>30x</td>
<td>968s</td>
<td>176x</td>
</tr>
<tr>
<td>lusearch</td>
<td>7s</td>
<td>69s</td>
<td>10x</td>
<td>157s</td>
<td>22x</td>
</tr>
<tr>
<td>burntet</td>
<td>5.2s</td>
<td>65s</td>
<td>12x</td>
<td>427s</td>
<td>82x</td>
</tr>
</tbody>
</table>

$t_0$: original execution time without trace capture
$t_{TOD}, t_{STIQ}$: execution time with trace capture
$\rho_{x}$: runtime overhead ($t_x/t_0$)

Table 4.1: Runtime overhead of trace capture.

running the x86_64 Linux 2.6.24 kernel. We used the Sun HotSpot 1.6.0_22 32 bits JVM in server mode for both the debuggee program and the indexing server.

We used the **avrora** and **lusearch** benchmarks of the DaCapo v9.12 benchmark suite [5], as well as a toy benchmark called **burntest** that stresses STIQ capture and indexing by performing almost only method calls and field accesses (it consists in repeatedly navigating a large in-memory tree). For DaCapo benchmarks, we use the **small** dataset size, and force two driver threads. For both STIQ and TOD, the JDK classes were configured to be out of scope.

We first present global results (capture overhead, indexing speed and query efficiency) that show the competitiveness of our approach. We then give a detailed accounting of the time and space resources needed for individual features.

### 4.6.1 Global results

Table 4.1 shows the impact of trace capture on the debugged program. It varies between 10x and 30x for STIQ and between 22x and 176x for TOD\(^{11}\). The overhead of STIQ is much lower than that of TOD, as well as that of other omniscient debuggers: the Omniscient Debugger [31] has an overhead of around 120x, while Chronicle (discussed in Sect. 4.7) reports a 300x overhead. Also, STIQ has an overhead comparable with other deterministic replay systems like Nirvana [4], which reports a 5x-17x overhead. Nirvana however is only concerned about deterministic replay, not trace indexing.

With respect to trace capture, even though the numbers are comparatively favorable to STIQ, the capture overhead still remains high; further effort is necessary in this regard.

Table 4.2 indicates the time needed to index the captured traces. For the DaCapo benchmarks, STIQ actually uses less time to perform the initial replay and build the indexes than to capture the trace. For **burntest** on the other hand, the indexing is very slow, as that workload consists only in method calls and field accesses, with no extra deterministic computation in between. STIQ is (at least) one order of magnitude faster than TOD to build the indexes.

\(^{11}\)This shows that the published worst-case runtime overhead of 80x for TOD [45] was not actually the worst case.
CHAPTER 4. SUMMARIZED TRACE INDEXING AND QUERYING

Table 4.2: Replay and indexing time (and ratio to original execution time).

<table>
<thead>
<tr>
<th>Workload</th>
<th>STIQ</th>
<th>TOD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>replay</td>
<td>indexing</td>
</tr>
<tr>
<td>avrora</td>
<td>95s (17x)</td>
<td>46s (8.4x)</td>
</tr>
<tr>
<td>lusearch</td>
<td>19s (2.7x)</td>
<td>13s (1.8x)</td>
</tr>
<tr>
<td>burntest</td>
<td>39s (7.5x)</td>
<td>375s (72x)</td>
</tr>
</tbody>
</table>

Table 4.3: Space usage.

<table>
<thead>
<tr>
<th>Workload</th>
<th>STIQ</th>
<th>TOD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>trace</td>
<td>index</td>
</tr>
<tr>
<td>avrora</td>
<td>5GB</td>
<td>0.27GB</td>
</tr>
<tr>
<td>lusearch</td>
<td>1.1GB</td>
<td>0.16GB</td>
</tr>
<tr>
<td>burntest</td>
<td>1.5GB</td>
<td>2.7GB</td>
</tr>
</tbody>
</table>

Table 4.3 shows the size of the captured execution traces, as well as the size of the created indexes. STIQ traces are much smaller than those of TOD, showing the benefit of using a deterministic replay system versus exhaustive trace capture. It is notable that for the DaCapo benchmarks, STIQ produces indexes that are much smaller than the trace itself; for burntest the index is almost twice as big as the trace, again because burntest is all about method calls and field accesses, which are the two kinds of events that are indexed. Also worthwhile to note is the fact that TOD indexes are always bigger than the already bulky traces.

Table 4.4 shows the query response time of STIQ and TOD. For stepping queries, we divide each thread of the execution trace into 100 equal intervals and starting at the beginning of each interval we alternately perform step over and step out operations until the root of the control flow is reached. As we get closer to the control flow root, step over operations must skip a greater number of events. For memory inspection queries, we first realize a (non-timed) pass that collects the locations to inspect: we divide each thread into 20 equal intervals and start scanning the trace at the beginning of each interval, collecting accessed locations until 20 distinct locations are found. After the collection phase, we once again divide each thread into 20 equal interval and inspect the content of each location at the beginning of each interval.

Table 4.4: Average (and maximum) query response time.
CHAPTER 4. SUMMARIZED TRACE INDEXING AND QUERYING

<table>
<thead>
<tr>
<th>Workload</th>
<th>object ids map</th>
<th>field reads</th>
</tr>
</thead>
<tbody>
<tr>
<td>avrora</td>
<td>9.5%</td>
<td>66%</td>
</tr>
<tr>
<td>lusearch</td>
<td>17%</td>
<td>53%</td>
</tr>
<tr>
<td>burntest</td>
<td>41%</td>
<td>47%</td>
</tr>
</tbody>
</table>

Table 4.5: Cost of capture features as percentage of total capture time.

<table>
<thead>
<tr>
<th>Workload</th>
<th>control flow</th>
<th>memory locs</th>
<th>snapshots</th>
<th>strings</th>
</tr>
</thead>
<tbody>
<tr>
<td>avrora</td>
<td>56%</td>
<td>28%</td>
<td>14%</td>
<td>0.6%</td>
</tr>
<tr>
<td>lusearch</td>
<td>14%</td>
<td>71%</td>
<td>11%</td>
<td>4%</td>
</tr>
<tr>
<td>burntest</td>
<td>1.3%</td>
<td>97%</td>
<td>0.8%</td>
<td>0.7%</td>
</tr>
</tbody>
</table>

Table 4.6: Size of the different indexes as percentage of total index size.

The experimental results clearly show the benefit of our approach. STIQ queries are guaranteed to take \(O(\log n)\) disk accesses and \(O(1)\) CPU time; in practice they never reach the one second mark, and take only a dozen milliseconds on average. In contrast, some TOD queries can take an extremely long time, thus jeopardizing the interactivity of the debugging session\(^\text{12}\).

Overall, we consider our approach successful. Capture overhead, indexing times and trace sizes are all significantly better than TOD. In addition, STIQ really shines at query processing, always guaranteeing interactive-compatible response times. We are not aware of any system that gives such strong guarantees on query process times.

4.6.2 Cost of individual features

This section gives a detailed accounting of the cost of the different features of STIQ for both capture and indexing. This is useful to pinpoint optimization targets.

Table 4.5 shows the cost of two important features used at capture time. As mentioned in Sect. 4.3.1, we must resort to a global map to store the ids of the instances of certain problematic classes. This has a non-negligible cost, that could be avoided if the JVM was modified to allow additional fields to be added to the \texttt{Object} class. The non-determinism of memory caused by thread scheduling requires the capture of the values of each memory read. This represents about half the capture time.

Table 4.6 show how the index size is distributed among the different indexes\(^\text{13}\). The distribution varies widely from a workload to another, but it is worthwhile to note that our lightweight snapshots use comparatively very little space.

\(^{12}\)Note the average query times for TOD are high in great part because of a few extremely long outliers; many queries still execute in a few dozen milliseconds.

\(^{13}\)The strings index stores the values of the strings used in the program. As it is not directly used for queries and has very limited impact in general, we did not mention it elsewhere in this chapter.
4.7 Related work

We now discuss related work in the areas of omniscient debugging, deterministic replay, and analysis of captured execution traces.

Omniscient debugging

Several omniscient debuggers have been presented in Section 2.1. We concentrate here on those that tackle scalability issues.

Chronicle\textsuperscript{14} by Robert O’Callahan is an omniscient debugger for native Linux programs that is designed to deal with large execution traces. As TOD (Chapter 3), it relies on exhaustive trace capture, and it creates an on-disk index of the execution trace. It performs compression of both trace and index data. It is interesting to note that for indexing memory accesses it uses the principle of \textit{spatial locality}: contiguous instructions that access contiguous memory locations produce a single event. However it does not create an individual index for each memory location, and thus suffers from the same limitation as TOD: it is possible that a large number of entries have to be scanned before finding the correct one. The runtime overhead of trace capture (300x) is also much higher than what we achieve with STIQ.

ODB [31] is an omniscient debugger for Java that stores the execution trace in RAM, in the same process as the debugged program. Because the amount of available storage is limited, they resort to discarding the oldest events to make room for the new ones. Lienhard \textit{et al.} [36] discard the events that relate to objects that have been garbage collected. In both cases, discarding events can limit the usefulness of the approach, as bugs can have occurred much before the symptoms appear, or in the context of objects that are no longer in use.

Deterministic replay

Replay-based systems have been presented in Section 2.4. Flashback [51] and Jockey [48] are deterministic replay systems for native Linux programs. Flashback relies on a modified kernel while Jockey relies on program instrumentation. They both take periodic snapshots of the state of the debugged process and record the interactions between the program and its environment. Snapshots are based on a fork of the process and take advantage of the copy-on-write mechanism of the kernel to avoid having to explicitly copy the entire address space. However, the fact that snapshots have to stay in memory make it necessary to discard older ones. Both systems have a runtime overhead lower than ours (2x-4x for Flashback, up to 30% for Jockey), but they do not properly handle multithreaded programs. Nirvana [4] is a deterministic replay system for native programs that properly supports multithreaded programs. Like our own system, it records the results of memory reads to account for

\textsuperscript{14}Although there are no formal publications about this open-source tool, Amber/Chronicle is a serious endeavor that has been successfully used to debug the Firefox web browser. Information can be found on this page: \url{http://weblogs.mozillazine.org/roc/archives/2006/12/more_about_ambe.html}. 
CHAPTER 4. SUMMARIZED TRACE INDEXING AND QUERYING

scheduling-induced non determinism. Its runtime overhead is between 5x and 17x, which is slightly better than what we achieve with STIQ.

Retrace [58] is a deterministic replay system for uniprocessor VMWare virtual machines. It has an extremely low runtime overhead (around 5%) and produces very compact traces. Such a low runtime overhead is possible because the recorded system is the entire (virtual) machine, and therefore the amount of interaction with the environment is limited to mostly IO operations; in particular, thread scheduling and the associated non-determinism on memory locations need not be captured, as the scheduling itself is a deterministic part of the recorded system.

4.8 Conclusion

This chapter represents the culmination of our work on the performance of omniscient debugger. STIQ is a scalable omniscient debugging approach based on summarized execution trace indexing and querying that favorably compares with previous approaches on the three essential levels: trace capture overhead, indexing speed and query response time. In particular, it leverages deterministic replay for a lower runtime overhead, and indexes only summarized information about bounded-size execution blocks for fast indexing and querying. Importantly, it guarantees that all queries require only $O(\log n)$ disk accesses and $O(1)$ CPU time; in practice they never reach the one second mark, and take only a dozen milliseconds on average. Such efficient querying is key to providing an interactive debugging experience; we are not aware of any omniscient debugging system that provides such strong guarantees.

We only presented the core queries of omniscient debuggers (stepping, memory inspection and causality links). However our indexing scheme could easily support other useful queries, such as finding the events that occur on a particular source code line, or the history of objects beyond the history of their individual fields (e.g., when objects are passed around as method arguments).

An interesting property of our approach is that the indexing and querying scheme is independent from the technique used for trace capture and replay. The only requirement is that it must be able to obtain (a) memory write and method entry/exit events for index construction (in the present work, these are obtained through an initial replay that generates a semi-exhaustive trace), and (b) exhaustive event lists of arbitrary execution blocks for processing queries (in this work, this is achieved by taking lightweight snapshots that permit to replay such blocks). Although this work provides both the capture and the indexing mechanism, we feel that the capture is still too slow to be really practical. It is our hope that this work will encourage the building of improved capture mechanisms that can be plugged into our indexing system so as to obtain a practical omniscient debugger. It would be particularly interesting to assess how an extremely efficient capture system such as Retrace [58] could be used for this purpose.

Beyond performance, another challenge for making omniscient debuggers practical is their
ability to handle more abstract programming paradigms. The next chapter presents our incursion in the realm of debugging aspect-oriented programs.
Chapter 5

Omniscient debugging of aspect-oriented programs\textsuperscript{1}

Contents

\begin{itemize}
\item[5.1] AOP and debugging \hfill 85
  \begin{itemize}
  \item[5.1.1] AspectJ \hfill 85
  \item[5.1.2] Debug model for AOP \hfill 85
  \end{itemize}
\item[5.2] State of the practice \hfill 86
  \begin{itemize}
  \item[5.2.1] The AJDT debugger \hfill 87
  \item[5.2.2] AOP debugging with TOD \hfill 87
  \end{itemize}
\item[5.3] Omniscient debugging for AOP \hfill 88
  \begin{itemize}
  \item[5.3.1] Improving intimacy of residue evaluation \hfill 89
  \item[5.3.2] Identification of AOP activities \hfill 89
  \item[5.3.3] Obliviousness/intimacy trade-off \hfill 90
  \item[5.3.4] Bird’s eye views \hfill 90
  \end{itemize}
\item[5.4] Related work \hfill 91
\item[5.5] Conclusion \hfill 92
\end{itemize}

\textsuperscript{1}The contents of this chapter are based on our SAC 2008 paper “Extending Omniscient Debugging to Support Aspect-Oriented Programming” [42].
When you catch bugs early, you also get fewer compound bugs. Compound bugs are two separate bugs that interact: you trip going downstairs, and when you reach for the handrail it comes off in your hand.

Paul Graham

While omniscient debugging seems to be slowly attracting more attention in industrial settings, new challenges are appearing with the emergence of new programming languages and paradigms. In particular, because it adds more possible loci for late binding, Aspect-Oriented Programming (AOP) [21] admittedly makes it more difficult for programmers to mentally reconstruct the execution flow of a program. Appropriate development tools, in particular debuggers, are required to support AOP. This chapter shows how we extended TOD (Chapter 3) to support AOP.

AOP provides means for proper modularization of crosscutting concerns, whose implementation would otherwise be scattered across several modules [21]. In most AOP approaches, modularization is achieved by defining aspects that affect the structure or the behavior of a base program that is mostly unaware, at least syntactically, of these aspects. One drawback of this approach is that it makes the comprehension of AOP-based systems more difficult: understanding a piece of code might require the understanding of the whole system, or at least of its aspects [52].

The time-consuming task of debugging has a significant impact on the cost of software [56]. Most of the time is usually spent locating the cause of the bug, often using a tedious trial-and-error approach, while actually fixing the bug can be trivial [19]. A strategy frequently used by programmers is to mentally simulate the execution of the program [19]. Thus the complexity of debugging increases with the level of abstraction of the programming paradigm because the correspondence between source code and runtime behavior becomes less direct. For instance, in object-oriented programming, one cannot always know by looking at the source code which method will be evaluated as a result of a method call, because of the dynamic dispatch mechanism. This is even more true for AOP, where the behavior of a given piece of code can be altered to an arbitrary degree by an aspect in another source code file. Section 5.1 details these difficulties.

This chapter shows how omniscient debugging can be extended so as to embrace aspect-oriented programming. We first present an analysis of the current situation for debugging aspect-oriented programs, focusing on the case of AspectJ, since it is the best supported
language to date (Sect. 5.2). We then describe a set of extensions to TOD, our prototype omniscient debugger (Chapter 3), that greatly enhance the understanding of the dynamics, and therefore the task of debugging, of aspect-oriented programs (Sect. 5.3).

5.1 AOP and debugging

This section briefly introduces the AspectJ [28] language, and then explains the issues brought by AOP to software understanding and debugging.

5.1.1 AspectJ

AspectJ [28] extends the Java language with a new unit called *aspect* that permits to implement crosscutting concerns modularly. AspectJ supports two kinds of crosscutting: *dynamic crosscutting* makes it possible to define additional behavior to be executed when certain conditions occur in the base program; *static crosscutting* makes it possible to modify the static structure of a program, e.g. adding new methods or modifying the class hierarchy.

In AspectJ a *join point* represents a well-defined point in the execution of a program, such as method call, field write, exception handler execution, etc. The (static) location of a join point in the source code is called a join point *shadow*. A join point may also specify a dynamically-evaluated condition, called the *residue*, to determine at runtime whether a join point shadow actually is the expected join point.

Join points of interest are grouped into a *pointcut* in order to specify the places where an aspect actually affects a base application. Pointcuts are specified using several primitive *pointcut designators* (PCDs) which can be combined using the standard logical operators. For example, the aspect in Figure 5.1 defines a pointcut named *cond* that combines several primitive PCDs in order to select calls to method *foo* of class *A* when the actual type of the receiver is *B* (a subclass of *A*) and the parameter is less than 3; additionally it exposes the parameter as context information.

Finally, the crosscutting behavior that should be applied upon occurrences of join points matched by a given pointcut definition is called an *advice*, which is a method-like construction that defines additional behavior to execute at certain join points. Advices must be explicitly bound to pointcuts. In the example of Figure 5.1, an advice is called *before* the pointcut *cond* matches and prints an informative message using exposed context information.

5.1.2 Debug model for AOP

The debug model for AOP developed in [18] is helpful in understanding the difficulties of debugging AO programs. Of particular interest are a classification of AOP *activities*, a comprehensive *fault model* and a definition of *debugging obliviousness* and *debugging intimacy*. 
public aspect Foo {
    pointcut cond(int x): call(* A.foo(int))
        && target(B) && args(x) && if(x<3);

    before(int x): cond(x) { System.out.println("Bingo: "+x); }
}

Figure 5.1: Example aspect in AspectJ.

Classification of AOP activities The execution of an AO program consists of different activities. Beyond base code execution, there is one explicit activity, namely advice execution, and several implicit activities used to coordinate advices with base code [17, 18]: dynamic aspect instantiation and selection, i.e. determining which aspect instance should apply; residue evaluation; aspect activation, i.e. gathering context information and transferring control to the advice; and bookkeeping for specific features, such as maintaining a thread-local stack for cflow pointcuts. A much more detailed decomposition is given in [17].

Fault model The existence of implicit AOP activities increases the difficulty of debugging AO programs because many instructions are executed at runtime that are not explicit in the source code. The following are examples of AOP-specific fault types [18]: (1) incorrect pointcut descriptor, when a pointcut declaration does not have the intended effect, (2) incorrect aspect composition, when several aspects match the same join point and are not executed in the expected order, (3) adverse changes on base program, when an aspect alters the functionality of the base program in such a way that it ceases to work properly, and (4) incorrect context exposure, when the context is not exposed as intended to an advice.

Debugging obliviousness and intimacy In the context of debugging AO programs, debugging obliviousness is the capacity to ignore all AOP-related activities. Conversely, debugging intimacy is the capacity to observe all activities in their full details [18]. Instead of considering obliviousness and intimacy as exclusive alternatives, our proposal describes how to support a range of options between these two extremes (Section 5.3). This makes it possible for the programmer to choose the appropriate level of intimacy depending on the debugging task at hand.

5.2 State of the practice

This section analyzes tools that can be currently used to debug AspectJ programs, and emphasizes on their limitations with respect to the debug model of Section 5.1.2.
5.2.1 The AJDT debugger

A major asset for the use of AspectJ is the AspectJ Development Tools (AJDT), a set of plugins for the Eclipse IDE [10]. It provides various features that help understanding AO programs, in particular: (a) markers that show join point shadows in the base source code and permit to jump from the shadow to the corresponding pointcut definition (and vice versa), and (b) high-level visualizations that show the scattering of join point shadows in whole packages.

Debugging in AJDT is rather ad-hoc, neither fully oblivious nor fully intimate. When the execution of the debugged program is halted at a breakpoint, the programmer can use the traditional step-over and step-into operations.

Step-into provides a certain level of intimacy. Invoking step-into when an advice is about to be executed actually steps into the code of the advice, but with an extra step where the debugger shows the first line of the file that defines the aspect: this actually corresponds to the dynamic aspect selection AOP activity, but this is not explicit in the debugger (and rather misleading).

The level of intimacy in the evaluation of residues is however not consistent. In the pointcut definition of Figure 5.1 the residue comprises two conditions: (1) the actual type of the target must be B (a subclass of A), and (2) the argument x must be less than 3. When invoking step-into in a.foo(2), only the if condition is stepped into: the condition on the target is silently stepped over. This is due to the semantics of the step-into operation of Java: the execution halts when it reaches another source code line, but the instanceof operation implementing the type test is considered to be on the same line than the rest of the AOP activities.\footnote{There is a step-into mode that halts at each bytecode (STEP_MIN instead of STEP_LINE), but it is not used by Eclipse, nor by any other debugger we know of. It would probably require a disassembled view of the bytecode of the debugged activity to be useful.}

Step-over, on the other hand, provides full obliviousness: all AOP activities on the current line are ignored, including advice execution. But there is no way to step into a method call on the current line while ignoring AOP activities.

5.2.2 AOP debugging with TOD

The original TOD as presented in Chapter 3 is able to debug AspectJ programs but has no special provisions for handling the obliviousness/intimacy trade-off.

Figure 5.2 shows the events that are registered when executing a piece of code affected by an aspect. Lines 2 to 6 correspond to AOP-specific activities: lines 2, 3 and 6 correspond to aspect activation, line 4 corresponds to residue evaluation and line 5 to aspect selection.

Compared to AJDT, TOD provides a bit more debugging intimacy: lines 2 and 3, that are a part of the context exposure mechanism, are not shown in AJDT; line 5 explicitly shows a call to aspectOf, while AJDT showed the first line of the aspect source file. For the residue
These events correspond to the execution of \texttt{a.foo(2)} with the aspect of Fig. 5.1. The $2$ and $3$ represent synthetic local variables added by the AspectJ compiler.

Figure 5.2: List of events corresponding to a conditional pointcut.

(a) debugging obliviousness is obtained by collapsing the events that correspond to AOP-related activities into a single line. Full intimacy is shown in (d), and two intermediate levels of intimacy are shown in (b) and (c). Each icon represents a kind of AOP activity.

Figure 5.3: The obliviousness/intimacy trade-off in an event list.

evaluation, both TOD and AJDT behave in the same inconsistent way: the \texttt{if()} condition is shown but the test on the actual type of the target is not. In the case of TOD the reason is that only method calls and actions that change the state of the program are registered; the \texttt{instanceof} operation implementing the type test is not registered.

5.3 Omniscient debugging for AOP

In this section we describe several extensions to omniscient debugging that facilitate the debugging of AO programs. These extensions have been integrated in TOD.
5.3.1 Improving intimacy of residue evaluation

Section 5.2 showed that the level of intimacy for the evaluation of join point residues is not consistent in existing debugging solutions. In the case of TOD, only method calls and actions that modify the state of the program are registered; tests such as `instanceof` are not. While this permits to reduce the number of events to register, this also hides useful information. In order to support full debugging intimacy, an omniscient debugger must register the outcome of all the tests that occur during that activity. For an omniscient debugger that captures events through a customized runtime, the runtime has to be able to emit events indicating the outcome of tests. For an omniscient debugger that relies on code instrumentation to generate events, such as TOD, this requires the insertion of additional instrumentation to the program: event generation code is now inserted at locations where an edge of the control dependence graph of the program is traversed [61]. Figures 5.3c and 5.3d show how full intimacy in residue evaluation adds details to the event list of Figure 5.2.

5.3.2 Identification of AOP activities

In aspect language implementations, instructions that represent AOP activities are woven with base code instructions so as to provide the semantics of aspects. This weaving process can be invasive (the base program is modified to include those instructions, either at the source level or at the binary level), or noninvasive (AOP-specific instructions are performed by an AOP-aware runtime) [18]. In both cases it is possible to determine at runtime the AOP activity corresponding to each executed instruction, as well as the pointcut corresponding to each AOP-specific instruction. The case of noninvasive weaving is the easiest: the runtime can be extended to provide the required information, as it is fully responsible for implementing AOP semantics.

Because the current implementations of AspectJ use invasive weaving, we use a tagging scheme [17] to identify AOP activities. Each instruction is given, at aspect weaving time, a tag that indicates its activity, as well as the identifier of the corresponding pointcut (for AOP-specific instructions). Additionally, runtime propagation rules are defined to ensure, for example, that code from the base program is considered as base code when called directly from the base program, but as advice code when called from an advice. In this way it is always possible to determine the activity to which the currently-executing instruction belongs.

Once the activity of each instruction is identified, it is possible to tag all the events registered by the omniscient debugger with the corresponding activity and pointcut identifier where applicable. This enables debugging obliviousness, as events tagged as AOP-specific can be concealed (Fig. 5.3a). Moreover this also improves intimacy, as the role of each event can be made explicit (Fig. 5.3d).

---

³We implemented the tagging mechanism as an extension of the abc AspectJ compiler.
CHAPTER 5. DEBUGGING OF ASPECT-ORIENTED PROGRAMS

5.3.3 Obliviousness/intimacy trade-off

In Sections 5.3.1 and 5.3.2 we showed how full intimacy and full obliviousness can be achieved. However an AOP debugger should permit to choose the appropriate level of detail for the task at hand. Figure 5.3 shows increasing level of detail for the same sequence of events, in our extension of TOD.

If obliviousness is required, sequences of AOP-related events between two base code events are collapsed into one line\(^4\) (Fig. 5.3a). This line provides a visual summary of the activities of the collapsed sequence: the programmer therefore knows at a glance that, before the call to \texttt{foo} a conditional pointcut was evaluated, the advice was indeed called and some context was exposed.

If intimacy is required, the sequence can be gradually expanded to show the details of the AOP activities (Fig. 5.3b to 5.3d). First only the event that corresponds to the advice call is shown; this permits to easily step into the execution trace of the advice in the case where no fault in AOP-specific activities is suspected. Otherwise the details of the implicit AOP activities can be revealed: for instance if the advice was erroneously executed, residue evaluation events can be shown (Fig. 5.3c). If maximum detail is needed, it is possible to show the complete sequence of events (Fig. 5.3d). The icons in front of each event help understanding the AOP activities.

5.3.4 Bird’s eye views

As an omniscient debugger registers the whole history of the debugged program, it can provide certain summarized views on its execution. These views are useful to abstract away from the

---

\(^4\)A completely oblivious debugger would not even show that line, but it seems rather counterproductive.
History of the pointcut of Fig. 5.1. The residue column shows the tests that passed and those that did not. Clicking on the operation shows the corresponding event in its context.

Figure 5.5: Pointcut history view (mock-up).

details of execution events. For example TOD provides murals that show the evolution of the density over time of events that meet certain criteria (see Sect. 3.1.5). With the event tagging scheme mentioned in Sect. 5.3.2, our extension of TOD provides aspect murals that show the activity of an aspect during the execution of the program. Such temporal views of AOP activity are the dynamic counterpart of the static crosscutting views of AJDT (Sect. 5.2.1). They are particularly helpful in understanding the interplay between aspects and base code, as well as between different aspects. Figure 5.4 shows an aspect mural where two aspects are selected and their activity is represented by two distinct colors. It is also possible to show the activity of aspects on methods of a single class, or of a single instance.

Another interesting view that could be constructed would show the execution history of the join point shadows of a particular pointcut. Such pointcut history views would be particularly useful for pointcuts with a residue, as they show which occurrences of join points matched and which ones did not. It would even be possible to examine in details the individual residue conditions, facilitating the resolution of the incorrect pointcut descriptor AOP fault (Sect. 5.1.2). On Figure 5.5, on the second line the condition on the type of the target was not verified, shortcircuiting further evaluations; on the last line the condition on the argument was not verified. Note that while it is easy to determine the number of tests that passed during the residue evaluation, determining the precise pointcut condition corresponding to each test requires a non-trivial static analysis (not tackled yet).

5.4 Related work

The comprehensive debug model for AOP presented in [18] has been discussed at length in this chapter. The model also presents topics that are orthogonal to omniscient debugging such as the ability to use edit-and-continue debugging, or to introduce new aspects at runtime. It introduces Wicca, a dynamic AOP system for the C# language, which excels at debugging intimacy but lacks debugging obliviousness.

Static analysis is an important tool for program understanding. The whole execution traces presented in [61] offer a compact yet comprehensive representation of program activity and permit to perform advanced semantic queries, such as dynamic slices, that identify the parts of a program that directly or indirectly affect the value of a given variable at a given
program point. A solution for computing slices of AO programs is presented in [62].

Using omniscient debugging for AOP was first explored in [46]. Their approach is based on slicing and therefore focuses on how to delimit regions of interest in the source code for a particular debugging task. Our approach in contrast focuses more on letting the programmer explore the details of the activities related to aspects, at the level of runtime events; abstraction on the execution trace is provided by summary views (aspect murals and pointcut history). Combining both approaches seems very promising.

5.5 Conclusion

In order to ease the debugging of aspect-oriented programs, we propose a number of extensions to omniscient debuggers: collapsable list of events to hide unwanted details of AOP activities, activity icons to help understand AOP activities, more thorough event model for observing the details of residue evaluation, and high-level views such as aspect murals to help understand the interplay between aspects and base code. These extensions, implemented on TOD, are designed to address issues observed in both traditional breakpoint-based debuggers and current omniscient debuggers when debugging programs with aspects. Improving the debugging experience is crucial for the widespread acceptance of AOP in industry.

There are still interesting issues that we have not addressed. A systematic evaluation of the attainable intimacy for various AspectJ features not discussed here, such as around advice and control-flow pointcuts, would be highly useful. Addressing the obliviousness/intimacy trade-off for intertype declarations would also be of great value, considering the importance of structural aspect mechanisms in practice [2].
Chapter 6

Conclusion and perspectives

This chapter summarizes our contributions to making omniscient debugging practical and discusses the associated perspectives:

- Efficient processes for implementing omniscient debugging engines. Our first-generation process (Chapter 3) is highly scalable, but requires a database cluster and suffers from long query response times in some situations. Our second-generation process (Chapter 4) resolves these issues by using compact indexing and providing strong guarantees on query response times, thus ensuring the interactivity of the debugging experience.

- A working prototype (Chapter 3) integrated into the widely-used Eclipse development environment. This prototype provides many features that help navigate in execution traces and supports partial traces to reduce the size of execution traces.

- An extension of omniscient debugging to support Aspect-Oriented Programming (AOP). This extension shows that omniscient debugging can be adapted to new paradigms that have different cognitive characteristics than classical object-oriented languages: in AOP, the chasm between the source code and what is actually executed can be wide, and thus makes the task of understanding such programs more difficult.

6.1 Indexing and querying processes

Our first omniscient debugging engine (Chapter 3) adopts a rather brute-force approach in which all the events that occur in the debugged program are captured to produce an execution trace; each attribute of each event is then separately indexed so that arbitrary queries can be executed. Although queries that use a single index can be executed very quickly, those that need to combine several indexes can take an arbitrarily long time to execute, as the indexes are linearly scanned to find the events that match in all the indexes. Moreover, this exhaustive indexing scheme requires a lot of space and processing power; fortunately, it is
easy to parallelize, and in our tests it displays a good scalability, as the throughput increases linearly with the number of database nodes, up to a certain limit.

Our second engine (Chapter 4) adopts a subtler approach. On the capture side, only non-deterministic events are recorded, thus reducing the capture overhead; the complete trace can be obtained through deterministic replay. For indexing, we divide the execution trace into individually replayable execution blocks, and we index summarized information for each block. This yields a dramatic reduction in the size of the indexes, and offers strong guarantees on query response times, thus ensuring an interactive debugging experience.

### 6.2 Prototype for Java and Eclipse

TOD (Chapter 3), our prototype omniscient debugger for Java based on the first engine described above, provides many features that help navigate in execution traces: bidirectional stepping, object and stack frame reconstitution, direct traversal of causal links, event murals, event and object bookmarks, navigation history, and custom formatters. It also supports partial traces to reduce the size of execution traces.

TOD is deeply integrated into Eclipse: it automatically selects the source code line corresponding to the currently inspected event, and provides contextual actions at the source code level, such as finding all the events that correspond to a specific line.

The availability of an omniscient debugger for a mainstream development environment makes it easier to assess its usefulness in real-world situations. We have been able to use it extensively ourselves to debug newer versions of the TOD database itself, and to understand issues that arose in our use of highly complex software such as the abc AspectJ compiler.

### 6.3 Extension for debugging aspect-oriented programs

We presented an extension of TOD to support the debugging of AspectJ programs (Chapter 5). Although TOD natively supports AspectJ programs (as AspectJ is compiled to standard Java bytecode), debugging such programs without special assistance is difficult because aspect-specific logic is woven into the base code by the AspectJ compiler, thus “polluting” the execution trace. Our extension lets the user select the desired level of detail, going from full intimacy (i.e. exposing all the aspect-specific events) to full obliviousness (i.e. hiding all aspect-specific events), with several steps in-between. It also offers high-level overviews of aspect activity by showing aspect-related events in different colors in the event murals.

---

1[^abc]

[^abc]: http://abc.comlab.ox.ac.uk/introduction
6.4 Perspectives

While we feel that this work is an important step towards practical omniscient debugging, there are still unresolved issues, as discussed below.

**Trace capture overhead** The capture of execution traces still incurs an important runtime overhead on the debugged program. While the snapshot and replay mechanism presented in Chapter 4 represents an important improvement over previous approaches, an overhead of 30x is still too high in many situations. This issue can be mitigated using partial traces, but this is not always a satisfactory solution, as it is possible for the cause of a bug to hide in out-of-scope code. We hope that the runtime overhead can be improved using techniques such as virtual-machine-based replay [58] that have an extremely low overhead.

**Implementing omniscient debuggers for Java** In both TOD and STIQ, we implemented trace capture using bytecode instrumentation. Although this solution seemed the easiest at the beginning, it became more complicated as we captured traces of more complex programs, as we frequently had to deal with Sun HotSpot JVM implementation issues that are not covered by the JVM specifications. In retrospect, we think that it might even have been easier to perform trace capture by using modifications to the HotSpot JVM. This would also probably reduce the runtime overhead, are some operations that are quick at the machine level cannot be expressed in (verifiable) Java bytecode.

**Integrating STIQ in TOD** Although TOD, our prototype omniscient debugger for Java, is currently usable in practice, it is still based on our first-generation omniscient debugging engine. We unfortunately did not have the resources to migrate our prototype to STIQ, our second generation engine, which currently lacks a user interface. It would be extremely interesting to experience the improvement in query response time in real-world debugging situations.

**Usefulness of omniscient debugging** The usefulness of omniscient debugging in resolving real-world bugs has not been formally studied, although we do have anecdotal evidence through our own use of our prototype. We conjecture that one of the reasons such a study was never conducted is that there is no efficient enough omniscient debugger available to operate on. We now argue that our work on guaranteeing low query response time makes it possible to conduct such a study: although the runtime overhead is still high, the execution trace can be captured in advance and the participants will still be able to work with a truly interactive debugger.
Bibliography


BIBLIOGRAPHY

PLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA ’04, pages 331–344, New York, NY, USA, 2004. ACM.


