Summarized Trace Indexing and Querying for Scalable Back-in-Time Debugging

Guillaume Pothier and Éric Tanter

PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile – Chile
www.pleiad.cl

Abstract. Back-in-time debuggers offer an interactive exploration interface to execution traces. However, maintaining a good level of interactivity with large execution traces is challenging. Current approaches either maintain execution traces in memory, which limits scalability, or perform exhaustive on-disk indexing, which is not efficient enough.

We present a novel scalable disk-based approach that supports efficient capture, indexing, and interactive navigation of arbitrarily large execution traces. In particular, our approach provides strong guarantees in terms of query processing time, ensuring an interactive debugging experience. 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. Benchmarks on a prototype for Java show that the system is fast in practice, and outperforms existing back-in-time debuggers.

1 Introduction

Execution traces are a valuable aid in program understanding and debugging. Most research is centered on the capture of execution traces for offline automatic analysis [7,17,20]. However, there has been a recent surge of interest in *interactive* trace analysis through back-in-time, or omniscient, debuggers [5,8,9,10,11,12,13]. Such debuggers allow forward and backward stepping and can directly answer questions such as "why does variable x have value y at this point in time?", thus greatly facilitating the analysis of causality relationships in programs.

The navigation operations provided by back-in-time debuggers are based on a small set of conceptually very simple queries. To achieve *interactive* navigation, those queries must execute extremely quickly, regardless of the size of the execution trace. It is therefore necessary to build and use indexes, otherwise queries would require scanning arbitrarily large portions of the execution trace. Interactive navigation in large execution traces requires an efficient indexing scheme tailored to the core set of back-in-time debugging queries:

M. Mezini (Ed.): ECOOP 2011, LNCS 6813, pp. 558–582, 2011.

Bidirectional stepping. These queries are similar to the usual stepping operations provided by traditional debuggers, with the added benefit of being able to perform them both *forward* and *backward* in time. *Step into* operations are very simple, as they consist in navigating to the next or previous event in the trace. *Step over* and *step out* operations, on the other hand, are more complex, as they require to skip all the events that occurred inside a method call. As the number of events to skip is potentially huge, it is not efficient to just perform a linear scan of the trace.

Memory inspection. Back-in-time debuggers support the inspection of the values of memory locations (such as object fields and local variables) at any point in time. To retrieve the value of a location at a particular point in time, the query to process consists in determining the last write operation to that location before the currently observed point. Again, as the last write can have happened much before the current observation point, it is not efficient to linearly scan the trace.

Causality links. Back-in-time debuggers support navigating via causality links, *e.g.* by instantly jumping to the point in time where a memory location was assigned its currently observed value. The corresponding query is actually the same as the one used to perform memory inspection: the last write operation to the location gives both the written value and the point in time at which it was written.

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 [8,11]. To handle larger traces without losing information, a disk-based solution is mandatory [13], but this typically reduces the efficiency of the system. In addition, most back-in-time debuggers rely on directly capturing *exhaustive* executions traces [5,8,11,13]. 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* [1,3,15,16,19]. It consists in capturing only a *minimal trace* of non-deterministic events during the initial execution of a program. The minimal trace can then be deterministically replayed to obtain the exhaustive trace without affecting the execution of the debugged program. This is much cheaper than capturing an exhaustive trace, and thus greatly reduces the probe effect. 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 [15,16]. 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 paper 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 back-in-time debugging engine called STIQ, for Summarized Trace Indexing and Querying (Section 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.
- We present an efficient deterministic replay system for Java (Section 3). This system supports partial deterministic replay through *lightweight snapshots* that are both fast to obtain and easy to serialize. We explain how these 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 [14] for efficient control flow indexing (Section 4), as well as the principle of temporal locality of memory accesses to reduce the amount of information to index (Section 5).
- We demonstrate through benchmarks that the approach enables a highly interactive back-in-time debugging experience (Section 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 back-in-time debugging system that provides either such efficient index building, or such strong guarantees in query response time.

Finally, Section 7 discusses related work and Section 8 concludes.

2 Summarized Trace Indexing and Querying

Interactive back-in-time 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 paper, 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 in $\mathcal{O}(\log n)$ time, and the relevant execution block is then scanned to find the exact answer in $\mathcal{O}(1)$ time (as the size of blocks is bounded).

This section gives an overview of the complete process, whose steps are detailed in subsequent sections, and presents the overall system architecture.

2.1 Process Overview

The STIQ process consists of four phases: trace capture, initial replay, summarized indexing, and querying.

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 so as to resolve race conditions.

Initial replay. Although the minimal trace produced by the capture phase is sufficient to deterministically replay the debugged program, it is not directly useful for our indexing process: it contains memory read events, whereas the memory writes are those that must be indexed. 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*.

Summarized indexing. 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 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) [14], 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

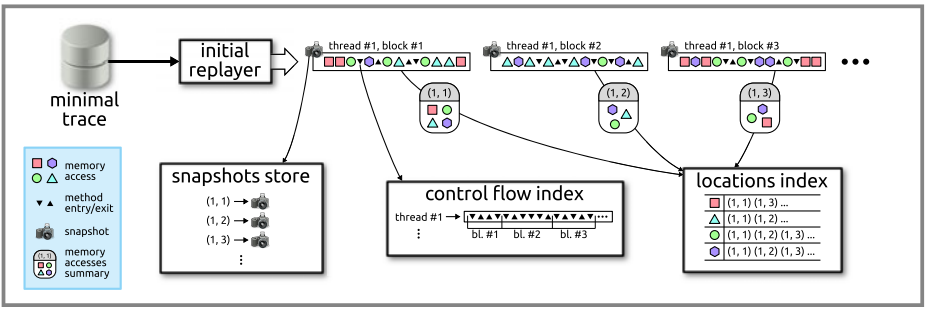


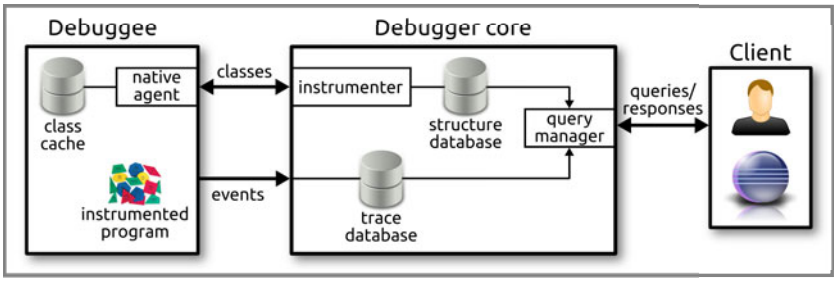
Fig. 1. Summarized indexing process

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 $\mathcal{O}(\log n)$ disk accesses and $\mathcal{O}(1)$ 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 $\mathcal{O}(\log n)$ disk accesses) and the block is replayed and then scanned to find the exact event of interest in$\mathcal{O}(1)$ 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. 6).

2.2 System Architecture

Our system uses an out-of-process database to store and index the execution trace. The overall architecture is depicted below:



It consists of three elements:

1. The *debuggee*, which is the Java VM that executes the program to debug. It contains a special native agent that intercepts class loading so that classes

are instrumented prior to execution (either by sending their bytecode to an out-of-process instrumenter, or by looking them up in a class cache if they have already been instrumented in a previous session). When executed, instrumented code emits events that are sent to the debugger core.
2. The *debugger core*, which consists in (a) an instrumenter that receives the original classes from the debuggee and inserts the event emission code before sending the modified classes back to the debuggee, (b) a structure database that records information about the classes, methods and fields of the debugged program, (c) a trace database that stores and indexes the events emitted by the debugged program, and (d) a query manager that uses the database indexes to quickly answer queries.
3. The *client*, which is the user interface of the debugger. It presents the user with views over the debugging session and controls to interactively navigate in the execution trace using back-in-time debugging metaphors: stepping backward and forward, navigating runtime data dependencies, etc.

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 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 3.2 discusses the replayer, and Section 3.3 details how and when snapshots are taken.

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, and thus must be recorded so as to enable deterministic replay. These operations are:

- *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.[1]

[1] In the case of Java, only the contents of heap memory is non-deterministic, as the virtual machine does not allow concurrent access to stack memory.

Dealing with memory non-determinism. 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 [3]. This is of limited usefulness with multicore architectures, however, as in this case concurrency occurs at the hardware level. Another strategy, which we use in our system, consists in recording the value obtained by every memory read [1].

Avoiding heap simulation. Although capturing memory reads is enough to allow a fully accurate replay, it still requires to simulate the state of the whole 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. The simulated heap would occupy as much memory as the heap of the original program.

Fortunately, it is possible to completely avoid simulating the heap by recording the outcome of the above control-flow-altering operations, even though they are deterministic. This has a very small impact on the trace capture overhead, but drastically reduces the memory requirements of the system, thus freeing valuable memory for the indexing process. Moreover, as the heap is not needed anymore for replay, the only information needed to start replaying at arbitrary execution block boundaries can 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 very little space.

Identification of memory locations. The reconstitution of program state at arbitrary points in time requires the indexing of memory locations; it is therefore necessary to be able to uniquely identify each memory location. 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 non-problematic 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. 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 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 [13]. 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.

Unfortunately, because of polymorphism 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. Whenever the execution of an in-scope method starts, the top of the stack is checked to decide if method arguments must be captured; similarly, whenever an out-of-scope method exits, the top of the stack is checked to decide if the return value should be captured.

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 consumes 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.

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 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 methods 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 tradeoff 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.

[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 paper.

It is 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 exist 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[3]. 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 inside 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 Indexing of Control Flow

We now turn to the indexing techniques. This section describes the indexing of control flow, and Section 5 describes the indexing of memory accesses. While *step into* queries simply consist in moving to the next/previous event in the execution trace, efficiently executing *step over* and *step out* queries requires an index: otherwise it would be necessary to linearly scan the execution trace to skip the events that occurred within 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) [14], 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 so that

[3] We 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.

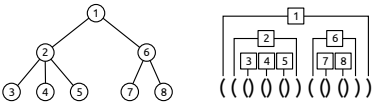


Fig. 2. A tree and its balanced parentheses representation

the block corresponding to a given node can be determined during queries. 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.

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[4] with n nodes, the lower bound is $2n - \Theta(\log n)$ bits [14]. A classical way to represent trees using $2n$ bits is the *balanced parentheses sequence* (see Figure 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 (or closing) 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 RMM Tree [14] 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 $\mathcal{O}(c^2)$ with a data structure using $2n + \mathcal{O}(n/\log^c n)$ 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) [14].

[4] Specifically, ordinal trees, where a node can have any number of ordered children.

Algorithm 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 ← getCFlowTree(t)
3:     p_call ← eventToPosition(t, b, i)
4:     p_ret ← tree.getClose(p_call)
5:     (t_ret, b_ret, i_ret) ← positionToEvent(t, p_ret)
6:     return (t, b_ret, i_ret)                    ▷ By construction t = t_ret
7: end function
```

Algorithm 2. Event to position.

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

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

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 $\left\lfloor \frac{4096}{10} \right\rfloor^3 \cdot 4096 \cdot 8 \simeq 2 \cdot 10^{12}$ bits in its leaves and occupies around $4096 \cdot \sum_{i=0}^{3} \left\lfloor \frac{4096}{10} \right\rfloor^i \simeq 280$GB, thus requiring roughly 2.005 bits per original tree node (slightly more than 1 bit per parenthesis).

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.

Algorithm 3. Position to event.

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

```
1: function PositionToEvent(t, p)
2:     (tree, map) ← getCFlowIndex(t)
3:     b ← map.getBlockId(p)
4:     p_0 ← map.getPos(b)   ▷ p_0 is the position of the RMMTree corresponding to the beginning of block b
5:     block ← getBlock(t, b)
6:     i ← 1
7:     while p_0 < p do
8:         if block[i] is call or return event then
9:             p_0 ← p_0 + 1
10:        end if
11:        i ← i + 1
12:    end while
13:    return (t, b, i)
14: end function
```

To perform a step over operation[5], 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 *findReturn* function (Algorithm 1) is thus the basis of the step over operation.

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 2 and 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.[6] 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.

[5] We describe forward step over; backward step over and step out are similar.

[6] In Algorithm 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. 6).

5 Indexing of Memory Accesses

Two of the essential features of back-in-time 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[7].

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 of the index (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 [13]. 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. 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. 3.1. We explain how stack locations (local variables) are handled in Sect. 5.3.

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

[7] Although 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. Memory location reconstitution.

```
1: function GetLastWrite(loc, (t, b, i))
2:     index ← getLocationIndex(loc)
3:     (b_2, threads) ← index.getAtOrBefore(b)
4:     for t_2 in threads do
5:         block ← getBlock(t_2, b_2)
6:         if b_2 = b and t_2 = 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 (t_2, b_2, k)
14:                break
15:            end if
16:        end for
17:    end for
18: end function
```

individual index for each 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. 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 $\mathcal{O}(\log n)$ disk accesses.
2. Within the location index, search the execution block(s) that occurred at the same time or just before the block b, which 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 $\mathcal{O}(\log n)$ 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. 3.3): the more threads execute concurrently at a given time, the less events there are in the corresponding blocks.[8] The time required to replay those blocks is therefore bounded and does not depend on the size of the trace.

As shown in Sect. 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.

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

[8] Modulo the number of available CPU cores, but this is also a constant.

built. As explained in Sect. 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. 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. 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. 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. 3b) is thus to avoid as much as possible the costly reordering by downstream stages.

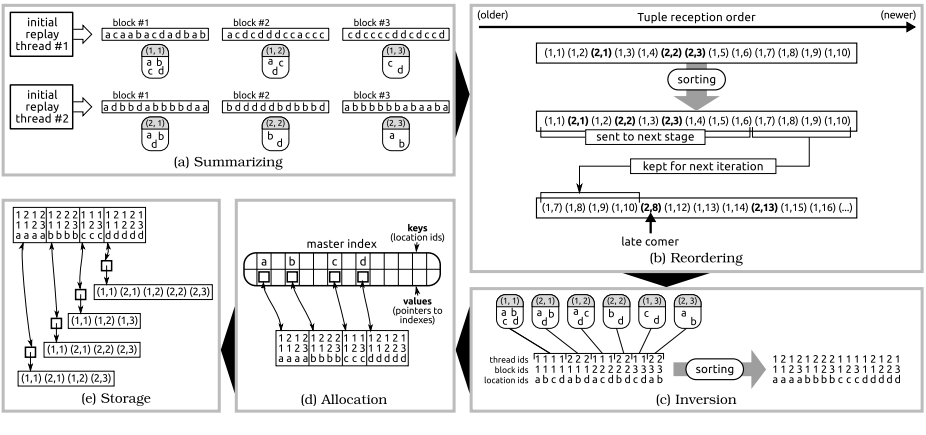


Fig. 3. The five stages of the indexing pipeline

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 late 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. 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[9] to store the difference between successive tuple components, a 4KB singles page contains around 800 indexes on average.

[9] Gamma codes [4] represent an integers x in (roughly) $2 \log_2 x$ bits. Small numbers are thus encoded in very few bits.

- 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. Indexes thus migrate from singles page to n-shared pages to BTrees as more tuples are added.

5.3 Local Variables

Having a separate index for each memory location implies that each location can be uniquely identified. As explained in Section 3.1, our trace capture system assigns a unique id to each 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.

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 [12,13], our previous disk-based back-in-time debugger for Java, which to the best of our knowledge still represents the state of the art up to now. (We compare to other related systems in Section 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, 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 [2], 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.

6.1 Global Results

Table 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[10]. The overhead of STIQ is much lower than that of TOD, as well as that of other back-in-time debuggers: the Omniscient Debugger [8] has an overhead of around 120x, while Chronicle (discussed in Sect. 7) reports a 300x overhead. Also, STIQ has an overhead comparable with other deterministic replay systems like Nirvana [1], which reports a 5x-17x overhead. Nirvana however is only concerned about deterministic replay, not trace indexing.

Table 1. Runtime overhead of trace capture

Workload	t_0	STIQ		TOD	
		t_{STIQ}	o_{STIQ}	t_{TOD}	o_{TOD}
avrora	5.5s	163s	30x	968s	176x
lusearch	7s	69s	10x	157s	22x
burntest	5.2s	65s	12x	427s	82x

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

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 2. Replay and indexing time (and ratio to original execution time)

Workload	STIQ			TOD
	replay	indexing	total	
avrora	95s (17x)	46s (8.4x)	141s (25x)	152min (1664x)
lusearch	19s (2.7x)	13s (1.8x)	32s (4.5x)	16min (138x)
burntest	39s (7.5x)	375s (72x)	414s (80x)	52min (606x)

Table 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

[10] This shows that the published worst-case runtime overhead of 80x for TOD [13] was not actually the worst case.

Table 3. Space usage

Workload	STIQ		TOD	
	trace	index	trace	index
avrora	5GB	0.27GB	35GB	65GB
lusearch	1.1GB	0.16GB	6.2GB	11.3GB
burntest	1.5GB	2.7GB	20GB	39.4GB

Table 4. Average (and maximum) query response time

Workload	STIQ		TOD	
	stepping	memory	stepping	memory
avrora	<1ms (0.24s)	19ms (0.5s)	12ms (6.8s)	41s (96min)
lusearch	<1ms (0.37s)	27ms (0.48s)	5.2ms (1.6s)	1.9s (4min)
burntest	6.9ms (0.65s)	8.6ms (0.17s)	17ms (0.74s)	3.4s (17min)

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.

Table 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 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.

The experimental results clearly show the benefit of our approach. STIQ queries are guaranteed to take $\mathcal{O}(\log n)$ disk accesses and $\mathcal{O}(1)$ 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, completely ruining the interactivity of the debugging session[11].

[11] Note that 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.

Table 5. Cost of capture features as percentage of total capture time

Workload	object ids map	field reads
avrora	9.5%	66%
lusearch	17%	53%
burntest	41%	47%

Table 6. Size of the different indexes as percentage of total index size

Workload	control flow	memory locs	snapshots	strings
avrora	56%	28%	14%	0.6%
lusearch	14%	71%	11%	4%
burntest	1.3%	97%	0.8%	0.7%

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.

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 5 shows the cost of two important features used at capture time. As mentioned in Sect. 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 `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 6 show how the index size is distributed among the different indexes[12]. The distribution varies widely from a workload to another, but it is worthwhile to note that our lightweight snapshots use comparatively very little space.

7 Related Work

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

Back-in-time debugging. TOD [12,13] is our previous attempt at a scalable disk-based back-in-time debugger for Java. It uses a specialized distributed

[12] 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 paper.

database to speed up indexing and querying. It is based on exhaustive trace capture and exhaustive indexing of events. As a consequence, its runtime overhead is higher than STIQ (up to 176x vs. up to 30x), and it is very resource hungry (traces are up to 13x larger than STIQ, and indexes up to 177x larger). Moreover, many queries in TOD involve a conjunction on several indexes, requiring a linear scan that can take a long time in some cases (more than a minute). In contrast, our system guarantees $\mathcal{O}(\log n)$ disk accesses and $\mathcal{O}(1)$ CPU time for all queries, in practice not exceeding a few hundred milliseconds.

Amber/Chronicle[13] by Robert O'Callahan is a back-in-time debugger for native Linux programs that is designed to deal with large execution traces. As TOD, 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 *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 that what we achieve with STIQ.

The Omniscient Debugger [8] and Unstuck [5] are tools for Java and Smalltalk respectively that store 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 *et al.* [11] 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.

The Whyline [6] is a debugging system for Java that provides richer queries than most back-in-time debuggers: it 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 the internal state of the program (memory locations, control flow), but also with its textual and graphical output, down to individual pixels. Although the Whyline can analyze relatively large execution traces (*e.g.* 35 million events), its scalability is limited by the fact that the analysis is performed in memory. ZStep [9] is an early back-in-time debugger for Lisp that does not claim great scalability, but instead explores the user interaction aspect of back-in-time debugging. It can also relate graphical output to the event that produced it.

Deterministic replay. Flashback [16] and Jockey [15] are deterministic replay systems for native Linux programs. Flashback relies on a modified kernel while

[13] 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: `http://weblogs.mozillazine.org/roc/archives/2006/12/more_about_ambe.html`.

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

DejaVu [3] is a deterministic replay system for Java based on modifications of the JVM. It supports multithreaded programs and 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.

Retrace [19] 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.

Capture and analysis of execution traces. Capture of execution trace for automatic offline analysis is a well studied topic. Zhang *et al.* [20] present several lossless compression techniques used to record whole execution traces of native programs. These compression algorithms support direct navigation in the compressed traces. Tallam *et al.* [17] show that it is possible to extend control flow traces to indirectly capture runtime data dependencies. Xin *et al.* [18] 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 useful to correlate several executions of the same program. Using the above techniques, relatively complex queries (*e.g.* calculating dynamic slices or matching instruction flows in different versions of the same program) can be executed in seconds or minutes instead of the hours or days it would take using a naive approach. In contrast, with our system, simple queries specific to the typical tasks of back-in-time debugging can be executed in at most a few hundred milliseconds instead of the seconds or minutes it would take without indexing.

8 Conclusion

This paper presented STIQ, a scalable back-in-time debugging approach based on summarized execution trace indexing and querying that favorably compares

with previous approaches on 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 only require $\mathcal{O}(\log n)$ disk accesses and $\mathcal{O}(1)$ 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 back-in-time debugging system that provides such strong guarantees.

In this paper we only presented the core queries of back-in-time 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 this paper 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 paper 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 back-in-time debugger. It would be particularly interesting to assess how an extremely efficient capture system such as Retrace [19] could be used for this purpose.

References

1. Bhansali, S., Chen, W.-K., de Jong, S., Edwards, A., Murray, R., Drinić, M., Mihočka, D., Chau, J.: Framework for instruction-level tracing and analysis of program executions. In: VEE 2006: Proceedings of the second international conference on Virtual execution environments, pp. 154–163. ACM Press, New York (2006)
2. Blackburn, S.M., Garner, R., Hoffman, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo benchmarks: Java benchmarking development and analysis. In: OOPSLA 2006: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications, pp. 169–190. ACM Press, New York (2006)
3. Choi, J.-D., Srinivasan, H.: Deterministic replay of Java multithreaded applications. In: SPDT 1998: Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools, pp. 48–59. ACM Press, New York (1988)
4. Elias, P.: Universal codeword sets and representations of the integers. IEEE Transactions on Information Theory 21(2), 194–203 (1975)

5. Hofer, C., Denker, M., Ducasse, S.: Design and implementation of a backward-in-time debugger. In: Proceedings of NODE 2006. Lecture Notes in Informatics, vol. P-88, pp. 17–32. Gesellschaft für Informatik, GI (2006)
6. Ko, A.J., Myers, B.A.: Debugging reinvented: Asking and answering why and why not questions about program behavior. In: ICSE 2008: Proceedings of the International Conference on Software Engineering, pp. 301–310 (2008)
7. Larus, J.R.: Whole program paths. In: PLDI 1999: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation, pp. 259–269. ACM Press, New York (1999)
8. Lewis, B.: Debugging backwards in time. In: Ronsse, M., De Bosschere, K. (eds.) Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003), Ghent, Belgium, vol. cs.SE/0310016 (2003)
9. Lieberman, H., Fry, C.: ZStep 95: A reversible, animated source code stepper. In: Stasko, J., Domingue, J., Brown, M.H., Price, B.A. (eds.) Software Visualization — Programming as a Multimedia Experience, pp. 277–292. The MIT Press, Cambridge (1998)
10. Lienhard, A., Fierz, J., Nierstrasz, O.: Flow-centric, back-in-time debugging. In: Oriol, M., Meyer, B. (eds.) TOOLS EUROPE 2009. Lecture Notes in Business Information Processing, vol. 33, pp. 272–288. Springer, Heidelberg (2009)
11. Lienhard, A., Gîrba, T., Wang, J.: Practical Object-Oriented Back-in-Time Debugging. In: Ryan, M. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 592–615. Springer, Heidelberg (2008)
12. Pothier, G., Tanter, É.: Back to the future: Omniscient debugging. IEEE Software 26(6), 78–95 (2009)
13. Pothier, G., Tanter, É., Piquer, J.: Scalable omniscient debugging. In: Proceedings of the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2007), pp. 535–552. ACM Press, New York (2007); ACM SIGPLAN Notices, 42(10)
14. Sadakane, K., Navarro, G.: Fully-functional succinct trees. In: Proc. 21st Annual ACM-SIAM Symposium on Discrete Algorithms, SODA (2010)
15. Saito, Y.: Jockey: a user-space library for record-replay debugging. In: Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging (AADEBUG 2005), pp. 69–76. ACM Press, New York (2005)
16. Srinivasan, S.M., Kandula, S., Andrews, C.R., Zhou, Y.: Flashback: a lightweight extension for rollback and deterministic replay for software debugging. In: ATEC 2004: Proceedings of the Annual Conference on USENIX Annual Technical Conference, pp. 3–3. USENIX Association, Berkeley (2004)
17. Tallam, S., Gupta, R., Zhang, X.: Extended whole program paths. In: International Conference on Parallel Architectures and Compilation Techniques, pp. 17–26 (2005)
18. Xin, B., Sumner, W.N., Zhang, X.: Efficient program execution indexing. In: Gupta, R., Amarasinghe, S.P. (eds.) PLDI, pp. 238–248. ACM, New York (2008)
19. Xu, M., Malyugin, V., Sheldon, J., Venkitachalam, G., Weissman, B., Inc, V.: Retrace: Collecting execution trace with virtual machine deterministic replay. In: In Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation, MoBS (2007)
20. Zhang, X., Gupta, R.: Whole execution traces and their applications. ACM Trans. Archit. Code Optim. 2(3), 301–334 (2005)