Department of Computing
Imperial College London

MEng Individual Project

A Time Travelling Debugger for Python

Niklas Steidl

supervised by

Dr. Robert Chatley

June 2016

Abstract

Time travelling debuggers (TTD) allow programmers to step backwards through the execution of their program, helping them to explore code and find the cause of bugs. This project presents an efficient TTD for the Python[28] language. Through modifying the Jython[15] Python interpreter, we implement a) *tdb*, a simple re-execution based TTD, b) *odb* a logging based TTD and c) a design for an efficient hybrid TTD.

Existing TTDs are either implemented by logging changes to a program's state during execution or by taking snapshots of the program and re-executing to a given time point. Such debuggers run slowly and / or use large amounts of memory. A Python TTD written in Python would be straightforward to implement, but slow as the debugger itself would be interpreted in addition to the program being debugged. Our hybrid design embeds TTD functionality in the interpreter. In order to support responsive debugging of non-trivial programs, we have implemented interpreter level instruction counting and a copy-on-write mechanism to maintain history. This is difficult as the internal representation of objects and data structures must be modified. Combined, we use these features to efficiently travel backwards through the execution of a program.

Our implementation performs better or on a level with with state of the art TTDs. The performance of *tdb* and *odb* are respectively 4 and 3 orders of magnitudes better than epdb, an existing Python TTD, when running the *bm_call_method*, *bm_call_simple*, *bm_fannkuch*, *bm_nbody*, *bm_raytrace* CPython performance benchmarks[1]. The instruction counting method used in *odb* and *tdb* has half the overhead of epdb when running the textitpybench benchmark. These initial findings show that our hybrid design is viable, and could be applied to other languages.

[1] `https://hg.python.org/benchmarks`

Acknowledgments

I would like to express my sincere thanks to:

My supervisor, Dr. Robert Chatley, for agreeing to supervise my project and giving me great advice when I doubted my project.

My personal tutor, Dr. Tony Field, for his words of encouragement, and the guidance has given me over the past 4 years.

My family, for supporting me in any way they could. I appreciate everything you have done for me.

Last but not least, my friends, who made the last 4 years fly by. In particular, my group project dream team: Stephen, Adrian, Bradley, Luke, George, Tom and Oskar.

Contents

Chapter 1

Introduction

A Bug's cause is often disconnected from its manifestation. Identifying a bug as having occurred is not the same as finding its cause. By the time a programmer has observed a bug, they have likely executed beyond the statement that caused it. Any 'real' program is too long to step through line by line from start to finish, with current tools. The programmer commonly guesses where they think the bug originated and sets a breakpoint. It can take a fair amount of time to reach the point in execution where the bug emanates from. A method may be called many times, but only exhibits the bug for a single invocation. If the programmer's guess is after the bug occurred, all this work needs to be repeated.

A debugger that allows a user to step backwards and forwards through a program's execution and inspect any value present at that time would vastly improve the efficiency of the debugging process. If the developer happens to step over the cause of the bug, they can simply step backwards. Such a debugger is known as a time travelling debugger, or a TTD. TTDs are not a new invention. They have existed in academia since the late 1960s, but were only commercially developed beginning in the late 2000s with underwhelming adoption. In recent years, research in TTDs has spiked again, spurred by an inspiring talk by Bret Victor[36] on improving the state of programming.

In theory, implementing a TTD is fairly straightforward. A program can't be executed in reverse, but the effect can be emulated. The simplest way to do this is to track how far the program has executed, and re-execute to any point in the program in the past that we would like to inspect. Conversely, all changes to the data of the program could be logged in order to be able to replicate its state at any point in time. To be able to provide the same information available to a regular debugger, a logging TTD must also store an abstraction of the execution path. However, logging imposes a large overhead on the forward execution of a program, and re-executing to an earlier point in the program can be slow for large programs. A common solution to this is to create a checkpointed TTD[2, 8, 19, 31] , which makes checkpoints or snapshots of the program at various points in time. The temporal distance that must be re-executed is then limited by the time between checkpoints. The standard implementation of checkpoints uses the Unix `fork` system call, to spawn child processes, which share memory with the original process. These child processes act as checkpoints, serializing the execution of the program. When one of the processes modifies the shared memory,

a unique copy is created for that process. This is known as a copy-on-write scheme, and is efficient in terms of processing overhead and memory usage.

This project proposes a design for a performant, hybrid TTD by modifying the Jython[15] Python interpreter. We implement the functionality required for time travel at the level of the interpreter, to improve performance and efficiency when compared to a TTD written in Python. The hybrid is a record and replay style TTD, which uses a persistent data structure to create checkpoints. It re-executes instructions to reach time points between checkpoints. In order to know how far forward to re-execute, we track the number of instructions executed. This is implemented natively in the interpreter, to avoid the penalty of each instruction of the debuggee program being counted by another piece of interpreted code. Many existing solutions use Unix's `fork` system call to create checkpoints. Our solution embeds the copy-on-write behaviour of `fork` into the interpreter, allowing for finer granularity and improved performance. This is more efficient, as changes to a value only require copying the underlying object, not the entire virtual page it is stored on. The challenge is, that we must modify the interpreter.

Modifying the interpreter to implement TTD functionality is messy, but provides a performance boost. If we implemented our TTD in Python code, each instruction of the debuggee program would be wrapped in debugger code which would also be interpreted. This is slow, and difficult for the JVM's JIT compiler to handle. Interpreter level implementation avoids the overhead of being interpreted and is more likely to be JIT'ed by the JVM.

We have chosen to implement such a debugger for the popular dynamic programming language, Python [30]. As a dynamic language, Python users stands to benefit more from a TTD than its statically typed counterparts. Often the best way to understand Python code can be to run it and a powerful TTD would allow a user to explore and understand code more quickly and easily.

1.1 Contributions

A summary of the contributions of this thesis is given below:

- **Efficient instruction counting** A key component of any TTD is an absolute measure of time. We have implemented an efficient version in the form of interpreter level instruction counting, which has 48% lower overhead than existing methods and imposes an overhead an order of magnitude less than existing solutions for Python.
- **A simple re-execution TTD for Python, *tdb*** *tdb* can perform the basic TTD navigation commands, but is limited in performance due to its lack of checkpoints.
- **An implementation of objects that store their own history using an efficient copy on write approach** *odb* uses the copy on write approach described in chapter 4 and implemented in chapter 6. The performance described in sections 7.2 and 7.4 make a considerable improvement over a comparable

`fork` based implementation, and has equivalent performance when compared to a persistent data structure TTD.

- **An implementation of built in Python data structures that track their own history** Specifically lists and dicts, however any collection that derives from either of these inherits the history logging properties, for example sets.
- **A performant logging based TTD that utilizes the above copy-on-write objects, *odb*** *odb* has an overhead 3 orders of magnitudes lower than epdb. When compared to Lienhard's object aliasing, *odb* performs with a similar (7x) if not better (as low as 4x) overhead of , while logging more information.
- **A design for implementing a hybrid Python TTD utilising interpreter level copy-on-write** Section 4 describes how to take advantage of Python's implementation details regarding locals and object fields to create an efficient checkpointing method.

Chapter 2

Time Travelling Debuggers

There are many names for Time Travelling Debuggers (TTD), for example Omniscient Debugger, Tracing Debugger or Reversible Debugger. We shall continue to refer to them in general as a TTD, making no assumptions about the underlying implementation. Instead, we will define the spectrum of TTD implementations.

2.1 Tracing / Logging TTDs

On one end of the TTD continuum, we find Tracing Based TTDs. This type of TTD logs all changes to the environment of a program in order to produce a comprehensive database of events. This database can then be searched to answer questions such as what are the values of variables over time. Generally, a special interface is required to query this data. Some implementations, like Visual Studio IntelliTrace[16](Subsection 3.2.3), allow this database of events to be treated as a regular debug session, with the additional benefit of being able to step backwards in time. This type of debugger allows users to record a bug once and then debug it repeatedly, potentially on a different machine. This type of TTD does not allow the user to resume execution from a past point in time.

Tracing Based TTDs tend to have high performance overheads and high data write rates to store the logged data. The size of the logged data can often become unreasonably large. However, the benefit is that all information is stored, so 'reverse execution' is very fast.

2.2 Replay / Re-execution TTDs

On the other end of the continuum are Replay Based TTDs. Since most operations are not reversible, reverse execution is not strictly possible. Instead, the illusion of reverse execution is created by keeping track of the temporal distance travelled forwards in a program, and re-executing to a desired time point from the beginning of the program. This is achieved by logging only the non-deterministic events, such as user input or

system calls. Then in re-execution or replay mode, the statements that caused the non-deterministic events are replaced with the logged values.

Re-execution Based TTDs have a low runtime overhead for forward execution. However, to 'reverse execute' they must re-run the program which is slow.

(For an example see subsection 3.2.5 on the Elm language's TTD)

2.3 Record and Replay TTDs

Pure Replay Based approaches are not often used in practice, instead the Record and Replay approach is used. The state of the program is periodically stored in a **snapshot** or **checkpoint** as it executes. To 'reverse execute', the snapshot before the desired timepoint is restored and the remaining distance is re-executed as described above.

Record and Replay TTDs are the middle ground between a pure logging and pure re-execution based TTD. The overhead of forward execution depends on the frequency of snapshots, the more frequent the larger the overhead. Similarly, the 'reverse-execution' latency is also based on this frequency. The distance required to re-execute can be bounded by the maximum temporal distance between checkpoints. Implementations might employ a variable checkpoint frequency or discard old checkpoints in order to avoid running out of space.

The bound on the distance which needs to be re-executed improves the performance of a Record and Replay TTD over a Re-execution TTD, however this comes at the cost of storing checkpoints.

(Examples: 3.1.1 IGOR or 3.1.12 TARDIS)

2.4 Persistent Data TTDs

A Persistent Data TTD is a type of tracing TTD. Instead of storing data in an external database, it stores a log of events in memory. This log could be stored in the address space of the running program, through a manipulation of the representation of objects, or simply stored in a data structure managed by the debugger.

Persistent Data TTDs tend to allow for faster read and write times than their externally stored counterparts. However, this comes at the cost of having a limited storage size. As a result, many Persistent Data TTDs either impose a hard limit on the number of events they can store or the time they can run for. Alternatively, they may employ a type of garbage collection to remove old events, but bugs that require a very large chain of events may not be caught by such a debugger if key events are lost.

Persistent Data TTDs have a lower overhead than their basic logging based counterparts, but have a limited capacity.

(Examples: subsection 3.1.6 ODB or subsection 3.1.8 Practical Object Oriented Back In Time Debugging)

2.5 Time Travelling Debugger Commands

The semantics of a standard debugger's commands are well known. The debugger continues execution until it is paused, either by the user or due to hitting a break point. Once paused, the debugger can step through single blocks of execution or continue. For the standard Python debugger, pdb, these commands are defined in the documentation[27] as follows:

step Execute the current line, stop at the first possible occasion (either in a function that is called or on the next line in the current function).

next Continue execution until the next line in the current function is reached or it returns. (The difference between `next` and `step` is that `step` stops inside a called function, while `next` executes called functions at (nearly) full speed, only stopping at the next line in the current function.)

return Continue execution until the current function returns.

continue Continue execution, only stop when a breakpoint is encountered.

We shall define the semantics of the reverse counterparts of the standard debugging commands. We shall prefix the complementary reverse commands with 'r'.

rstep Reverse-execute to the previous line, stop at the first possible occasion

rnext Reverse-executes to the previous line, steps over function calls and back up and out of function entry

rreturn Reverse-executes to the function call

rcontinue Reverse-execute until we reach a breakpoint

Chapter 3

Background

3.1 Related Research

The following research is academic research that relates to the implementation of a TTD. It is presented in chronological order to give an understanding of the progress of the field.

3.1.1 IGOR - Brown and Feldman - 1988

IGOR[9], developed in 1988 for the Dune distributed operating system, was one of the earliest implementations of a TTD. The major goal was to "provide a practical way to attack large and complicated problems."[31] IGOR relies on check pointing and re-execution to allow for reversible execution. To use IGOR is not entirely straightforward. A modified compiler, library, linker and loader, as well as a system with a modified kernel are all required. In order to implement check pointing, Feldman and Brown implemented two system calls: `Pagemod`, for returning a list of memory pages since the previous call of the function and `ualarm`, a modified alarm system call that counts in user CPU time as opposed to the real time at a very fine granularity. Using these new system calls, IGOR saves a copy of every page in memory that has been modified since the last check. Without `pagemod`, IGOR would need to access memory that was modified, or in the worst case, copy all of memory for each checkpoint. Given that IGOR is intended as a debugger, the high granularity of `ualarm` is required to ensure that the debuggee has actually done some work in the elapsed time. When comparing IGOR with other TTDs, we are only interested in runtime performance, therefore we will not discuss the performance overhead added to the pre-execution steps. IGOR's execution overhead ranged from 40% to 380% for various programs run with checkpoint intervals of 0.1 to 1 second. During these runs, IGOR wrote on average 34 pages per checkpoint. The execution timing for the interpreter was approximately 140 times slower than actual execution. Like most TTDs after it, IGOR is limited to single thread applications. The approach used by Feldman and Brown is outdated and too 'low level' for current systems. Particularly, it wouldn't scale well to the size of today's memory and speed of processors, however as an inspiration for many future implementations it was important to analyse.

3.1.2 SPYDER Execution Backtracking - Agrawal, De Millio and Spafford - 1991

Agrawal, De Millio and Spafford's SPYDER system[1] is an advanced debugging tool that, among other things, uses execution backtracking to perform backwards debugging. The state of a running program is represented by the value of the variables in the program and the location of the program control. The history of the program is recoded by associating a change set with each assignment statement. SPYDER uses so called structured backtracking. This approach stores change sets with control flow, so that all the possibly modified variables are added. This means that if and while statements can be stepped over, but has the drawback that they cannot be stepped into. The authors argue that this is in line with forward execution semantics and therefore not an issue. The bound on the space required to store these change sets in the usual case is independent of the execution and instead based on the length of the program. SPYDER was implemented for C, but is truly a prototype, as only simple expressions are supported. For example pointers, function calls and arrays are not implemented. Later approaches note that the backtracking is condensed to a coarse set of points, which does not necessarily provide the flexibility required for backwards movement and debugging.[5]

3.1.3 Zstep 95 - Lieberman and Fry - 1997

Zstep 95[19] is a program debugging environment with a focus on helping the programmer "understand the correspondence between static program code and dynamic program execution"[19]. The capabilities of ZStep 95 were well ahead of its time, realizing many of the concepts made popular in Brett Victor's talk "Inventing on Principle"[36] around 15 years later. Zstep 95 is a "reversible animated source code stepper."[19] It features an animated view of the program's execution, updating the source code as the program executes. Synchronized with this is a graphical display of the source code, where all the graphical objects are linked directly to the source code that created them (see figure 3.1). This means a user can select a visual line, and see the source code that created it or vice versa. Along with this, it allows the user to step backwards in the execution of the program!

Zstep 95 recognizes the work done by past implementations of reversible debugger's, but chose to focus on the UI, an aspect the authors believed had thus far been neglected. As a result of this focus, Zstep 95 keeps a complete history of the program and its output without any regard for performance.

As a prototype, Zstep 95 was extremely successful. It was implemented for Common Lisp 2, and worked with a re-implementation of the graphics library. The authors note that the approach does not guarantee to work for programs with complex side effects, however they argue that the approach is no less successful at debugging such programs than traditional debuggers.

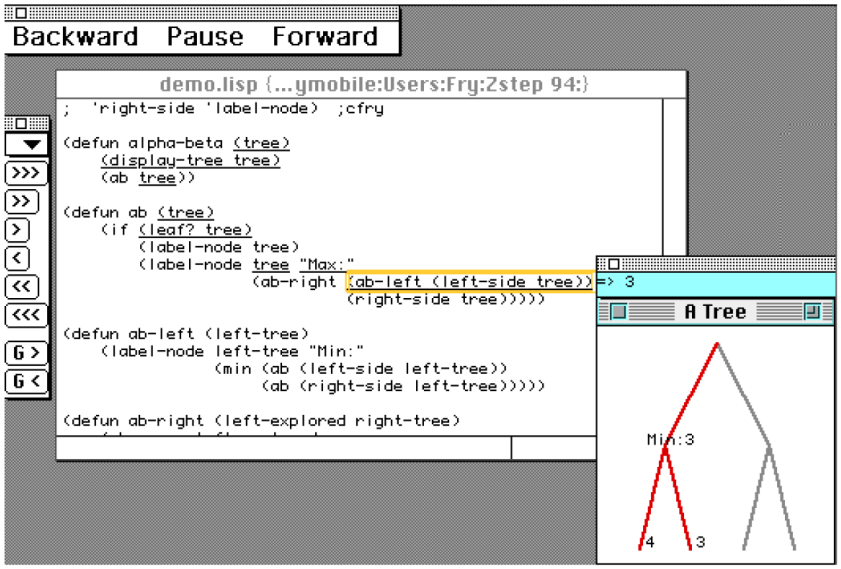


Figure 3.1: ZStep 95's graphical debugger window [19, P.4]

3.1.4 BDB - Boothe -2000

BDB[5] is a prototype, bi-directional debugger developed by Bob Boothe at the University of Southern Maine in 2003. BDB is a twist on the common approach of IO logging and re-execution. By embedding a program counter and call depth counter in the debuggee BDB is able to efficiently re-execute up to a desired point. By checkpointing, the time or re-execution is bounded. Boothe identifies three issues in the re-execution approach: "(1) locating the desired earlier point while re-executing forward, (2) the overhead involved in re-executing, and (3) ensuring deterministic re-execution"[5] The embedded counters address problem 1, checkpoints address 3 and the remaining issue is addressed by "true-execution rather than emulation."[5] A traditional debugger (such as gdb) uses trap based debugging where traps are inserted into the code, and the program is executed until it hits a trap statement. Boothe points out that such trap based approaches can lead to expensive context switching if for example the command `continue 1000` was used, or `next` called over a recursive call. Therefore, such an approach is not ideal for reverse execution. By embedding counters, the execution only traps once the counter variable has reached a specific value. Boothe's paper outlines how to implement standard debugger commands such as `next`, `step`, `continue` and `finish` in terms of a program counter and call depth counter, and more importantly, how to implement their temporal complements. (See figure 3.2 for an example of how `previous` uses the counters to reverse-execute)

Checkpointing is performed by forking the debuggee process using the `fork` unix system call. This takes advantage of the efficiency of fork due to the operating system's copy on write policy. To avoid the number of checkpoints becoming unreasonable, exponential thinning is employed. This ensures the property that "no execution point ever be in a checkpoint interval whose size is greater than the distance from the current position back to that point" [5]. Re-execution is possible in a time proportional to the temporal distance moved back. The number of checkpoints therefore grows

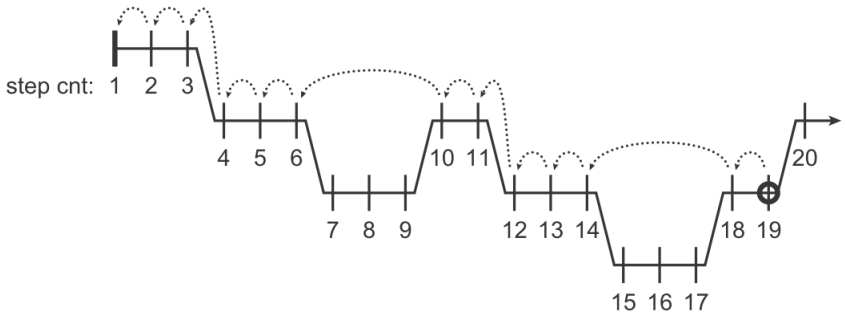


Figure 3.2: The sequence of time points visited by BDB when continuously executing the `previous` command. Function calls are stepped over (hence jumping over the trough at 15-17)[5, Fig. 5]

logarithmically.

Boothe identifies I/O logging as the most difficult task in implementing the debugger. Given an IO bound program they experience an overhead of 5100% and a slowdown factor of 52. However, replaying the IO incurs only half the overhead of creating it.

3.1.5 Reversible Debugging Using Program Instrumentation - Fuchs - 2001

Fuchs' paper[10] discusses an efficient reversible debugging scheme using automatic assembly level instrumentation, without modifying the programming language or compiler. While the concept is relevant, it won't be further explored in this thesis, as all trends point towards instrumentation at higher levels being more successful over time.

3.1.6 ODB - Lewis - 2003

ODB[18] is an omniscient debugger written for Java by Bill Lewis in 2003. ODB deviates from the standard approach of a logging based TTD, in that it stores its logs in memory. The history of ODB is limited to a 31 bit address space of roughly 10 million events. ODB is implemented at the bytecode level. ODB maintains a list of timestamps. Every time an event occurs, the debugger pauses execution. A new timestamp is created, and the trace associated with the time and event are created. Each variable has a history list, which consists of pairs of timestamps and values. History lists for local variables and arguments are attached to the traces as they are generated.

The performance of ODB is not spectacular, and highly dependent on the specific program. Lewis is aware of this, particularly as performance was not a goal for the project, stating "the important thing to recognize here is that ODB represents a worst case implementation - we can only do better. The ODB is completely naïve,

it does no optimization at all."[18] Similarly to Zstep 95, the main focus of ODB is its presentation of debugging state rather than the implementation of an extremely efficient TTD. The principles are, that the interface should update values as time steps are changed, state should be easily recognized and navigation should be simple. (See figure 3.3) However, despite performance not being Lewis' goal, it is still interesting to look at purely for comparison with other techniques. The slowdown of execution when using ODB varied from 2x to 300x in testing. For complex programs, the minimum overhead was 7x. Additionally it is important to note, that the address space can be filled in roughly 20 seconds (on a 700mhz Apple G3). Lewis points out that bugs that don't fit in this range can likely be dealt with through the use of garbage collection, marking methods to not be instrumented or using an event analysis engine to decide when to enable recording.

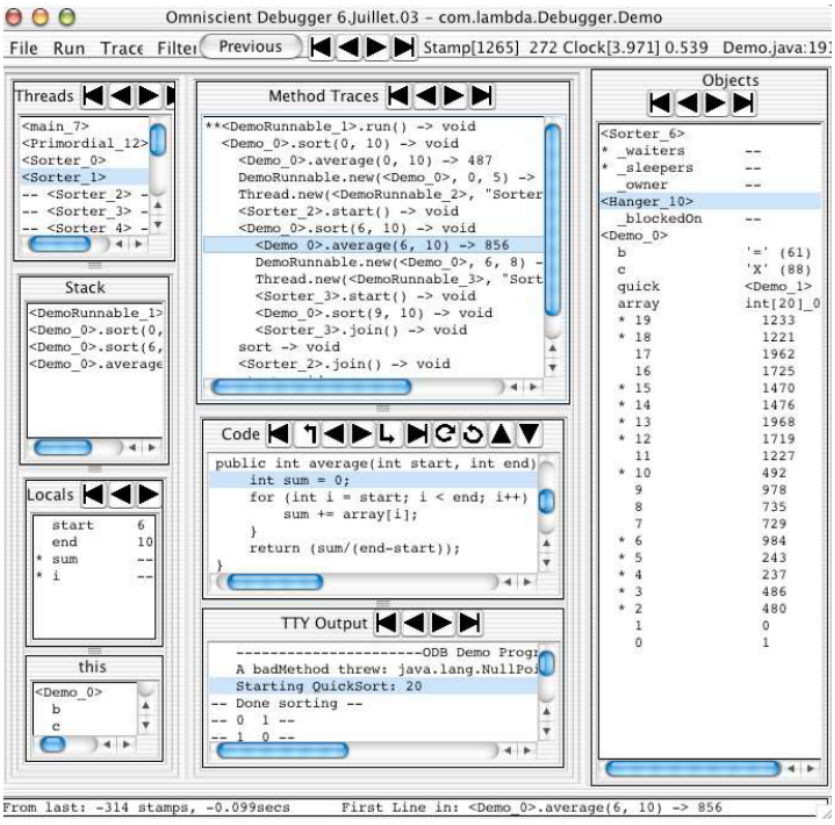


Figure 3.3: The ODB debugger window[18, Fig.1]

3.1.7 Why Line - Ko and Myers - 2004

Why Line[17] is not a TTD, but an interrogative debugger. It allows the user to ask questions about why or why not a program exhibited certain behaviour. The similarity to a TTD is in the tracing Why Line performs. Why Line performs tracing through the use of byte code instrumentation, additionally logging IO. In order to highlight the effect of specific code on the graphics output, Why Line has implemented a custom emulator for Java's Graphics2D class. This is similar to the approach presented by

Lieberman and Fry in Zstep95. In terms of performance Why Line is more comparable to a tracing debugger, with a slowdown of around 15 times on complex programs and as little as 1.7 times on simple ones.

3.1.8 Object Oriented Debugging - Lienhard, Girba and Nierstrasz - 2008

The paper "Practical Object-Oriented Back-In-Time Debugging"[20] is an award winning paper describing the implementation of a TTD based around objects. It differs from the other approaches, in that it doesn't follow a logging or re-execution model for replay. Instead, it relies on encoding the history directly into the representation of object references on the heap. The researchers point out that other approaches are limited by slow execution and large history logs. By storing object history in the same address space as the program, they solve this problem using garbage collection, as un-referenced objects are garbage collected. However, they lose some backtracking capabilities as unreachable objects and their history are removed by the garbage collection as well. Therefore, it is possible that the root cause of a bug might not be visible to such systems.

The main approach relies on modifying the VM or runtime to store object references as objects on the heap. These new references are referred to as aliases.(See figure 3.4) An alias contains a pointer to the previous alias as well as the origin object that instantiated it. This allows pointer traversals to retrieve the history of an object as well as determine how objects are passed to a method call.(See figure 3.5)

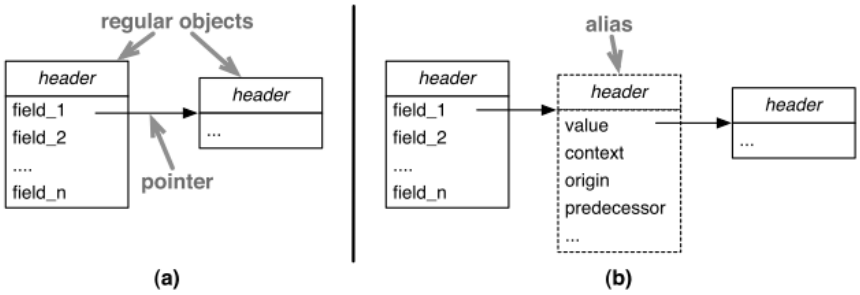


Figure 3.4: **a:** typical object with references as direct pointers and **b:** references represented by an alias object from[20, Fig. 1]

In the worst case, execution suffered an overhead of 7 times. The baseline VM with object history tracing introduced a performance overhead of 15%. As object history is stored on the heap with the objects themselves, looking up previous values is extremely fast, as there is no need for re-execution and the lookup is equivalent to a linked list traversal. A potential downside of this approach is, that only the history of data accessible from the current program point is available. The benefit of this approach is, that memory consumption is restricted to an upper bound in the best case, and slowly grows in the worst case (potentially exceeding the memory of the vm).

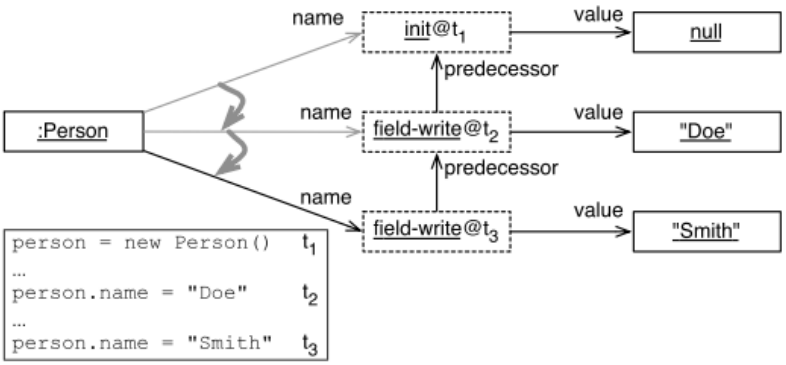


Figure 3.5: History of object values represented through a graph of aliases from [20, Fig. 3]

3.1.9 epdb - Sabin - 2011

epdb[31], or the Extended Python Debugger is a reversible Python debugger developed by Patrick Sabin at the Vienna University of Technology. epdb is a record and replay style debugger. It is written in Python, and runs on top of CPython by hooking into the interpreter by extending the standard pdb debugger.

In order to create a snapshot, Sabin used the `fork` system call to clone the process. This makes creating snapshots quite quick. To avoid recording non-deterministic calls, epdb simply takes a snapshot after any non-deterministic event. This way instead of re-executing the non-deterministic command, the snapshot will simple be restored when reverse debugging. To avoid long-running commands, EPDB takes a snapshot of the result of any instruction that takes longer than 1 second to execute. This helps speed up re-execution, for example a `sleep()` command for 10 seconds would not be re-executed but instead the snapshot could be loaded. The drawback to using `fork` is that the OS must manage a large number of processes, placing an upper bound on the number of snapshots that can be maintained. Sabin found that roughly 1000 snapshots could be stored, which equates to 16 minutes of execution with a 1 second checkpoint interval.

In order to step back in time, a measure of time is required. For this, epdb counts instructions using trace information supplied by CPython. Sabin notes that this implementation is quite slow (15 to 110×), and would be faster if implemented at a lower level.

epdb provides a feature called timelines. Timelines allow the past execution of a program to be cloned up to a certain point, and from that point forwards be executed normally. This is useful for answering questions about how the program would have performed after changing a specific value.

3.1.10 Tralfamadore - Head - 2012

The Tralfamadore debugger[13] is based on an execution recording and static analysis engine of the same name. It provides source code level debugging using a cpu level execution log. While it shows that interesting work on time travelling debuggers is actively undertaken, Tralfamadore is not particularly relevant to this project, because it focuses on a single point in code that is potentially hit multiple times throughout a trace.

3.1.11 Expositor - Phang, Foster and Hicks - 2013

Similarly to Tralfamadore, Expositor[23] validates the idea that TTDs are still relevant, however it is focused on creating a scriptable debugging experience using a TTD. In the case of expositor, the TTD is the commercial product UndoDB[35] discussed in the commercial applications section.

3.1.12 Tardis - Barr and Marron - 2014

The TARDIS[2] time-travelling debugger was implemented by Mark Marron and Earl Barr as part of Microsoft Research. In their paper "Tardis: Affordable Time-Travel Debugging in Managed Runtimes"[2] they outline a method for implementing time travelling debuggers by piggybacking on a managed runtime / VM such as the Java JVM. They then evaluate the performance of the realization of this design.

TARDIS is the most recent of the implementations discussed in this thesis. Therefore, it has the advantage of learning from its predecessors. The authors reflect, that while TTDs have existed for decades, they have not been widely adopted. They attribute this to "prohibitive runtime overheads"[2] of 10-100 times and "long pauses when initiating reverse execution."[2] From their usability research, they found wait times of longer than 10 seconds cause "rapid system abandonment."[2] Their goals for Tardis were therefore to create an affordable time travelling debugger, with an execution overhead of less than 25% and a time travel latency below 1 second.

Marron and Barr base the starting point of their TTD on the fact that managed runtimes have "already paid much of the cost of providing core features - type safety, memory management, and virtual IO."[2] They adopt the standard approach of taking snapshots of the program state at intervals, logging non-deterministic events and re-executing from checkpoints to achieve 'reverse execution'. The managed runtime aids in simplifying this. Memory management allows the heap to be inspected and manipulated. Type information allows a good level of precision to the introspection. Combined, these allow for less information to be included in the snapshots, as only live objects and program state need to be included in a snapshot. The garbage collector (GC) allows for additional optimization. A generational garbage collector's remembered-sets reduce the cost of walking the live heap. Lastly, a write barrier ensures only modified memory locations need to be included in the snapshot. The virtualisation of resources provided by a managed runtime allow calls to non-deterministic events such as IO to be intercepted and manipulated for replay.

The authors describe the implementation of a non-intrusive TTD, which can retrofit existing runtimes. On forward execution, snapshots are taken at regular intervals. Additionally, at each loop head, procedure call and any point where the garbage collector may be invoked snapshots are created. Snapshots and logged IO need to have a total order, so a unique trace timestamp is implemented by incrementing a tracetime counter whenever a built-in procedure that can produce IO may execute, or a snapshot is taken. Re-play is done in the standard manner of restoring the earliest checkpoint before the desired time point, and forward executing to that point (replaying IO where necessary). This non-intrusive approach does not change the allocator, memory layout of objects or garbage collector.

A more involved implementation relies on the garbage collector. The best results are achieved if a generational GC is used. Long-lived values won't be copied multiple times, as each explicit snapshot operation will only walk the nursery as opposed to the whole heap. Additionally implementing a write barrier allows the TTD to only save modified objects. As the size of snapshots can be quite large, using a separate thread to compress the heap before saving offers a significant performance boost.

For the non-intrusive snapshotting, a runtime overhead of less than 22% at 5.6MB/s of logging was achieved. After modifying the garbage collector, performance of TARDIS was improved to an overhead of 11% at 1.3MB/s of logging. The latency of time-travel was 0.65 seconds in the worst case, a huge improvement over the 10 second abandonment threshold. This project was such a success it is being adapted to be included as the default debugger for the Microsoft Edge Browser.

3.2 Commercial Products

Commercial products do not tend to divulge their implementation details, but instead are useful as an indication of adoption and trends.

3.2.1 Undo DB

Undo DB[35] is a commercial DB which uses "a 'snapshot-and-replay' technique that stores periodic copy-on-write snapshots of the application and non-deterministic inputs (system calls, thread-switches, shared memory reads, etc)."[35] The execution overhead is 2-4 times on average, but can reach 10x in pathological cases. This performance is achieved by scaling the frequency of snapshots dynamically, and maintaining only 25 snapshots. By default, the memory consumption is kept low. UndoDB uses copy-on-write to create snapshots, so changes between snapshots are not stored. Similarly to other TTDs, "during replay, non-deterministic events are not performed directly, but their effects are synthesized based on the contents of the event log."[35] According to the UndoDB website, the performance is "many orders of magnitude better"[35] than gdb's. For example, running the gzip compression on a 16MB file takes gdb 21 hours to record using 63GB compared to UndoDB's 2.6 seconds and 17.8MB of memory used.

3.2.2 Chronon Time Travelling Debugger

The Chronon Time Travelling Debugger[6] is a commercial TTD for Java. It runs on Windows, Linux, OSX and Solaris, and has plugins available for eclipse and Inteilij IDEA. As a commercial product, the source code is not available to inspect how Chronon works, nor are real performance figures given. The website states, that Chronon "reads data from the recording file, almost like reading a database, to instantly reproduce the state of a program at any point in time"[32] and that the original executable is not required for replay. This indicates, that Chronon falls into the category of a logging based TTD. Alongside the ability to step backwards in time, Chronon offers other advanced features such as method history, variable history and post-execution logging. Post-execution logging allows a developer to generate a log file from a recording, by indicating where log messages should be evaluated in the source code. However, these calls to the logger are not done directly in the source code, but dynamically after execution, similarly to inserting a breakpoint. Instant-execution path is another interesting feature. It allows a developer to see the paths taken by a method call, but unlike code coverage, "the Execution path in Chronon shows you a separate execution path for each different call to a method. Thus a million different calls to the same method would result in a million different execution paths."[6] In order for the two previously mentioned features to be possible, without re-executing the code, Chronon must also log the execution path. This is somewhat contradictory to a statement made about the minimal overhead Chronon places on an application:

> "The Chronon Recorder puts minimal overhead on your applications. The overhead is not much more than what simple logging inside your application, making Chronon suitable for even the most intensive applications. We have spent years perfecting our 'prediction' technology which analyses your Java bytecode as it is loaded and creates predictions as to how it will execute. Thus only the most minimal, non-deterministic portions of your applications are recorded and even those in a highly compressed format. Thats much, much less than what traditional logging would generate."[21]

From this statement, it would also be reasonable to assume that Chronon operates at the virtual machine level. Rather than instrumenting bytecode directly, it hooks into the JVM or JIT compiler to perform its logging. This is in line with the observation, that recent TTDs have been implemented at higher abstraction levels, leading to improved performance. A clear advantage of the approach taken by Chronon is its ability to handle multithreaded programs with low overhead, something that few TTDs allow.

3.2.3 Visual Studio IntelliTrace

Microsoft has provided the IntelliTrace historical debugging feature as part of its Visual Studio IDE[37] since 2009. IntelliTrace is a pure logging TTD. It records the events of a debugging session and plays them back, allowing the user to step forwards and backwards. According to the documentation, IntelliTrace "is not enabled by default because it adds considerable overhead. Not only does IntelliTrace have to intercept every method call your application makes, but it also has to deal with a much larger

set of data when it comes to showing it on the screen or persisting it to disk."[16] One of the best features of IntelliTrace is, that the recording can be shared, allowing a bug to be replayed by multiple different developers at once. The main limitation of IntelliTrace is, that it does not allow for the local values of variables to be changed while in replay mode or to continue execution past the replay point.

3.2.4 GDB

In version 7 of GDB[11], reversible debugging was introduced. This allows users to step forwards and backwards. It is only compatible with certain GDB targets, which must explicitly enable recording. GDB is a low level, native debugger. To facilitate reverse-debugging it logs machine instructions executed in the debugee with the corresponding change in memory and registers. To reverse execute, it sequentially reverses the change of each logged instruction. This can be very slow as mentioned in subsection 3.2.1.

3.2.5 Elm Debugger

The Elm time traveling debugger[7] is an extremely impressive TTD build by Laszlo Pandy in 2013. The power of this debugger is due to the properties of the language it works on. Elm is a functional language, based on the functional reactive programming paradigm. The purity of functional programming, as well as the immutability of objects makes a TTD straightforward to implement for ELM. Purity with respect to ELM means that side-effects are modelled explicitly, meaning "To perform a side-effect, you first create a data structure that represents what you want to do. You then give that data structure to the language's runtime system to actually perform the side-effect."[7] This is similar to how a runtime takes care of things like memory management. So in ELM to replay events without side-effects, the runtime simply needs to be told not to perform any of the side-effects. Immutable data ensures that data is not overwritten. Lastly, ELM's basis on FRP means that events are already tracked over time via 'signals'. "This makes managing replay a matter of recording the incoming events to the program and discarding the outgoing events. As long as no one acts on the outgoing events, there will not be unwanted side-effects."[7] Perhaps the most impressive aspect of ELM's debugger is its support for hotswapping. Hotswapping is the act of replacing code as a program is running. The combination of the two allows the user to rewind time, change a value, and play back execution without re-starting the execution.

In order to use Elm's debugger (similar to PDB in Python) one sets watch or trace points in the code for values that should be tracked.

> The watch function: `Debug.watch : String → a → a` will track a value over time (See the y-velocity in the example)
> The trace function: `Debug.trace : String → Element → Element` will visually trace a selected value over time.

There is some room for improvement for the Elm debugger. Having to modify the code to enable tracing and watching is less than ideal. For example, Elm does not

implement any checkpointing. Therefore the entire program must be re-executed on replay. Another desired feature would be to save a recording for replay at a later date. This is particularly useful for testing and bug reports. Pandy also suggests that graphing watched values over execution time would be a useful feature.

3.3 Research Summary

As the sections above have shown, there are many variations of TTD implementations, but all rely on solving a few crucial problems. Regardless of the type of TTD being implemented, some representation of time must be uniquely associated with execution. A tracing based TTD must track the state of the program, and provide an interface for viewing the data. A record and replay style TTD, must have a mechanism for saving and restoring state, as well as a controlled manner of replaying execution, accounting for non-deterministic statements.

Mapping execution time to a unique identifier is the simplest problem, with one general solution. TTDs count number of statements executed. The implementations differ in the level at which the counting is performed. Some instrument assembly code or use bytecode counters [10, 5]. Others count as they step through the program [31, 5] or modify the runtime [20, 2].

The solutions to logging state differ only at the implementation level. Many favor instrumenting bytecode [17, 18] while others perform the same at the VM/runtime level [20, 18]

The solutions to creating snapshots generally fall into two cases. Some make use of the operating system to perform efficient cloning of processes through the use of a copy-on-write memory policy.[5, 8, 31] Other implementations try to improve performance, by snapshotting state at a higher level than pages of memory.[2, 13, 20]

There are two common solutions to replaying non-determinism. Either the affected calls are dynamically patched at runtime or a checkpoint is made after every non-deterministic event, negating the need for re-execution.

Logging and re-execution based TTDs both have their place. Over time both solutions have been successful, and continue to be so, with neither solution dominating. However, there is a trend within the level of implementation. Earlier solutions were implemented at a lower level, with some requiring a specific OS to be run on [8] and implementing their own system calls. More recently, the trend points towards taking advantage of existing services provided by the runtime, for example memory management, as seen in [2], [31] or [20].

3.4 Python Implementations

Our TTD targets Python[30], so it requires a Python interpreter. As writing a new interpreter is beyond the scope of this project, we chose to modify an existing Python interpreter. Luckily, there are many different interpreters for the Python language. In

fact, there is no official implementation of Python. Python is actually the specification of the dynamic programming language invented by Guido van Rossum. However, when referring to Python, people often mean the CPython[28] interpreter and environment too. Some alternative implementations of Python include Jython[15], PyPy[25] and Iron Python[14]. We evaluate these environments for their suitability for implementing support for a TTD. The original implementation of Python was entirely open source, a principle the community has continued to follow in other implementations.

3.4.1 Python Modules

Before discussing the specific implementations of Python, the introspective modules of Python merit a discussion. Due to the high amount of introspection that is allowed by the Python language, it is possible to implement a Python TTD written in Python (see subsection 3.1.9). The main modules that enable this are the `inspect`, `trace`, `dis` and `sys` modules. In particular, the function `sys.settrace` allows a Python program to implement a Python debugger. The `dis` module can disassemble Python bytecode. `inspect` provides information about the running program, including linking the current execution state to the line of code in a file.

However, a TTD with functionality implemented by the interpreter or in C is likely to be faster than one implemented in Python and for this reason we consider the various implementations of Python interpreters.

3.4.2 CPython

CPython[28] is the original implementation of Python. When someone refers to the Python interpreter, they are generally referring to CPython. Cpython was developed in the 1980s by the creator of Python, Guido van Rossum, as the original implementation of the Python language and is written in C. It is a simple bytecode interpreter that makes use of a stack machine. CPython is implemented using reference counting rather than fully fledged garbage collection. This can be faster and less resource intensive, but pays a price in ease of development. Reference counting is hard-coded into the CPython code base. It is prone to human errors, which cause memory leaks if references are not properly added. Furthermore, the hard-coded nature makes it near impossible to experiment with other garbage collection techniques.

CPython is a fast implementation of Python, but the cost of modification seems to be quite high. Being written in C means it has a relatively low performance overhead, but loses out on features such as garbage collection provided by a VM or managed runtime.

3.4.3 Jython

Jython[15] is a successful implementation of Python using the Java virtual machine (JVM) created by Jim Hugunin. Like CPython, it is a simple stack machine. As a full re-implementation of Python, one of the disadvantages of using Jython, from

a developer's perspective, is that it lags behind CPython in its implementation of features.

As Jython is built on top of the JVM, it can take advantage of certain features provided by the JVM, such as memory management and typing. Given memory management, Jython is well poised towards implementing the object-oriented debugger described in Nierstrasz and Lienhard's 2008 paper.[20] However, there are certain features of Python's introspective modules that Jython does not implement, as it runs on the JVM. Furthermore, the translation of Python bytecode to Java bytecode and subsequent execution is obscured, making a TTD implementation somewhat difficult.

Jython has a lower overhead in terms of modification mostly as a result of being implemented in Java. However, the reliance on the JVM might prove to be more of a hindrance than a help.

3.4.4 PyPy

PyPy[24] is an implementation of Python written in Python. Many consider this to be a great improvement over Jython and CPython, as they would argue Python is simpler to write than C or Java. If the self-hosting nature of PyPy wasn't convincing enough, it is also roughly seven times faster than CPython on a number of benchmarks[26]. This is because PyPy makes use of just in time (JIT) compilation. Code that will be run in a loop over many iterations is detected and compiled to machine code by the JIT compiler, which can then be run directly on hardware rather than being interpreted.

Besides being fast, PyPy has a design that lends itself to developing a TTD (see section 5.2 for more details). Additionally, PyPy is extremely quick to adopt new Python features and reach parity with CPython. This is thanks to its strong and active developer community. Adapting PyPy might prove to difficult, as it is too clever for its own good (see 5.4) , and interpreter internals must be written in a proprietary subset of Python.

3.4.5 Iron Python

Iron Python[14] is a Python implementation that runs on Microsoft's CLR. Originally started within Microsoft as an open source project by Jim Hugunin, Iron Python allows Python code to access the .NET framework. This works because Iron Python is built on the DLR, Microsoft's Dynamic Language Runtime, and extension of the Common Language Runtime. Iron Python has always been an open source project, but in 2010 Microsoft abandoned work on it, leaving the community in charge. The last update to Iron Python was in 2014.

Iron Python is very similar to Jython due to both being created by Jim Hugunin. Similarly to Jython, Iron Python benefits from running in a VM, which is desirable for implementing a TTD according to our background research. However, Iron Python has more or less been abandoned, so Jython seems to be a better fit.

Chapter 4

Design

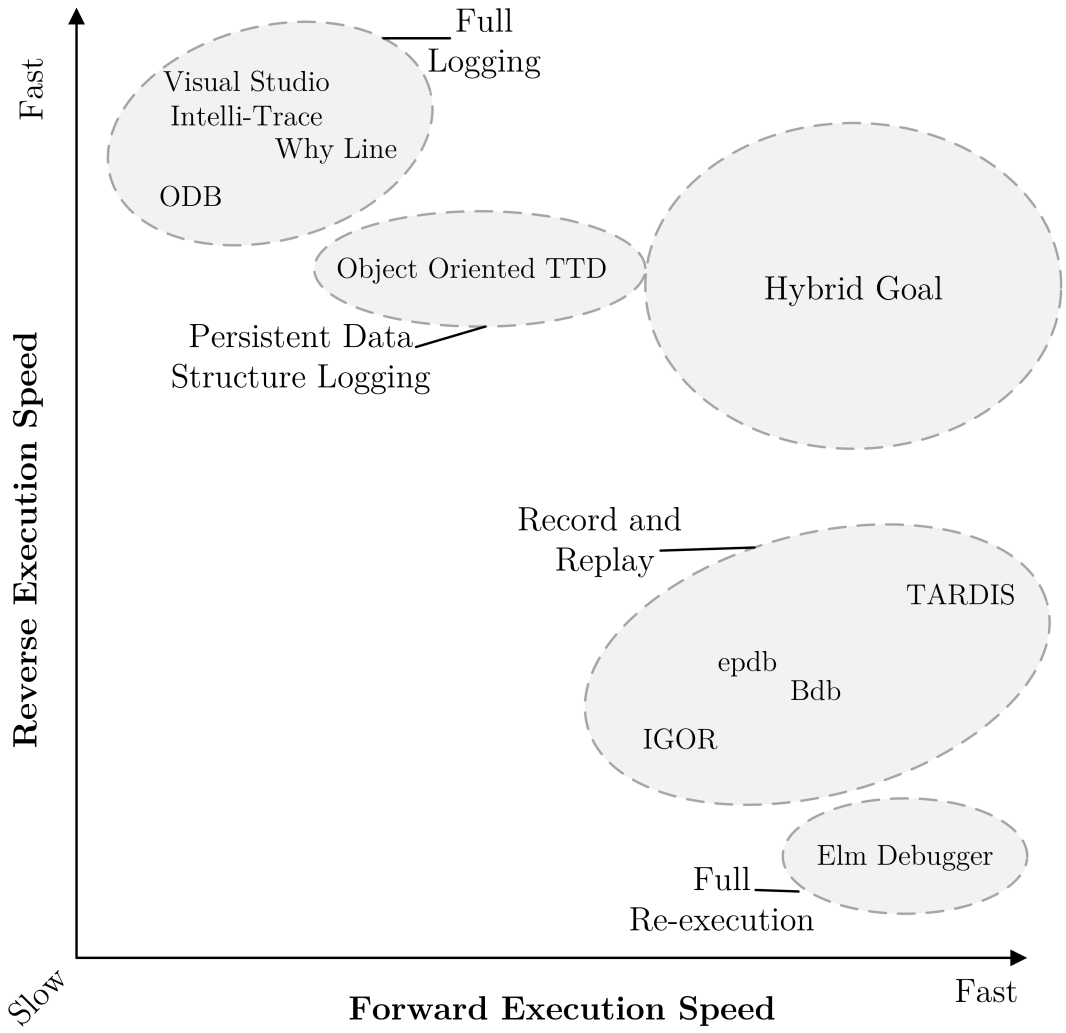


Figure 4.1: TTD landscape

TTDs sit on a landscape (figure 4.1) where implementations make trade-offs between

forward execution speed and reverse execution speed. In the top left corner lie full logging TTDs (ie Visual Studio's intelli-trace or ODB), which execute slowly forward but make up this overhead with fast reverse execution. In the opposite corner, full re-execution TTDs have low execution overhead, but can be slow to travel back in time. ODB, which stores history in the address space of the debugger sits just center of a full logging approach (Persistent Data Structure). To the center of a full re-execution approach lies Tardis, which relies heavily on snapshots, event logging and re-execution(Checkpointed Re-execution). The speed vs memory trade-off lies along this same diagonal line. This project aims to split the middle in the region between Tardis and ODB to achieve an improved balance, ideally moving towards the top right corner where both forms of execution are fast.

4.1 Required Functionality

To achieve the desired hybrid approach, some basic functionality present in existing TTD solutions is required. We shall describe these functions briefly in the sections to follow.

4.1.1 General requirements

Debugging and Introspection The hybrid TTD must provide basic debugging capabilities. These include stepping through the execution of the program, and inspecting its state at said points of execution. Introspective features include:

- listing the line of code currently being executed as well as the event type (line, call, return, exception)
- showing the current call stack
- navigating the current call stack
- displaying a variable's value

Absolute measure of time To travel in time, an absolute measure of time must be defined. According to Barr and Marron this measure should impose a total order on events and snapshots and ensure that every execution of a statement has a unique timestamp.[2] Note, that the term timestamp is overloaded, and in these cases does not mean the actual time. A desirable feature of such a measure of time is, that it's granularity is as coarse as possible. This allows us to maximize the number of events represented in a given address space.

4.1.2 Logging based TTD basic requirement

Logging of changes to variables This is the key feature of a logging based TTD. By recording changes to every variable along with its timestamp, the execution of the

program is partially serialized. This event log can be searched to determine the cause of a bug, but further exploratory debugging from a past point is not possible.

4.1.3 Re-execution based TTD basic requirement

Re-execution to an arbitrary time point This is the main feature of a re-execution based TTD. As reverse execution isn't possible, we re-execute to give the user the illusion of travelling backwards in time. Re-execution is based on the absolute measure of time, therefore if there are any branches in control flow that are non-deterministic they must be recorded and replayed in order to reached the desired execution point.

4.1.4 Performance improvements

Checkpoint Creation In order to speed up the reverse execution latency of a re-execution based TTD, checkpoints are used. A checkpoint must encapsulate the state of a program at a given timepoint. It therefore requires at least as much information as a logging based TTD can provide about the state of a program at a given timepoint. A checkpoint may need to include additional information in order for the execution of the program to be resumed at a later point.

Checkpoint Restoration A checkpoint and re-execution based TTD must be able to restore a checkpoint to avoid needing to re-execute large amounts of code. The time taken to restore a checkpoint should be less than the time required to re-execute between any given checkpoints. Restoring a checkpoint likely requires the ability to jump to an arbitrary line of code, and place arbitrary frames on the call stack of the interpreter.

4.2 Objectives

To guide the implementation of our project, we propose the following performance based objectives:

O1 **(Comparably) Low overhead instruction counting ($\leq 15\times$)**
In order to travel back in time, we need an absolute measure of the execution of a program. To do this we essentially count the number of instructions executed. As this is a key component of any TTD we would like the overhead incurred as a result to be low. An existing python TTD, epdb[31] (see section 3.1.9), incurs an instruction counting overhead of between 15 and 110 times. As epdb is implemented in python code, we believe an interpreter level implementation should improve performance.

O2 **Forward execution overhead comparable with other state of the art TTDs ($\leq 30\times$)**
For a debugger to be considered usable, it must not be too slow. Many existing implementations have forward execution overheads that range between 7 and 300

times. A worst case overhead of 30 times would be a substantial, but hopefully attainable improvement.

O3 **Comparably low memory consumption for snapshots ($\geq$ 10 million events in 2GB of memory)**
TTDs must store some data. Snapshots tend to use less space than full logging solutions, but both can be quite memory intensive. We would like our TTD to be competitive not only in terms of speed, but also in resource usage. However, the amount of data stored depends heavily on the complexity of the program. We therefore limit our scope to what we consider reasonable. The programs which we target, will tend to be simple, self-contained pieces of code such as implementations of abstract data types for an algorithms class or slightly more complex programs. We assume the programs are being run on a modern machine typical of a developer, and 4GB of memory is available to the debugger. Bill Lewis' Omniscient debugger, ODB[18], can store up to 10 million events in a 2GB address space. He states that 10 million events are enough to debug many problems. We would like to at-least match this value.

4.3 Hybrid TTD Design

As the debugger executes a program, it increments a global timestamp known as the `instruction count` for each instruction executed. An instruction is defined as the smallest amount of code that a debugger can stop at. This instruction count is the absolute measure of time used by the debugger to navigate backwards in time.

During forward execution, information is also tracked that allows the debugger to determine when to create a checkpoint. The simplest heuristic is fixed interval check pointing, which creates a new checkpoint when the execution time since the previous checkpoint exceeds a pre-determined interval. To signal a new checkpoint's creation a global checkpoint counter is incremented after being stored in a mapping from checkpoint number to instruction count.

The act of creating a checkpoint is mostly implicit, as checkpoints are created in a copy-on-write manner. Objects in the address space of the program being debugged are instantiated with history logging enabled. This history logging replaces single values with lists of values indexed by checkpoint number. When a value is changed, it is appended to the end of this list only if the current checkpoint counter exceeds the checkpoint number of the last added value. If the last value in the list has the same checkpoint number as the current value then it is replaced by the new value.

This implicit checkpointing mechanism ensures that checkpoints only contain values that have changed, performing copy on write at the object level rather than the page level (as is the case in OS fork). This finer granularity leads to an efficient implementation, as long as the memory required to implement it is not excessive.

Checkpoints must also contain the call stack of the interpreter, as it is required to restore them. Much of the information in a standard stack frame is derivative from the function call that placed the frame on the stack. We therefore only store minimal information. This includes the locals map, file name, function name, line number and

parent frame for every frame on the stack. As future checkpoints may have a call stack that contains a prefix of the current checkpoint, the frames are stored using the same copy-on-write mechanism used to store object history, as shown in figure 4.2. It is important to note, that the locals map of the frame is not copied in any way. Instead a reference to it is stored, which ensures the map will not be garbage collected once the original frame has been popped off the stack. The combination of storing frames and storing the history of variables in local maps enables us to serialize the state of the interpreter.

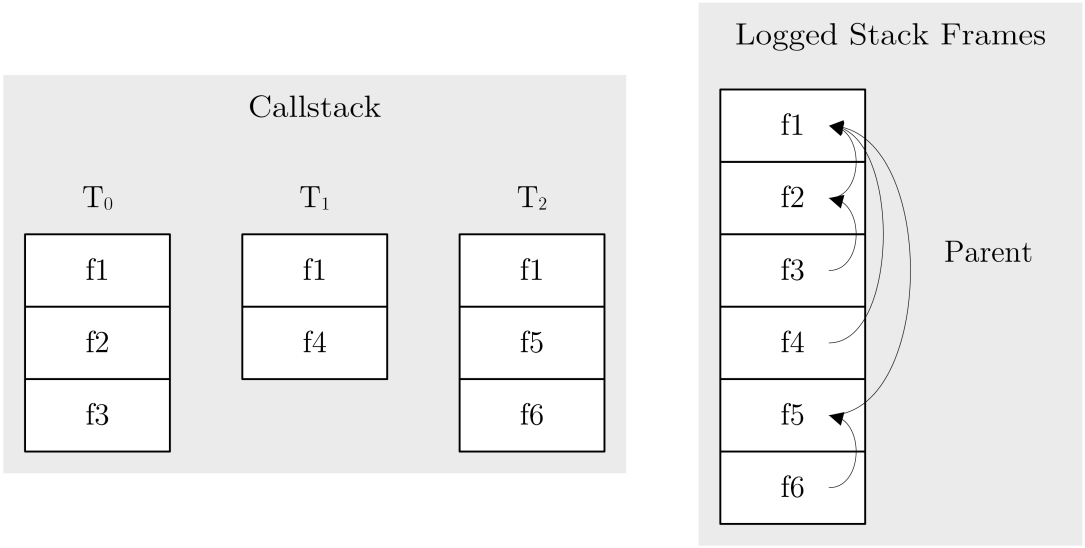


Figure 4.2: The call stack shown at timestamps T_0, T_1, T_2 would be logged as the list of frames shown on the right. This list only stores one entry for each unique frame that existed on the call stack. The parent pointers are used to recreate the state of the call stack on the left. For example starting at f6 and following the parent pointer reproduces the call stack at T_2. Additionally, as the locals of each frame track their own history, there is no problem with only logging each frame once.

When a user wishes to reverse execute, the debugger determines the instruction count to which they would like to execute, and queries the checkpoint list. If a checkpoint exists with the exact instruction count, it is restored. If not, the latest checkpoint with an instruction count smaller than the desired checkpoint is restored, and the remaining 'distance' is re-executed. This process can be seen in figure 4.3

Restoring a checkpoint consists of two parts. Firstly, the call stack is restored. The information necessary to restore the call stack is stored in an internal data structure accessible to the interpreter. Once the call stack has been restored, every value must be restored to the value it had at that time point. As with creating storing value history, restoring history is an on-demand operation. All that is required is for the global instruction counter to be set to the checkpoint's instruction count and a replay mode flag to be enabled. Only when a value is actually requested, is the list of past values traversed to find the correct value. Similarly to how the creation of checkpoints is limited to only store values that have changed, restoration is limited only to values that are accessed.

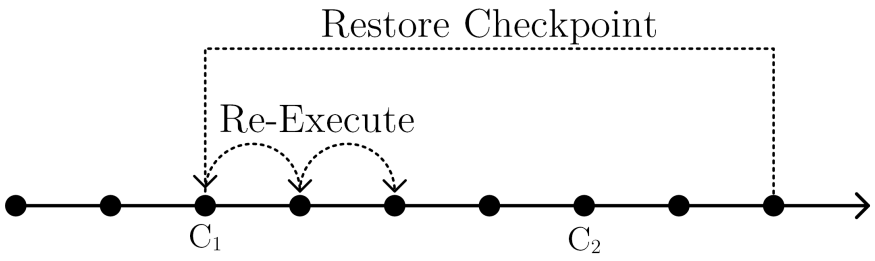


Figure 4.3: To jump back in time efficiently, we restore the closest checkpoint before the timestamp, and then re-execute any remaining steps.

The key feature of the proposed hybrid TTD design is, that it is implemented at the level of the interpreter, which maximizes its efficiency. Since checkpoints are implemented by modifying objects 'in-place' rather than storing a copy of the heap, the runtime overhead of restoring a check pointed value is deferred to the access time. Similarly, the overhead of creating a checkpoint is spread out over all the instructions that are executed between two checkpoints.

Chapter 5

Interpreter Choice

Choosing an interpreter was an important step in the implementation of this project. Rather than spending an unreasonable amount of time writing an interpreter from scratch, we decided modifying an existing implementation would provide a better result given the limited amount of time. However, choosing an interpreter was harder than anticipated, and we ended up experimenting with both Jython and PyPy. In short, both options were attractive because they are implemented in higher level languages, which take advantage of a runtime / virtual machine. Jython was used in our initial experiments. We then switched to PyPy due to an apparent deficiency in Jython, and later returned to Jython after not being able to brave the learning curve of PyPy. The details of Jython's implementation are discussed in section 5.1. PyPy is discussed in detail in section 5.2. A direct comparison between the two is made in section 5.3 and a more detailed account of the experience of working with both implementations is given in section 5.4

5.1 Jython - Details

Jython is an implementation of a Python interpreter written in Java. Version 2.7 of Jython is meant to be compliant with version 2.7 of CPython. In 2.7, the focus was placed on compatibility of Jython with CPython rather than performance improvements. However, any performance improvements in the JVM translates directly to a performance improvement in Jython. For performance reasons, Python code is compiled into Java byte code rather than interpreting each Python instruction one by one. Overall Jython's speed is comparable with CPython, and it's compatibility is quite comprehensive.

5.1.1 Interpreter Design

Jython consists of a Java based interpreter, with the modules of the standard library being written in a mixture of Java and Python code (similarly to how CPython consists of a mixture of C and Python code). Certain modules (such as `sys`) must be written in Java in order to interface with the JVM correctly. Similarly, any modules that are

written in C in CPython must be translated to Java. Additionally, any modules where performance is critical are also written in Java.

The Java based modules make use of a special set of attributes to help facilitate their translation. These attributes (ExposeType, ExposeGet, ExposeMethod etc) are used to specify the public interface of each Java object as a Python module. A useful feature of the attributes is, that the ExposeGet attribute can be used on both fields directly and on methods. This allows Java backing fields to be dynamically converted to Jython objects when they are accessed by Python code. Code can be simplified, as the module can access the backing field without having to worry about any type conversions and there is no need for a second field to track the equivalent information as a Jython object. The attributes also allow type hierarchies to be specified, as Python supports multiple inheritance and Java does not.

As with any Python implementation, Jython must define the built in Python types. Jython defines an automatic mapping between Java primitives and Python primitives when exposed on a Java class. However, in Python everything is an object. The primitives are therefore defined as Java classes, for example the PyInteger class, which corresponds to an int in Python. PyInteger defines methods that are specific to the Python implementation of integers. These include operations such as add. In Python users are not directly concerned with types, so the PyInteger implementation handles integer overflow by returning a long, which can hold the value. If a long can't hold the value, Java's BigInteger class provides immutable arbitrary precision integers.

For complex types such as List, Dictionary or Frames, Jython defines the classes PyList, PyDictionary and PyFrame. Such classes are simple Java classes that expose the fields and methods required by the Python language specification. In many cases, they are backed by equivalent Java classes where appropriate. PyList uses an ArrayList as a backing value.

5.1.2 Generating Java Bytecode

The translation from Python to Java bytecode is performed by walking an AST built when a Python module is imported. The AST is passed to a compiler. This compiler then creates Module, Class, Function and Code objects (as in Python). The properties of the code object are filled in as they would be when Python is compiled under CPython to a Code object, with the exception of the Code object's `co_code` property. This would normally hold the bytes which represent the Python bytecode, however in Jython this information does not exist. The reason for this is, that the Python bytecode means nothing to Jython[1], and it instead writes Java bytecode directly using the `asm` library.

The dynamic nature of Python code is achieved in Jython by way of a lookup table. Each Python module becomes a Java class with a lookup table. Functions and methods are transformed into Java methods, and called using an int passed to the lookup table. This allows functions to be treated as first class citizens.

[1]This is technically not true, as Jython has a module which is able to interpret Python bytecode. What Jython is not capable of is producing Python bytecode, and for this reason the co_code field of the code object is left blank.

5.2 PyPy - Details

The PyPy project consists of two main components, the Pypy interpreter and the RPython translation toolchain. RPython is a restricted subset of the Python language. For example, in RPython functions cannot be defined at runtime, and a variable may not ever contain incompatible types (eg int and object). These restrictions allow static analysis techniques to be applied to the program so it can be reasoned about.

5.2.1 The PyPy Interpreter

The PyPy Interpreter is almost entirely written in RPython. As RPython is a subset of Python, it can be run on top of any Python interpreter, without invoking the RPython translation toolchain to translate the interpreter to an executable. This comes at the cost of speed, but allows for quick testing of interpreter changes as the translation takes roughly 40 minutes on a modern machine. Additionally, as the interpreter is being run within a Python environment standard Python debugging tools can be used to debug it.

PyPy and CPython use similar bytecode and data structures for their interpretation of Python. This is one of the reasons why PyPy can quickly adopt new features and reach parity with the CPython implementation (unlike Jython). The overall architecture of the PyPy interpreter is one of many abstractions. According to Benjamin Peterson, "PyPy's powerful abstractions make it the most flexible Python implementation."[22] The main abstraction is called object spaces. An object space encapsulates the knowledge required to represent Python objects and perform operations on them. This allows the bytecode interpreter to be a simple stack machine, which need only push and pop objects on the stack and call methods on them, without knowing the specifics of handling these objects. For example, the definition of addition shows that the interpreter does not need to inspect the operands, but delegates to the object space. A distinct advantage of such a solution is that new data type implementations can be implemented without modifying the interpreter. Additionally, object spaces can be used to "intercept, proxy, or record operations on objects."[22] These features make PyPy an attractive candidate for implementing an object oriented time travelling debugger OR a trace based TTD. Object spaces appear to be an abstraction that would allow for the implementation of an object oriented debugger. [2]

PyPy provides a standard object space. This level of indirection means that a data type may have multiple implementations. "Operations on data types are then dispatched using multimethods." According to Guido van Rossum, the creator of Python, multimethods are "a function that has multiple versions, distinguished by the type of the arguments"[29], meaning that at runtime depending on the type, the object space invokes a different version of the method. This allows the interpreter to pick the most efficient representation of data, all while being completely transparent to the application. For example, arbitrary sized numerical values can be stored as longs if they are small enough, which is both more memory and computationally efficient.

[2] As a side note, the authors of Practical Object Oriented Debugging published a paper in which they used PyPy to implement a Smalltalk VM in under a week that outperformed the existing interpreter.[4]

PyPy distinguishes between Application level code and Interpreter level code. Interpreter level code must be written in RPython and then translated. It operates on the object space of wrapped Python objects. Application level code is written in Python and run by the byte code interpreter. Some interpreter level code is written in application level code, and there is therefore support for converting between the two at runtime. For example print statements are application level code, but often included in interpreter level code.

5.2.2 RPython Translation Toolchain

The steps of the translation process can be seen in figure 5.1 First, the translator loads the RPython program into its process using standard Python module loading. The translator is written in Python and can make use of dynamic features that the RPython program is restricted from using.

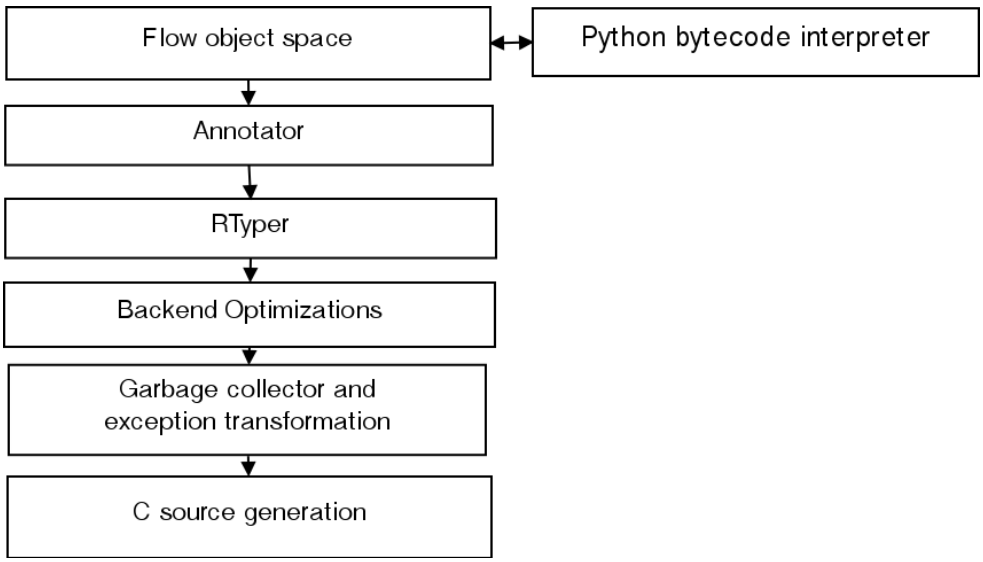


Figure 5.1: RPython translation steps from [22, Fig. 19.1]

Translator uses a process called 'abstract interpretation' to build a flow graph. This is achieved by using a a special object space called the flow object space. The flow object space consists of only variables and constants. Constants are values known during the translation whereas values are not. Constant folding is applied, performing any operation that consists solely of constants. The flow object space records operations the interpreter asks it to perform, without performing them. It records the branches of the conditional control flow, to create a flow graph consisting of blocks of one or more operations linked together.

The annotation phase assigns a type to the results and arguments of each operation Rtyping expands high-level flow graph operations to low-level ones based on type information provided by the annotator. Optimizations are then performed on the low level flow graph. These optimizations include many standard compiler optimizations such as constant folding, dead code removal or function inlining. Before the flow graph can be passed to the back end for translation, it needs some back end specific

transformations. Exception passing must be transformed, and garbage collection implemented, as Python has automatic memory management and low level languages like C do not. In CPython reference counting garbage collection is used. This is not an ideal solution, as it hard-codes the Garbage Collection into the code base, which is prone to human errors. PyPy uses garbage collection transformers to convert the flow graph. This eliminates the chance that a programmer may forget to add references to the garbage collection scheme. Similarly implementing a write barrier based garbage collection is trivial compared to implementing one in CPython. Lastly, the chosen back end generates low level code. In PyPy this is a C compiler.

5.3 PyPy and Jython comparison

Many see PyPy being written in Python (or a subset of it) as a huge advantage from the point of view of a developer. However, writing RPython is not always straightforwards. RPython is not specified formally, and often changes depending on the needs of the PyPy project. Additionally, the RPython translator works on whole programs. Therefore, changes to the PyPy interpreter require the entire interpreter to be re-translated, which can be very time consuming. Tangent to this issue, modules containing RPython code must be included and translated at translation time of the interpreter, and therefore can't simply be added at runtime.

The layers to PyPy's architecture make it easy to implement various garbage collection schemes and perform interesting transformations on the code without modifying the interpreter. However, these layers can make tracking down a bug difficult. Firstly, the bug might not be reproducible on the untranslated interpreter. This means a developer needs to use a debugger on machine generated C code. These levels of indirection can also cost performance.

The speed of PyPy is thanks to its JIT compiler. The intention of the JIT generator is to be very simple, requiring only 2 hints, but these are often not enough, and additional tweaking and hints must be added. In the context of writing an interpreter to support a TTD the JIT would be a challenge to support. Luckily PyPY has the option to disable JIT, and the design benefits of PyPy may prove to be worth the loss in performance.

The main advantage of Jython over PyPy is its accessibility. Building Jython is significantly faster than PyPy and simpler to set up. While some will argue Python is a better implementation language than Java, when modifying a large codebase statically typed languages are often easier to work with. From a personal point of view, I have more experience with Java and am more confident in assessing the effect of my changes.

As Jython runs on the JVM, it can take advantage of features such as memory management, which Barr and Marron place so much emphasis on. The JVM is well known for allowing users to tweak memory management settings and for this reason is a good candidate for the basis of a TTD that uses a persistent data structure for logging. While the start-up time of Jython is generally slower than Python, the runtime speed is comparable. Particularly in situations with hot spots, the JVM's JIT speeds up

Jython performance. This represents a particular advantage over PyPy as the performance of a TTD built in Jython will get the benefit of the JIT without additional work.

While the architecture of Jython is not as clean or versatile as PyPy, Jython's implementation of Python objects does provide a level of indirection in which to place hooks. This is akin to PyPy's object spaces without the ability to easily swap them out. This still makes Jython an attractive option for writing a TTD, although the implementation may not be as clean.

5.4 Interpreter Choice and Experience

The choice of an interpreter implementation was difficult, as the aims of the project shifted over time, as well as the implementation plan. After some initial research, Jython was chosen to perform some feasibility testing and prototyping. The initial features implemented in the interpreter were limited to counting the instructions executed by a program being debugged. This was a fairly simple task. As the experiments also looked at how to implement logging of changes to variables, Jython appeared to be ill-suited, due to the lack of direct bytecode interpretation. At this stage in the project, the Java bytecode generation aspect of Jython was not yet understood by my. This meant that, when stepping through the Jython interpreter running a user program, the generated bytecode was skipped by the Java interpreter and made working with Jython seem impossible.

At this point, it was decided to attempt to reproduce the work done so far in PyPy. This was fairly simple as the majority of code was written in standard Python. As previously explained, the PyPy interpreter can be run as interpreted Python code, which is very slow, or compiled to an executable, which is faster but takes a long time to build. The biggest obstacle with using PyPy was writing code that was valid RPython. Since the interpreted version of PyPy need not be RPython compliant, the feedback loop for determining if code was valid RPython took on average an hour (at best 9 minutes once I learned how to optimize the build commands to strip away everything but the bare minimum). This difficult experience proved to be too much of a disadvantage, and I decided PyPy would not be viable, especially when the project became more complex. The only consolation was, that I was not alone in this. David Beazley, a Python and C veteran renowned in the Python community gave a talk at PyCon US 2012 entitled "Tinkering with PyPy"[3]. In the talk, Beazley describes his failed attempts to understand and modify PyPy using only the documentation available. He gave other similar talks at subsequent conferences.

Finally, I returned to Jython. This was somewhat difficult, as the developer documentation for PyPy was generally quite good (RPython aside) and Jython's documentation was either non-existent or out of date by over five years. However, the time spent using PyPy was not all wasted. Again, the majority of my code was written in plain Python, so I was able to re-use it once the modifications to Jython reached parity with PyPy. More importantly than this code, stepping through PyPy gave me a better understanding of how Python works 'under the hood' and many of the concepts in PyPy could be found in Jython. My experience with PyPy allowed me to understand

Jython's bytecode generation, which had previously stumped me.

Both PyPy and Jython are suitable for writing a TTD, however I believe the accessibility of Jython trumps the architecture of PyPy. Furthermore, as I would be working in a foreign codebase, an implementation written in a strongly typed and less dynamic language is much easier to reason about when making changes.

Chapter 6

Implementation

The design of the hybrid debugger is based on functionality present at either end of the TTD spectrum (as described in figure 4.1). We attempted to implement the Hyrid TTD in 3 steps.

- *tdb* - a re-execution based debugger, see section 6.1
- *odb* - an omniscient logging based debugger, see section 6.2
- An experimental hybrid of the above, see section 6.6

6.1 Tdb: Re-execution Debugger

tdb is a very basic implementation of a re-execution based TTD. It's architecture is broken into modifications to the interpreter written in Java, and Python modules, which exposed those implementations as an interactive debugger. An overview can be seen in figure 6.1.

TdbTraceFunction is an extension of the standard PythonTraceFunction Java class used by Jython to send trace events to a debugger written in Python. TdbTrace-Function implements the absolute measure of time used by Tdb. This information is exposed to the debugger, written in Python, through the modules _tdb.

The base debugger, called TBdb, is a copy of Python's included Bdb. Bdb is Python's base debugger class, which allows a developer to write a debugger simply by overriding 4 methods that correspond to the events traced by the debugger: line, call, return and exception. However, since this implementation is specific to a standard debugger we found it easier to re-write the class and hence TBdb is the time travelling version. TPdb is a copy of Pdb that is specific to a TTD. It extends TBdb as well as Cmd, a basic command line interface. Cmd provides functionality to parse user input and call the appropriate function. TPdb provides the functionality of the various user debugger commands. The core of TPdb is a loop, which processes user input. If the user enters a command that requires travelling back in time, a Re-Execute exception is thrown, which signals that the program should be restarted and executed uninterrupted to

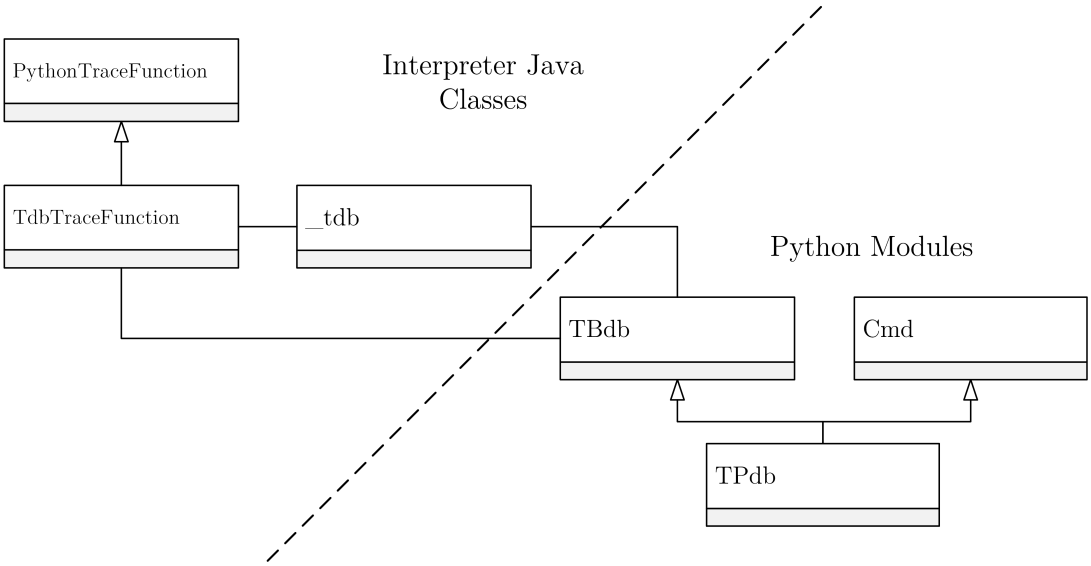


Figure 6.1: Architecture of Tdb debugger

the desired instruction count. An overview of this control flow can be seen in figure 6.2

6.1.1 Absolute Measure of Time

As mentioned before, an absolute measure of time is a key component of a TTD. The measure of time used in *tdb* is the number of instructions executed. For *tdb* we define a single instruction to be the amount of code executed by a step instruction in pdb. This creates a one to one mapping between the steps at which a program can be halted in pdb, and an instruction count in *tdb*. The time travelling debugger will therefore only be able to stop at precisely the same points in execution as pdb. This is good, as we would like the granularity of the measure of time to be as coarse as possible while still allowing precise debugging.

TdbTraceFunction implements this instruction count as a global variable. The instruction counter is incremented whenever a trace event is dispatched to the debugger attached to the TdbTraceFunction. This allows us to use the presence of a trace function as a filter to ensure only instructions in user level code increment the counter. Similar to other TTDs, *tdb* also skips any tracing in the standard library. This avoids tracing potentially a large amount of instructions that the user has not written.

The motivation behind implementing the measure of time as part of the interpreter is for maximum performance. A standard debugger is able to mitigate some of the overhead of tracing functions, by not tracing functions that are stepped over. This is done by removing the trace function from a frame that a user has decided to skip. This is not possible in a TTD, as the trace function provides the means of implementing an absolute measure of time.

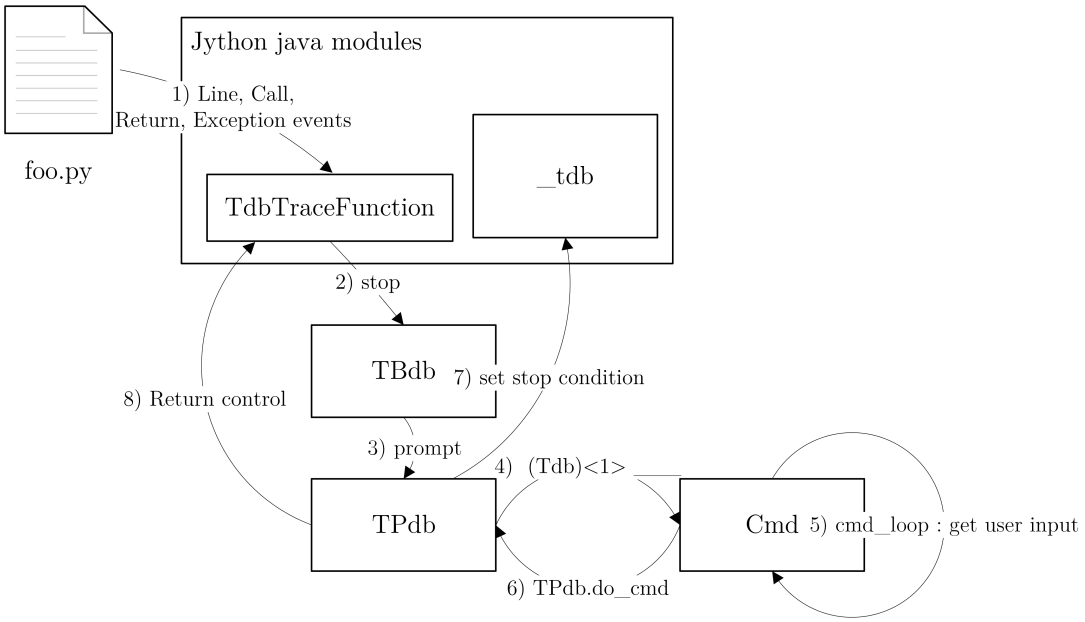


Figure 6.2: (1) the script being debugged is run by Jython, which attaches the TdbTraceFunction. The TdbTraceFunction recieves line, call, return and exception events (2) TdbTraceFunction runs the program until the stop condition is met, and then calls TBdb (3)TBdb stops and calls to TPdb to prompt the user (4) TPdb presents a prompt to the user and (5) calls to the CMD module to run the cmd_loop and wait for user input. (6) CMD passes back to TPdb a valid command, calling do_cmd with the cmd. (7) Tpdb sets the next stop condition using the _tdb module. (8) TPdb returns control to the TdbTraceFunction, which continues to run the program

6.1.2 Forward Execution

A standard Python debugger based on Bdb performs forward execution by comparing the frame of the current trace event to a stop condition. The stop condition consists of a stop frame and a return frame. However, when reverse executing, it is not possible to compare frames that haven't been created yet. For this reason, *tdb* uses an approach based on Boothe's "Efficient algorithms for bidirectional debugging"[5] described in section 3.1.4. This approach uses an instruction counter as well as a call depth counter to allow for controlled debugging in both directions.

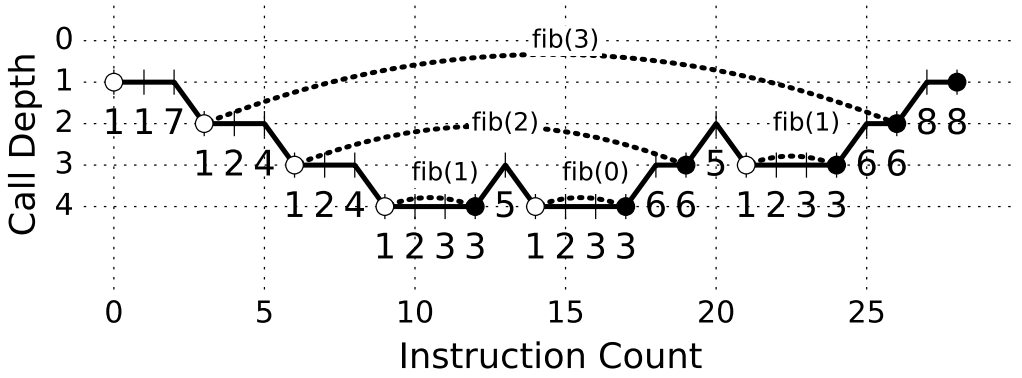
The stop condition consists of three values, a stop instruction count (*stop_ic*), the stop call depth (*stop_depth*) and a stop event (*stop_event*). A negative depth value is used to denote that any depth is acceptable. A stop instruction count is always provided as it can be generated from any instruction count by incrementing or decrementing the current instruction count depending on the direction of travel. Lastly, the stop event is a string which stores the event type. The debugger will only stop on an event type which matches the stop event (if the stop event is not null). The basic stop condition is as follows:

```
if (stopIc >= 0 && instructionCount >= stopIc) {
    if (stopDepth == -1 || stopDepth >= callDepth) {
        if (stopEvent == null || label.equals(stopEvent)) {
            ... // stop and interact with user
        }
    }
}
```

To help explain the semantics of debug navigation commands, we will use the following simple program:

```
1 def fib(n):
2     if n <= 1 :
3         return n
4     f1 = fib(n-1)
5     f2 = fib(n-2)
6     return f1 + f2
7 f = fib(3)
8 print f
```

The program has been simplified, for the purpose of improving the clarity of examples. Specifically, lines 4,5 and 6 could be combined into a single line. This program can be represented by the call graph shown below:



Each vertical tick represents an **execution point** in the program. These are all the points at which a debugger can pause execution. Below each tick is the line number of the instruction being executed.

The y axis represents how many calls deep the current instruction is. [1]

Open circles ∘ denote a call event. A call event is not the event that calls a function, but rather the first event in the new function. Note therefore, that a call event will always be preceded by a downward sloping line, from the location of the function call. In the instance of the program above, there is only one function, so all call events have line number 1. However, the call location varies. For example line 7: `f = fib(3)` or lines 4 and 5: `f1 = fib(n-1)`, `f2 = fib(n-2)` Instruction 0 is marked as a call, as it is the entry point into the program.

[1] Note, the call depth starts at 1 as side-effect of how the debugger is invoked. The last instruction that returns to call depth 0 represents the program exiting

The program computes the third Fibonacci number (0 indexed) in a recursive fashion. This can be expanded as: `fib(3) = fib(2) + fib(1) = (fib(1) + fib(0)) + fib(1)` These calls are labeled on the graph above with dotted arcs.

Closed circles • denote a return instruction. In this example, the fib function returns on line 3 or 6. The program exits on line 8, which is recorded as a return event by the debugger.

In later examples, a diamond ⋄ will denote the start position of the debugger.

Step Command

The step command executes the next instruction, possibly stepping into a function to do so. The counter based stop condition is `stop_instruction_count = current_instruction_count + 1` and `stop_call_depth = any_call_depth`. In *tdb*, any call depth is flagged by a -1.

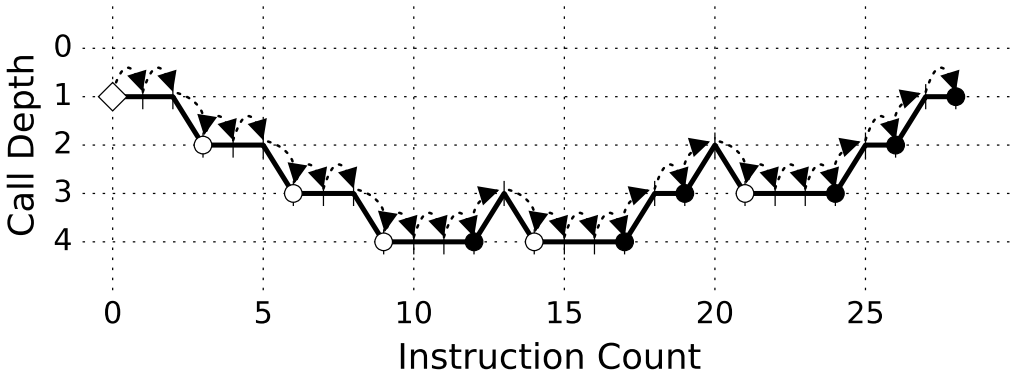


Figure 6.3: Starting at the first instruction, repeatedly calling the step command will navigate to every execution point in the program.

Next Command

The next command executes the next instruction, stepping over any functions that might be called. This is analogous to setting a stop condition where the instruction count is greater than the current count, and the call depth is less than or equal to the current call depth.

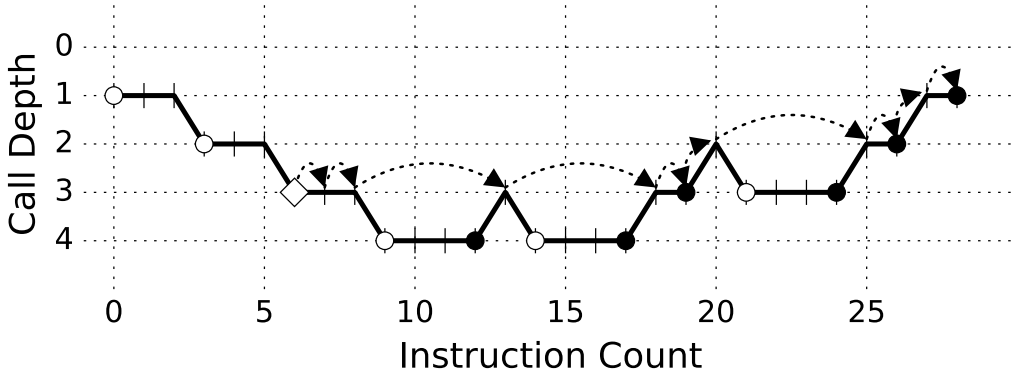


Figure 6.4: This graph shows the next command being called 10 times, starting at instruction number 6. Note that at instruction 8 we do no step into the function, but rather to the instruction after its return event. At instruction 18 we step up and out of a function. After stepping out of the function, we continue stepping at call depth 2, skipping the function at instructions 20-24.

Return

The return command executes to the return of the current function. The debugger stops at the next return event where the instruction count has been increased and the call depth has been decreased. There is one exception to this rule, which is that if the current instruction is a return instruction, then we simply stop at the next event.

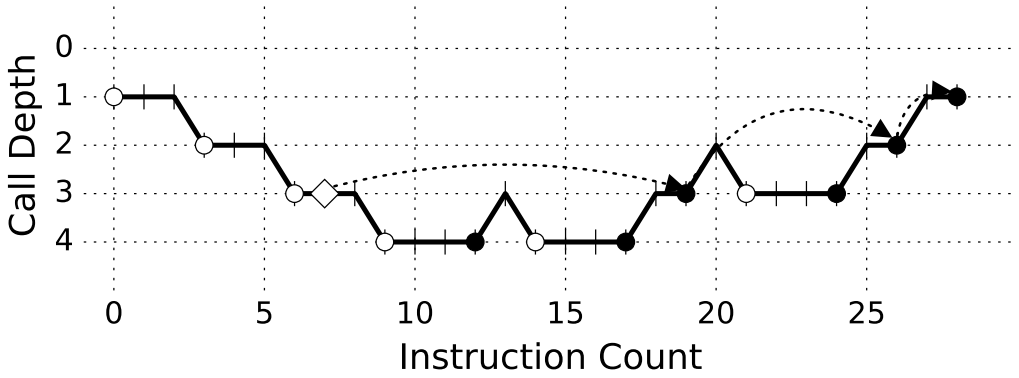


Figure 6.5: This graph shows the return command being called 3 times starting at instruction 7. The first return jumps to instruction 19, which is the return event that corresponds to the call at 6. Any of the events at call depth 3 between instructions 6 and 19 (ie 6,7,12,13,17,18) will return here.

Continue

The continue instruction, executes until a breakpoint is reached or the program terminates. Handling breakpoints in forward execution is very simple. Breakpoints are indexed by a file name, line number pair. For each trace event received during forward execution, the debugger checks the stop condition as usual, and additionally stops if the current event's line and file name match any of the breakpoints.

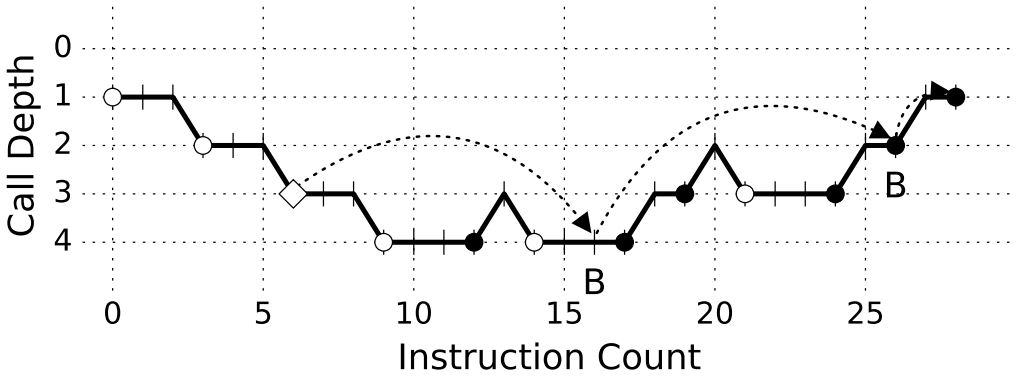


Figure 6.6: This graph shows the effect of running the continue command until the program exits. Notice the two breakpoints denoted by a **B**. If no breakpoints were set, the program would jump the the last execution point without stopping, however since there are breakpoints, execution is caught.

6.1.3 Reverse Execution

Reverse execution is performed in name alone, *tdb* actually performs reverse execution by forward executing up to the desired instruction count. This is achieved by throwing a Re-Execute exception, which breaks execution. The exception is caught in the main loop, which enters the debugger into 'replay' mode. In replay mode, the prompt is disabled until the desired instruction count is reached. As far as the user is aware, the program has jumped back in time to desired instruction, giving the illusion that the program traveled backwards in time.

RStep

RStep, similarly to its regular counterpart is the simplest command to implement. The debugger is set to re-execute and stop when it reaches the current instruction count minus one.

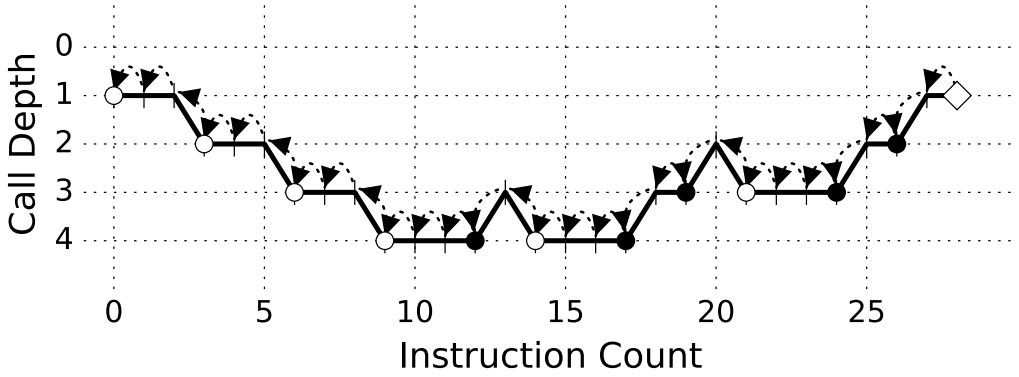


Figure 6.7: RStep navigates to each execution point in reverse order.

RNext

The RNext command steps backwards through the program, stepping over any function returns back to the call of the function. RNext is a bit more involved to implement. We stray from Boothe's implementation, as it requires additional passes. We instead trade-off execution time for memory usage. The instruction count of calls are stored in a stack. Every call event, the instruction count is added to the stack, and every return event the instruction count is popped off the stack. To determine the instruction count that we should jump to after an RNext command, we look at the previous event. If it is a return event, we jump to the instruction specified by the top of the stack. Otherwise, we perform an RStep, stepping backwards a single instruction.

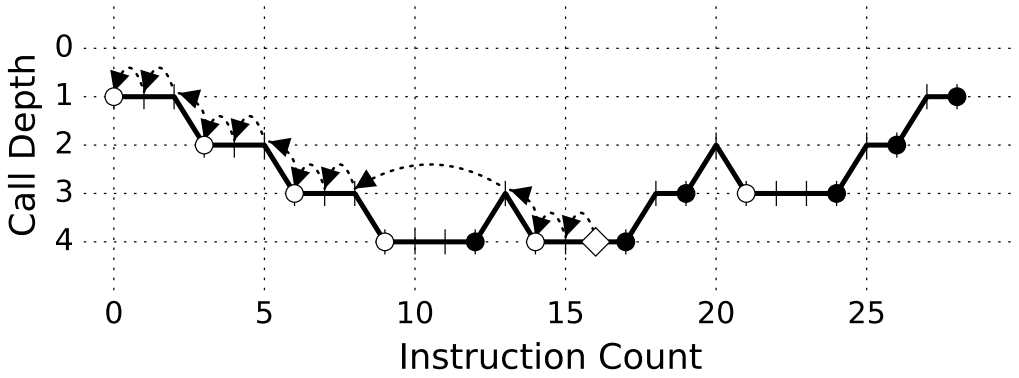


Figure 6.8: This graph shows the effect of calling RNext 12 times starting at instruction 16. RNext moves backwards over the execution points. If it encounters a return instruction, if jumps to the instruction before the function call, skipping the functions execution. For example at instruction 13, rnext jumps to 8 instead of entering the function at depth 4.

RReturn

RReturn jumps to the call location of the current function, ie the instruction before the debugger received a call event. The implementation for RReturn follows from RNext. We peek at the top of the stack to determine this call instruction count, and jump to it.

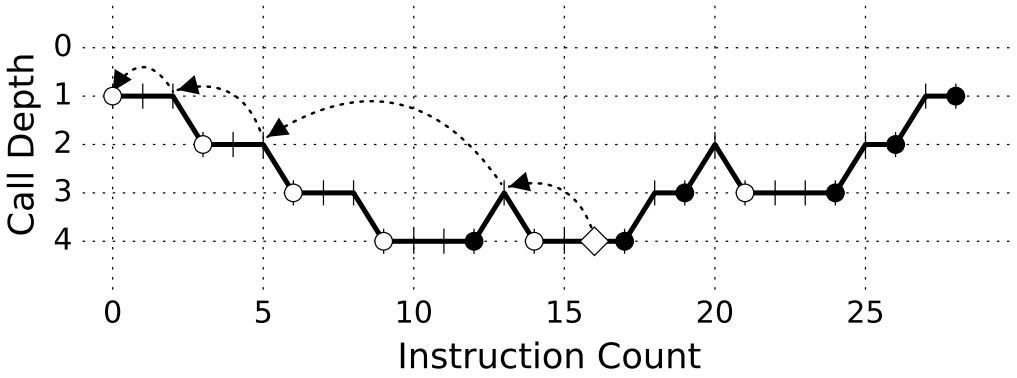


Figure 6.9: This graph shows the effect of executing the RReturn instruction 4 times from instruction 16. The location reached after calling RReturn is the first execution point before the previous call ∘ at the same depth as the current instruction, ie the call location.

6.2 Odb: Omniscient Debugger

The omniscient debugger (*odb*) is a time travelling debugger written in the style of a pure logging TTD. The goal of this debugger is to provide logging capabilities that could later be transformed into a mechanism for efficiently creating checkpoints. Odb must log enough information to recreate the state at any point in execution that a debugger would be able to stop at to provide a seamless debugging experience.

Odb is implemented in the *odb* module, which is backed by the Java modules in the *_odb* package. Additionally, when the interpreter detects that the *odb* module is actively logging the execution of a program, certain data types are switched to a logging mode. These data types include the default implementations of lists and maps, which are used to store object attributes, and local variables.

6.2.1 Event Logging

odb must log enough information to re-create the experience of debugging a program. To do this, *odb* hooks into the interpreter in the same manner as a regular debugger, through `sys.setodbtrace`, a specialized version of `sys.settrace`. However, unlike a typical debugger, *odb* only uses the trace function as a flag to determine which frames belong to the program being debugged. The OdbTraceFunction is a Java class, which is

attached as the trace function for any frames in the debugee. OdbTraceFunction passes trace events (of type Line, Call, Return, Exception) and their associated information to the *odb* instance for logging. Besides this logging, the debugee program is allowed to execute without interruption.

The Line, Call, Return and Exception events that are generated throughout the execution of a debugee program are stored in the 'events' list. Similarly, information about every frame that appeared on the stack during the execution of the program is stored in the 'frames' list.

A standard event consists of a timestamp, line number, and a frame index. The index points to the position of an OdbFrame object in the frames list. This frame object is the recorded information that corresponds to the Python stack frame on which the event was executed. An event is encoded as a long value as shown in figure 6.10. The first 32 bits make up the frame index, which holds the index of the frame in the frames list. The next 32 bits are split, with the type taking up the last 2 bits and the line number using the remaining 30. To convert between the constituent components of the event and the long representation, standard bit manipulation techniques are used. This approach is much more memory efficient when compared to using a Java object, as an object requires at least 2 words for its header, 3 times the size of a long. However, this efficiency is only realized if the long is stored as a true primitive and not a boxed long, as discussed in section 6.2.1.

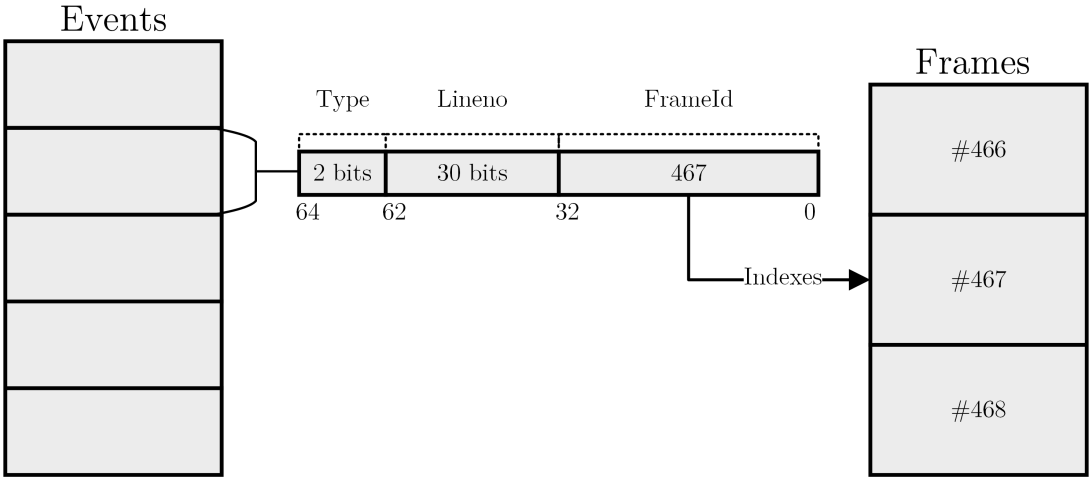


Figure 6.10: The layout of an event encoded as a long

The OdbFrame object is significantly larger than an event, occupying 72 bytes per instance. This larger size is a result of a trade-off. By storing more information in the frame objects, the event objects can be smaller. This is particularly effective, since many events from the same frame share information.

Index `int` This field stores the index of the current frame object in the frame list. This allows for quick retrieval of frames adjacent to the current frame.

Parent `OdbFrame` This field stores a pointer to the parent of the frame object. During execution, this would be the previous frame on the stack, however since all frames are stored in the frames list it is possible that a frame's parent will be

multiple frames above it. This field helps facilitate the next, rnext, up and down commands

Name `String` To re-create the debug experience, we store the name of the function being executed as it is not always possible to retrieve this from source code. This value is extracted from the code object of the frame being logged. By storing this value on the frame, the events need not duplicate this storage. Regardless of whether the event or frame stores the file name, duplicate file names are not duplicated in memory as Java interns these strings.

Filename `String` Just as with the Name field, the file name is stored for the purpose of recreating the debugging experience and reducing the amount of information that needs to be stored by individual events.

Lineno `int` The line number where the function was called from. This would appear to be extraneous information, as we have a reference to the Call timestamp, however the Call event is one event too late. Similarly, the line number of the event at `events[frame.timestamp -1]` is not always correct as a single line may have multiple calls OR a return event may be generated due to an event.

Timestamp `int` This timestamp corresponds to the time at which the frame first existed, ie the timestamp of this frame's call event. It is always true that `frames[events[i].frameid].timestamp == i` The timestamp field is used to quickly jump back to the call of a given function in the 'rreturn' command.

Return Timestamp `int` This timestamp indexes into the events list to the return event for this frame.

Return value `PyObject` As return a value is not necessarily stored as a local before it is returned, we ensure to store it if it exists.

Locals `HistoryMap<String,PyObject>` The locals field is a reference to the locals dictionary used in the execution of the program. As this dictionary has been instrumented to store history, a reference is sufficient. Essentially, the reference is keeping this locals dictionary from being garbage collected, as the standard PyFrame frame objects are garbage collected when they are popped off the stack.

Initializing the Top Frame

As we don't know whether the first event will be a Line, Call or Exception event, the creation of any of the events needs to ensure that *odb* is set up for logging. The `initializeParent()` method creates the first frame in the frames list. This has two purposes. The first is to initialize a pointer to the globals maps. Secondly, by adding an additional parent frame, we guarantee that all frames that are created as a result of a call event will have a parent frame. This reduces any edge cases we need to handle.

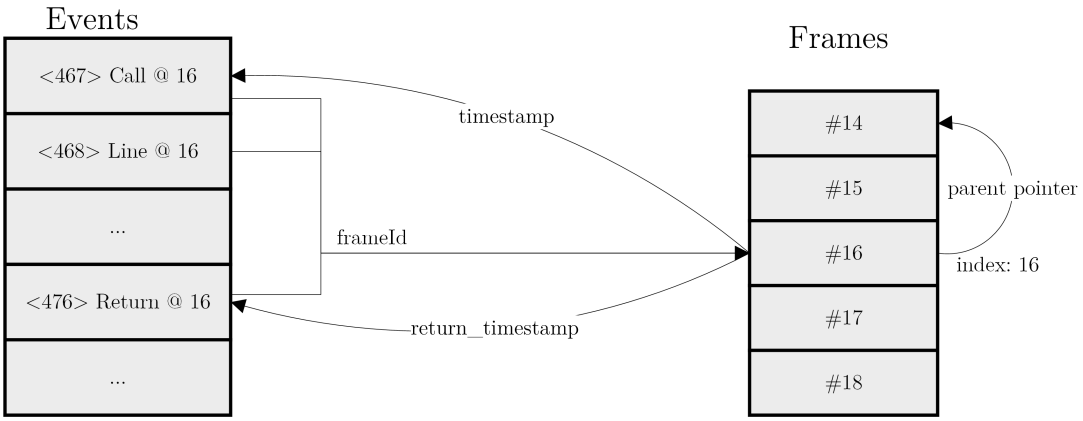


Figure 6.11: Each OdbEvent indexes into the frames list via the frameId value. OdbFrames in turn index into the events list to access the call event at `events[frame.timestamp]` and the return event at `events[frame.return_timestamp]`. Each frame's `parent` field points directly to its parent frame in order to allow for call stack restoration. Lastly, each frame stores its own index in the frame's list in the `index` field to facilitate relative movement in the frame stack.

Line Event

A line event is the simplest form of event that *odb* logs, and requires only a frame as input (in addition to access to the global timestamp). Logging a line event is as simple as encoding the event type, line number and frame id in a long, and storing it in the events list. Finally, the global timestamp is incremented. As no new frame is added to the frames list, the global frame index doesn't need to be modified.

Call Event

A call event creates not only a new event, but also a new OdbFrame. First the new OdbFrame is created, and populated with the necessary information from the PyFrame (the frame used by the interpreter). This frame is then appended to the frames list, and the *currentFrameId* counter is incremented. This counter is necessary to efficiently track the current OdbFrame, as the current frame is not always the last frame in the list. Once the new frame has been created, a call event is encoded and added to the events list. Finally, the global timestamp is incremented.

Return Event

A return event completes the logging of the information stored in an OdbFrame. In particular, the return timestamp and value are recorded. As the parent frame may not be the previous entry in the frames list, it is explicitly recorded in the OdbFrame object. The parent is then used to update the *currentFrameId*

Exception Event

An exception event is created in almost an identical manner to the line event. The line number, frame id and event type are encoded as a long and stored in the events list. Before the global timestamp is incremented, an additional exception object is created, and stored to the exceptions list. This exception object stores the type, value and traceback (if present) of an exception along with the event's timestamp.

Performance considerations

The performance of a logging TTD relies on two factors, the memory footprint of the stored information and the overhead of creating and traversing said information.

One of the most important considerations is the decision of the data structure used to store the event history. The event history consists of events, which are indexed by a monotone continuous integer value. That is to say, the natural numbers in order, with no missing values. Events are only appended to the list. The append operation should therefore be fast and efficient, as the appends are happening during the execution of the program and affect the runtime overhead of the debugger. Preferably the data structure should not require resizing and copying a backing array to append data past an initial capacity, as such a scheme performs poorly for large amounts of data. Fast traversal and random access are also desirable qualities.

A linked list meets half of these requirements, as it appends quickly and is ideal for quick traversals. However, a linked list is not particularly memory efficient, using 24 bytes per entry and has poor random access performance. An array list is slightly more memory efficient, although the append operation suffers in performance at larger sizes. To satisfy all the criteria, the Big List[33] implementation was chosen. Big List is a list that is optimized for lists with more than one million elements. It is essentially a tree with 1000 element arrays at the leaves. The tree structure allows for quick indexing, and efficient appends. Additionally, the memory overhead of a Big List is low compared to a linked list. As previously explained, events are stored as primitive long values. BigList has a special long implementation called LongBigList, which stores unboxed longs, maximizing performance in terms of memory and runtime. For these reasons LongBigList was chosen to store event history.

6.2.2 Logging Variable History

Logging variable history is achieved by adding an extra dimension to each variable. A single variable is represented by a HistoryValueList. This HistoryValueList represents the value of a given variable at various points in time. Each entry in the list is a HistoryValueList, which consists of the timestamp, and the value the variable had at that timestamp. The list is ordered by timestamp, and values are only appended to the end. That is to say, there is no history re-writing, so only monotone increasing timestamps are valid. However, if a new value is appended with the same timestamp as the top value, the top value is simply replaced. This is the copy-on-write property of HistoryValueList. Variables are only logged to a HistoryValueListwhen if their value is changed, as a result the get operation of a HistoryValueList must 'interpolate' values.

If a value is requested at a timestamp which is not present in the list, the value with the largest timestamp below the requested timestamp is returned. This can be seen in figure 6.12.

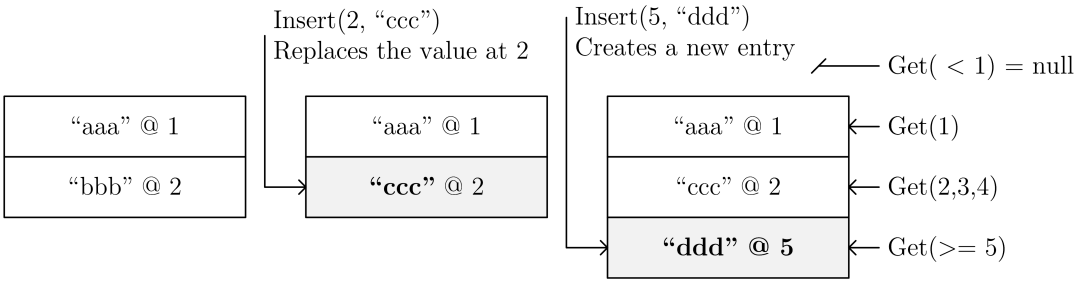


Figure 6.12: Operation of a HistoryValueList with string values

The properties of a HistoryList mean that it can be implemented either as a Map where the timestamp is the key or any ordered collection. The most common operation will be an append or peek operation. Additionally, the size of each HistoryList is highly dependent on the variable it is backing. Therefore, the implementation should be performant in terms of memory and runtime for all list sizes, with emphasis being placed on low memory usage. A linked list satisfies many of these properties, but has a relatively high memory overhead. A hash map is another possibility, however expensive re-hashing is required when the backing array's capacity is exceeded. Additionally, Java's implementation of an ordered hash map uses a linked list, which adds to the memory overhead. Instead, a simple ArrayDequeue is used. ArrayDequeue is backed by an array, which is re-sized using Java's efficient Array copy method.

Thus far we have described a mechanism for storing the history of a single variable. What has not been explained is how to substitute this representation of a variable into a program so that all accesses and modifications of the variable are redirected to this HistoryValueList. The implementation is similar to Leinhard's object aliasing solution described in section 3.1.8. This is possible, because just as in Smalltalk, all values in Python are objects. More importantly, local and global variables in Python are stored in a dictionary. In the Jython implementation, locals and globals are stored in a PyStringMap, which is a dictionary with performance optimizations for key values that are Strings of PyStrings. PyStringMap as well as the more general PyDictionary are both backed by an object that implements the Java Map interface. By replacing this standard map with one that makes use of HistoryValueLists, we can log variables. The replacement of the backing map is performed during instantiation. We have chosen to restrict *odb* to run on a program from beginning to end without disabling the logging. Therefore, any frame or object that is created in a script being run by *odb* will know at runtime whether it should enable logging. This provides the benefit of not requiring a conditional to check whether logging is enabled as well as its overhead. An overview of this implementation can be seen in figure 6.13

HistoryMap

HistoryMap is a pseudo implementation of the Java Map interface, which is backed by a Map of a generic type key to HistoryValueList. HistoryMap implements size,

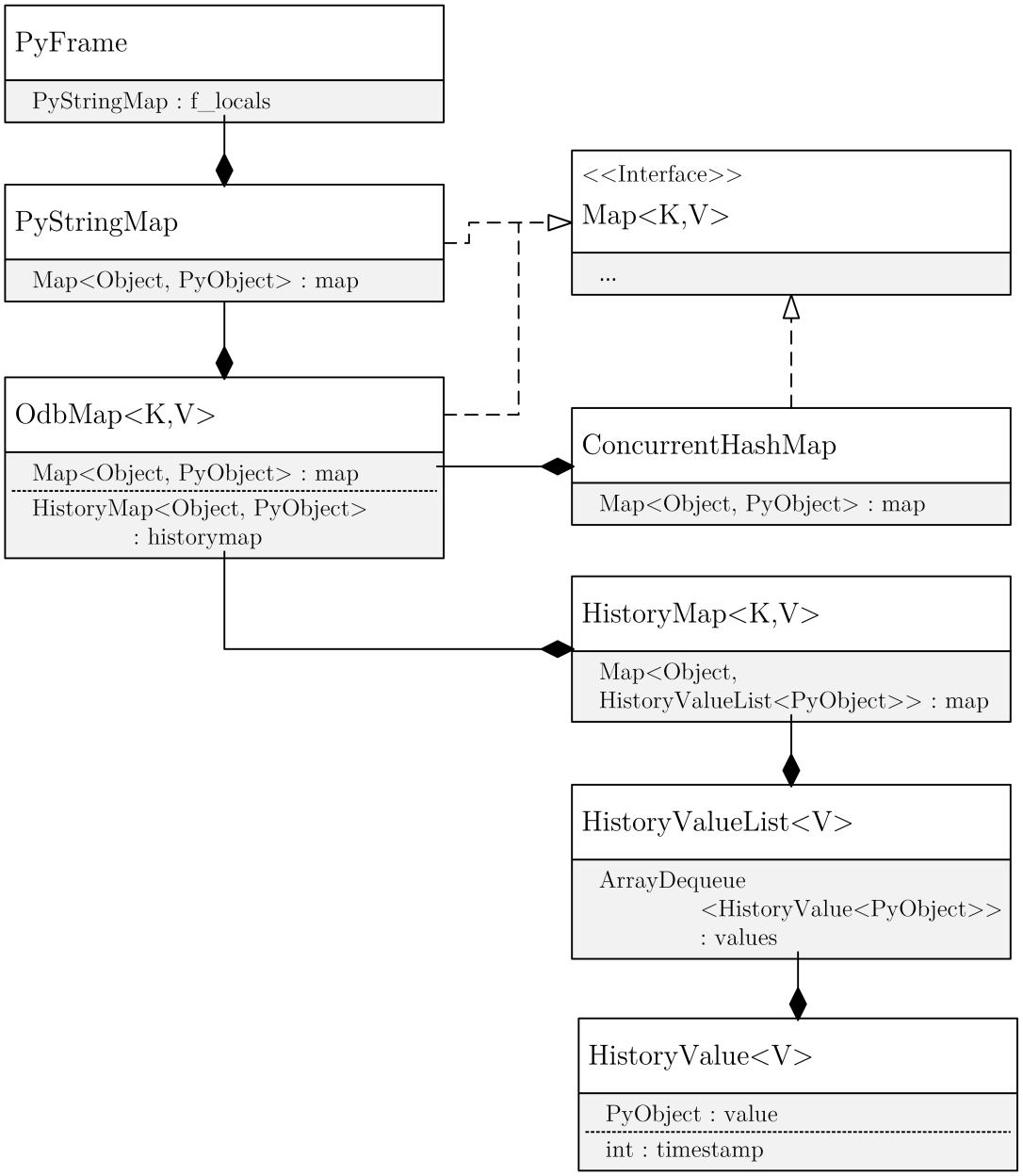


Figure 6.13: Rough UML diagram of how locals in a PyFrame object are logged using an OdbMap backed by a HistoryMap

get, put, remove, clear, contains and putall with an additional timestamp argument. The timestamp parameter is used to call methods on the HistoryValueList elements of the map. As the size of the map will change overtime, and we would like to refrain from doing an expensive traversal to determine the size, we store the size as a HistoryValueList of integers. When an element is removed, we insert a `null` into the HistoryValueList. A null flag is sufficient, as Jython uses PyNone to represent Python null values.

The steps for *put* are as follows

- Check that key and value are both not null
- Lookup the key in the map to get the HistoryValueList
- If the key didn't exist, put a new HistoryValueList with the given timestamp and value into the map
- Otherwise insert the new timestamp and value in the HistoryValueList
- Increment the size HistoryValueList

The steps for *remove* are as follows:

- Check that key is not null
- Lookup the key in the map to get the HistoryValueList
- If the key doesn't exist return
- Otherwise insert the new timestamp and a null in the HistoryValueList
- Decrement the size HistoryValueList

An example of how a history map stores changes to a map see figure 6.14.

```
0 : {“a” : “aaa”, “b” : “bbb”, “d” : “ddd”}
1 : remove(“d”)
2 : put(“c”:“ccc”)
3 : put(“b”:“BBB”)
4 : put(“d”:“DDD”)
```

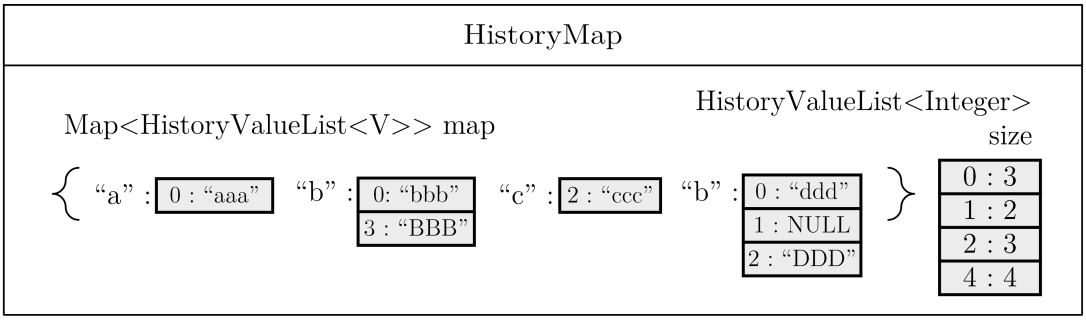


Figure 6.14: The following shows the state of a HistoryMap after performing the operations listed above at the corresponding timestamp. Note that the size HistoryValueList does not contain an entry for timestamp 3 as its value would also be 2.

OdbMap

OdbMap is an implementation of the Java Map interface, which uses HistoryMap as a backing object. OdbMap is implemented using a delegate pattern, with a slight twist. In addition to the HistoryMap, a standard Java map object is also used to provide fast lookups at runtime. OdbMap delegates all write operations to both maps. If the interpreter is executing forward through a program, the OdbMap delegates read operations to a standard map. Once normal execution has finished, and replay mode has been enabled, read operations are delegated to the HistoryMap using the global odb timestamp. To provide a memory efficient implementation of EntrySet, KeySet

and ValueSet, OdbMap creates the desired collection dynamically using an iterator rather than creating a new collection.

6.2.3 Logging Everything Else

Once logging has been implemented for dictionaries, the majority of logging has been handled.

Object fields/attributes Python object instances store attributes in dictionaries, allowing us to re-use the OdbMap to log history.

Lists We implemented lists in a similar manner to maps. Again, the special case was removing objects, which in this situation required shuffling elements in addition to inserting a null value.

Tuples Tuples are immutable and as such need no work.

Arrays and other built in types Arrays would be simple to implement now that lists have been implemented, but it was decided place efforts elsewhere in the project as implementing further collections and objects is not particularly challenging but very error prone.

Stdout The output of a program plays an important role in debugging. Therefore, it is important to also ensure that *odb* logs it. *odb* replaces `stdout` with an instance of OdbStringIO. To ensure that `stdout` logging is in sync with the rest of *odb*'s history, OdbStringIO it is logged using a HistoryValueList of strings. These saved values are then fetched whenever *odb* navigates forwards over a timestamp that OdbStringIO associates with a string.

6.3 ODB operation

Navigation commands in ODB are fairly straightforward to implement as all the information required is present in either the events or frames list. Rather than reuse the call graph diagrams from the *tdb* sections 6.1.2 and 6.1.3, we have instead provided a layout of the same *fib3.py* program as stored by *odb*. This diagram can be seen in figure 6.15. Examples in this section will reference this diagram.

6.3.1 Step

To step, *odb* increments the `currentTimestamp`, and updates the `currentFrameId` by decoding the long at `events[currentTimestamp]`.

The step command can be visualized in figure 6.15 as moving down the events list one item.

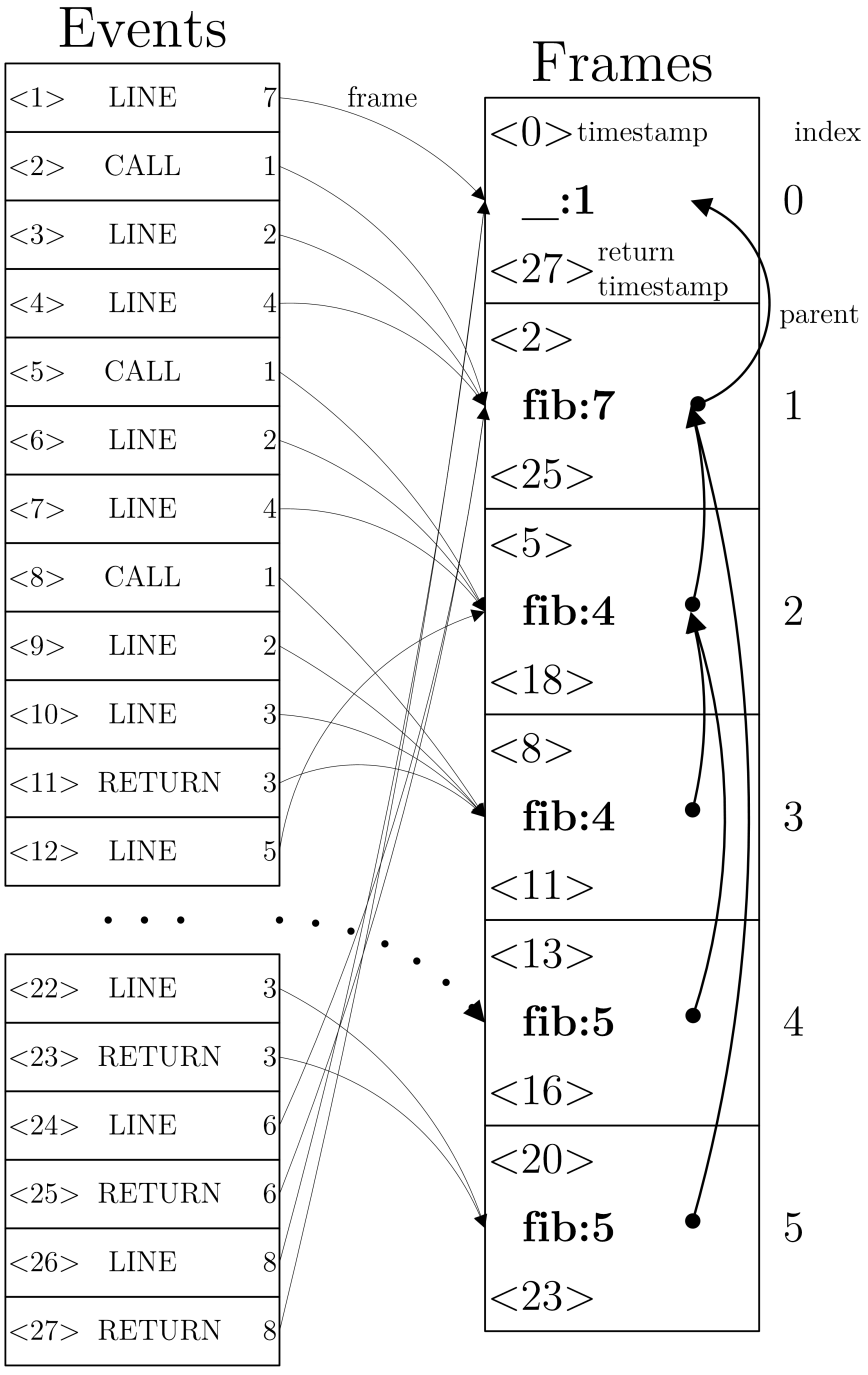


Figure 6.15: This diagram shows the contents of the 'events' and 'frames' list after *odb* has executed the *fib3.py* program. Events are structured as `<timestamp> EVENT_TYPE linenumber`. Each event's timestamp is also its index in the 'events' list. The arrow shows which frame the event points to by its `frameId` value. From top to bottom, each frame contains its `timestamp`, `methodname:linenumber`, `return_timestamp`. The arrow indicates the parent of the frame, and the number on the right its index in the 'frames' list.

6.3.2 Next

The next instruction performs a linear search on the events list, starting at the index given by the `currentTimestamp`. It stops at the first event, which has a frame index equal to the current frame index OR a breakpoint that matches the event's line number. If there are no breakpoints set, we can take advantage of the structure of the events and frames. Since next does not enter into calls, when we encounter a call event we lookup the corresponding `return_timestamp` from the event's frame and continue the search from there.

For example, let us assume *odb* is currently at event 1, with no breakpoints set, and we called the **next** command. *odb* would inspect the next event, 2, which is a call event. Therefore, it looks up the frame that event belongs to, frame number 1, and jumps to the corresponding `return_timestamp` (25) plus 1, 26. From figure 6.15 we can see, that the events between 2 and 25 inclusive all belong to a child of frame 0. Next doesn't step into functions, so we would like to ignore these events, which is exactly what jumping to 26 achieves.

6.3.3 Return

Return has a special case. If the current event is a return event, then the debugger simply performs a Step command. If no breakpoints were set, the return instruction could simply set the `currentTimestamp` to `frame.return_timestamp`. However, since we would like to support breakpoints, it must perform a linear search on the events list from `currentTimestamp` to `frame.return_timestamp`, stopping if the line matches one of the breakpoints for this file.

For example, we will again *odb* is currently at event 1, with no breakpoints set, and we have called the **return** command. Since no breakpoints are set, *odb* looks up the frame that event 1 belongs to, frame number 0, and jumps to the corresponding `return_timestamp`, 27. Figure 6.15 clearly shows, that event 27 is a return event that belongs to frame 0.

6.3.4 Breakpoints

Breakpoints are handled by the *BreakpointManager* class. The breakpoint manager contains a set of breakpoints, as well as an index. The index acts like a unique key in a database, incrementing for each breakpoint added to the set, but never decrementing. Breakpoints contain fields to store their index, file name and line number. Breakpoints in the set must be unique in terms of file name and line number. When clearing a breakpoint, they can be looked up by either the index number or file name line number pair. The main function of the breakpoint manager used by the other debugger commands is, the *getBreakpointLinesForFile* method, which returns the line numbers that have a breakpoint in a given file. This method is used frequently in the methods *firstBreakpointBetween* and *lastBreakpointBetween*. For a given start and end point, *firstBreakpointBetween* performs a linear search over the events. If the file and line number of an event matches a breakpoint, that event's timestamp is returned. If no

breakpoint is hit, the end timestamp is returned. *lastBreakpointBetween* performs the same search in reverse.

6.3.5 Continue

If there are no breakpoints, continue jumps to the last event. Otherwise, continue performs a linear search on the events list, stopping at the first breakpoint. Continue is implemented as a simple combination of the jump instruction and *firstBreakpoint-Between*.

```
int do_continue(){
    int start = OdbTraceFunction.getCurrentTimestamp();
    int end = OdbTraceFunction.getEvents().size()-1;
    return do_jump(firstBreakpointBetween(start, end));
}
```

6.3.6 Reverse Execution

The reverse commands are almost all identical to their forward execution counterparts, with the exception of the order of linear search being reversed.

6.4 Other Debug Commands

Where displays the call stack of the current function. To re-create the call stack at any point in time, *odb* simply traverses the `parent` pointer of the current frame until a `null` value is reached.

Up moves up the call stack one frame. Similarly to where, *odb* achieves this by setting the current frame equal to the frame pointed to by its `parent` field. The difference between *odb* and a standard debugger is, that navigating up the call stack jumps to that point in time, so one can continue to step through the program after moving up the call stack.

Down moves down one frame. As there are potentially many children for each frame depending on the point in the program, a child field is not stored in an OdbFrame. Instead, as this is an uncommon operation a linear search through the frames list is used to find the next frame that has a parent equal to the current frame.

Default prompt The default prompt allows the user to type in an expression. The expression is evaluated in a copy of the locals and globals at the current timestamp.

6.5 Omniscient Debugger Specific Commands

The all-knowing nature of an omniscient or logging debugger allows for some additional commands to be defined

6.5.1 NextF, PrevF

These are the frame commands, which allow us to navigate the frame list regardless of its call stack structure. NextF moves to the next frame, and PrevF moves to the previous frame. This is similar to return and rreturn, however these commands do not halt for breakpoints.

6.5.2 Events, Frames

A useful debugging technique is to be able to quickly see the execution path of a program. One way to visualize this is, to see the events and frames stored by *odb*.

events displays the events list in groups of 20 using the format:

```
<%instruction count%> %event type% %filename%:%line number%
```

frames displays data in a similar format for frames. Both use an arrow to indicate the current frame / event.

6.5.3 History

history displays the history of a variable for the current frame. This is useful for seeing changes of a variable over time quickly. The history command looks up the current frame and accesses its locals map, which is a HistoryMap. It then retrieves a sub list of the specific variable's history using the *getBefore* method. This method takes a variable name as a key, and a timestamp to start the history search. The returned list contains all HistoryValue entries with a timestamp greater than the given timestamp.

6.5.4 Eval

eval is a powerful command that quickly evaluates an expression over all the events in a frame. This can be used to test assertions dynamically when debugging.

Since the expression being evaluated is the same, we start by compiling the expression to a PyCode object. This allows us to re-use this code object, saving the extreme overhead of compilation.

Next we perform a slightly modified linear search over the events list starting at the `currentframe.timestamp` and ending at `currentframe.return_timestamp`. Whenever a call event is encountered, we skip to the event after the corresponding return. Depending on the nature of the program, this can skip large numbers of events.

Any events between this call and return occur in a different frame, and therefore have a different locals map. Therefore, skipping these events is not a problem, as they are not considered valid for the expression being evaluated. An exception is made for recursive methods, by checking the file name and line number of the method.

When a valid event is reached, the frame's locals at the event's timestamp are used to evaluate the expression. If there is an exception, it is discarded. Exceptions are common when evaluating expressions with many variables, as the evaluation of the expression will raise a NameError for all events before the point where all variables have been instantiated.

Finally, a list of the expression results at specific times is returned. To avoid over saturating this list, only new values are appended. A variable, `m`, which is assigned to once, but is present in 1000 valid events would only display a single value when running the command `eval m`.

The two optimizations described above have a significant impact on the performance of the eval command. Running eval on a frame with a range of around 300k events, where roughly 3000 events are 'valid', took several minute without optimization. Applying the event skipping optimization improved this to a matter of tens of seconds. Profiling revealed just how slow compiling was, the precompiliation optimization improved the speed of eval to sub second time.

6.6 Hybrid TTD

The hybrid debugger makes use of the technique implemented separately both in *tdb* and *odb*. *tdb* provides the basis for re-execution and *odb*'s object level copy-on-write history can be extended to create checkpoints. To combine *tdb* and *odb*, a method for restoring the execution state of a checkpoint is required. A set of experiments was undertaken to assess the technical feasibility of our hybrid design. As a result of these, we conclude that it should be possible to create a combined TTD from *tdb* and *odb*, and suggest this as a project for future work.

The biggest challenge in implementing the hybrid TTD is restoring the call stack without re-executing the entire program. An interpreter, which processes bytecode instructions one at a time, would easily allow for this to be implemented. Given a serialization of the call stack, the program starts by placing the first method called on the stack. It then pseudo-executes this method to the next call in the call stack, skipping all other instructions, but updating the program counter (in the case of Python this is the frame's f_line field). As part of the call stack serialization, the opcode stack must also be serialized, so that it too can be restored for each frame. Conceptually, the checkpoint has now been restored, as the *odb* style object history is restored on demand. Therefore, any regular execution from this point would make use of the logged values from the checkpoint.

This challenge was further complicated by our departure from PyPy. PyPy interprets Python bytecode in the manner described. Jython, unfortunately does not. As described in 5.1.2, Jython translates Python bytecode into Java bytecode. This method improves the performance of Jython execution, but has the side effect of entangling the Java call stack with the Jython call stack. Particularly, the generated Java bytecode makes calls into the Jython interpreter. As a result, a single call in Python code becomes many Java calls. As the JVM does not allow us to add arbitrary stack frames, nor jump to a line in a program, the implementation described above is not possible. We explored two solutions to this problem.

6.6.1 Modifying Java Bytecode Generation

The first idea involved modifying the bytecode compiler. Every method on the call stack could be replaced by a fake version. These fake methods are clones of the original, with previously executed code removed. Method calls which correspond to a call on the call stack are made to these fake methods instead. Special care would need to be taken with loops and complex control flow. To implement this idea, a new visitor could be derived from the existing compiler's visitors. The call stack checkpoints would need to contain enough information to determine the lines which created each frame on the call stack. In particular, multiple method calls in a single line would need to be dealt with. An example of how a Python program could be modified to produce the same results as the bytecode re-writing technique can be seen in figure 6.16. We foresaw many complications in the implementation of this method, as well as a high potential for the semantics of the original program to be modified.

```
1  #...
2  #definitions of foo and baz above
3
4  def bar(f):
5      a = 2**f
6      # ... some more instructions
7  #-> b1 = baz() #stopped before this call
8      b2 = bar()
9      return b1 + b2
10
11 def main():
12     a = foo()
13     b = bar(a)
14     c = baz(b)
15
16 ######################################################################
17 #Methods added to recreate call stack: main -> bar
18
19 def fake_bar(f):
20         # instructions before stopped point are removed
21     b1 = baz() #stopped before this call
22     b2 = bar() # calls after the stopped point are normal
23     return b1 + b2
24
25 def fake_main():
26         # instructions before next method on call stack are removed
27     b = fake_bar(a) # fake method replaces call
28     c = baz(b)
29
30 fake_main() #replaces the call to main()
```

Figure 6.16: This code snippet shows a simple program stopped on line 7. The call stack consists of *main*, which calls *bar*. In order to re-store this call stack, and the execution up to line 7, we introduce the *fake_main* and *fake_bar*. *fake_main* is a clone of *main* with the executed lines (12 in main) removed. The call to *bar* is replaced by *fake_bar* since *bar* is next on the call stack. Notice in *fake_bar* that the call to *baz* remains, as it has yet to be executed.

6.6.2 Interpreting Python Bytecode

Our second idea took a more pragmatic approach. Jython can not produce Python bytecode, but given Python 2.5 bytecode, Jython can interpret it. This functionality comes from a legacy file in Jython. The PyBytecode class has an interpret function, which interprets a provided byte string as Python bytecode. This would provide us with the required functionality at a performance cost (as well as being limited to woefully outdated Python 2.5 code). To generate the Python bytecode, we can call `python25 -m pycompile file.py` to generate python bytecode. Then from Jython we run the file with `jython -c 'import pycimport; import file'`, which enables the bytecode meta-importer and imports the file module. This process could easily be embedded into the Jython interpreter.

Work on the second idea was started. The functionality of the *pycimport* module was assessed and *odb* was shown to work in interpreted mode. However, in lieu of polishing the *odb* and *tdb* implementation, work on checkpoint restoration was abandoned.

6.7 Testing

Python does not directly support attaching multiple debuggers, as there is only a single global trace function. For this reason, testing was particularly important in developing *odb*. There were two main components to test, debugging compatibility and logging correctness.

6.7.1 Debugger

To test the compliance of *odb* in relation to pdb, we developed a type of model based test. This style of testing was a result of hand written unit tests missing test cases. The model consists of an array of tuples. Each tuple corresponds to a instruction count, and the elements of the tuple store the instruction count after executing a specific navigation command at that instruction count. For example, the first entry might be (1,12,...) which means that a step instruction would result in the debugger reaching instruction count 1, and a next instruction would jump to instruction count 12. The model implements the debugger commands in the simplest manner possible, by updating its state with what the model says it should be:

```
def do_next(self):
    self.ic = self.model[self.ic].next_ic
    self.check_model()
```

Test cases are generated for ever command at every point in the program. When each test is run, it performs the same commands on the model and the actual debugger, and the *check_model* method compares the state of the two.

6.7.2 Logging

Since logging was essentially implemented by replacing the backing Map of the PyStringMap object, we focused our tests on ensure compatibility between the existing backing object and the new OdbMap. This was achieved by using the test generators provided by Google's guava-test library[12]. Using these test generators, we were able to specify the properties that our map implementation should have, and generate a test suite of over 300 tests to run on the existing backing map and OdbMap in both recording mode and replay mode. This gave us the confidence that the two functioned equivalently, at least in recording mode. To more thoroughly test replay mode, we wrote specific tests for the HistoryMap class. Lastly, we supplemented these Java tests with Python tests of the collections and objects that used the Map.

Chapter 7

Evaluation

Unless otherwise stated, all benchmarks were run on an HP Compaq dc8300 with 16GB of ram, and an Intel Core i7-3770 3.40GHz with 8 cores. Any benchmarks run using Jython limit the JVM's heap size to 4GB, as it was decided this was an amount of memory that modern systems can afford to allocate.

7.1 Instruction Counting Overhead

Implementing instruction counting efficiently is important to the performance of the debugger, as it is the basis for all reverse execution. We tried to improve the performance overhead incurred in Sabin's epdb[31], as this is a Python based TTD that implements instruction counting without modifying the interpreter. Sabin's implementation reportedly ran at roughly 15 times slower than standard CPython, with a maximum overhead of 110 times. As a benchmark, he used the standard Python benchmark, *pybench*. While *pybench* is no longer considered a reliable or accurate benchmark, we will use it to be able to compare our performance with epbd.

To evaluate the performance improvement gained by implementing instruction counting in the interpreter, we set up a 3 way comparison. we started by running *pybench* on an unmodified version of Jython as the control variable. The benchmark was run for 10 rounds (the default) and a warp factor of 5 using the following command:

```
Jython -J-Xmx4096M pybench.py -w 5
```

The warp factor is a parameter used by *pybench* to speed up tests. Tests need to run long enough for the timing to be considered accurate. A high warp factor decreases the amount of time the tests run for. As Jython uses a JIT and pybench was run for 10 rounds, the warp factor needed to be reduced from the default of 10 to 5 as compensation for the speed up over time afforded by the JIT. The warp factor was determined experimentally, as the default warp factor resulted in a "warp factor set to high error".

Next, we ran *pybench* through the instruction counting program used by Sabin using the following command:

```
Jython -J-Xmx4096M countinst.py pybench.py -w 5
```

Lastly, we ran *pybench* under *tdb* debugger to enable interpreter level instruction counting enabled using this command:

```
Jython -J-Xmx4096M -m tdb pybench.py -w 5
```

The graph in figure 7.1 shows the performance difference between the instruction counting implemented in Java and Python for the various components of *pybench*. The instruction counter written in Python had an average runtime of 41 seconds. The interpreter level instruction counting average runtime was only 19 seconds. The average performance improvement is around 48%, meaning that the interpreter level instruction counting has an overhead equal to roughly half the overhead of software level instruction counting. Interestingly, the interpreter level counter was actually slower for the TryRaiseExcept and WithRaiseExcept tests but much faster for the TryFinally and TryExcept tests. This is likely attributed to Jython's slow exception raising in general, as a single Python exception is translated into multiple Java exceptions.

7.1.1 Summary

The goal, O1, was to have an instruction counting overhead that was comparably low, around 15 times. The average overhead was a slowdown of 2.27 times using the interpreter level instruction counting. The maximum overhead was 15 times for the TryExcept test. We feel we have met this goal comfortably, however we would have liked to have seen more of an improvement relative to the software instruction counter. The software level instruction counter, written in python, was on average 5.78 times slower than standard Jython, peaking at an overhead of 58 times. This conflicts with Sabin's results on CPython, which were on average 15 times slower but reached overheads exceeding 100 times. It therefore appears, that Jython has some inefficiencies in its implementation of tracing, and that such an experiment would yield even better results on CPython.

7.2 Forward Execution Runtime Overhead

A TTD needs to log data as it executes forward in time. This comes at the cost of performance. To measure the overhead, we executed a number of standard Python benchmarks with and without a TTD attached. Each benchmark program was run 10 times, and the average runtime was recorded. The benchmarking programs are: *bm_call_method*, *bm_call_simple*, *bm_fannkuch*, *bm_nbody*, *bm_raytrace* from the CPython 'benchmarks' repository's[1] performance folder.

Additionally, to provide a point of comparison, we ran Sabin's epdb on CPython to determine its overhead on the same benchmarks. As a result of epdb's slow performance, we had to adjust the benchmarks to ensure they would terminate. In particular,

[1] `https://hg.python.org/benchmarks`

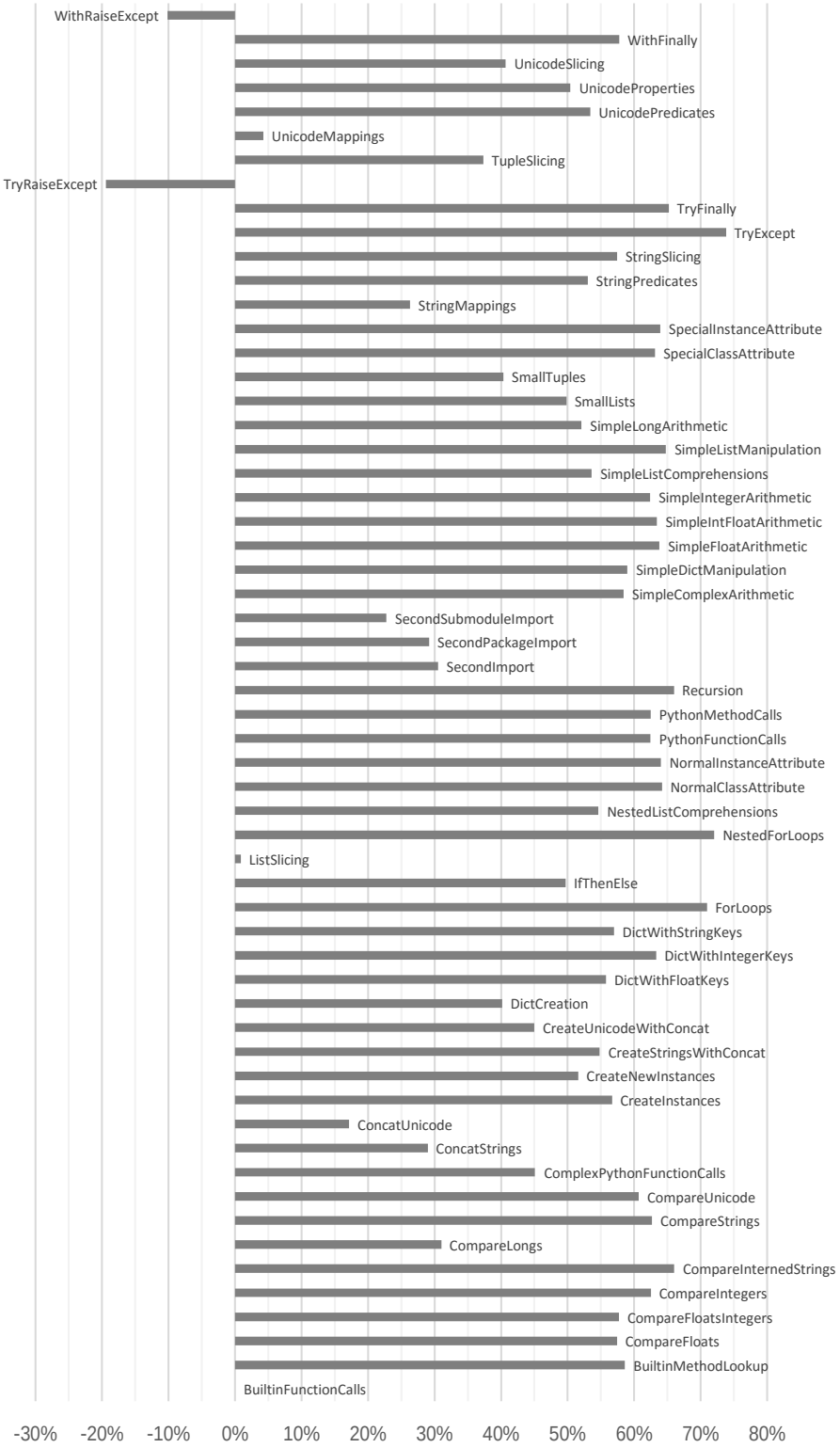


Figure 7.1: Performance improvement gained by implementing instruction counting in Java as opposed to Python

	Interpreter and Debugger			
Benchmark Name	Jython	Jython pdb	Jython Odb	Jython Tdb
bm_call_method.py	0.030	0.735	0.296	0.058
bm_call_simple.py	0.015	0.686	0.256	0.041
bm_fannkuch.py	0.101	0.615	0.892	0.116
bm_nbody.py	0.071	0.626	0.506	0.091
bm_raytrace.py	0.252	1.873	1.043	0.255

Table 7.1: Average time in seconds required to run each benchmark program on Jython

bm_call_method, *bm_call_simple* were simplified to perform 100 times fewer calls and *bm_raytrace* reduced from a 100x100 pixel grid to a 20x20 pixel grid.

In order to provide a comprehensive analysis of the performance of our TTDs, we ran the benchmark on 7 different interpreter/debugger combinations:

Jython The baseline performance test for benchmarks running on the Jython interpreter.

Jython pdb A comparison of the standard debugger running on Jython. A breakpoint was set on the last line of the top level function to ensure the debugger was actually invoked.[2]

Jython odb Our omniscient debugger implementation. No breakpoints set, as the omniscient debugger runs the entire program by default.

Jython tdb Our re-execution debugger implementation. No breakpoint set, as *tdb* traces every frame by default.

CPython The baseline performance test for benchmarks running on the CPython interpreter. Python version 3.4 was used for all CPython based runs.

CPython pdb The standard debugger running on CPython. As with the Jython benchmarks, a breakpoint was set on the last line.

CPython epdb Another implementation of a time travelling debugger, running on CPython. Provided for comparisons sake.

The results of the Jython and Cpython benchmark runs can be seen in tables 7.1 and 7.2 respectively. To help visualize the relative overhead, these results have been compiled into the graph shown in figure 7.2. For each of the Jython runs, the time taken was divided by the time Jython required to complete the benchmark. The bars of the graph therefore represent how many times slower the various debuggers are than Jython or CPython running without a debugger attached. Notice the y axis has a log scale.

[2]If we did not include a breakpoint the overhead would be negligible. This is because pdb has an optimization, which checks if there are any breakpoints before running. If there are no breakpoints, the trace function is removed from the current frames, and the program will execute without any overhead. While this test slightly misrepresents the performance of pdb, we believe it is still fair as it demonstrates the overhead of pdb when actually debugging a program. Furthermore, the breakpoint is set in the outermost frame, so inner frames may still drop the trace function, reducing the overhead.

	Interpreter and Debugger		
Benchmark Name	CPython	CPython pdb	CPython epdb
bm_call_method.py	0.006	0.496	203.627
bm_call_simple.py	0.005	0.492	211.342
bm_fannkuch.py	0.014	0.396	210.897
bm_nbody.py	0.015	0.775	125.576
bm_raytrace.py	0.017	0.435	175.506

Table 7.2: Average time in seconds required to run each benchmark program on CPython 3.4

pdb The overhead of debugging a program using pdb is more significant for CPython than Jython with an average of 49 times slower compared to Jython's 16 times. However, these values are skewed a bit. The two *bm_call_method/simple* benchmarks had the largest impact on the overhead of the debugger. The overhead of running pdb on these two benchmarks was 76 and 99 times for CPython and 25 and 45 times for Jython. This is due to the benchmarks being almost entirely call based, which are slower to debug than simple line events. Excluding these two benchmarks, the average slowdown of running pdb on CPython and Jython respectively reduces to 29 and 7 times.

epdb epdb is shockingly slow. The overhead of epdb ranges from 8k to a staggering 42k times slower than CPython. While these overheads are immediately apparent from Sabin's report, the execution times we observed are similar enough to give us confidence that epdb has been setup correctly.

tdb The overhead of *tdb* is on average just 1.45 times! *tdb* must trace every instruction to track the instruction count, so this low overhead is quite impressive. To achieve this, the instruction counting and call depth tracking code is written as high performance java code embedded in the Jython interpreter. When the TdbTraceFunction determines the debugger should stop, it makes a call to the TBdb python module to allow for user interaction.

odb As a logging TTD, *odb* can be thought of as a taking a checkpoint at each execution point in the program. One would therefore expect *odb* to have a significant forward execution overhead, when logging the program. *odb*'s results are better than expected, when compared to the overhead that pdb exhibits. With a range of 4 to 17 times slower than standalone Jython, *odb* outperforms pdb in all but one benchmark. There is no competition between *odb* and epdb, with *odb*'s largest overhead being over 500 smaller than epdb's smallest overhead. The comparison is not entirely fair, as epdb takes care of resource handling and non-deterministic events. The lack of such events or the use of resources such as files allows us to claim the comparison is still valid.

7.2.1 Summary

We believe that the comparatively low overhead of *odb* can be attributed to the main logic being written at the interpreter level. The trace function (a python function) is never actually called by the OdbTraceFunction (the java function that implements tracing for *odb*). It is purely used as a flag to determine which frames are debugee code. However, when compared to *tdb*, which is also implemented at the interpreter level, we can see that *odb* has a logging overhead of roughly 4-5 times.

Goal O2 was to have a forward execution overhead comparable with state of the art TTDs, aiming for a worst case overhead of 30 times. Both *odb* and *tdb* satisfy this goal. We believe our hybrid design would perform similarly to *odb*, with a slight performance improvement, as less information would need to be logged.

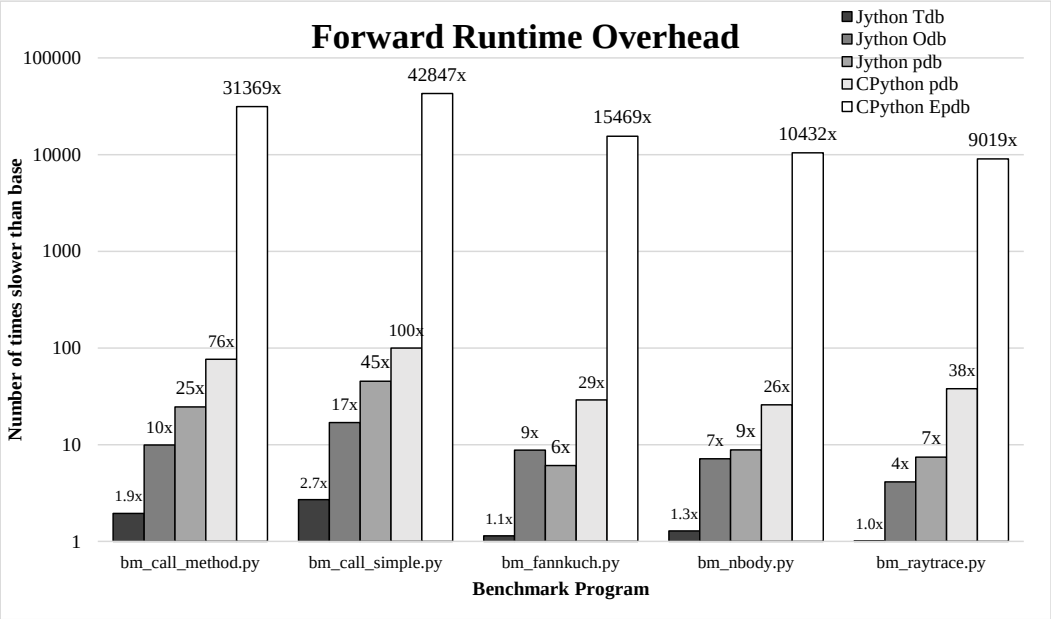


Figure 7.2: This graph shows the relative overhead of each debugger. The values represent how many times slower the debugger took to execute than the same interpreter without a debugger. The Jython values are compared to the Jython interpreter, and the Cpython values are compared to the Cpython interpreter. Therefore, note that a lower overhead does not necessarily mean the absolute speed of the implementation is faster. For absolute time, see tables 7.1 and 7.2

7.3 Reverse Execution Latency

An important characteristic of a successful TTD is that reverse execution does not take too long. Depending on the implementation, the time to jump back in time (or the reverse execution latency) may be linear depending on the temporal distance being travelled or even bounded by a constant. This would be an important metric to assess

the hybrid design by. However, *odb* and *tdb* have predictable performance. *tdb* re-executes from the beginning of a program, so its latency decreases linearly depending on the distance travelled back in time. For example, re-executing to the first instruction is the equivalent of only running the first instruction, while re-executing to the previous instruction requires re-executing all but the last instruction. *odb* has already logged all the required information, and given the limit on the number of events which can be stored in memory, jumping to a past timepoint will take a constant amount of time. We have therefore not run any specific benchmarks, as the hybrid implementation was not completed.

7.4 Snapshot Creation Performance

While we have not been able to fully implement a hybrid debugger that makes use of our copy on write method to create snapshots, *odb* can be analysed as a TTD that snapshots the state at every execution point. Many existing implementations measure the efficiency of snapshot creation by the amount of data logged per second. We believe this is not a comparable metric, as it depends heavily on the program and performance of the test machine. Instead, we offer an analysis of the amount of memory used by *odb*.

To test the limits of our event logging, we wrote two very simple programs.[3] The first tries to create as many events as possible by simply iterating through a loop, and the second tries to create as many frames as possible by calling a function in a loop. To find a rough limit, we modify the number of iterations, observing the heap in the visualVM profiler.

While finding the limit is quite difficult due to garbage collection, these programs allow us to find the usable limits of *odb*, by stopping the program if it takes too long to run. From these experiments, we determined that *odb* can log between 50 million events executed by 8.5 million function calls /frames and 100 million events executed by a single function/frame. Of course, debugging a program that makes and edits a 2 million element list would limit the amount of memory available for logging events. As a comparison, ODB, which is also written in java, can store 10 million events in a 2 GB address space. Storing between 50 and 100 million events in a 4GB address space is therefore an improvement and satisfies goal O3. We credit this efficiency with our implementation of events as 64 bit integers, and the use of a LongBigList to store the events without boxing.

7.5 Compared to the state of the art

The cutting edge of TTD implementation has followed a trend of taking advantage of a managed runtime's features in order to efficiently implement a TTD. This project extends these proposed by the Tardis paper discussed in section 3.1.12. However, rather than simply re-implementing a Tardis like system for Python, a number of techniques

[3] As stated at the start of this chapter, Jython was run with 4GB of heap space.

from other successful TTDs are combined with the hope of marrying both ends of the TTD spectrum to produce a more effective TTD. We also took inspiration from the implementation by Leinhard et al. (section 3.1.8), which used replaced heap references with aliases to store history. This inspiration manifested itself in our approach to copy one write logging, stored at the interpreter level.

While we were not able to implement the planned hybrid debugger in full, we can discuss the performance of *odb*. *odb* has a forward execution overhead on par with the Leinhard et al. implementation and many times better than the epdb, another python TTD. However, Tardis remains unrivaled with an overhead of just 11%, but to do so requires a modified version of the .NET CLR. No other TTD has come close to this figure, and neither did we. This doesn't discount our approach however, as our copy on write logging appears to be efficient when compared to the number of events that the logging TTD ODB by Bill Lewis can store.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

The aim of this project was to implement a hybrid Python TTD, by combining existing TTD styles to create a novel solution with improved performance. We have demonstrated the viability of our hybrid TTD design by successfully implementing two TTDs, *tdb* and *odb*, demonstrating the functionality required by this design. However, the construction of a hybrid of the two was not achieved.

Prior to this thesis, only one fully fledged Python TTDs existed, epdb. Respectively, *tdb* and *odb* are now the fastest known implementations of a re-execution TTD and persistent data structure TTD for Python.

The minimal overhead imposed by the re-execution style debugger, *tdb*, and the logging style debugger, *odb*, lead us to believe that a hybrid of the two would be competitive with state of the art TTDs. While we were unable to complete the hybrid implementation, we believe it to be straightforward given our design and enough time.

We have shown, that Sabin's thoughts on improving the performance of instruction counting, by implementing it as part of the interpreter, are true. The results we observe from running his instruction counting overhead experiments on Jython are less than the overhead Sabin observed using CPython. We surmise that the performance improvement gained by implementing interpreter level instruction counting would be greater on CPython, and Jython's tracing interface could be optimised.

Our work has contributed further evidence in support of Barr and Maron's claim that piggybacking on a VM or managed runtime simplifies the implementation of a low overhead TTD. Our design makes use of the JVM's garbage collection, Java objects and exploits the JVM's JIT compiler speeds up instruction counting and logging.

The straightforward nature of our design could be extended to other languages. Particularly, our work on embedding object history into the interpreter can be applied to any language where the underlying representation uses objects as an atomic unit of data. Based on our work, we believe that *tdb* and *odb* could successfully be combined to form a production quality hybrid TTD, and suggest this as a future project.

8.2 Future Work

The nature of this project necessitated certain constraints, notably the handling of multi-threaded programs and programs with side-effects such as file manipulation. These provide the basis for many improvements to be made to the project. These improvements range from solving technical challenges, to transforming *tdb* into a user friendly and powerful debugger. These ideas are described below:

8.2.1 Finishing the Hybrid

A logical piece of future work would be to finish the implementation of the hybrid debugger. The challenge in this work is restoring the call stack, as Jython translates Python code to Java bytecode and runs it directly. Some additional details have been provided in section 6.6.

8.2.2 Handling Side-Effects

Sabin's epdb focuses fairly heavily on the handling of resources outside of the debugger such as the file system. For a TTD to be useful for a wider range of applications, it must be able to handle side effects to the environment in some way.

8.2.3 Advanced checkpointing heuristics

While a TTD may implement efficient checkpointing, at a certain point it will run out of memory. To deal with this, more advanced heuristics for checkpointing, and discarding of checkpoints could be implemented. This could include exponential checkpoint culling, which tries to distribute checkpoints throughout the lifetime of the program so that more recent checkpoints are closer together. Additionally, adaptive checkpoint intervals or heuristics could be used to improve the runtime overhead of the debugger, by reducing the number of checkpoints made depending on the type of code being executed.

8.2.4 Graphical Interface / IDE integration

One of the pieces of feedback from the project fair was that participants preferred a graphical interface. While this was beyond the scope of the project, we agree that a TTD would be greatly enhanced by adding a graphical interface. The more easily a developer can find information (or perhaps even discover it accidentally) the faster they can solve bugs. Additionally, it was suggested to create an IDE plugin.

8.2.5 Multi-threading

Our implementation is not suitable for multi-threaded programs. Handling multi-threaded programs may make for an interesting future project. Common approaches are to multiplex all threads into a single thread for TTD execution (which incurs a large overhead).

8.2.6 Time Travelling Interpreter

Given the ability to re-do this project, I would have forgone the effort of trying to modify an existing interpreter, and instead written a prototype interpreter with time travel in mind. The decision was made to use existing interpreters as the barrier of entry and workload for writing a new interpreter seemed too high. However, in retrospect the amount of time spent trying to understand existing interpreters could have been spent writing a toy interpreter to demonstrate the concept.

8.2.7 PyPy again?

Despite PyPy's complexity, we believe it, or similar interpreters, are the future of interpreter design. For example, Stackless Python[34], a Python implementation that doesn't depend on the C call stack for its own stack, now uses PyPy as its underlying implementation. I believe someone experienced in the nuances of PyPy would be able to quickly implement an efficient TTD.

Bibliography

[1] H. Agrawal, R.a. De Millo, and E.H. Spafford. "An Execution, Backtracking Approach to Debugging". In: *IEEE Software* 8.3 (1991), pp. 21–26. DOI: 10.1109/52.88940. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=88940.

[2] Earl Barr and Mark Marron. "TARDIS: Affordable Time-Travel Debugging in Managed Runtimes". In: *Oopsla* (2014), pp. 67–82. ISSN: 9781450325851. DOI: 10.1145/2660193.2660209. URL: http://research.microsoft.com/pubs/212395/TimeTravelDbg.pdf.

[3] David Beazley. "Tinkering with PyPy". PyCon US 2012. Mar. 10, 2012. URL: http://pyvideo.org/video/659/keynote-david-beazley.

[4] Carl Friedrich Bolz et al. "Back to the Future in One Week-Implementing a Smalltalk VM in PyPy." In: Springer.

[5] Bob Boothe. "Efficient algorithms for bidirectional debugging". In: *ACM SIGPLAN Notices* 35.5 (2000), pp. 299–310. ISSN: 1-58113-199-2. DOI: 10.1145/358438.349339.

[6] *Chronon Time Travelling Debugger*. URL: http://chrononsystems.com/products/chronon-time-travelling-debugger.

[7] *Elm Debugger*. URL: http://debug.elm-lang.org/ (visited on 11/22/2015).

[8] Stuart I. Feldman and Channing B. Brown. "IGOR: A System for Program Debugging Via Reversible Execution". In: *SIGPLAN Not.* 24.1 (1989), pp. 112–123. ISSN: 0-89791-296-9. DOI: 10.1145/69215.69226. URL: http://portal.acm.org/citation.cfm?id=69226&CFID=66428822&CFTOKEN=77494100.

[9] Robert J. Fowler, Thomas J. LeBlanc, and John M. Mellor-Crummey. "An integrated approach to parallel program debugging and performance analysis onlarge-scale multiprocessors". In: *Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging - PADD '88* (1988), pp. 163–173. ISSN: 0897912969. DOI: 10.1145/68210.69231. URL: http://portal.acm.org/citation.cfm?doid=68210.69231.

[10] W.K. Fuchs. "Reversible debugging using program instrumentation". In: *IEEE Transactions on Software Engineering* 27.8 (2001), pp. 715–727. ISSN: 0098-5589 VO - 27. DOI: 10.1109/32.940726. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=940726.

[11] *GDB: The GNU Project Debugger*. URL: https://www.gnu.org/software/gdb/.

[12] Google inc. *Guava: Google Core Libraries for Java*. Dec. 9, 2015. URL: https://github.com/google/guava.

[13] Christopher C D Head. "Debugging through Time with the Tralfamadore Debugger". In: (2011).

[14] Jim Hugunin. *Iron Python*. URL: http://ironpython.net.

[15] Jim Hugunin. *Jython*. URL: http://www.jython.org.

[16] *IntelliTrace Features*. URL: https://msdn.microsoft.com/en-us/library/dd572114.aspx (visited on 11/22/2015).

[17] a.J. Ko and B.a. Myers. "Debugging reinvented: Asking and Answering Wha and Why Not Questions about Program Behavior". In: *2008 ACM/IEEE 30th International Conference on Software Engineering* (2008), pp. 301–310. ISSN: 978-1-60558-079-1. DOI: 10.1145/1368088.1368130. URL: http://portal.acm.org/citation.cfm?doid=1368088.1368130.

[18] Bil Lewis. "Debugging Backwards in Time". In: (Oct. 2003). URL: http://arxiv.org/abs/cs/0310016.

[19] Henry Lieberman and Christopher Fry. "ZStep 95: A reversible, animated source code stepper". In: *Software Visualization: Programming as a Multimedia Experience*. 1997, pp. 277–292. ISBN: 978-0-262-19395-5.

[20] Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz. "Practical object-oriented back-in-time debugging". In: *In 22nd European Conference on Object-Oriented Programming (ECOOP'08), volume 5142 of LNCS*. Springer, 2008, pp. 592–615.

[21] *Performance — Chronon*. URL: http://chrononsystems.com/what-is-chronon/performance (visited on 01/08/2016).

[22] Benjamin Peterson. "PyPy". In: *Structure, scale and a few more fearless hacks*. Ed. by Amy Brown. The architecture of Open Source applications Ed. by Amy Brown Greg Wilson ; 2. Mountain View: Creative Commons, 2012. ISBN: 978-1-105-57181-7.

[23] Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. "Expositor: Scriptable time-travel debugging with first-class traces". In: *2013 35th International Conference on Software Engineering (ICSE)* (2013), pp. 352–361. ISSN: 978-1-4673-3076-3. DOI: 10.1109/ICSE.2013.6606581. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6606581.

[24] *PyPy*. URL: http://pypy.org.

[25] *PyPy Documentation*. URL: http://doc.pypy.org/en/latest/index.html.

[26] *PyPy Speed Center*. URL: http://speed.pypy.org.

[27] Python Software Foundation. *pdb - The Python Debugger*. Jan. 1, 2016. URL: https://docs.python.org/2/library/pdb.html.

[28] Guido Rossum. *CPython*. URL: https://www.python.org.

[29] Guido Rossum. *Five-minute Multimethods in Python*. Mar. 29, 2005. URL: https://www.artima.com/weblogs/viewpost.jsp?thread=101605 (visited on 01/08/2016).

[30] Guido Rossum. *Python Language*. 1995.

[31] Patrick Sabin. "Implementing a reversible debugger for Python". 2011. URL: http://mips.complang.tuwien.ac.at/Diplomarbeiten/sabin11.pdf.

[32] *Technology — Chronon*. URL: http://chrononsystems.com/what-is-chronon/technology (visited on 01/08/2016).

[33] Thomas Mauch. *BigList*. Nov. 5, 2015. URL: http://www.magicwerk.org/page-collections-overview.html.

[34] Christian Tismer. *Stackless Python*. 20, Jan 2000. URL: https://bitbucket.org/stackless-dev/stackless/wiki/Home.

[35] *UndoDB Reversible Debugging Tool for Linux — Undo Software*. URL: `http://undo-software.com/undodb/` (visited on 11/22/2015).

[36] Bret Victor. "Inventing on principle". CUSEC 2012. 2012. URL: `https://vimeo.com/36579366`.

[37] *Visual Studio IntelliTrace*. URL: `https://www.visualstudio.com`.