Università
della
Svizzera
italiana

Faculty
of Informatics

USI Technical Report Series in Informatics

Space-Time Views For Back-In-Time Debugging

Mohammad R. Azadmanesh[1], Matthias Hauswirth[1]

[1] Faculty of Informatics, Università della Svizzera italiana, Switzerland

Abstract

Omniscient debuggers promise to greatly simplify debugging. They allow developers to explore the entire execution history of a program run, navigating back in time from a failure to its root cause. While many omniscient debuggers provide features to move forward and backward in an execution, they usually only visualize one single (current) program state at any given time.
We believe that omniscient debuggers should use visualizations that explicitly represent time, allowing developers to see and compare multiple program states, and to visually follow the flow of information throughout a program execution.
In this paper we study the most obvious such visualization, which we call "space-time view". Space-time views are tables where columns represent space—the various memory locations of the program—and rows represent time—the various operations performed throughout the execution. Our goal is not to present a particular implementation of space-time views, but to provide the foundations for many different such visualizations. The key problem with space-time views is their extreme size for realistic executions. Any useful space-time view has to deal with this size. Based on a set of realistic execution histories, we automatically generate space-time views, and we study their size and their internal structure. Our observational study helps visualization developers to understand the most important scaling problems with space-time views, and to develop effective visualization techniques that are meaningful in practice.

Report Info

Published
May 2015

Number
USI-INF-TR-2015-2

Institution
Faculty of Informatics
Università della Svizzera italiana
Lugano, Switzerland

Online Access
www.inf.usi.ch/techreports

1 Introduction

Zeller [1] states "[In debugging,] understanding how the failure came to be [...] requires by far the most time", and Bracha [2] points out that omniscient debugging approaches providing that understanding are "worth more than a boatload of language features". Indeed, omniscient debuggers [3, 4, 5] promise to drastically simplify the debugging problem by providing developers access to the complete execution history of a program. This allows developers to navigate back in time, answering questions such as "Why is this reference null?" or "Where did this value come from?"

In this paper, we are interested in visualizations that allow a programmer to track the flow of values throughout a program execution. This is particularly relevant when tracing backwards along the infection chain, from a program failure back to its cause. An infection chain may involve operations that are *far apart in time* (e.g., a statement during program initialization, and a statement in an entirely different method executing when handling some request), and memory locations that are *far apart in space* (e.g., a field in a heap object, and a local variable in an activation record of some method).

Textbooks explaining execution histories often represent an execution in a tabular form [1, 6], similar to **Figure 1**, with columns representing different memory locations, and rows representing execution steps. We call these tables "space-time views". Those simple and intuitive visualizations are used for illustrat-

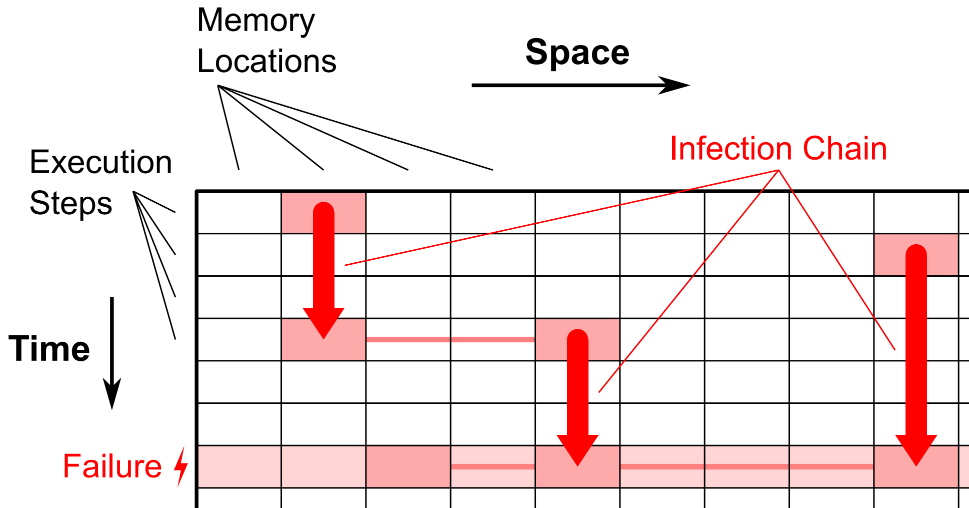

Figure 1: Space-Time View

ive purposes, for execution histories that can be shown in tables with around a dozen rows and columns. Moreover, such examples often use only a subset of modern programming language features, avoiding complexities such as heap allocation, polymorphism, or recursion. This is of course intentional, given that those visualizations do not serve as real debugging aids, but as ways to explain the principles behind techniques such as dependency analysis or slicing.

Histories of realistic executions are much larger, and they involve the entirety of features of modern languages. While there is an intuitive understanding that realistic histories will lead to very large space-time views, it is not clear how large they actually would get, how they would be structured, and to what degree their size could be reduced with appropriate techniques. In this paper we study these questions based on histories of a large number of real-world Java unit test executions, from which we automatically generate space-time views which we then characterize. We study two independent techniques to reduce the size of space-time views: "operation abstraction", which combines multiple rows, and "history slicing", which filters irrelevant columns and rows.

The remainder of this paper is structured as follows: Section 2 provides a brief introduction of space-time views. Section 3 shows that the intricacies of real-world programming languages require space-time views with a low level of abstraction. Section 4 precisely defines space-time views for Java bytecode execution histories, and Section 5 characterizes such views based on a set of unit test executions. Section 6 presents "'operation abstraction" and "history slicing", two techniques to reduce the size of space-time views. Section 7 discusses related work, Section 8 discusses limitations, and Section 9 concludes.

2 Space-Time Views

Figure 1 shows a sketch of a space-time view. Columns represent the space dimension, consisting of all memory locations, while rows represent the time dimension, consisting of all execution steps. The last row shown in the figure represents a failing execution step. The purpose of debugging is to find the cause of that failure. For this reason, developers usually navigate backwards along the infection chain to determine where the values used by the failing instruction came from.

2.1 Space

The space dimension consists of a set of columns. Each individual column represents a single memory location L_x, which holds a single scalar value (i.e., a value of a primitive type, or a reference to some object). One complete row in the table represents a snapshot of the program state. It corresponds to a flat representation of all the information a traditional debugger could possibly show at a given point in time.

In Java, like in most modern programming languages, space has a certain structure. **Figure 2** structures the space with a hierarchy of column headers: At the top of the hierarchy we distinguish between different

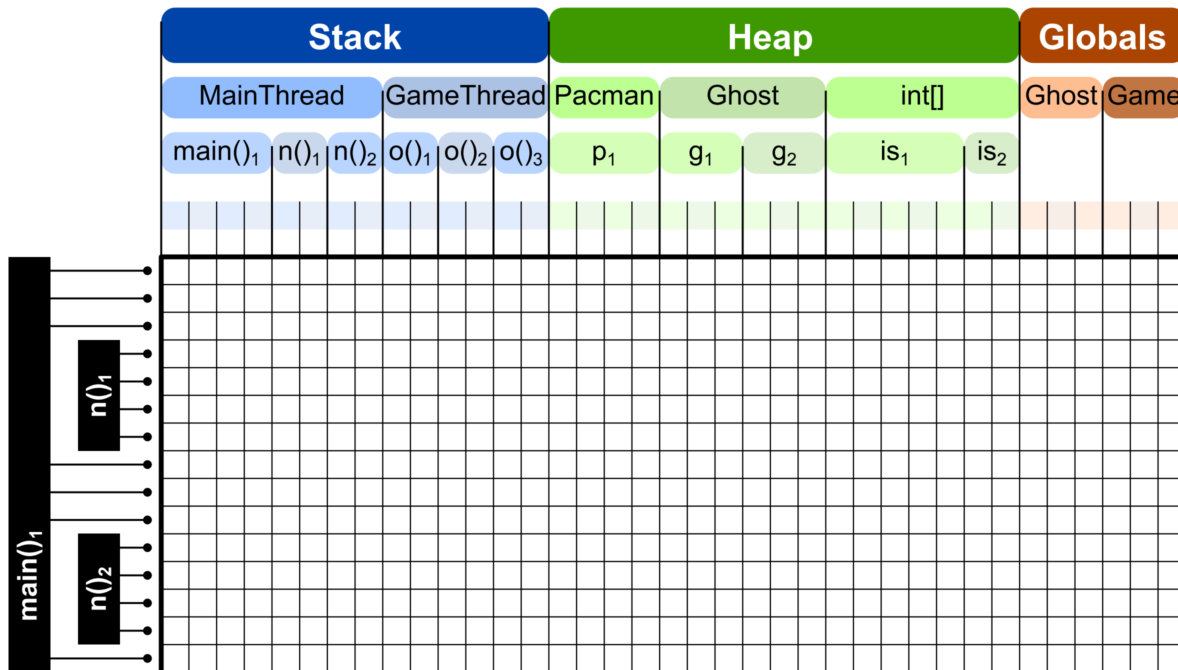

Figure 2: Structure along space and time dimensions

memory regions, usually the stack(s), the heap, and the space for global variables. Each of these regions can be further subdivided.

- **Stack.** In a single-threaded program, there is only one stack, while in multi-threaded programs, we have one stack per thread. Each thread's stack is further subdivided into activation records (or stack frames). Each stack frame holds the local variables and parameters of a method invocation.
- **Heap.** In a typed language like Java, we can group heap objects by their types. Heap objects can be either regular objects, consisting of a set of fields, or they can be arrays, consisting of a sequence of array elements (and a read-only length field).
- **Global.** In Java-like languages, global variables are (static) members of classes, and thus they can be grouped by their class.

2.2 Time

The time dimension consists of a sequence of rows. Each row corresponds to one execution step S_y of the program. Depending on the abstraction level, an execution step could correspond to the execution of a bytecode instruction, the evaluation of a subexpression, the execution of a program statement, or the execution of a line of source code. We will discuss the different abstraction levels in Sections 3 and 6.1. As Figure 2 shows, the time dimension can be structured by using the row headers to represent a thread's call tree (with one node for each method invocation). For multi-threaded programs, the steps from different threads will be interleaved, and multiple call trees (one for each thread) will need to be presented.

2.3 Cells

Each cell in the table holds the value of a memory location (column) at the time of a given execution step (row). A particular implementation of a space-time view could show the value *before* that step and/or the value *after* that step. Moreover, if a step accesses a memory location, the cell can indicate whether there was a read access, a write access, or a read/write access.

Figure 3 presents an example: S_1 writes L_1 and L_8. While usually a single step only writes to a single memory location, there are cases (e.g., at higher abstraction levels) where one step writes to multiple locations. S_1 does not read from any memory location. While most steps will read from at least one location,

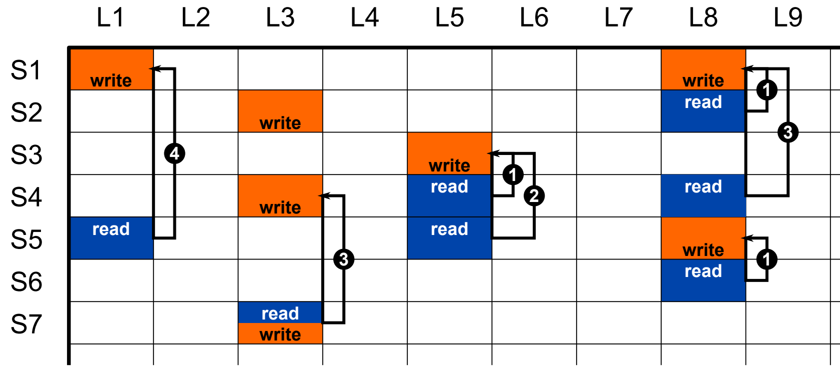

Figure 3: Steps S_y accessing memory locations L_x

there are cases (e.g., when initializing a variable with a constant value) when a step only writes but does not read. S_2 represents a more common situation: it reads a value from one location (L_8), then potentially computes something based on that value, and then stores the result in a different location (L_3). S_4 reads from more than one location. This is quite common. S_6 only reads, but does not write. This is quite rare. S_7 reads and writes the same memory location. This can happen for example when incrementing a variable (`i++`). Note also that in general a memory location can be written more than once (e.g., L_8 is written twice, and L_3 is written three times).

2.4 Infection Chain: Information Flow

While debugging, a developer usually starts at the failing execution step (for example at S_7 in Figure 3). Each step may read multiple memory locations (S_7 only reads one location, L_3). The failing step must have failed for a reason, and the developer tries to find that reason by navigating back along all information flows reaching the failing step. For each memory location read by the failing step (in S_7 this means for L_3), the developer can look when that location had last been written by scanning upwards (back in time) until he finds the most recent write in the memory location's column (L_3 was most recently written in S_4). The developer can repeat this search transitively (e.g., S_4 reads L_5 and L_8, L_5 is written by S_3, L_8 is written by S_1), until he finds the root cause of the failure. This chain, from the failure to the root cause, is called the infection chain.

2.5 A Notion of Place

The advantage of using a space-time view for this kind of debugging process is that the information is presented in an overall context: every memory location and every execution step has its fixed place in a two-dimensional plane. Adjacent rows represent execution steps adjacent in time, and most of the adjacent columns represent memory locations that belong to the same object or array. The disadvantage is that both the space and the time dimension can grow large, much larger than what can be shown on a screen or on a sheet of paper. It is the goal of this paper to quantify how large space-time views can grow, and to investigate techniques to reduce their size, while keeping their positive properties.

3 The complexity of statements

When developing a space-time view visualization, one has to decide what constitutes an execution step. As stated previously, this could be a source code line, a statement, or something else. We now discuss why source code lines and statements can be problematic. Many modern debuggers are integrated with an IDE, and their primary visualization is the source code of the program. Such a debugger visualizes a point in the program execution by highlighting the source code line with the currently executing statement. When single-stepping, the debugger executes the statement, and stops at the next one.

Statements, however, are a relatively weak abstraction for debugging. A single statement may interact with the program state in many different ways. Below is an illustrative example of a single statement with a large, intricate set of interactions with the program state:

```
int a = b ? c(d=1, e&&f()) : new G(h++).i(--j);
```

This statement involves several intricate language features:

- *Assignment in an expression context* (d=...) and *pre/post increment/decrement operators* (`h++`, `--j`) mean that this assignment statement does not just set the value of variable a, but also of variables d, h, and j.
- *Conditional operators* (?:) and *short-circuit evaluation* (`&&`) lead to conditional control flow inside expressions, which means that not all effects of the statement take place unconditionally. Thus the statement cannot be considered a simple atomic unit of execution.
- *Method calls* in expressions mean that parts of the statement execute before the method is called, and parts execute after the method returns. In the example, the statement contains multiple method calls, parameters of one method call are computed with other method calls, and return values of one method call are used as target objects for a second method call.

A traditional debugger deals with this complexity by highlighting the same statement multiple times (e.g., before and after each method call it involves), without any indication about which stage of the execution that statement is in. This can be highly confusing already for a traditional debugger, and it is equally problematic when using statements as execution steps in a space-time view.

4 Bytecode-Level Space-Time Views

Based on the discussion in Section 3, we first study the most fundamental, finest granularity execution steps describing a Java program execution: bytecode instructions. Most application developers will not be familiar with bytecode instructions, and thus this low level of abstraction seems to be excessive. However, as Section 3 has shown, the two higher level abstractions familiar to developers, statements and source code lines, are broken. For this reason we first define our model on the low level, and in Section 6.1 we then show how to abstract to a higher level that is close to source code statements.

4.1 Local Variables

We first focus on simple expressions that use operators, literal values, and local variables. The evaluation of an expression with more than one operator produces intermediate values. As described in the previous section, intermediate values in an expression can escape to various memory locations and intermediate steps within a statement can affect control flow. Thus, a visualization that is supposed to help developers to precisely understand when and where variables were written—in arbitrary programs written in real-world languages—needs to explicitly represent such intermediate values as well as the intermediate execution steps that lead to those values.

Bytecode-level space-time views explicitly represent these intermediate values and steps. This makes the visualization much larger, but it shows the complete picture and serves as the baseline for discussing various visual optimizations. We show intermediate values by explicitly visualizing the operand stack locations as they are modeled on the level of a language runtime environment (such as the JVM). We show the intermediate steps by showing instructions (in our case, JVM bytecode instructions) instead of statements in the rows of the table.

Table 1 shows the space-time view corresponding to the execution of the statement `x = 3*x+a`, assuming that at the beginning *x* was 4 and *a* was 5. The view shows a total of five memory locations, and how their values change over time: two local variables, *x* and *a*, and three operand stack slots.

Each row represents the execution of an instruction. `iconst 3` stores the value 3 in operand stack slot #1. `iload x` loads the current value (4) from local variable *x* and stores it in stack slot #2, and `iload a` works similarly for *a*.

The body cells of the table show the current value of each memory location. In case the value is read by the current instruction, the cell also contains an 'R'. In case the value is written, the cell contains a 'W'. In cases where the location is read and written by the same instruction, the cell shows 'R', the old (read) value, 'W', and the new (written) value. We see such a read/write access for `imul`, which reads the value 4 from slot #2 and the value 5 from #3, multiplies them, and stores the intermediate result, 20, back into #2. The next instruction, `iadd`, works similary. Finally, `istore x` stores the result into local variable *x*.

Table 1: Space-time view for accessing local variables

	Stack				
	Activation record				
	Operand stack			Local variables	
	#1	#2	#3	x	a
// initial state					
				4	5
x = 3*x+a;					
iconst 3	W 3			4	5
iload x	3	W 4		R 4	5
iload a	3	4	W 5	4	R 5
imul	3	R 4 W 20	R 5	4	5
iadd	R 3 W 23	R 20		4	5
istore x	R 23			W 23	5

We treat operand stack slot locations slightly differently from other memory locations: operand stack slot reads are destructive, that is, the value disappears after a read. This is in line with JVM semantics, where a read from the operand stack essentially corresponds to a "pop". Our treatment does not affect the characterization reported in the next section, because it does not affect reads and writes in any way (it only clears cells after a read and makes the table slightly more readable). Note that for other memory locations, reads are *not* destructive, that is, there can be multiple reads of the same value.

As the header rows of the table indicate, all five memory locations are part of an activation record (a stack frame), corresponding to the temporary data needed throughout a single invocation of a method. The visualization will include a separate activation record (and thus a separate set of memory locations or table columns) for each and every method invocation.

In this paper we annotate the space-time views with comments and source code statements in bold font. While these annotations are invaluable for the understanding of this low-level code, they are not part of the bytecode-level space-time views we characterize in the next section. This means that they are not included in the metrics we report. We discuss how to build space-time views with execution steps that are similar to source code statements in Section 6.1.

4.2 Heap Accesses

The Java heap contains two kinds of data: objects (instances of Java classes) and arrays. An object consists of a set of fields. The space-time view contains one column for each field. An array consists of a length field and a number of elements. The contents of the length field determines how many elements it contains. The space-time view contains a column for the length field, and one column for each element.

Java provides bytecode instructions for reading and writing heap data: `getfield` reads and `putfield` writes a field of an object, `*aload` reads an element and `*astore` writes an array element, and `arraylength` reads the (read-only) length field of an array.

```
Sprite pacman, ghost;  Sprite[] sprites;
...
pacman.x = ghost.x;
sprites[0] = sprites[sprites.length-1];
```

The above snippet shows example code accessing the heap, and **Table 2** shows the space-time view of an execution of this code. Some cells in this table contain references (pointers to heap objects). We represent those references by using a unique ID for each object. In this example, s_1 and s_2 are the IDs of the two Sprite objects, and ss_1 is the ID of the Sprite[]. An object's ID is also shown in the table header, in the heap region, to group the columns that represent the fields of that object.

The first row of Table 2 shows the initial state of the system, before the first of the two statements starts executing. We see three local variables: `pacman` and `ghost` point to `Sprite` objects, and `sprites` points

Table 2: Space-time view for heap accesses

	Stack								Heap						
	Activation record								Sprite				Sprite[]		
	Operand stack					Local variables			s_1		s_2		ss_1		
	#1	#2	#3	#4	#5	pacman	ghost	sprites	x	y	x	y	length	[0]	[1]
// initial state															
						s_1	s_2	ss_1	1	2	3	4	2	s_1	s_2
pacman.x = ghost.x;															
aload pacman	W s_1					R s_1	s_2	ss_1	1	2	3	4	2	s_1	s_2
aload ghost	s_1	W s_2				s_1	R s_2	ss_1	1	2	3	4	2	s_1	s_2
getfield Sprite.x	s_1	R s_2 W 3				s_1	s_2	ss_1	1	2	R 3	4	2	s_1	s_2
putfield Sprite.x	R s_1	R 3				s_1	s_2	ss_1	W 3	2	3	4	2	s_1	s_2
sprites[0] = sprites[sprites.length-1];															
aload sprites	W ss_1					s_1	s_2	R ss_1	3	2	3	4	2	s_1	s_2
iconst 0	ss_1	W 0				s_1	s_2	ss_1	3	2	3	4	2	s_1	s_2
aload sprites	ss_1	0	W ss_1			s_1	s_2	R ss_1	3	2	3	4	2	s_1	s_2
aload sprites	ss_1	0	ss_1	W ss_1		s_1	s_2	R ss_1	3	2	3	4	2	s_1	s_2
arraylength	ss_1	0	ss_1	R ss_1 W 2		s_1	s_2	ss_1	3	2	3	4	R 2	s_1	s_2
iconst 1	ss_1	0	ss_1	2	W 1	s_1	s_2	ss_1	3	2	3	4	2	s_1	s_2
isub	ss_1	0	ss_1	R 2 W 1	R 1	s_1	s_2	ss_1	3	2	3	4	2	s_1	s_2
aaload	ss_1	0	R ss_1 W s_2	R 1		s_1	s_2	ss_1	3	2	3	4	2	s_1	R s_2
aastore	R ss_1	R 0	R s_2			s_1	s_2	ss_1	3	2	3	4	2	W s_2	s_2

to an array of sprites. The `Sprite` objects are initialized so their `x` and `y` fields are set to (1, 2) and (3, 4). The `Sprite` array has length 2 and contains two references, one to each sprite.

4.3 Global Accesses

Java's static fields are a form of global variables. They exist outside the heap. Java provides the `getstatic` and `putstatic` instructions for reading and writing static fields. A space-time view contains one column for each static field. These columns are grouped by class, and they are placed under the "Globals" header. The following code snippet shows a read access of the static field `Calc.PI`.

```
class Calc { public static double PI; }
...
double pi = Calc.PI;
```

Table 3 shows the corresponding space-time view, with a column for memory location `Calc.PI`, and a row for the `getstatic` bytecode instruction reading that value from that field (and another row storing the value in local variable `pi`).

Table 3: Space-time view for reading a global variable

	Stack		Globals
	Activation record		Calc
	Operand stack	Local variables	
	#1	pi	PI
// initial state			
			3.14
pi = Calc.PI;			
getstatic Calc.PI	W 3.14		R 3.14
dstore pi	R 3.14	W 3.14	3.14

4.4 Method Calls

On the bytecode level, method calls are represented by `invoke*` instructions. Those instructions allocate an activation record for the callee, copy all parameter values (and, for non-static calls, the call target object reference) from the caller's operand stack to the callee's activation record, trigger the execution of the callee, and when the callee returns, copy the return value (if any) from the callee's activation record to the caller's operand stack.

Table 4: Space-time view for method call

	Stack											Heap	
	Activation record for A.m(B b)$_1$						Activation record for B.n(A a,int i)$_1$					A	B
	Operand stack			Local variables			Operand stack	Local variables			Ret.	a_1	b_1
	#1	#2	#3	this	b	r	#1	this	a	i		...	...
// initial state													
				a_1	b_1							...	...
int r = **b.n(this, 3)**;													
aload b	W b_1			a_1	R b_1							...	...
aload this	b_1	W a_1		R a_1	b_1							...	...
iconst 3	b_1	a_1	W 3	a_1	b_1							...	...
invokevirtual_arg	b_1	a_1	R 3	a_1	b_1					W 3		...	...
invokevirtual_arg	b_1	R a_1		a_1	b_1				W a_1	3		...	...
invokevirtual_arg	R b_1			a_1	b_1			W b_1	a_1	3		...	...
return 1;													
iconst 1				a_1	b_1		W 1	b_1	a_1	3		...	...
ireturn				a_1	b_1		R 1	b_1	a_1	3	W 1	...	...
int r = b.n(this, 3);													
invokevirtual_ret	W 1			a_1	b_1						R 1	...	...
istore r	R 1			a_1	b_1	W 1						...	...

```
class A {
  void m(B b)       { int r = b.n(this, 3); }
}
class B {
  int n(A a, int i) { return 1; }
}
```

The above code snippet shows an example method call, where method `A.m()` calls method `B.n()`. **Table 4** shows the corresponding space-time view, with the sequence of executed instructions. Execution starts with some instructions in method `A.m()`, is followed by the code of `B.n()`, and ends with the remainder of `A.m()`.

To make the information flows through parameters and return value explicit, we break each `invoke*` instruction into multiple execution steps: one `invoke*_arg` instruction for each argument to be passed, and one `invoke*_ret` instruction for passing the return value. Note that we introduce a "return value" memory location to activation records of methods with non-void return types. This location allows the callee's `*return` instruction to pass information to the caller's `invoke*_ret`.

4.5 Allocations

In Java, objects and arrays are allocated on the heap with `new` and `*newarray` bytecode instructions, respectively.

```
class Sprite {
  int x, y;
  Sprite() { super(); y = 5; }
}
...
Sprite s = new Sprite();
int[] is1 = new int[2];
```

The above code snippet illustrates object and array allocation. **Table 5** shows the corresponding space-time view. We see the `new` instruction allocating a `Sprite` object. This instruction initializes all fields of the object to their default values. Then `invokespecial` calls the constructor of the `Sprite` class. That constructor invokes the super class (Object) constructor. We do not trace the Java library, and thus we do not see what happens inside the Object constructor. Next, the Sprite constructor sets the `y` field to 5. The time between the start of the constructor and the execution of `y=5` is somewhat dangerous because code may see a not completely initialized object. The space-time view with its explicit representation of state over time can be helpful in debugging problems related to this behavior.

The array allocation simply consists of the `newarray` instruction, which initializes all elements of the array to their default values and the length field of the array to 2. The `new` and `newarray` instructions can write to an arbitrary number of memory locations: they essentially "zero out" the heap space used for new heap objects.

Table 5: Space-time view for allocations

	Stack							Heap				
	Activation record				Activation record for Sprite.Sprite()$_1$			Sprite		int[]		
	Operand stack		Local variables		Operand stack		Local variables	s_1		is_1		
	#1	#2	s	is	#1	#2	this	x	y	length	[0]	[1]
// initial state												
Sprite s = **new Sprite**();												
new	W s_1							W 0	W 0			
dup	R s_1	W s_1						0	0			
invokespecial_arg	s_1	R s_1					W s_1	0	0			
// entered Sprite.Sprite() constructor												
super();												
aload this	s_1				W s_1		R s_1	0	0			
invokespecial_arg	s_1				R s_1		s_1	0	0			
// entered Object.Object() constructor												
// Object.Object() constructor is not traced												
// returned from Object.Object() constructor												
y = 5;												
aload this	s_1				W s_1		R s_1	0	0			
iconst 5	s_1				s_1	W 5	s_1	0	0			
putfield Sprite.y	s_1				R s_1	R 5	s_1	0	W 5			
return	s_1						s_1	0	5			
// returned from Sprite.Sprite() constructor												
Sprite s = new Sprite();												
astore s	R s_1		W s_1					0	5			
int[] is = **new int[2]**;												
iconst 2	W 3		s_1					0	5			
newarray int	R 3 W is_1		s_1 s_1					0	5	W 2	W 0	W 0
int[] is = new int[2];												
astore is	R is_1		s_1	W is_1				0	5	2	0	0

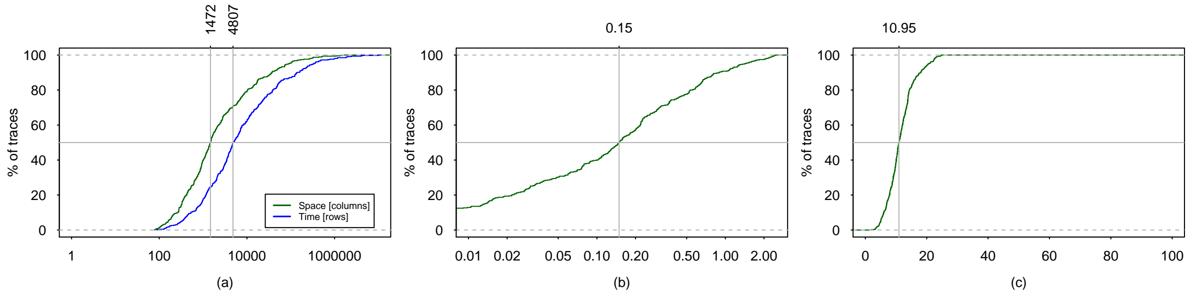

Figure 4: (a) Size (columns and rows) and (b) density of space-time views, and (c) percentage of rows containing pseudo-instructions for parameter passing

5 Characteristics of Bytecode-Level Space-Time Views

We expect space-time views as defined in the previous section to grow large, making it difficult to create effective visualization tools. To understand exactly how large these views really become, we pick a realistic debugging scenario. We chose EJML[1], an open source Java library that comes with an extensive suite of unit tests. During development, unit tests often fail, and when they fail, developers have to find the root cause of that failure. Unlike complete production runs of an application, such unit test runs are scenarios for which a complete execution history can be collected without requiring terabytes of traces. Moreover, it is a scenario in which developers could benefit from effective space-time view visualization tools.

We run each unit test method separately, and we trace all EJML code. We do not trace the Java runtime library. **Figure 4** shows the key characteristics of the resulting space-time views. It consists of three plots. Each plot represents the cumulative distribution (over all traces) of a certain metric. The first plot shows the width and height of the view: the median number of columns (memory locations) is 1472, and the median number of rows (instructions) is 4807. A Full HD screen with its 1920 by 1080 pixels would provide 1.3 pixels per column and 0.2 pixels per row of such a median space-time view. To be able to read the cells, only about 1% of the rows and columns could be visible on the screen. Thus, the views are large, but not as large as we expected.

[1]https://code.google.com/efficient-java-matrix-library/

On average, 48% of the columns are used for the operand stack and return values, another 48% for local variables, 3.5% are used for arrays, and only 0.85% for objects. This small percentage of heap locations is surprising. We believe this is due to the fact that EJML, a matrix library, is computation-heavy. With 0.02%, almost no static fields are used, which is not surprising for well-designed Java code.

The second plot shows the density of the view, the percentage of cells that contain memory accesses (reads, writes, or read/writes). In the median, only 0.15% of space-time view cells represent accesses, 99.85% can be ignored. For a significant number of traces, the density is even smaller. This large number of untouched cells is the cost of having visual stability: the x and y axes have a fixed, well-defined meaning. It also means that interaction techniques such as folding can help to greatly reduce the size of the visualization.

Section 4.4 has shown that method calls lead to multiple pseudo-instructions (one for each parameter and return value). The third plot shows the fraction of rows that correspond to such pseudo-instructions. The median over all traces is 11%. This could be reduced with a more compact representation of method invocations, at the cost of giving up the ability to keep flows of parameters separate from each other.

6 Optimizations

We now discuss "operation abstraction" and "history slicing", two optimizations that help in drastically reducing the size of space-time views.

6.1 Operation Abstraction

Section 5 has shown that almost half the columns correspond to operand stack locations. These locations are not intuitive to application programmers because they are not represented in source code. We thus would like to eliminate them. However, based on the discussion in Section 3, we do not want to use statements as execution steps. We thus use "operations", an abstraction somewhere between bytecode instructions and language statements.

To perform "operation abstraction", we simply group all instructions that directly communicate through the operand stack. This combines instructions that compute expressions such as `a[b+c]+o.x` into a single group. Each such group is an operation, which requires a single row. For each group, we remove all its reads and writes to the operand stack. The writes of the operation then correspond to the union of all remaining writes of its constituent instructions. The same applies to its reads. We end up with a set of high-level operations, each with zero or more reads, writes, and read/writes. Those operations are roughly equivalent to statement expressions and subexpressions computing actual arguments of method calls in the Java language. Finally, given that we deleted all reads and writes to the operand stack, we can also eliminate all operand stack columns.

Consider the following example:

```
class Sprite {
  int y = 5;
  public int getY() { return y; }
}
...
int y = s.getY();
```

Assume `s` is a `Sprite`. The last statement calls the method `getY()` on this object and assigns the returned value to the local variable `y`. **Table 6** shows the space-time view for this statement after operation abstraction. The first column shows the Java statement containing the abstract operation. In the statement, we highlight the abstract operation in bold. The first and the third rows show an example where the statement consists of more than one operation. While our implementation of operation abstraction produces tables like this, we do not automatically generate the labels in the first column. The generation of such labels would require access to the source code, and the development of a source code compiler that unambiguously maps bytecode instructions to AST nodes (instead of just to source file lines).

6.2 History Slicing

The idea of "history slicing", our second optimization, is quite simple: instead of visualizing *all* execution steps and memory locations, we only visualize those steps and locations that are part of the infection chain

Table 6: Space-time view after operation abstraction

	Stack				Heap
	Activation Record		Sprite.getY()		Sprite
	Local vars.		Local vars.	Ret.	s_1
	s	y	this		y
// initial state:	s_1				5
int y = **s.getY()**;	R s_1		W s_1		5
return y;	s_1		R s_1	W 5	R 5
int y = s.getY();	s_1	W 5		R 5	5

(i.e., in Figure 1, steps with shaded cells, locations with thick vertical arrows). This makes perfect sense, given that these are exactly the execution steps and memory locations the programmer will need to look at when traversing the infection chain.

To determine a history slice, we need to pick an execution step from which to navigate backwards. This step is called the "slicing criterion". When debugging, the slicing criterion usually is the failing execution step. In this study, the criterion is an assert statement (where a unit test would fail).

A history slice is similar to a program slice (more concretely, a dynamic backward slice). The concepts are not the same, though: while a program slice represents a set of source code locations, a history slice represents a set of execution steps. Moreover, unlike in program slicing, we do not consider control dependencies when slicing, as we will discuss in Section 8.

6.3 Benefits of Optimization

Figure 5 shows how the two optimizations affect the size of space-time views. Operation abstraction reduces the rows by about 75% and the columns by about 30%. This is a significant improvement, and it comes with a notion of execution step that is more familiar to developers. However, it also causes some loss of precision, because each row may now read from multiple locations and write to multiple locations, and the internal information flow inside the abstract operation will be hidden. History slicing is even more effective, shrinking rows and columns by about two orders of magnitude. Many of the resulting space-time views will now fit on a single screen, making debugging practical even without powerful navigational aids. However, slicing removes a lot of contextual information that may be relevant for understanding. Thus, in a space-time view visualization tool, slicing probably should be a feature the user can switch on and off.

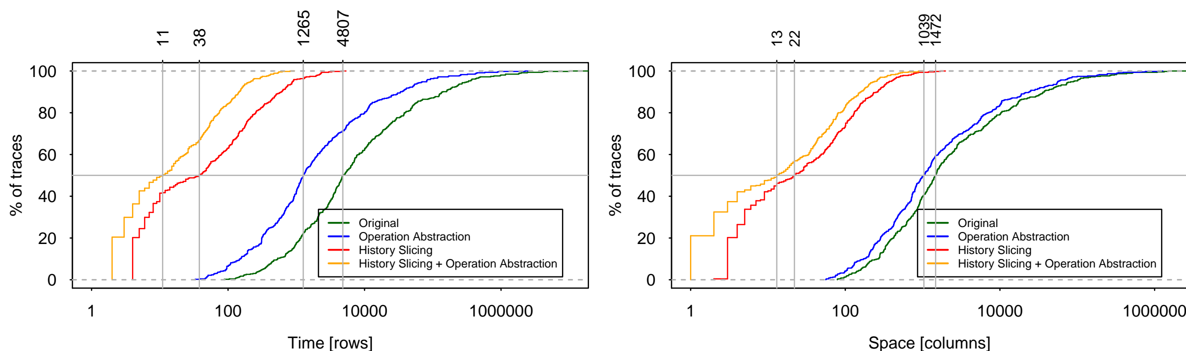

Figure 5: How optimizations reduce space-time view size

Figure 6 shows that both optimizations improve density and thus reduce the amount of white space. This leads to more compact visualizations.

Figure 7 shows the benefit of slicing in terms of the visual distance between a read and the corresponding write of a memory location (see Figure 3 for a visualization of this distance). History slicing reduces the median number of rows the user has to scan through (from the execution step that reads a value from a memory location to the execution step that wrote that value) from 295 down to 2 for bytecode-level his-

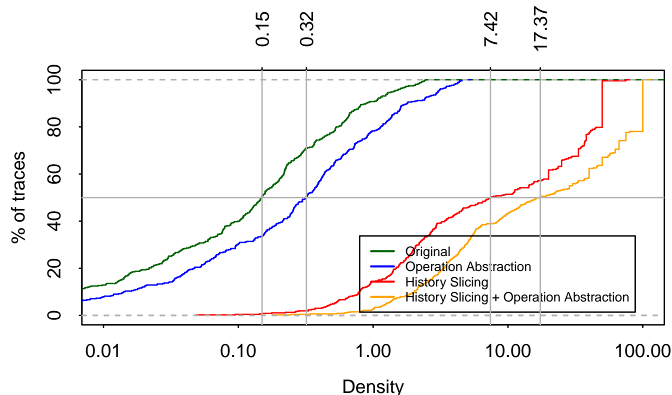

Figure 6: How optimizations affect space-time view density

tories. When combined with operation abstraction, the median number of rows drops from 25 for the full history to 1 for the sliced history.

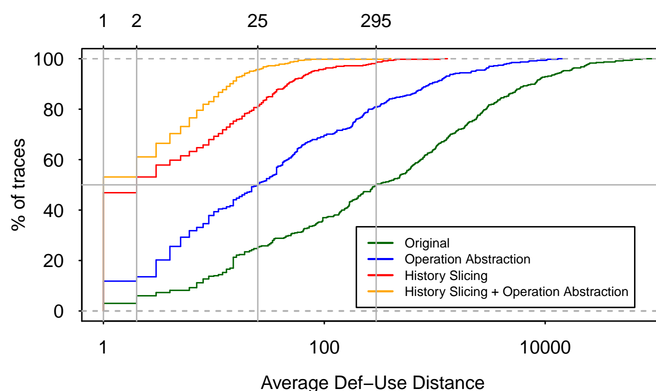

Figure 7: Distance from memory location reads to their corresponding writes

7 Related Work

Space-time views integrate space, time, dependencies, and slicing for omniscient debugging.

Space. The data display debugger (DDD) [7] provides a visualization of program state, allowing developers to gradually unfold and explore data structures of a running program. Memory Graphs [8] automatically visualize such structures, without requiring the manual unfolding by the developer. By comparing multiple memory graphs representing specific program states, memory graphs can be used for isolating cause-effect chains [9]. Memory graphs could provide an intuitive two-dimensional visualization of a selected row in a space-time view, or of the difference between two rows.

Time. A rich body of work exists that visualizes program executions in various ways, but that does not provide all the details needed for understanding information flow. Jinsight [10] is a tool for understanding program executions. Its visualizations include call trees (which could be useful as row header of a space-time view), heap structures (to understand references between heap objects), and it provides ways to collapse repetitive behavior into higher-level patterns. Jive [11] visualizes program executions using UML sequence and object diagrams. These diagrams focus on method invocations: they show which object invoked which method on which other object, and which object points to which other object. Code-Bubbles [12] allows developers to incrementally compose a visual representation of the important code, data, and notes throughout a debugging session. EvoSpaces [13] visualizes execution histories as a movie. Such time-based animated visualizations make it hard to juxtapose two related points in the program execution (such as the setting of a variable value and the use of that value) that are far apart in time.

Dependencies and slicing. Prior work focused on visualizing the dependencies in programs, or on using dependencies to produce more effective visualizations. SeeSlice [14] is a tool to visualize program slices. While the actual slicing operations used by the tool require the detailed information contained in a space-time view, the SeeSlice visualizations omit data flow information and only focus on code (source files, functions, and statements). Moreover, program slices are sets of source code statements, while space-time views show execution steps. The Whyline [15] aims at the same goal as space-time views: helping to find the causes of program failures. However, the WhyLine does not provide a global view of the execution, it shows a timeline of dependent execution events. The Whyline study found that users were reluctant of following data dependencies. We believe that making data (not just the dependencies) explicit—as columns of space-time views—will make the navigation of data dependencies more natural.

Omniscient debugger GUIs. ODB [3] is an omniscient debugger for Java with a GUI similar to a traditional debugger: it presents the state of the system at a given point in time. Its "method trace" serves as a visual history, but it only shows method calls and returns. TOD [5] is similar to ODB. TOD's "control flow view" extends the method trace with information about executed statements. Its "event murals" provide a high-level overview of behavior. Both ODB and TOD provide ways to navigate dependencies and to inspect the state at a selected point of execution, but they do not integrate space and time into a single visualization.

Amber [16] is a back-in-time debugger for native applications. It presents a "timeline" showing the subset of execution steps that are the result of a query over the execution history. The GUI focuses on the time dimension but does not explicitly represent the space dimension. However, like ODB and TOD, it allows users to select a specific execution step and to investigate the state at that point.

Unstuck [17] is a back-in-time debugger for Smalltalk Squeak. In addition to a "method trace", it also provides an "object history" that shows the subset of events in which a specific object was involved. Lienhard et al. [4] implemented another back-in-time debugger in Squeak. They argue that most subtle bugs are caused by aspects observable in explicit information flow, and thus, like us, they don't represent implicit information flow via control-flow dependencies. Compass [18], a related debugger in Squeak, provides a "navigation history". This is not an execution history, but a history of the points in the execution history the developers looked at. The use of a navigation history is similar in spirit to CodeBubbles, the Whyline, and also to Jikes RDB [19], a debugger used to inspect the internal state of virtual machines. Developers can navigate their exploration history like a user can navigate their web browser history (using a backward button), and they have the ability to bookmark points they looked at. The Compass debugger also provides a *graph-based visualization of the method trace*. That visualization omits most details: it just represents each method invocation as a node in a big call tree. This is similar to a high-level abstraction over the y-axis of a space-time view: instead of showing each and every instruction inside each and ever method invocation; one could collapse all rows representing a method invocations into a single row, and would then end up with the same kind of tree structure.

8 Limitations

We now discuss the limitations of this work.

Only unit tests. While we traced test runs with 47449928 executed bytecode instructions overall, we did not study histories of complete production runs. We believe that due to their overhead, omniscient debuggers will first be used for debugging unit test failures, thus that is what we focused on.

Only one test suite. Our study focuses on one test suite. While that suite is reasonably large (we analyzed 611 different unit test methods), it would be interesting to generalize our results by using a number of different test suites.

No implicit information flow. We focus on *explicit* information flow. Using our bytecode-level space-time views, it is possible to determine where exactly a certain value came from. However, the space-time views described in this paper do not take control-flow into consideration. This means they do not allow the tracking of *implicit* information flow. To track implicit information flow, the space-time views would have to be extended. They would need one additional row and column for each control flow instruction (i.e., branch instruction). The column would represent a pseudo memory location representing the implicit information flow from the branch to all instructions dominated by that branch.

No tool. While we specify the meaning of space-time views, we do not provide a specific implementation of a visualization tool. The goal of this paper is to provide the foundations and inspiration for developers of such tools. The large size of the views shows that a tool needs to be highly interactive and that it has

to integrate techniques such as folding and unfolding of the hierarchically structured time or space, and slicing to automatically fold irrelevant regions of time and space.

9 Conclusions

While space-time views are used in programming and program analysis textbooks, to the best of our knowledge, we are the first to define the precise meaning of space-time views for a real-world language like Java. We characterize the space-time views we automatically generated from a set of real-world unit test execution traces. We propose two approaches to reduce the amount of information: "operation abstraction" and "history slicing", and we study their effectiveness. We hope that our results will help visualization developers to design more effective tools for visualizing and navigating program executions captured by the omniscient debuggers that are now at the forefront of program analysis research.

Acknowledgments

The first author has been supported by the Swiss National Science Foundation under grant 200021_135245.

References

[1] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

[2] G. Bracha, "Debug mode is the only mode," http://gbracha.blogspot.ch/2012/11/debug-mode-is-only-mode.html.

[3] B. Lewis, "Debugging backwards in time," in *Proceedings of the Fifth International Workshop on Automated Debugging*, ser. AADEBUG'03, October 2003.

[4] A. Lienhard, T. Gîrba, and O. Nierstrasz, "Practical object-oriented back-in-time debugging," in *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ser. ECOOP '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 592–615.

[5] G. Pothier, E. Tanter, and J. Piquer, "Scalable omniscient debugging," in *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, ser. OOPSLA '07. New York, NY, USA: ACM, 2007, pp. 535–552. [Online]. Available: http://doi.acm.org/10.1145/1297027.1297067

[6] D. J. Barnes and M. Kölling, *Objects First with Java: A Practical Introduction using BlueJ*, 5th ed. Prentice Hall / Pearson Education, 2012, ch. 7, pp. 257–258.

[7] A. Zeller and D. Lütkehaus, "Ddd—a free graphical front-end for unix debuggers," *SIGPLAN Not.*, vol. 31, no. 1, pp. 22–27, Jan. 1996. [Online]. Available: http://doi.acm.org/10.1145/249094.249108

[8] T. Zimmermann and A. Zeller, "Visualizing memory graphs," in *Revised Lectures on Software Visualization, International Seminar.* London, UK, UK: Springer-Verlag, 2002, pp. 191–204. [Online]. Available: http://dl.acm.org/citation.cfm?id=647382.724787

[9] A. Zeller, "Isolating cause-effect chains from computer programs," in *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, ser. SIGSOFT '02/FSE-10. New York, NY, USA: ACM, 2002, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/587051.587053

[10] W. D. Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. M. Vlissides, and J. Yang, "Visualizing the execution of java programs," in *Revised Lectures on Software Visualization, International Seminar.* London, UK, UK: Springer-Verlag, 2002, pp. 151–162. [Online]. Available: http://dl.acm.org/citation.cfm?id=647382.724791

[11] P. Gestwicki and B. Jayaraman, "Methodology and Architecture of JIVE," in *Proceedings of the 2005 ACM Symposium on Software Visualization*, ser. SoftVis '05. New York, NY, USA: ACM, 2005, pp. 95–104. [Online]. Available: http://doi.acm.org/10.1145/1056018.1056032

[12] A. Bragdon, S. P. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, Jr., "Code bubbles: Rethinking the user interface paradigm of integrated development environments," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 455–464. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806866

[13] P. Dugerdil and S. Alam, "Execution trace visualization in a 3d space," in *Information Technology: New Generations, 2008. ITNG 2008. Fifth International Conference on*, April 2008, pp. 38–43.

[14] T. Ball, S. G. Eick, and S. G. Eick, "Visualizing program slices," in *In IEEE/CS Symposium on Visual Languages.* IEEE Computer Society Press, 1994, pp. 288–295.

[15] A. J. Ko and B. A. Myers, "Finding causes of program output with the java whyline," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '09. New York, NY, USA: ACM, 2009, pp. 1569–1578. [Online]. Available: http://doi.acm.org/10.1145/1518701.1518942

[16] R. O'Callahan, "Efficient collection and storage of indexed program traces," Unpublished manuscript. https://www.cs.purdue.edu/homes/xyzhang/fall07/Papers/Amber.pdf.

[17] C. Hofer, M. Denker, and S. Ducasse, "Design and implementation of a backward-in-time debugger," in *Proceedings of NODE 2006*. Gesellschaft für Informatik (GI), September 2006, pp. 17–32.

[18] J. Fierz, "Compass: Flow-Centric Back-In-Time Debugging," Master's thesis, University of Bern, 2009.

[19] D. Makarov and M. Hauswirth, "Jikes rdb: a debugger for the jikes rvm," in *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, ser. PPPJ '13. New York, NY, USA: ACM, 2013, pp. 169–172. [Online]. Available: http://doi.acm.org/10.1145/2500828.2500847