Thin Slicing

Manu Sridharan
Stephen J. Fink
Ras Bodik

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-184
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-184.html

December 18, 2006

Thin Slicing

Manu Sridharan
University of California, Berkeley
manu_s@cs.berkeley.edu

Stephen J. Fink
IBM T.J. Watson Research Center
sjfink@us.ibm.com

Rastislav Bodík
University of California, Berkeley
bodik@cs.berkeley.edu

Abstract

Program slicing provides a systematic technique to identify parts of a program relevant to a particular seed. Unfortunately, slices of modern programs often grow too large for human consumption in program understanding or browsing tools. We argue that unwieldy slices arise primarily from an overly broad definition of relevance, rather than from analysis imprecision. While a traditional slice includes all statements that may *affect* a value of interest, not all such statements appear equally relevant to a human.

As an improved method of finding statements relevant to a human, we propose *thin slicing*, a technique based on value-flow relevance. A thin slice consists of statements that may "copy-propagate" a value to the seed, and excludes statements that may indirectly affect value flow. For example, for a seed that reads a value from a container object, a thin slice includes statements that store a value into the container, but excludes statements that manipulate pointers to the container itself. We also show how a tool can support incremental thin slice expansion to explain aliasing and control dependence, yielding a traditional slice in the limit.

Experimental results show that thin slices usually include the most relevant statements for a set of debugging and program understanding tasks. Furthermore, using breadth-first inspection to simulate realistic use of a slicing tool, thin slices reveal interesting target statements after inspecting up to 4.52X fewer statements than traditional slicing for debugging tasks, and up to 34.2X fewer for program understanding tasks. Finally, experimental results show that an effective thin slicing algorithm scales well to relatively large Java benchmarks, suggesting that thin slicing represents an attractive option for practical tools.

1. Introduction

> *"Thin-slicing is part of what makes the unconscious so dazzling. But it's also what we find most problematic about rapid cognition. How is it possible to gather the necessary information for a sophisticated judgment in such a short time?"* Malcom Gladwell, *Blink: The Power of Thinking Without Thinking*

Motivation Today's programmers rely more on libraries and frameworks than ever before. While rich libraries promote code reuse, a productive developer must rapidly digest huge and complex Application Programmer Interfaces (APIs) to use them. Today's standard libraries already contain more code than a programmer can hope to master, and frameworks further compound the complexity.[1] In many frameworks, programmers spend hours learning simple usage patterns [20] and even more time debugging, which requires deeper understanding.

When trying to understand object-oriented code, heap-allocated data and myriad associated data structures present key difficulties. Levels upon levels of pointer indirection can make manually tracing the flow of data through the heap prohibitively difficult. For these situations, programmers could benefit from a tool that abstracts away irrelevant details of heap behavior during code inspection and debugging.

Slicing is a well-known technique for focusing on a subset of the program that is relevant to a statement or value of interest, called a *seed*. Slicing applies to a variety of program understanding tasks, ranging from testing and debugging to reverse engineering [29]. Weiser [30] originally defined a slice as an *executable* program subset in which the seed statement performs the same computation as in the original program. Weiser's definition is elegant and intuitive, but imposes a rather broad definition of relevance: any statement t that could possibly *affect* the computation at the seed statement s must appear in the slice. We claim that this definition pollutes slices with many statements that may affect a seed but do not help for typical program understanding tasks.

Consider a value stored in a deeply nested data structure, *e.g.*, a hash table which holds trees with lists at each tree node. A backwards slice for a read from one such list must include all statements that construct and manipulate the nested data structure. For many program understanding tasks, the programmer needs information about the values stored in the list, but doesn't care about the management of meta-data which holds onto the value through pointers. For these common cases, the backwards slice presents far too much information for the task at hand.

Furthermore, modern programs typically rely heavily on well-tested data structures provided by standard libraries; the internal implementation of these data structures, though included in the slice, do not interest end-user programmers. Note that the slice grows too large necessarily by its definition, as opposed to conservative analysis imprecision.

Our Approach This paper presents *thin slicing*, a dependence analysis technique based on a new notion of relevance. We demonstrate this alternate relevance notion on the Java program fragment of Figure 1, which manipulates `String`s stored in a container. Given full names as input, the example extracts the first names and stores them in a `Vector` (the `readNames()` function), and then later prints out the first names (the `printNames()` function). The `main()` method illustrates a use of the code in a web application, storing and retrieving the names from a `SessionState` object. The example contains a bug: when the program receives as input full name "John Doe", line 24 erroneously prints "FIRST NAME: Joh".

Unfortunately, a traditional slicer doesn't help much with this bug, as a slice seeded with the `firstName` expression at line 24[2] includes all the code in the example. The slice must include all the code to construct and populate the `Vector` passed to

[1] According to [20], the J2SE 1.4 standard library includes at least 21,000 methods and Eclipse [2] includes more than 100,000.

[2] For clarity, we refer to relevant expressions as being in the slice, which actually includes the corresponding statements.

```
1  class Vector {
2    Object[] elems; int count;
3    Vector() { elems = new Object[10]; }
4    void add(Object p) {
5      this.elems[count++] = p;
6    }
7    Object get(int ind) {
8      return this.elems[ind];
9    } ...
10 }
11 Vector readNames(InputStream input) {
12   Vector firstNames = new Vector();
13   while (!eof(input)) {
14     String fullName = readFullName(input);
15     int spaceInd = fullName.indexOf(' ');
16     String firstName = fullName.substring(0,spaceInd-1);
17     names.add(firstName);
18   }
19   return firstNames;
20 }
21 void printNames(Vector firstNames) {
22   for (int i = 0; i < firstNames.size(); i++) {
23     String firstName = (String)firstNames.get(i);
24     print("FIRST NAME: " + firstName);
25   }
26 }
27 void main(String[] args) {
28   Vector firstNames = readNames(new InputStream(args[0]));
29   SessionState s = getState(); s.setNames(firstNames);
30   ...;
31   SessionState t = getState();
32   printNames(t.getNames());
33 }
```

Figure 1. Example showing the advantages of thin slicing. The six statements with underlined expressions are in the thin slice seeded with line 24, while the traditional slice contains all displayed code. Some functions with behavior not essential to the example (like `print()`) have been elided for clarity.

`printNames()`, and the code in `main()` retrieving the `Vector` from the `SessionState` object, which all affect line 24. As in this example, slices for Java programs typically include most of the program.

What lines of code are most relevant for the debugging task in this example? The bug lies at line 16, which incorrectly passes `spaceInd-1` (rather than `spaceInd`) to `String.substring()`. Seeing how this erroneous `String` flows to where it is printed would almost immediately lead the user to the problematic line. In this case, the flow traverses a `Vector`: line 17 adds the `String`, and line 23 retrieves it. If this code were part of a larger Java program, manually tracing this flow through the `Vector` could be difficult, as such programs often include many unrelated uses of `Vector`s that the user must filter out.

A thin slice is based on the notion of *value-flow relevance*. Intuitively, a statement s is value-flow relevant to statement t if s *copy-propagates* a value v to t, *i.e.*, if s is part of a chain of assignments that copy v to t. In Figure 1, the value-flow relevant statements for the seed, highlighted with underlining, are almost exactly the statements most relevant to the bug. We are interested in the pointer value in `firstName` at line 24, and the thin slice allows us to easily trace its flow (relevant expressions are underlined):

- Line 23 copies the value returned by `Vector.get()`.
- `Vector.get()` obtains the value from an array read (line 8).
- The value is copied into the array in `Vector.add()` (line 5).
- `Vector.add()` gets the value from the actual parameter at line 17.
- Line 17 passes the value returned at line 16, the buggy statement.

Unlike a traditional slice, the thin slice does not provide a standalone executable. However, for use in a tool, the thin slice more effectively identifies statements most relevant to the bug.

Advantages of Thin Slicing One reason that thin slicing works well is that it ignores flow into base pointers of field accesses, focusing just on the value flowing through the accesses. For example, line 8 reads `this.elems[ind]`. A thin slicer ignores the values of the two dereferenced base pointers (`this` and `this.elems`), focusing solely on statements that can write into the array (*i.e.*, line 5). In contrast, a traditional slicer or data dependence analysis includes statements influencing both `this` and `this.elems`, which contributes to the blowup in slice size. For many program understanding tasks, base pointer manipulation matters less than actual copying of the value through the heap.

Value-flow relevance gives thin slices a notion of soundness, despite the fact that many "affecting" statements from the traditional slice are excluded. A thin slice is sound with respect to value flow: for any non-base-pointer value v used by the seed, the thin slice contains all statements that copy-propagate v to the seed. In contrast, if slices were shrunk using some ad-hoc method, such as setting a constant limit on slice size, the user could not easily characterize what is in the presented slice subset and what is missing. Apart from making thin slices more understandable, this notion of soundness can be essential for some applications. For example, when performing an audit to see if untrusted inputs can flow to a sensitive statement s (*e.g.*, web form inputs flowing to a database query [19]), a thin slice will show all statements that incoming values passed though, soundly indicating whether the values have passed through some "cleansing" function before reaching s.

We have also developed a technique to incrementally and hierarchically expand thin slices, to explain aliasing of base pointers. For example, given a field read `x := y.f` and a field write `w.f := z` in a thin slice, a user may ask how aliasing between `y` and `w` arises, causing value flow from `z` to `x`. This question can be answered via two more thin slices, respectively showing how some object o flows to both `y` and `w`. This expansion can repeat recursively for further aliasing queries, yielding a traditional data slice in the limit.

The same technique applies to understanding statements upon which the seed is transitively *control dependent*, *i.e.*, conditional branches that affect whether thin slice statements execute. To produce executable slices, traditional slicers must include all transitive control dependences. Unfortunately, Java's semantics make many statements a type of conditional branch. For example, if a statement might throw an exception, many statements will be control dependent on its successful execution. Also, each virtual call `x.m()` is a conditional, branching on the runtime type of `x` to determine which implementation of `m()` to invoke. Due to the number of potentially branching statements, control dependences quickly cause a huge blowup in the size of a traditional slice. In practice, we found that when control dependences are relevant, they are often apparent from browsing the source code around a statement in the thin slice, so exposing one or two levels of control dependence from such statements suffices.

Of course, thin slicing does not provide a panacea: in certain cases thin slices with expansion grow too large to effectively identify statements of interest. However, in most cases we tested, thin slices both include the most relevant statements and and effectively narrow the scope of the program navigated during a program understanding task.

The current paper focuses on static thin slicing for Java, but the techniques are more broadly applicable. As value-flow relevance is a general concept, thin slicing should apply to many programming languages. Our hierarchical expansion technique relies on properties of Java pointer accesses, however, and many not work as well for languages like C (see Section 4). Value-flow relevance can just as easily be computed over dynamic data dependences (Section 2 defines thin slices over a concrete semantics).

Contributions This paper makes the following contributions:

- We define value-flow relevance and thin slicing, and present simple modifications to existing slicing algorithms for computing both context-insensitive and context-sensitive thin slices.
- We present a method for hierarchically expanding thin slices to explain aliasing and control dependence, in the limit yielding a traditional slice.
- We compare thin slicing and traditional slicing for several debugging and program understanding tasks, using a methodology that simulates realistic use of a slicing tool (details in Section 5). Our results show that (1) thin slices usually include the most relevant statements for the tasks, and (2) thin slices reveal interesting target statements after inspecting up to 4.52X fewer statements than traditional slicing for debugging tasks, and up to 34.2X fewer for program understanding tasks. We also show that our thin slicing algorithm scales to relatively large Java benchmarks.

The rest of this paper is organized as follows. Section 2 defines thin slices more formally, and Section 3 presents algorithms for computing thin slices as deltas from a traditional slicing algorithm. Section 4 describes our technique for expanding thin slices to explain aliasing, and our handling of control dependence. Section 5 gives our experimental evaluation, Section 6 discusses related work, and Section 7 concludes.

2. Defining Thin Slices

The section defines a simple language of pointer assignments, and then formally define thin slices over that language. We also describe how we to express other Java language features in the simple language.

We will illustrate aspects of the thin slice definition on the example code in Figure 2. Recall that intuitively, a thin slice for s contains statements copy-propagating values to s, where dereferences of base pointers are ignored. So, the thin slice for statement 8 is $\{8, 7, 2\}$, showing how the `B` object allocated at line 2 flows to `w`. The thin slice for statement 6 is $\{6, 5, 2\}$, showing flow of the `B` object to `z.f`, while ignoring flow into the base pointer `z`.

2.1 Language and Semantics

We define thin slices over a very small language of Java-like pointer assignments. For the purposes of thin slicing, most other Java language features can be suitably translated into this language, and the language is similar to representations commonly used for Java flow analysis (*e.g.*, [16]). The grammar rules are as follows:

$$\begin{array}{rcl} \mathit{locname} & ::= & x \mid x.f \\ \mathit{expr} & ::= & \textbf{new}\, C \mid \mathit{locname} \\ s & ::= & \mathit{locname} := \mathit{expr} \mid s_1 \,;\, s_2 \end{array}$$

Note that in the following discussion, we use meta-variables m and m' to range over $locname$s. The language is clearly a toy, as it only allows straight-line code and has no primitive types or functions, but these features can easily be added without impacting the thin slice definition. We discuss handling of other language features in Section 2.3.

```
1  x = new A;
2  y = new B;
3  w = x;
4  z = x;
5  w.f = y;
6  v = z.f;
7  w = y;
8  z = w
```

Figure 2. Simple example to illustrate the definition of thin slicing.

A $locname$ is a name for a *location*, a runtime storage slot for a value. We have two types of locations at runtime: (1) a *variable location* l_x for each variable x, and (2) a *field location* $l_{o.f}$ for each field f of each object o. Since the only values in our language are objects, the runtime store σ maps locations l to objects o.

The function $\Gamma_\sigma(m)$ maps a $locname\ m$ to the corresponding location in σ:

- $\Gamma_\sigma(x) = l_x$
- $\Gamma_\sigma(x.f) = l_{o.f}$, where $\sigma(l_x) = o$.

Given a statement s of the form $m := e$, let $dstloc(s, \sigma) = \Gamma_\sigma(m)$ and $srcloc(s, \sigma) = \Gamma_\sigma(e)$; note that $srcloc(m := \textbf{new}\ C, \sigma)$ is undefined. Intuitively, a statement s copies the value from its $srcloc$ to its $dstloc$. Notice that $srcloc$ does not include the location of the base pointer for a field access; this leads our thin slice definition in Section 2.2 to ignore base pointers, as desired.

Given Γ, we define an instrumented operational semantics that describes updates to both the store σ and a set of *live copies* ϕ, which will be used to define the thin slice in Section 2.2. A live copy at program point p is a pair of memory locations (l_i, l_j) such the pointer in l_i at p was copied from l_j and not subsequently overwritten. In Figure 2, (l_w, l_x) is a live copy after line 3, but not after line 7. Live copies correspond to dynamic reaching definitions, defined over locations rather than variables and statements.

The three rules of the semantics are of the form $\langle s, \sigma, \phi\rangle \Downarrow (\sigma', \phi')$:

$$\frac{l_i = \Gamma_\sigma(m) \qquad o \text{ fresh} \qquad \sigma' = \sigma[l_i \mapsto o] \qquad \phi' = \phi - \bigcup_k (l_i, l_k)}{\langle m := \textbf{new}\ C, \sigma, \phi\rangle \Downarrow (\sigma', \phi')} \quad (1)$$

$$\frac{l_i = \Gamma_\sigma(m) \qquad l_j = \Gamma_\sigma(m') \qquad \sigma' = \sigma[l_i \mapsto \sigma(l_j)] \qquad \phi' = (\phi - \bigcup_k (l_i, l_k)) \cup (l_i, l_j)}{\langle m := m', \sigma, \phi\rangle \Downarrow (\sigma', \phi')} \quad (2)$$

$$\frac{\langle s_1, \sigma, \phi\rangle \Downarrow (\sigma', \phi') \qquad \langle s_2, \sigma', \phi'\rangle \Downarrow (\sigma'', \phi'')}{\langle s_1\ ;\ s_2, \sigma, \phi\rangle \Downarrow (\sigma'', \phi'')} \quad (3)$$

Rule 1 handles assigning a new object to a $locname\ l_i$ by updating σ to map l_i to a fresh object name. Existing live copies into l_i are killed, but no copy is added, since the allocation creates a new object rather than copying a pointer from an existing location. Rules 2 and 3 respectively handle assigning one $locname$ to another and sequencing in the expected manner. Note that for simplicity, these semantics assume that all variables and fields are initialized before they are read.

2.2 The Thin Slice

To define thin slices, we first need to define the notion of *value flow*. Value flow exists from location l_j to location l_i at program point p iff the value (an object pointer for this language) in l_i at p was copied from l_j through a (possibly empty) sequence of assignments. Given l_i and l_j and a set of live copies ϕ, we define the value flow predicate $valFlow(l_i, l_j, \phi)$:

$$valFlow(l_i, l_j, \phi) = \\ l_i = l_j \vee \exists\, l_1, \ldots, l_k \mid (l_i, l_1) \in \phi \wedge (l_1, l_2) \in \phi \\ \wedge \ldots \wedge (l_k, l_j) \in \phi$$

The locations $l_1, \ldots, l_k$ are intermediate locations through which the value flows. As an example, after statement 8 in Figure 2, $\phi = \{(l_{o_1.f}, l_y), (l_v, l_{o_1.f}), (l_w, l_y), (l_z, l_w)\}$. Hence, we have $valFlow(l_z, l_y, \phi)$, since a value is copied from l_y to l_w at line 7 and then from l_w to l_z at line 8. We also have $valFlow(l_v, l_y, \phi)$, since a value is copied from l_y to $l_{o_1.f}$ at line 5 and then from $l_{o_1.f}$ to l_v at line 6 (o_1 is the object allocated at line 1).

Now we have all the pieces needed to define the thin slice. Let the program P be $s_1; s_2; \ldots; s_n$. The thin slice for statement s_k, written $T(s_k)$, is defined as follows ($\perp$ is the empty map):

$$T(s_k) = s_k \cup \{\, s_i \mid i < k \wedge \langle s_1; \ldots; s_{i-1}, \perp, \emptyset \rangle \Downarrow (\sigma, \phi) \\ \wedge\ dstloc(s_i, \sigma) = l \\ \wedge\ \langle s_i; \ldots; s_{k-1}, \sigma, \phi \rangle \Downarrow (\sigma', \phi') \\ \wedge\ srcloc(s_k, \sigma') = l' \\ \wedge\ valFlow(l', l, \phi')\ \}$$

This definition states that a thin slice for statement s of the form $m := e$ must include (1) s itself and (2) all statements s' such that there is value flow from the location written by s' to the location holding the value of e. We say that any statement in the thin slice for statement s is *value-flow relevant* to s.

Note that our definition of a thin slice is based on concrete semantics. An analysis could directly compute such a thin slice using dynamic data dependences. When computing thin slices statically for a realistic language, the thin slicer must of course (1) compute thin slices that capture all possible runtime behavior, and (2) abstract some of the semantics for decidability and scalability. In Section 3, we give algorithms for statically computing thin slices for Java.

Thin Slice vs. Traditional Slice Thin slices differ in two key ways from traditional slices: (1) control dependences are excluded, and (2) dependences of base pointers are excluded. Our handling of base pointers makes value-flow relevance narrower than classic flow dependence [11], the notion used by traditional slicing for handling data dependence. A statement s is flow dependent on statement s' if s' updates *any* memory location read by s. So, a statement `x := y.f` in store σ is flow dependent on statements updating l_y and on statements updating $l_{o.f}$, where $\sigma(l_y) = o$. In contrast, for this statement we have $srcloc(\texttt{x := y.f}, \sigma) = l_{o.f}$, and hence the value-flow relevant statements are only those that flow values into $l_{o.f}$. So, thin slices include a subset of statements in a traditional data slice.

2.3 Other Language Features

Although the previous presentation considers a simple toy language, it is straightforward to extend the language for standard imperative language features.

Concerning primitive values and expressions: each primitive expression can be treated as if it either 1) creates a new value, or 2) propagates value flow from each argument to the result. Under the first option, the thin slice for `a := b + c` would include that statement alone. Under the second option, thin slices of `b` and `c` are included in the thin slice for `a`. Our implementation uses choice 2), but the issue rarely arose in the experiments reported.

Handling of most other Java program constructs is fairly straightforward. Loops and if-then-else constructs require adding a conditional branch statement and the definition of its semantics, but otherwise do not affect the thin slice definition. Parameter passing and return values for method calls can be modeled directly in the language using a heap-allocated stack frame, with fields holding the parameters, locals, and return value of the call. Reads and writes of array contents can be modeled as field reads and writes, where the field is the array index.

Straightforward extensions also apply to virtual dispatch, exceptions, and other object-oriented language features.

3. Computing Thin Slices

Computing a thin slice entails computing interprocedural reaching definitions for statements, including definitions of heap locations. We briefly present two simple algorithms to compute thin slices, one context insensitive and one context sensitive, applying well-understood concepts from slicing.

In both thin slice approaches, we first compute a subset of the System Dependence Graph of Horwitz et al. [12]. We exclude control dependence edges, and include a subset of the standard SDG edges for heap accesses. Graph construction relies on a pre-computed points-to analysis, to determine def-use information for loads and stores, and a pre-computed call graph, for connecting method calls with possible callees.

For a simple language of Section 2.1 enhanced with procedure calls, the graph contains the expected structure of the data dependence subset of an SDG:

1. For a statement `x := e`, where `x` is a local, we add edges to all statements of the form `y := x` or `y.f := x`. We operate on an SSA representation, so these edges are added flow sensitively.
2. For an actual parameter node for a call to method `m()`, we query the pre-computed call graph to find the possible call targets $m_1, \ldots, m_k$. Then, for each m_i, we add an edge from the actual parameter node to the corresponding formal parameter node. Return values are handled similarly.

Our thin slicing algorithms differ from the standard SDG handling of data dependence, and from each other, in their treatment of definitions of heap locations (*i.e.*, statements of the form `x.f := e`) as described below.

3.1 Context-Insensitive Variant

Our first algorithm builds an SDG variant and computes flat (context-insensitive) graph reachability to compute thin slices. In this approach,

- For a statement `x.f := e`, we add an edge to each statement `y := w.f` such that the pre-computed points-to analysis indicates `x` *may-alias* `w`.

Note that we add edges to statements *in other procedures*, and not just to statements in the same procedure. The advantage of this approach is that we need not introduce parameter call and return nodes for each heap location involved in a slice, as is done with traditional slicing [13]. In practice, this reduces the graph size by a large factor, since def-use chains through heap locations often span many procedure boundaries.

Having computed the graph, a simple transitive closure gives the thin slice for a particular seed. It is straightforward to construct the graph and do the traversal in a demand-driven fashion. A potential disadvantage of this approach is that it may return unrealizable paths [25] due to lack of context sensitivity (Section 5 shows this issue is not significant in practice).

3.2 Context-Sensitive Variant

The context-sensitive thin slicing algorithm uses an SDG variant closer to that of traditional slicing, created compositionally from program dependence graphs (PDGs) for each procedure. In contrast to the previous approach, this approach handles field writes as follows:

- For a statement `x.f := e`, we add an edge to each statement `y := w.f` *in the same procedure* such that the pre-computed points-to analysis indicates `x` *may-alias* `w`.

We handle interprocedural heap flow the same way as the standard SDG, with heap reads and writes modeled as extra parameters and return values to each procedure [5, 13]. Our implementation introduces such parameters using the same heap partitions as used by the preliminary pointer analysis. Discovering the appropriate set of parameters for each procedure requires an interprocedural mod-ref analysis [28], computed using the result of the pre-computed points-to analysis.

Having built the graph, we compute context-sensitive reachability as a partially balanced parenthesis problem [24]. Our implementation relies on a backwards, demand-driven tabulation algorithm [25].

In our experience, constructing an SDG using heap parameters can be very expensive for large programs. Furthermore, we found that for realistic usage patterns, context sensitivity did not provide much benefit for thin slicing. See Section 5 for details.

Our implementation handles the full Java Virtual Machine bytecode language excluding concurrency. We model array reads and writes by collapsing array indices using a pseudo-field `arr` for the array contents: `a := b[i]; c[j] := d` becomes `a := b.arr; c.arr := d`. We do not discuss language features such as dynamic dispatch, exceptions, bytecode subroutines, default field values, and type tests; these complicate the implementation but do not introduce any subtle difficulties.

4. Expanding Thin Slices

Here, we discuss how thin slices can be expanded to explain aliasing between base pointers and control dependence. Given a thin slice T, the user may still ask:

1. Given statements `x := y.f` and `w.f := z` in T such that `w` and `y` may be aliased (causing value flow from `z` to `x`), what statements cause the aliasing?
2. Under what conditions can some statement s in T execute?

These questions do not arise with a traditional slice, since it includes all affecting statements (at the cost of producing large slices).

In an approach based on thin slicing, a tool will answer these questions driven by user requests. Section 4.1 discusses a technique for hierarchically expand thin slices to answer question 1. Section 4.2 discusses our handling of control dependences.

Example We use the example in Figure 3, a simple program fragment manipulating a file, to illustrate thin slice expansion. The `File` class displays a small part of its implementation, the tracking of whether the file is open using a `boolean` field. The `readFromFile()` function throws an exception if the file passed to it is not open. Finally, the `main()` method creates a file, erroneously closes it, and then passes it to `readFromFile()`, causing the exception. The `File` object is read from a `Vector` before being passed to `close()` and `readFromFile()`, complicating discovery of the bug.

```
1  class File {
2    boolean open;
3    File() { ...; this.open = true; }
4    isOpen() { return this.open; }
5    close() { ...; this.open = false; }
6    ...
7  }
8  readFromFile(File f) {
9    boolean open = f.isOpen();
10   if (!open)
11     throw new ClosedException();
12   } ...
13 }
14 main() {
15   File f = new File();
16   Vector files = new Vector();
17   files.add(f);
18   ...;
19   File g = (File)files.get(i);
20   g.close();
21   ...;
22   File h = (File)files.get(i);
23   readFromFile(h);
24 }
```

Figure 3. An example for showing expansion of thin slices, similar to an example we saw in our evaluation. The bug is an exception thrown at line 11, and understanding the bug requires an explanation of aliasing (Section 4.1) and following a control dependence (Section 4.2). We use single underlines to highlight relevant expressions in the initial thin slice, and double underlines for additional expressions included after expansion to explain aliasing.

4.1 Question 1: Explaining Aliasing

When a thin slice includes statements that copy a value through a heap read and write, sometimes the user needs to understand why those statements access the same heap location. For the example of Figure 3, suppose that the user asks for a thin slice from line 10 to determine why line 11 threw an exception. The computed thin slice will be $\{3, 4, 5, 9, 10\}$ (highlighted with underlines), the only statements that can touch the boolean `open` value. Clearly, these statements fail to completely diagnose the bug: the user still does not know which `File` passes to `close()` before being passed to `isOpen()`. The relevant question boils down to understanding which statements cause the '`this`' pointers of `close()` and `isOpen()` to be aliased.

We can expand thin slices to answer aliasing questions by computing two additional thin slices for the base pointers in question. These thin slices will show why some common object o flows to both of the base pointers, showing that they can be aliased. For Figure 3, the common object for the '`this`' parameters of `close()` and `isOpen()` is the `File` allocated at line 15. Double underlines in Figure 3 indicate the statements added to explain the flow of the `File` (the `Vector` class is elided for clarity). Note line 16 is still omitted, as it does not touch the `File` object. Given these thin slices, the user sees that line 20 closes the `File`, and that the bug could be fixed by either not closing the file or by removing it from the `Vector`.

Explaining aliases using additional thin slices yields an intuitive structure to heap-based flow, making it more understandable for the user. Suppose that statements `x := y.f` and `w.f := z` appear in a thin slice. Expanding the thin slice to show flow into `x` and `w` adds one more level of data dependences to the slice. If during expansion, statements `a := b.g` and `c.g := d` are added, the aliasing of `b` and `c` could be explained with another level of data dependences, and so on. If Figure 3 were changed so that the flow of the `Vector` to the `add()` and `get()` in `main()` was complex (*e.g.*, it got stored in a data structure), another level of thin slices would explain that flow. The ability to show these different levels of aliasing in a struc-

tured manner relies on the fact that only field reads and writes can dereference pointers in Java; in C, which allows for creating pointers to pointers and taking addresses of variables, explanations of why two statements access the same memory location may not be so simple.

We encountered one case in which an explanation of aliasing was necessary in our experiments, and we believe that many similar situations often arise in practice. In our programming experience, we have found that when such bugs arise, they can be tricky to debug, as values can be mutated in unexpected places. Analyses that find typestate bugs [6, 9], *e.g.*, reading from a file after closing it, could benefit from using thin slices to explain bugs that involve aliasing. Such tools sometimes hide error reports that involve aliasing, since there is no mechanism in the tool for explaining the aliasing succinctly [21].

Three additional technical points about explaining aliasing merit mention. First, thin slices that explain aliasing should be restricted to only show the flow of objects that can flow to both base pointers, eliminating some extraneous statements. Second, context sensitivity may be necessary to focus the aliasing explanations in some cases. For example, if the code of Figure 3 were part of a large program where many `File` objects were used, the user would likely want to ask about aliasing '`this`' in `isOpen()` for the particular call at line 9, rather than for all calls. Finally, since our analysis collapses array indices by default when handling arrays (see Section 3), the programmer may also wonder why two array accesses can access the same array index: this question can be answered through thin slices on the array index expressions, which will explain why they can have the same value.

4.2 Question 2: Control Dependence

In our experience, when a debugging or program understanding task requires viewing control dependences, the control-relevant statements usually lie lexically close to some statement in the thin slice. In Figure 3, the bug manifests at line 11, which throws the exception. As there no value flows into the `throw` statement, a thin slice from the throw statement will not aid debugging. However, code inspection immediately shows that the condition of the `if` statement at line 10 is relevant to the bug, as it directly controls whether the exception is thrown. With this information, the obvious next step is to thin slice from line 10 to learn more about the bug, as described in Section 4.1.

While this example may seem contrived, our experiments show that Figure 3 reflects the common case. For nearly all tasks in our evaluation, at most one or two control dependences were relevant, and they all lay syntactically close to statements in the thin slice. Also, for most statements in thin slices, control dependences are not needed at all for understanding the seed behavior. Hence, we believe that in practice, simply showing the thin slice statements in the source code suffices for identifying any relevant control dependences. If more information is needed for a statement controlling a branch, as in Figure 3, the user would simply take another thin slice for the branch statement.

5. Evaluation

We now present an empirical evaluation of thin slicing for debugging and program understanding tasks. Our experiments validate three hypotheses:

- **Thin slices lead the user to relevant statements.** For the debugging and program understanding tasks we considered, thin slices often contain the statements of interest. When control dependences are relevant, the statements are always lexically close to statements in the thin slice. Subjectively, we found a thin slicer very useful for understanding benchmarks in one experiment.
- **Thin slices focus on relevant statements more tightly than traditional slices.** We compared thin slicing to context-insensitive traditional slicing (the configuration that scaled), with identical handling of control dependences and a breadth-first strategy for inspecting statements, reflecting real-world use of a program understanding tool. The experiments show that exploring a traditional slice required inspecting 1-4.52X as many statements as thin slicing for the debugging tasks, and 1.17-34.2X as many statements for the program understanding tasks.
- **Thin slices can be computed efficiently.** Our context-insensitive thin slicing algorithm scales well to large programs: the cost of computing thin slices is insignificant compared to pre-requisite call graph construction and pointer analysis. The precision of this thin slicer sufficed for the program understanding tasks considered, and depended primarily on the precision of the preliminary pointer analysis. In contrast, we were unable to scale a context-sensitive traditional slicer [13] to our larger benchmarks.

Program	Methods	Bytecode Size (KB)	Call Graph Nodes	SDG Statements
Software-Artifact Infrastructure Repository				
`nanoxml`	541	35	817	22205
`jtopas`	337	24	397	23766
`ant`	11147	632	20164	584155
`xmlsec`	11192	678	17075	525886
SPECjvm98				
`mtrt`	470	32	514	19699
`jess`	1061	67	1466	46037
`javac`	1610	118	2127	71041
`jack`	592	55	1088	38114

Table 1. Benchmark Characteristics, derived from methods discovered by the on-the-fly call graph construction, including Java library methods. The number of call graph nodes exceeds the number of distinct methods due to limited cloning-based context-sensitivity in the points-to analysis. SDG Statements reports the number of scalar statements, but excludes parameter passing statements introduced to model the heap.

5.1 Configuration and Methodology

We implemented the thin and traditional data slicers using the T.J. Watson Libraries for Analysis (WALA) [3]. We utilized the call graph construction and pointer analysis algorithms provided by WALA, along with its tabulation solver for context-sensitive analysis [25]. We analyzed the Sun JDK 1.4.2_09 standard library code, for which WALA provides models of important native methods. WALA uses heuristics to analyze the most common uses of reflection in Java, but in general reflection and native methods may still cause some unsoundness, as is typical in Java static analysis implementations. All experiments were performed on a Lenovo ThinkPad t60p with dual 2.2GHz Intel T2600 processors and 2GB RAM. The analyzer ran on the Sun JDK 1.5_07 using at most 1GB of heap space.

Table 1 provides information about the programs used in our experiments. The table reports characteristics based on the program as discovered by an on-the-fly call graph construction, including library methods. We used a pointer analysis based on Andersen's analysis with on-the-fly call graph construction [4, 27], but with fully object-sensitive cloning for objects of key collections classes, as described in [9]. The precise handling of containers was key for

precision for all slicing variants, as the experiments often required tracing flow through containers. We excluded from the call graphs a few large standard libraries (*e.g.*,`javax.swing`, `java.awt`) which we deem *a priori* uninteresting for the tasks at hand; we believe a practical tool would do likewise. For all experiments reported, call graph construction and pointer analysis ran in under 5 minutes.

Scalability For the dependence graph traversal, we considered both the context-insensitive (flat graph reachability) and context-sensitive (tabulation) algorithms presented in Section 3.

In all cases, the time and space to compute the thin slice or traditional slice with the context-insensitive algorithm was insignificant compared to the preliminary pointer analysis. Context-insensitive thin slicing took under 6 seconds for all tests except `ant`, which took 47 seconds since a large number of interprocedural heap dependence edges had to be added. These low running times are not surprising, as context-insensitive slicing (thin or traditional) reduces to simple graph reachability on a demand-driven construction of the SDG program representation.

The tabulation-based algorithm scales to handle most experiments on the smaller test cases (`nanoxml`, `jtopas`, `mtrt`, `jack`). For the larger codes, we could not run in reasonable time and/or space. We believe our tabulation implementation is fairly well-tuned, having evolved over several years and been used in several studies reporting scalable interprocedural dataflow analyses(*e.g.*, [9]). For slicing, the key bottleneck comes from handling of the heap; the SDG statements introduced to model heap parameter-passing quickly explode, which blows up space of the SDG and time in tabulation. For the larger benchmarks, the full SDG quickly blows up to over 10 million nodes, and we suspect would grow much larger given adequate space.

In all results reported, we compare results from the context-insensitive thin slicer to a context-insensitive traditional slicer, which scaled to all benchmarks. This provides an apples-to-apples comparison, as all experimental parameters match exactly the two algorithms; the only difference was how each handled data dependences.

Measuring Slice Size Nearly all existing work measures the precision of a slice by its full size. However, in practice, once a user of a program understanding tool has discovered all the statements relevant to her original problem, she will not inspect the rest of the slice. Our experiments aim to simulate this realistic usage pattern.

For each task, we identify both a seed statement for the slice and a set of *desired statements*, *i.e.*, those statements whose discovery suffices for completing the task. For example, for a debugging task, the seed is the point of failure, and the desired statement is the cause of the bug. We then aim to measure how many statements in the slice the user must inspect to discover the desired statements.

We use a breadth-first traversal strategy to simulate the order in which statements are inspected by the user. Intuitively, statements "closer" to the seed seem more likely to be relevant to its behavior. Hence, we assume the user gradually explores statements of increasing distance (defined by the dependence graph of the slicer) from the seed until the desired statements are found; a breadth-first search of the dependence graph simulates this strategy. Note that a realistic slicing tool allows the dependence graph to be browsed in this manner [1], and that this technique has been used or proposed in other previous work [7, 33]. For thin and traditional slicing, we report the number of statements inspected using this breadth-first inspection strategy in our tables.

To our knowledge, ours is the first work to compare static slicing algorithms using a measure intended to simulate a realistic tool, rather than just comparing the full slice sizes. We note that the different measures can produce qualitatively different results. For example, in of our smaller test cases (`nanoxml-1`), context sensitivity reduces the traditional slice size from 8067 statements to 381 statements, but the number of statements explored in the traversal decreases only from 32 to 26. For the test cases we could analyze context sensitively, we noticed no reduction in traversal size for thin slices due to context sensitivity, though the full thin slice size was reduced. Given this result, the context-sensitive algorithm of Section 3.2 does not seem beneficial for thin slicing as used in practice.

Control Dependence As discussed in Section 4.2, relevant control dependences were always lexically close to statements in the thin slice for our experiments, like the example of Figure 3. Furthermore, the control dependences for most statements are not relevant, and it is not obvious how to automatically expose important control dependences. Hence, we manually pre-determined the relevant control dependences and fed that information to both slicers, simulating the actions of a user. This allows us to focus on the effectiveness of thin slicing's handling of data dependences vs. a traditional slicer.

5.2 Experiment: Locating Bugs

Our first experiment tested locating several bugs, (1) to see if thin slices include the buggy statement when slicing from the seed, and (2) to compare the number of inspected statements for thin and traditional slices. We investigated several injected bugs in the Java programs in the Software-Artifact Infrastructure Repository (SIR) [8]. SIR provides both several injected bugs for each program and test suites that can be used to expose the bugs. For each injected bug, we ran the corresponding test suite to discover a failure. Then, we ran both thin and traditional slicing from the failure point, measuring how many statements had to be inspected to find the bug (as described in Section 5.1).

Three points should be noted about the SIR programs and injected bugs. First, we were unable to include two SIR programs in these experiments, `jmeter` and `siena`. We could not determine the appropriate library dependences to build `jmeter`, and in our runs, no test cases exposed the injected bugs in `siena`. We plan to rectify this situation for the final version of this paper. Also, the suite provides several versions of each benchmark; we chose bugs from the most recent versions. Finally, some of the injected bugs represent bugs of omission, *i.e.*, bugs that deleted necessary code. If the omission bug removed an assignment to a local or a conditional branch, we chose as the desired target statements the immediate data or control dependent successor statements, respectively. We excluded bugs that deleted field writes, as there was no obvious relationship between the deleted write and the surrounding code in the method.

Table 2 presents results for our debugging experiment. Several of the injected buggy statements were quite close to the failure points of the programs, and hence both the traditional and thin slicers found the bugs very quickly. For example, with `jtopas-1`, the injected buggy statement itself fails with a `NullPointerException`. These sorts of bugs can be easily debugged without tool support, but we include them for completeness.

For bugs requiring inspection of more than 4 statements, using the traditional slicer requires inspecting 1.62-4.52X more statements than thin slicing to find the bug. The injected bugs in `nanoxml` in particular often require tracing a value as it is inserted and later retrieved from one or two `Vectors`, as in the example of Figure 1. Tracing this flow by hand can be difficult and time-consuming, and hence we think that thin slicing can have the greatest impact for this type of bug.

Debugging `nanoxml-5` required exposing statements causing aliasing (see Section 4.1), for reasons similar to those of the example in Figure 3. To simulate this user action, we ran the thin slicer in a configuration that included statements explaining one level of indirect aliasing. The results show that exposing such statements in this controlled manner is useful, as we still inspect significantly fewer statements than the traditional slice.

Bug	# Thin	# Trad.	Ratio	# Control
nanoxml-1	12	32	2.67	0
nanoxml-2	25	113	4.52	0
nanoxml-3	29	123	4.24	0
nanoxml-4	12	33	2.75	1
nanoxml-5	35	156	4.46	1
nanoxml-6	12	52	4.33	0
jtopas-1	1	1	1	0
jtopas-2	2	2	1	1
ant-1	2	2	1	1
ant-2	4	5	1.25	0
ant-3	34	55	1.62	15
ant-4	3	3	1	2
xml-security-1	2	2	1	1

Table 2. Evaluation of thin slicing for debugging. For each bug, we show the number of statements that must be explored in the thin slice and the traditional slice to discover the bug, using BFS traversal (see Section 5.1). We also give the ratio of traditional statements to thin slice statements, and the number of control dependences that must be exposed to find the bug; the numbers for thin and traditional slices include these control dependences. Slicing of any kind was not useful for five bugs in `xml-security` and one bug in `ant`; these bugs do not appear in the table.

Few control dependences are relevant for these debugging tests, validating our decision to ignore control dependence in thin slices. For all but one bug, the number of control dependences that need to be followed is 2 or less. These control dependences were always obvious from code surrounding the thin slice. The high number of control dependences for `ant-3` is due to the fact that the buggy function has 12 return statements, and one of them is directly control dependent on the bug; we included one control dependence for each return, as it is not obvious which one caused the bug. Nevertheless, all the control dependences are still near statements in the thin slice.

Finally, for five bugs in `xml-security` and one bug in `ant`, no type of slicing could help the user find the bug, and hence they do not appear in the table. The `xml-security` bugs all followed the same pattern:

```
long hash = computeHash(input); // buggy
assert hash == expectedHash; // fails
```

In `xml-security`, the `computeHash()` equivalent is complex, spanning several .class files, and the injected bugs were buried in the algorithm internals. In such cases, slicing from this assertion failure (whether static or dynamic) will inevitably bring in most or all of the code that computes the hash function. This example illustrates that slicing of course is not a panacea; delta debugging [31] or refactoring to test at a finer granularity may help in these situations. We find the fact that thin slicing was useful for 13 out of 19 inspected bugs encouraging.

In summary, we found that for these injected bugs, thin slices very often contain the buggy statements, and the bugs could be found more quickly with a thin slicer than a traditional slicer. Also, 11.5 statements on average required inspection with the thin slicer (ranging from 1 to 35), quite a manageable number; the average for the traditional slicer was significantly larger at 54.8 statements, ranging from 1 to 156.

5.3 Experiment: Understanding Tough Casts

Our second experiment involved using slicing to hand-validate the safety of *tough casts* in the SPECjvm98 benchmarks. A tough cast is a downcast in a program that cannot be verified by precise and scalable pointer analysis [17]; we used the same pointer analysis used to construct our call graph. Checking these tough casts with thin slicing represents a useful exercise for understanding a code base, and can be useful for refactoring or adding parameterized types or annotations.

```
1  class Node {
2    final int op;
3    static int ADD_NODE_OP = 1;
4    Node(int op) { this.op = op; }
5  }
6  class AddNode extends Node {
7    AddNode(...) {
8      super(ADD_NODE_OP); ...
9    }
10 }
11 simplify(Node n) {
12   int op = n.op;
13   switch (op) {
14   case ADD_NODE_OP:
15     AddNode add = (AddNode) n;
16   ...
17   }
18 }
```

Figure 4. An example illustrating a tough cast. Expressions in the thin slice used to understand the safety of the cast are underlined.

The cast at line 15 in Figure 4, adapted from the `javac` benchmark, is a tough cast. This cast cannot fail because the value of the `op` field of `AddNode` objects is `ADD_NODE_OP`, as guaranteed by line 8, and no other subclasses of `Node` (not shown) have `ADD_NODE_OP` in their `op` field. We can discover this fact through thin slicing by following a control dependence from the cast, and then computing a thin slice for line 12 to see what value `op` gets for different subclasses of `Node`. Typically, tough casts are those that (1) are not used to cast values retrieved from a container (due to lack of generics) and (2) are not dominated by an explicit `instanceof` check ensuring their safety.

Our experimental configuration involved first manually identifying those statements that showed each tough cast could not fail (the desired statements of Section 5.1) with the help of the thin slicer, and then comparing the BFS traversal sizes of the thin and traditional slices from the cast to these desired statements. For each SPEC benchmark, we investigated 10 tough casts at random, or all tough casts if there were fewer than 10. Note that The `compress` and `db` benchmarks had no tough casts, and `mpegaudio` was excluded since its bytecodes are obfuscated, making understanding its casts difficult. Also, we failed to determine the reason for cast safety for 6 casts in `javac` and one cast in `jess`. In these cases, the safety of the cast relies on some subtle invariant that is not easy to determine for one unfamiliar with the code.

The thin slicer significantly eased the manual process of determining the desired statements for each tough cast. Although the code was unfamiliar to us, our thin slicing tool guided us through heap-based value flow, saving a great deal of time. The thin slicer was especially helpful when source code was *not* available, *e.g.*, for the `jack` benchmark, as we had to study a compiler representation of the bytecodes and could not use standard IDE-based source navigation tools.

Results for the tough casts experiment appear Table 3. Thin slicing helps understand tough casts more effectively than traditional slicing: the number of statements examined using a traditional slice exceeds the number using a thin slice by factors of 1.17-34.2X. In `javac`, the casts resemble Figure 4. The code includes a large num-

Cast	# Thin	# Trad.	Ratio	# Control
mtrt-1	22	51	2.32	0
mtrt-2	23	52	2.26	0
jess-1	6	7	1.17	2
jess-2	13	39	3	0
jess-3	6	6	1	2
jess-4	6	7	1.17	2
jess-5	6	7	1.17	2
jess-6	6	6	1	2
javac-1	57	910	16	1
javac-2	43	853	19.8	1
javac-3	65	2224	34.2	1
javac-4	45	855	19	1
jack-1	18	79	4.39	0
jack-2	57	151	2.65	0
jack-3	18	69	3.83	0
jack-4	18	79	4.39	0
jack-5	57	151	2.65	0
jack-6	35	132	3.77	0
jack-7	35	132	3.77	0
jack-8	35	132	3.77	0
jack-9	30	79	2.63	0
jack-10	57	151	2.65	0

Table 3. Evaluation of thin slicing for understanding tough casts. For each tough cast, we provide the number of statements that must be explored in the thin slice and the traditional slice to discover the reason the cast cannot fail, using BFS traversal (see Section 5.1). We also give a ratio of traditional slice to thin and the number of relevant control dependences.

ber of `Node` subclasses used pervasively in the program, resulting in large numbers for the traditional slicer.

The absolute numbers of inspected statements exceed those for the debugging tests, but they remain manageable for a user. The thin slicer required inspecting an average of 29.3 statements (ranging from 6-65), while the traditional slicer required an average of 280 (ranging from 6 to 2224). For `javac`, many of the thin slice statements are writes of opcodes in a large number of constructors (like in Figure 4), which could be quickly inspected to ensure that a suitable constant is written. For `jack`, the BFS traversal over-estimates the number of thin slice statements that need to be inspected; as once we understood the benchmark, we could terminate the search early at some statements which we knew would not cause the cast to fail.

In summary, we conclude thin slicing can effectively provide tool support to identify statements that ensure tough casts cannot fail. A traversal based on thin slicing typically touches significantly fewer statements than a traversal based on traditional transitive flow dependence.

6. Related Work

Since first being defined by Weiser in 1979 [30], slicing has inspired a large body of work on computing slices and on applications to a variety of software engineering tasks. We refer the reader to Tip's survey [29] and Krinke's thesis [14] for broad overviews of slicing technology and challenges. Here, we focus on the work most relevant to our own.

Our thin slicing algorithm is a straightforward adaptation of the SDG-based approach first presented by Horwitz et al. [12]. Our implementation of a traditional slicer is in fact tabulation-based, as suggested in [24] and the 20-year Retrospective to [12].

CodeSurfer [1] is a program understanding tool for C and C++ based on the analysis techniques of [13, 26]. CodeSurfer uses pointer analysis to allow navigation from a use of a variable to potential defs through a pointer. However, it is not clear whether the same can be done for uses and defs of heap locations (*e.g.*, asking where `*x.f` is defined), the critical problem for Java (since in Java, variables cannot be defined through pointers). Our evaluation metric of a breadth-first traversal strategy aims to simulate use of a tool like CodeSurfer, which allows for navigating the dependence graph.

Atkinson and Griswold [5] present a slicer relying a preliminary flow-insensitive pointer analysis. This work targets C, and so had to deal with difficulties arising from issues such as stack-directed pointers and unsafe memory access, which do not arise in Java. Larsen and Harrold [15] presented one of the first slicing approaches for object-oriented software, adding pseudo-parameters for fields to track dependencies through the heap. Our context- sensitive slicer implementation uses a similar approach, but relies on a partially context-sensitive preliminary pointer analysis to disambiguate locations with field- and object-sensitivity, and additional pseudo-parameters to soundly handle all fields that may be accessed transitively by callees.

In recent years, several papers have improved precision by integrating more precise static alias analysis into slicing. Liang and Harrold [18] present a novel approach to represent formal parameter objects with trees. Hammer and Snelting [10] present an enhancement to this approach, including a criterion for sound limiting of tree sizes for recursive data structures. Both these approaches are more powerful than relying solely on a preceding flow-insensitive alias analysis, since must-alias information on parameters can allow sound strong updates. It is not clear how far these algorithms scale; the experimental results of Hammer and Snelting address programs significantly smaller than the benchmarks considered here. In future work, we plan to incorporate aspects of Hammer and Snelting's approach for thin slices.

Orso et al. [23] present a classification of data dependence edges in an SDG, based on certainty of may-alias information, and the span (scope) of a program over which a data dependence flows. They propose an incremental slicing procedure to aid debugging, whereby a tool can provide progressively larger slices by including progressively more classes of data dependencies. Our thin slice expansion technique is similar in spirit, but goes in a different direction by expanding slices to include statements that indirectly give rise to the primary alias relations.

Mock et al. [22] showed that for C programs with heavy pointer use, using dynamic points-to data significantly improved slice precision over a conservative flow-insensitive pointer analysis. We suspect Java programs resemble C programs with heavy pointer use with regard to data dependence.

PSE [21] is a static analysis tool for localizing the causes of typestate errors in C and C++ programs by essentially computing a variant of a backward slice, with extra filtering based on the type of error. Their system is able to perform strong updates on the heap in some situations, using a technique we plan to try in our thin slicer. Their work unsoundly ignores may-aliasing in some configurations, partly due to the fact that traces involving aliasing are hard for developers to understand. If applied to a Java-like language, our technique for explaining aliases in thin slices may help solve this problem, as discussed in Section 4.1.

Recently, Zhang et al. have considerably improved the state-of-the-art in dynamic slicing [32, 33, 34, 35]. Thin slicing applies naturally to dynamic data dependences, and we believe dynamic thin slices could provide benefits similar to static thin slices. Zhang et al.'s work on improving scalability [34, 35] could be leveraged to create a more scalable dynamic thin slicer. Their recent work on

pruning dynamic slices [32] is complementary to ours: thin slicing and their heuristics for determining when a statement is unlikely to be relevant (based on which statements output good and bad values) could be fruitfully combined. Recent work [33] observes that using dynamic data dependences alone can often identify buggy statements in C programs; we suspect that in fact those data dependences considered by a thin slicer would also be sufficient. Finally, this work [33] also suggests exploring statements closer to the seed first when viewing a slice, an idea we also use in our evaluation.

7. Conclusions

We have described thin slicing, a novel approach to finding relevant statements for some seed computation. The key difference between thin slicing and traditional slicing is the use of the narrower notion of value-flow relevance, which only includes statements copy-propagating values to a seed rather than all statements that can possibly affect it. Our evaluation shows that relevant statements could be found with thin slices by inspecting up to 4.52X fewer statements than traditional slices for debugging tasks and 34.2X fewer statements for program understanding tasks. Furthermore, we show that a context-insensitive thin slicer, based on precise pointer analysis and call graph construction, can scale to large benchmarks, while our implementation of traditional context-sensitive slicing could not. Thin slicing provides the basis for a practical and effective program understanding tool, as it provides significant help to developers for finding relevant statements and scales to large, realistic Java programs.

References

[1] Codesurfer. http://www.grammatech.com/products/codesurfer/.

[2] The Eclipse Platform. http://www.eclipse.org/.

[3] T.J. Watson Libraries for Analysis. http://wala.sourceforge.net.

[4] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, DIKU, 1994.

[5] D. C. Atkinson and W. G. Griswold. Effective whole-program analysis in the presence of pointers. In *Foundations of Software Engineering*, pages 46–55, 1998.

[6] M. Das, S. Lerner, and M. Seigle. Esp: path-sensitive program verification in polynomial time. In *Conference on Programming Language Design and Implementation (PLDI)*, 2002.

[7] A. Diwan. Personal communication. 2006.

[8] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4), October 2005.

[9] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. In *International symposium on Software testing and analysis (ISSTA)*, 2006.

[10] C. Hammer and G. Snelting. An improved slicer for Java. In *Proceedings of the ACM-SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 17–22, 2004.

[11] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *Conference on Programming Language Design and Implementation (PLDI)*, 1989.

[12] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Conference on Programming Language Design and Implementation (PLDI)*, 1988.

[13] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.

[14] J. Krinke. *Advanced Slicing of Sequential and Concurrent Programs*. PhD thesis, University of Passau, 2003.

[15] L. Larsen and M. J. Harrold. Slicing object-oriented software. In *International Conference on Software Engineering (ICSE)*.

[16] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *International Conference on Compiler Construction (CC), Warsaw, Poland*, April 2003.

[17] O. Lhoták and L. Hendren. Context-sensitive points-to analysis: Is it worth it? In *International Conference on Compiler Construction (CC)*, 2006.

[18] D. Liang and M. J. Harrold. Slicing objects using system dependence graphs. In *ICSM*, pages 358–367, 1998.

[19] V. B. Livshits and M. S. Lam. Finding security errors in Java programs with static analysis. In *Proceedings of the 14th Usenix Security Symposium*, pages 271–286, Aug. 2005.

[20] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. In *Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[21] R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang. PSE: explaining program failures via postmortem static analysis. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 63–72, New York, NY, USA, 2004. ACM Press.

[22] M. Mock, D. C. Atkinson, C. Chambers, and S. J. Eggers. Improving program slicing with dynamic points-to data. *SIGSOFT Softw. Eng. Notes*, 27(6):71–80, 2002.

[23] A. Orso, S. Sinha, and M. J. Harrold. Classifying data dependences in the presence of pointers for program comprehension, testing, and debugging. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 13(2):199–239, 2004.

[24] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-12):701–726, November/December 1998.

[25] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1995.

[26] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. Speeding up slicing. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE), New Orleans, LA*, December 1994.

[27] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java using annotated constraints. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Tampa Bay, Florida*, October 2001.

[28] B. G. Ryder, W. A. Landi, P. A. Stocks, S. Zhang, and R. Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM Trans. Program. Lang. Syst.*, 23(2):105–186, 2001.

[29] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[30] M. D. Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, 1979.

[31] A. Zeller. Isolating cause-effect chains from computer programs. *SIGSOFT Softw. Eng. Notes*, 27(6):1–10, 2002.

[32] X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. In *Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[33] X. Zhang, N. Gupta, and R. Gupta. A study of effectiveness of dynamic slicing in locating real faults. *Empirical Software Engineering*, 2006. To appear.

[34] X. Zhang, R. Gupta, and Y. Zhang. Efficient forward computation of dynamic slices using reduced ordered binary decision diagrams. In *International Conference on Software Engineering (ICSE)*.

[35] X. Zhang, S. Tallam, and R. Gupta. Dynamic slicing long running programs through execution fast forwarding. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2006.