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 systematically identifies parts of a program relevant to a seed statement. Unfortunately, slices of modern programs often grow too large for human consumption. 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 point of interest, not all such statements appear equally relevant to a human.

As an improved method of finding relevant statements, we propose *thin slicing*. A thin slice consists only of *producer statements* for the seed, *i.e.*, those statements that help compute and copy a value to the seed. Statements that explain why producers affect the seed are excluded. For example, for a seed that reads a value from a container object, a thin slice includes statements that store the value into the container, but excludes statements that manipulate pointers to the container itself. Thin slices can also be hierarchically expanded to include statements explaining how producers affect the seed, yielding a traditional slice in the limit.

We evaluated thin slicing for a set of debugging and program understanding tasks. The evaluation showed that thin slices usually included the desired statements for the tasks (*e.g.*, the buggy statement for a debugging task). Furthermore, in simulated use of a slicing tool, thin slices revealed desired statements after inspecting 3.3 times fewer statements than traditional slicing for our debugging tasks and 9.4 times fewer statements for our program understanding tasks. Finally, our thin slicing algorithm scales well to relatively large Java benchmarks, suggesting that thin slicing represents an attractive option for practical tools.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Debugging aids

General Terms Languages, Reliability

Keywords slicing, debugging, program understanding

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?"* Malcolm Gladwell, *Blink: The Power of Thinking Without Thinking*

PLDI'07 June 11–13, 2007, San Diego, California, USA.

Motivation Large-scale object-oriented programs can be very hard to understand and debug. Pervasive use of heap-allocated data and complex data structures in these programs causes much of this difficulty. Multiple levels of pointer indirection in common data structures 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.

Program slicing is a well-known technique for identifying a subset of the program that is relevant to a statement or value of interest, called a *seed*[1]. Slicing applies to a variety of program understanding tasks, ranging from testing and debugging to reverse engineering [27]. Weiser [28] 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. This definition pollutes slices with many statements that indirectly affect a seed but are not pertinent to typical program understanding tasks.

Data structures are a key source of slice pollution. Slices often include internal implementation details of these data structures, which are almost always irrelevant to programmer 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 the statements that construct and manipulate all levels of this complex data structure. For many program understanding tasks, the programmer needs information about the values stored in the list, but doesn't care about other details of nested data structures containing the value. Furthermore, modern programs typically rely heavily on well-tested data structures provided by standard libraries, whose internal details rarely concern the end-user programmer. For these common cases, the backwards slice presents far too much information for the task at hand.

Our Approach This paper presents *thin slicing*, a program understanding technique that redefines relevance in a manner aimed at only including statements useful for developer tasks. For thin slicing, only *producer statements* for the seed are relevant, *i.e.*, those statements that help compute and copy a value to the seed. Statements that explain *why* producers affect the seed are excluded from a thin slice. In practice, producer statements alone are sufficient for many debugging and program understanding tasks.

We demonstrate the relevance notion of thin slicing 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()`

[1] The seed is often termed the *slicing criterion* in the literature [27]; we use 'seed' for brevity.

```
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();
30   s.setNames(firstNames);
31   ...;
32   SessionState t = getState();
33   printNames(t.getNames());
34 }
```

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 for line 24 contains all displayed code. The bodies of functions with inessential behavior (*e.g.*, `print()`) have been elided for clarity.

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".

Traditional slicing does not help in diagnosing this bug, as a slice seeded with line 24 includes all the code in the example. The slice must include all the code to construct and populate the `Vector` passed to `printNames()` and the code in `main()` retrieving the `Vector` from the `SessionState` object, which all affects 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.

A thin slice only includes *producer statements* for the seed. We say statement s is a producer for statement t if s is part of a chain of assignments that computes and copies a value to t. In Figure 1, the producer statements for the seed, highlighted with underlining, are almost exactly the statements most relevant to the bug in question. 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 an executable program; for example, statements initializing the `Vector` containing the erroneous `String` are not included. However, the thin slice more directly leads the user to the bug.

Advantages of Thin Slicing One reason that thin slicing works well is that it ignores value flow into base pointers of heap accesses, focusing just on the value read from or written to the heap. 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 includes statements influencing both the base pointers `this` and `this.elems` (dereferenced to access the array contents), contributing 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.

Thin slices have an intuitive semantic definition, making them easier to understand than some ad-hoc pruning of the traditional slice. Simply stated, a thin slice contains all statements flowing values to the seed, ignoring uses of base pointers. These well-defined semantics allow a user to reason about thin slices in a self-contained manner, since she knows that all producer statements are included in a thin slice. 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.

In cases where a thin slice alone is insufficient for some programmer task, it can be expanded with additional thin slices to ease the understanding of other relevant statements. One case in which statements outside the thin slice may be needed is to explain heap-based value flow, established through pointer aliasing. 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 heap-based 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 of the thin slice can be repeated recursively for further aliasing questions, yielding a structured method for studying the often large set of statements relevant to nested data structures.

A similar expansion technique applies to explaining why thin slice statements can execute, *i.e.*, showing their transitive *control dependences*. Traditional slicers must include all transitive control dependences. Unfortunately, Java's semantics make many statements a type of conditional branch, often yielding a huge number of control dependences. For example, if a statement might throw an exception, many statements will be control dependent on its successful execution. Similarly, each virtual call `x.m()` is a conditional expression because it branches on the runtime type of `x`.

In practice, we found that when control dependences are relevant, they can usually be identified from browsing the source code since they appear lexically near a thin slice statement, making their discovery straightforward. Further thin slices can be taken to understand these important conditionals. In the limit, hierarchically expanding a thin slice to show control and aliasing explanations yields a traditional slice; hence, any possibly relevant statement can eventually be discovered.

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, for most tasks we tested, thin slices with little or no expansion included the desired statements for the task with many fewer extraneous statements than traditional slices.

The current paper focuses on static thin slicing for Java, but the technique is more broadly applicable. Thin slicing itself relies on standard data dependence concepts [10] and hence 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). Also note that dynamic thin slices can be defined in a straightforward manner using dynamic data dependences.

Contributions This paper makes the following contributions:

- We define a thin slice as *producer statements* for a given seed.
- We present a method for hierarchically expanding thin slices to explain why producer statements affect the seed, in the limit yielding a traditional slice.
- We present simple modifications to existing slicing algorithms for computing both context-insensitive and context-sensitive thin slices.
- We present experiments comparing 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 6). Our results show that (1) thin slices usually included the desired statements for the tasks (*e.g.*, the buggy statement for a debugging task), and (2) thin slices revealed desired statements after inspecting 3.3 times fewer statements than traditional slicing for debugging tasks and 9.4 times fewer statements for program understanding tasks. We also showed that our thin slicing algorithm scales to relatively large Java benchmarks.

The rest of this paper is organized as follows. Section 2 defines producer statements and the thin slicing process, and Section 3 defines thin slices using traditional dependences. Section 4 describes our technique for expanding thin slices to explain heap-based value flow and control dependences. Section 5 presents algorithms for computing thin slices as variants of a traditional slicing algorithm. Section 6 gives our experimental evaluation, Section 7 discusses related work, and Section 8 concludes.

2. Defining Thin Slices

In this section, we define the producer statements included in a thin slice. We also show how statements excluded from the thin slice explain why the producer statements affect the seed. A simple example, seen in Figure 2, is used to illustrate these concepts. Section 3 defines the statements in a thin slice using traditional notions of dependence.

Slicing determines the parts of a program "relevant" to some *seed* statement. In traditional slicing, relevance is defined as any statement possibly affecting the values computed by the seed. As originally stated by Weiser [28], this relevance definition requires the slice to include an *executable subset* of the program in which the seed always performs the same computation as in the original program. Thin slicing differs from classical slicing primarily in its more selective notion of relevance.

```
1  x = new A();
2  z = x;
3  y = new B();
4  w = x;
5  w.f = y;
6  if (w == z) {
7    v = z.f;  // the seed
8  }
```

Figure 2. A small program to illustrate thin slicing. Directly-used locations (see Section 2) in the thin slice for line 7 are underlined.

With thin slicing, only *producer statements* for the seed are relevant. We define producer statements in terms of *direct uses* of memory locations (variables or object fields in Java). A statement s directly uses a location l iff s uses l for some computation other than a pointer dereference. For example, the statement `y = x.f` does not directly use `x`, but it does directly use $o.f$, where `x` points to o. A statement t is a *producer* for a seed s iff (1) $s = t$ or (2) t writes a value to a location directly used by some other producer.

Consider computing a thin slice for line 7 in the toy example of Figure 2. Line 7 directly uses an object field written at line 5 (since `w` and `z` are aliased), and therefore, line 5 is a producer. Similarly, line 5 directly uses `y`, which is written at line 3, making line 3 a producer as well. Hence, lines 5 and 3 are comprise the thin slice for line 7 (along with line 7 itself). In contrast, the traditional slice for line 7 is the entire example.

We call the non-producer statements in the traditional slice *explainer statements*. These statements show *why* the producer statements can affect the seed. Explainer statements can show one of two things about the producers:

Heap-based value flow When values flow between producers through heap locations, the locations are accessed using aliased pointers. Explainer statements show how these base pointers may become aliased.

Control flow The remaining explainer statements show the conditions (*i.e.*, the expressions in conditional branches) under which producer statements actually execute.

Consider again the example of Figure 2. Lines 2 and 4 show that `w` and `z` both point to the `A` object allocated at line 1. Hence, these lines are explainers for the heap-based value flow between lines 5 and 7 in the thin slice. Line 6 explains control flow, showing the condition under which the seed statement actually executes.

Thin slicing's separation of producer and explainer statements provides a natural, structured method for exploring a traditional slice. Traditional slices must include *transitive* explainer statements (*i.e.*, explainers for the explainers and so on), since any statement possibly affecting the seed is relevant for such a slice. While this transitivity can lead to an overwhelming number of explainer statements, thin slices structure them into a manageable hierarchy. Explainers for heap-based value flow in a thin slice can be shown using two additional thin slices, as shown in Section 4.1. The behavior of a conditional guarding a thin slice statement can also be understood through an additional thin slice. In this manner, more and more thin slices can be used to show explainers, in the limit yielding the traditional slice.

In practice, we have found that very few explainers are needed to accomplish typical debugging and understanding tasks. In our evaluation, over half the tasks could be completed with a thin slice alone. In most other cases, only one or two explainer statements

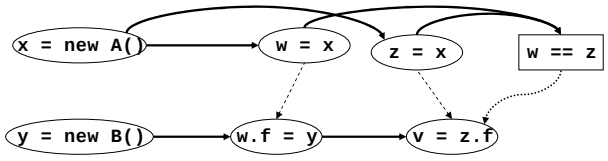

Figure 3. A dependence graph for the program of Figure 2. Thick edges indicate non-base-pointer flow dependences, used for thin slicing. Traditional slicing also uses base pointer flow dependences (the dashed edges) and control dependences (the dotted edge).

were required, and these explainers were lexically close to thin slice statements (further discussed in Section 4.2). Hence, thin slicing is highly effective at identifying the statements in a traditional slice most relevant to developer tasks.

3. Thin Slices as Dependences

In Section 2, we defined thin slices in terms of producer statements. Here we define thin slices in terms of the dependences typically used to define traditional slices. The thin slice for a seed s is a subset of those statements upon which s is transitively *flow dependent* (also known as *data dependent*), obtained by ignoring uses of base pointers in dereferences.

A statement s is flow dependent on statement t if the following three conditions hold [10]:

1. s can read from some storage location l.
2. t can write to l.
3. There exists a control-flow path from t to s on which l is not re-defined.

For Java-like languages, storage locations are either variables (local or global) or object fields, with the latter accessed through some field dereference of the form `x.f`. Traditional slices must include the transitive flow dependences of the seed.

Thin slices ignore *base pointer flow dependences*, thereby excluding statements explaining heap-based value flow. A base pointer flow dependence is a flow dependence due solely to the use of a pointer in a field dereference. For the statement `y = x.f`, flow dependences due to the use of `x` are base pointer flow dependences. Similarly, a statement of the form `p.f = q` has base pointer flow dependences due to the use of `p`. Ignoring base pointer flow dependences leaves only *producer flow dependences*, which transitively connect a statement to its producers. For example, `y = x.f` would have a producer flow dependence to some statement `z.f = w`, where `x` and `z` may be aliased.

Figure 3 shows an example dependence graph for the program of Figure 2. Nodes represent statements, and edges represent dependences between statements. As is standard for dependence graphs [11, 22], edges are drawn in the direction opposite of the dependences, so thin slicing requires computing backwards reachability. In Figure 3, the solid edges indicate the producer flow dependences, while the dashed edges indicate ignored base pointer flow dependences. The dotted edge is a control dependence, to be discussed shortly. The seed `v = z.f` is only reachable from `w.f = y` and `y = new B()` via solid edges, and these statements are the producers for the seed, as expected.

Note that thin slices also exclude control dependences, explainers of control flow. Intuitively, statement s is control dependent on conditional e if e can affect how many times s executes (Tip's survey [27] has a more formal definition). Figure 3 has a dotted control dependence edge from conditional `w == z` to `v = z.f`, the statement in its `if` block in Figure 2. Section 4 describes our empirical observation that important control dependences are nearly always lexically close to thin slice statements, and hence can be discovered easily.

```
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 4. 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 expressions in explainer statements for aliasing.

4. Expanding Thin Slices

Here, we discuss in more detail how thin slices can be expanded to show explainer statements, as discussed in Section 2. To review, explainer statements can answer questions of the following form about a thin slice T:

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

A thin slicing tool answers these questions when requested by the user. Section 4.1 discusses a technique for explaining aliasing using two additional thin slices. Section 4.2 discusses how relevant control dependences are usually "close" to thin slice statements, making their discovery relatively straightforward.

Example We use the example in Figure 4, a simple program fragment manipulating a file, to illustrate thin slice expansion. The example displays only a small part of the `File` 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.

4.1 Question 1: Explaining Aliasing

When a thin slice includes statements that copy a value through the heap, sometimes the user needs to understand why those statements access the same heap location. For the example of Figure 4, 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 produce the boolean open value. Clearly, these statements fail to diagnose the bug completely: the user still does not know which `File` is passed to `close()` before being passed to `isOpen()`. To diagnose this bug, the user must determine which statements cause the '`this`' pointers of `close()` and `isOpen()` to be aliased.

We can expand thin slices to explain aliasing by computing additional thin slices for the base pointers in question. Given aliased base pointers `x` and `y`, we compute thin slices seeded with the statements defining `x` and `y` (unique assuming SSA form). These thin slices will show why some common object o can flow to both `x` and `y`, causing them to be aliased. For Figure 4, the common object for the '`this`' parameters of `close()` and `isOpen()` is the `File` allocated at line 15. Double underlines in Figure 4 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 aliasing using additional thin slices yields an intuitive hierarchical 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 4 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 structured 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.

Array accesses can require explainer statements beyond those showing the aliasing of the array pointers. Say that we have statements `a = b[i]` and `c[j] = d` in the thin slice, such that there is value flow from `d` to `a`. In trying to understand this heap-based flow, the user may wonder both (1) how `b` and `c` can be aliased (the same question as with field accesses), and additionally (2) how the array indices `i` and `j` can have the same value. The latter question can be answered through thin slices on each of the array index expressions (with any necessary expansion).

Two additional technical points about explaining aliasing merit mention. First, the thin slices explaining aliasing should be restricted to only show the flow of objects that can flow to both base pointers, filtering statements showing flow of an object to just one of them. This filtering eliminates some statements irrelevant to explaining the aliasing. Second, context sensitivity may be necessary to focus the aliasing explanations in some cases. For example, if the code of Figure 4 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.

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, 8], *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 [15].

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 4, the bug manifests at line 11, which throws the exception. As 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 4 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. We also found that the vast majority of control dependences are unnecessary 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; the user can take additional thin slices from relevant conditionals to understand their behavior. Additional tool support may be useful for indicating non-obvious control dependences, *e.g.*, due to exceptions.

5. Computing Thin Slices

Computing a thin slice entails computing a statement's transitive flow dependences, ignoring uses of base pointers (as discussed in Section 3). As in previous work on slicing [11, 20], we compute thin slices using variants of graph reachability. Here, we first describe the basics of constructing our graph representation, a subset of system dependence graphs [11] (Section 5.1). Then, we briefly present two simple algorithms to compute thin slices, one context insensitive (Section 5.2) and one context sensitive (Section 5.3).

5.1 Graph Construction

In both thin slice algorithms, we first compute a subset of the system dependence graph (SDG) program representation of Horwitz et al. [11]. Previous work [3, 13] has described how to compute SDGs for Java-like languages, and we mostly re-use those techniques (slight differences are discussed in Section 7). Our implementation handles the full Java Virtual Machine bytecode language, excluding concurrency. Our representation differs in that we (1) exclude control dependence edges and (2) handle heap-based flow dependences differently, depending on the thin slicing algorithm (details in Section 5.2 and Section 5.3).

SDG construction relies on a pre-computed points-to analysis. We use the points-to analysis to compute a call graph for the program, necessary for tracking interprocedural dependences. We also use the points-to analysis to determine which heap locations can be defined (used) by field writes (reads), in order to track heap-based value flow. Section 6 shows that precise points-to analysis is key for effective thin slicing of Java programs.

Our representation of data dependences for local variables and method parameters is straightforward. At a high level, we represent such dependences as follows:

1. For a statement `x = e`, where `x` is a local, we add edges to all statements using `x`, excluding uses in pointer dereferences of the form `x.f`. 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.

5.2 Context-Insensitive Thin Slicing

Our first algorithm computes traditional (context-insensitive) graph reachability on our SDG variant to compute thin slices. In this approach, we represent data dependences for heap access statements as follows:

- For a statement `x.f := e`, we add an edge to each statement with an expression `w.f` on its right-hand side, such that the pre-computed points-to analysis indicates `x` *may-alias* `w`.

Note that we add direct edges to statements *in other procedures*. In contrast, the traditional SDG only includes interprocedural edges for parameter passing and return values. The advantage of this approach is that we need not model heap accesses using additional parameters and return values, as is done with traditional slicing [11]. In practice, not using heap parameters dramatically increases scalability without significant loss in precision (discussed further in Section 5.3 and Section 6).

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 [21] due to lack of context sensitivity (Section 6 shows this issue is not significant in practice).

5.3 Context-Sensitive Thin Slicing

The context-sensitive thin slicing algorithm uses an SDG variant closer to that used in traditional slicing, created compositionally from program dependence graphs (PDGs) for each procedure. Intraprocedurally, this approach handles heap accesses as follows:

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

We handle interprocedural heap flow in the same way as the standard SDG, with heap reads and writes modeled as extra parameters and return values to each procedure [5, 11]. Our implementation introduces such parameters using the same heap partitions used by the preliminary pointer analysis. Discovering the appropriate set of parameters for each procedure requires an interprocedural mod-ref analysis [24], computed using the result of the pre-computed points-to analysis.

Having built the graph, we compute context-sensitive reachability as a partially balanced parentheses problem [20]. Our implementation relies on a backwards, demand-driven tabulation algorithm [21].

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 6 for details.

6. Evaluation

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

- **Thin slices lead the user to desired statements.** For the tasks we considered, thin slices often contain the desired statements (*e.g.*, the buggy statement for a debugging task). When statements explaining pointer aliasing or control flow were relevant, the statements were always lexically close to statements in the thin slice. Subjectively, we also found a thin slicer very useful for understanding one set of benchmarks.
- **Thin slices focus better on desired statements than traditional slices.** We compared context-insensitive thin slicing to context-insensitive traditional slicing (the context-sensitive configurations did not scale) with identical handling of control dependences and a breadth-first strategy for inspecting statements, simulating real-world use of a program understanding tool. The experiments showed that finding desired statements in a traditional slice required inspecting 3.3 times more statements than a thin slice for the debugging tasks, and 9.4 times more statements for the program understanding tasks.
- **A precise pointer analysis is key to effective thin slicing.** We used a pointer analysis with object-sensitive handling [16] of key collections classes for the thin slicer. With a less precise pointer analysis, up to 17.2X more statements required inspection in thin slices to find desired statements.
- **Thin slices can be computed efficiently.** Our context-insensitive thin slicing algorithm scaled well to large programs, with the cost of computing thin slices insignificant compared to the pre-requisite call graph construction and pointer analysis. We were unable to scale a context-sensitive traditional slicer [11] to our larger benchmarks.

6.1 Configuration and Methodology

We implemented the thin and traditional data slicers using the IBM T.J. Watson Libraries for Analysis (WALA) [2]. We utilized call graph construction and pointer analysis algorithms provided by WALA, along with its tabulation solver for context-sensitive analysis [21]. We analyzed our benchmarks with 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. For pointer analysis and call graph construction, we used a variant of Andersen's analysis with on-the-fly call graph construction [4, 23], with fully object-sensitive cloning [16] for objects of key collections classes, as described in [8] (the importance of this precision is discussed later in the section). We excluded from the call graphs a few large standard libraries (*e.g.*,`javax.swing`, `java.awt`) which we deemed *a priori* uninteresting for the tasks at hand, since none of our tested tasks involved those libraries. 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 5.

In all cases, the time and space to compute the thin slice or traditional slice with the context-insensitive algorithm was insignificant

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 during 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.

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.

Our implementation of context-sensitive traditional slicing [22] scales to handle most experiments on the smaller test cases (nanoxml, jtopas, mtrt, jack). For the larger codes, our implementation could not complete in reasonable time and/or space. We believe our implementation is fairly well-tuned, as the analysis engine (based on tabulation [20]) has evolved over several years and been used in several studies reporting scalable interprocedural dataflow analyses (*e.g.*, [8]). For slicing, the key bottleneck comes from handling of the heap; as programs grow larger, the number of SDG statements introduced to model heap parameter-passing quickly explodes, dramatically increasing space and time requirements. For our larger benchmarks, the full SDG grew to over 10 million nodes before exhausting available memory; we suspect the number of nodes would grow much larger given adequate space. Note that heap parameters are also a scalability bottleneck in a commercial slicing tool [26].

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 for 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 of the desired statements for her original problem (*e.g.*, those causing some bug), 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, as in the work of Renieris and Reiss [19]. 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 technique) from the seed until the desired statements are found; a breadth-first search of the dependence graph simulates this strategy. Note that CodeSurfer [1], perhaps the most widely-used slicing tool, supports such dependence-graph browsing for viewing slices. The BFS evaluation metric has also been used in other recent work [19, 31, 34]. For thin and traditional slicing, our tables report the number of statements inspected using this breadth-first inspection strategy.

To our knowledge, ours is the first work to compare static slicing algorithms using a measure intended to simulate the usage of a realistic tool, rather than just comparing the full slice sizes. We note that the two measures produce qualitatively different results. For example, in one 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. We observed similar results for thin slices. Given these results, the context-sensitive algorithm of Section 5.3 does not seem beneficial for thin slicing as likely used in practice.

Control Dependence As discussed in Section 4.2, relevant control dependences were observed to be always lexically close to statements in the thin slice, as in the example of Figure 4. Furthermore, most control dependences were not useful for the tested tasks, and it is not obvious how to automatically expose important control dependences. Hence, we manually pre-determined the important control dependences for our tasks, and counted only those control dependences as inspected for both the thin and traditional slicers. This handling of control dependences allowed us to focus on the effectiveness of thin slicing's handling of data dependences compared with a traditional slicer's.

Threats to Validity One threat to the validity of our results is that our study of debugging tasks (Section 6.2) uses injected bugs from the SIR suite [7], which may not accurately reflect the characteristics of real bugs. Several techniques were used to make the injected bugs in the SIR suite realistic, described in detail in [7]. The bugs were of a wide variety: they could alter both the control and data flow of the program, and the resulting failures ranged from program crashes to incorrect output. Nevertheless, we intend to do a future study with real bugs to confirm that thin slicing still provides a significant benefit.

Our use of breadth-first search on the dependence graph to simulate programmer exploration of the slice may not accurately reflect how developers would use a slicing tool. If most developers are able to very quickly prune statements in a traditional slice irrelevant to their tasks, then the BFS metric would overstate the advantage of thin slicing. In the future, we aim to do a user study to obtain more definitive answers on the productivity benefits of thin slicing.

Finally, our use of whole-program pointer analysis and call graph construction for the thin slicer may not scale to larger benchmarks. These analyses also may not be suitable for use inside a development environment, as code edits could require expensive re-computation of the pointer analysis results. We plan to employ demand-driven, refinement-based pointer analysis [25] in the next version of the thin slicer to overcome these drawbacks.

6.2 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) [7]. 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 6.1).

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`. 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 buggy statement itself fails with a `NullPointerException`. These sorts of bugs can be easily debugged without tool support, but we include them for completeness.

Using the traditional slicer required inspecting 1 to 4.52 times more statements than thin slicing to find the bug. The total number of inspected statements for traditional slicing was 3.3 times higher than with thin slicing, a measure of the total inspection effort saved. The injected bugs in `nanoxml` in particular often required 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 4. 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 inspected significantly fewer statements than the traditional slice.

Few control dependences were 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 (as discussed in Section 4.2). 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 were still near statements in the thin slice.

The precision of our preliminary points-to analysis was key to the effectiveness of the thin slicer. The "ThinNoObjSens" and "TradNoObjSens" columns in Table 2 show our results to be considerably worse with a points-to analysis that does not treat container classes like `Vector` object sensitively. In cases involving such data structures, the number of statements inspected with the thin slice increased by up to a factor of 17.2X with the less precise analysis, likely making the thin slicing tool unusable.

```
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 void 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 5. An example illustrating a tough cast. Expressions in the thin slice used to understand the safety of the cast are underlined.

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 [29] 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.

6.3 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 (we used the same pointer analysis used to construct our call graph). For example, the cast at line 15 in Figure 5, 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. 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.

Tough casts present a good test of the efficacy of thin slicing in aiding program understanding. The safety of tough casts is often due to some global invariant of a program. These invariants are of-

Bug	# Thin	# Trad.	Ratio	# Control	# ThinNoObjSens	# TradNoObjSens
nanoxml-1	12	32	2.67	0	12	32
nanoxml-2	25	113	4.52	0	431	1675
nanoxml-3	29	123	4.24	0	472	1883
nanoxml-4	12	33	2.75	1	17	44
nanoxml-5	35	156	4.46	1	159	45
nanoxml-6	12	52	4.33	0	35	90
jtopas-1	1	1	1	0	1	1
jtopas-2	2	2	1	1	2	2
ant-1	2	2	1	1	2	2
ant-2	4	5	1.25	0	4	5
ant-3	34	55	1.62	15	251	501
ant-4	3	3	1	2	3	3
xml-security-1	2	2	1	1	2	2

Table 2. Evaluation of thin slicing for debugging. For each bug, we show the number of statements that must be inspected in the thin slice (the "Thin" column) and the traditional slice (the "Trad" column) to discover the bug using BFS traversal (see Section 6.1). 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. Finally, we give the number of inspected statements for thin and traditional slicing when container classes are not treated object sensitively [16] in the points-to analysis (the "ThinNoObjSens" and "TradNoObjSens" columns). 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.

ten (in our experience) undocumented, and discovering the invariants can aid the programmer in understanding the overall structure and behavior of the program. Furthermore, discovering these invariants by hand can be difficult, as it often requires tracing value flow through several disparate parts of the program. Hence, easing the understanding of tough casts with tool support aids overall program understanding and additionally can be useful for refactoring or adding parameterized types or annotations.

Our experimental configuration involved first manually identifying those statements that showed each tough cast could not fail (the desired statements of Section 6.1) 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. In the example of Figure 5, we can understand the tough cast 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`. 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 in Table 3. Thin slicing helped understand tough casts more effectively than traditional slicing: the number of statements examined using a traditional slice exceeded by 1.17 to 34.2 times the number examined using a thin slice. In total, 9.4 times more statements were examined with the traditional slicer than the thin slicer. In `javac`, the casts resembled Figure 5. The code includes a large number of `Node` subclasses used pervasively in the program, resulting in large numbers for the traditional slicer. The importance of object-sensitive container handling in the points-to analysis is seen for the `jack` casts, where the number of inspected statements increased by factors of 5.9-16.9X with less precise analysis.

The absolute numbers of inspected statements exceeded those for the debugging tests, but they remained 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 were writes of opcodes in a large number of constructors (like in Figure 5), which could be quickly inspected to ensure that a suitable constant is written. For `jack`, the BFS traversal overestimated the number of thin slice statements that needed to be inspected; 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.

7. Related Work

Since first being defined by Weiser in 1979 [28], 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 [27] and Krinke's thesis [12] 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. [11]. Our implementation of a traditional slicer is in fact tabulation-based, as suggested in [20] and the 20-year Retrospective to [11].

CodeSurfer [1] is a program understanding tool for C and C++ based on the analysis techniques of [11, 22]. CodeSurfer also uses pointer analysis to allow navigation from a use of a heap location to potential defs. 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. While CodeSurfer

Cast	# Thin	# Trad.	Ratio	# Control	# ThinNoObjSens	# TradNoObjSens
mtrt-1	22	51	2.32	0	22	51
mtrt-2	23	52	2.26	0	23	52
jess-1	6	7	1.17	2	6	7
jess-2	13	39	3	0	25	93
jess-3	6	6	1	2	6	6
jess-4	6	7	1.17	2	6	7
jess-5	6	7	1.17	2	6	7
jess-6	6	6	1	2	6	6
javac-1	57	910	16	1	57	910
javac-2	43	853	19.8	1	43	853
javac-3	65	2224	34.2	1	65	2267
javac-4	45	855	19	1	45	855
jack-1	18	79	4.39	0	303	758
jack-2	57	151	2.65	0	339	647
jack-3	18	69	3.83	0	304	603
jack-4	18	79	4.39	0	304	759
jack-5	57	151	2.65	0	339	647
jack-6	35	132	3.77	0	338	802
jack-7	35	132	3.77	0	338	802
jack-8	35	132	3.77	0	338	802
jack-9	30	79	2.63	0	304	759
jack-10	57	151	2.65	0	339	647

Table 3. Evaluation of thin slicing for understanding tough casts. The types of data in the table columns are described with Table 2.

allows navigation of all control and data dependences, thin slicing emphasizes producer statements and shows explainer statements using additional thin slices (see Section 2); our evaluation has shown that this technique quickly leads users to the most relevant statements.

Atkinson and Griswold [5] present a slicer relying on 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 [13] 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 [14] present a novel approach to represent formal parameter objects with trees. Hammer and Snelting [9] 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. [18] 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. [17] 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 [15] 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 [30, 31, 32, 33]. 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 [32, 33] could be leveraged to create a more scalable dynamic thin slicer. Their recent work on pruning dynamic slices [30] 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 [31] 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 [31] also suggests exploring statements closer to the seed first when viewing a slice, an idea we also use in our evaluation.

8. 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 a more selective

notion of relevance; thin slices only include statements producing the value at the seed, rather than all statements that can possibly affect it. Our evaluation shows that desired statements could be found with thin slices while inspecting 3.3 times fewer statements than traditional slices for debugging tasks and 9.4 times 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. Thin slicing provides the basis for a practical and effective program understanding tool, as it provides significant help for finding statements relevant to tasks and scales to large, realistic Java programs.

Acknowledgements This work is supported in part by the IBM OCR program, the National Science Foundation with grants CCF-0085949, CCR-0105721, CCR-0243657, CNS-0225610, CCR-0326577, CNS-0524815, and CCF-0613997 the University of California MICRO program, an Okawa Research Grant, a Hellman Family Faculty Fund Award, and a Microsoft Graduate Fellowship. This work has also been supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCHC020056. The views expressed herein are not necessarily those of DARPA.

We thank Susan Graham for her suggestion of the name "thin slicing." We also thank Dave Mandelin, Adam Chlipala, and the anonymous reviewers for their helpful comments.

References

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

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

[3] M. Allen and S. Horwitz. Slicing Java programs that throw and catch exceptions. In *PEPM '03: Proceedings of the 2003 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 44–54, New York, NY, USA, 2003. ACM Press.

[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] 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.

[8] 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.

[9] 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.

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

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

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

[13] L. Larsen and M. J. Harrold. Slicing object-oriented software. In *International Conference on Software Engineering (ICSE)*, 1996.

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

[15] 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.

[16] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, 2005.

[17] 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.

[18] 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.

[19] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *IEEE International Conference on Automated Software Engineering (ASE)*, 2003.

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

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

[22] 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.

[23] 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.

[24] 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.

[25] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[26] T. Teitelbaum. Personal communication regarding CodeSurfer. 2007.

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

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

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

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

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

[32] 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)*, 2004.

[33] 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.

[34] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken. Statistical debugging: Simultaneous identification of multiple bugs. In *Proceedings of the 23rd International Conference on Machine Learning*, 2006.