

# On the utility of cutpoints for monitoring program execution

Shachar Rubinstein<sup>1</sup>

School of Computer Science, Tel-Aviv University, Israel

August 2006

<sup>1</sup>shachar1@post.tau.ac.il

# Acknowledgements

I would like to thank:

- Prof. Shmuel (Mooly) Sagiv for his trust in me, guidance and invaluable support.
- Noam Rinetzkyl for his always positive outlook and encouragements, advice and ideas.
- The rest of Prof. Sagiv's group for their advice and assistance.
- Dr. Erez Petrank and Dr. Harel Paz, for pointing out the possibility of using garbage collection algorithms to detect cutpoints.
- The Jikes RVM group and researchers mailing list, especially Assoc. Prof. J. Eliot B. Moss, for their help in learning how to use this amazing software.
- Prof. Sivan Toledo and his students for providing computation resources.
- Prof. Amiram Yehudai for assisting in the area of design by contract.
- Anat Lotan for improving the thesis write up.
- Dr. Ran Shaham and Liam Roditty, who have helped me when I was searching for directions.
- Yotam Shtossel, Dr. Zur Izhakian and Daphna Amit for their company and companionship along the way.
- Dan, Micha, Carine, Asi, Noa, Irit, Orit, Dana and Yael, who endured and encouraged me during my work. I apologize if I have omitted anyone.
- For my loving family, without whom I would not be here today.
- The Israeli National Academy of Science for their financial support.

# Abstract

Sharing mutable data is a powerful programming technique, but it makes programs hard to understand. *Local heaps* and *cutpoints* are a notion introduced by Rinetzky et. al. ([29]) in order to understand and analyze programs.

In this work we develop a runtime tool for measuring the number of cutpoints which can occur in a given program. The tool encourages programmers to reduce the number of cutpoints, thus eliminating erroneous aliasing leading to cutpoints. We introduce a way to refine the results of the tool by adding a notion of *live* and *dead cutpoints* and an algorithm for their detection. Finally, we demonstrate a use for cutpoints by developing a new algorithm for runtime check of class invariants.

# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Background . . . . .	5
1.2	Main results . . . . .	6
1.3	Thesis organization . . . . .	7
<b>2</b>	<b>Local heaps and cutpoints</b>	<b>8</b>
2.1	Local heap . . . . .	8
2.2	Cutpoint . . . . .	9
2.3	Usage . . . . .	10
<b>3</b>	<b>Computing cutpoints</b>	<b>11</b>
3.1	Preliminaries . . . . .	11
3.2	Naive attempts . . . . .	11
3.2.1	Scanning the global heap . . . . .	11
3.2.2	Using a source list . . . . .	12
3.3	Our solution . . . . .	12
3.4	The cutpoint detection algorithm . . . . .	13
3.4.1	Acyclic data types . . . . .	14
3.4.2	A running example . . . . .	14
<b>4</b>	<b>Computing live and dead cutpoints</b>	<b>18</b>
4.1	Live and dead cutpoints . . . . .	18
4.2	Computing live cutpoints . . . . .	19
4.2.1	Collecting cutpoint referencing fields . . . . .	22
4.2.2	Finding liveness . . . . .	23
4.3	Computing external sources using source lists . . . . .	23
<b>5</b>	<b>Early detection of class invariant violations</b>	<b>35</b>
5.1	Design by contract . . . . .	35
5.1.1	The sharing problem . . . . .	36
5.2	Computing invalid class invariants . . . . .	40
5.2.1	Using cutpoints . . . . .	40
5.2.2	Holding cutpoints . . . . .	40
5.2.3	The cutpoint list . . . . .	41
5.2.4	Backward scan . . . . .	41

5.3 The computation algorithm . . . . .	42
<b>6 Results</b>	<b>46</b>
6.1 Motivation . . . . .	46
6.2 Measurements . . . . .	46
6.2.1 Top cutpoint producing methods . . . . .	46
6.2.2 Well-known classes effect . . . . .	46
6.2.3 Methods' maximum cutpoints disparity . . . . .	47
6.3 The benchmarks . . . . .	48
6.3.1 Soot: a Java optimization framework . . . . .	48
6.3.2 The Kawa language framework . . . . .	48
6.3.3 SPEC JVM98 benchmarks . . . . .	48
6.3.4 TVLA: 3-valued logic analysis engine . . . . .	48
6.4 Results . . . . .	49
6.4.1 Shared immutable objects . . . . .	49
6.4.2 String . . . . .	50
6.4.3 Methods' maximum cutpoints disparity . . . . .	50
<b>7 Related work</b>	<b>52</b>
<b>8 Future Work</b>	<b>54</b>
8.1 Suggestions for future work . . . . .	54
8.1.1 Prototype . . . . .	55
8.2 Limitations . . . . .	55
<b>A Prototype implementation</b>	<b>63</b>
A.1 Picking a platform . . . . .	63
A.1.1 Limitations . . . . .	63
A.1.2 The build process . . . . .	64
A.2 Common preliminaries . . . . .	64
A.2.1 Working on the user program . . . . .	64
A.2.2 Holding sources . . . . .	67
A.3 Computations specific . . . . .	68
A.3.1 Computing cutpoints preparations . . . . .	68
A.3.2 Live and dead cutpoints . . . . .	68
A.3.3 Early detection of class invariants violation . . . . .	68
A.4 Other implementation notes . . . . .	69
A.4.1 Uninterruptible code . . . . .	69
A.4.2 Summary of object header changes . . . . .	71
<b>B Results processing</b>	<b>72</b>
B.1 The prototype raw file . . . . .	72
B.2 The summary file . . . . .	73
B.3 Database processing . . . . .	75

# Chapter 1

## Introduction

### 1.1 Background

Understanding the behavior of heap manipulating (object oriented) programs is a challenge. Such programs exhibit complex relationships between the structure of the program and the reference structure of heap allocated objects. Aliasing between references makes programs hard to understand, debug, and verify. Visibility keywords such as *private* suggest that some data should be encapsulated, but do not prevent public methods from returning aliases to that (supposedly) internal data. Indeed sharing mutable data complicates reasoning about programs both informally and formally.

On the other hand, sharing mutable data is a powerful programming technique. For example, the model-view-controller design pattern [12] captures the essential structure of many graphical user interfaces: many controllers and views share the same object. Indeed it is obvious that while sharing and aliasing is problematic some sharing, e.g., temporary sharing created inside a simple procedure is usually harmless and very useful.

In [29], Rinetzky et. al. define the notion of *local heaps* and *cutpoint objects*. The local heap of a procedure contains only the objects reachable from the formal parameters. Cutpoints are objects which separate the local heap (that can be accessed by a procedure) from the rest of the heap (which—from the viewpoint of that procedure—is non-accessible and immutable).

Programs with few (or even no) cutpoints can be simpler to understand and to analyze. For example, in [30], a shape analysis for cutpoint-free programs was developed. The main idea is that the absence of cutpoints allows to extract the meaning of a procedure as an input/output relation which is independent of the sharing created in the calling context, and thus supports the notion of procedural abstraction. Gotsman et. al. [13] developed an analysis for programs with few cutpoints.

## 1.2 Main results

This thesis develops a runtime tool for measuring the number of cutpoints which can occur in a given program. The tool is totally automatic. The tool encourages programmers to reduce the number of cutpoints, this eliminating erroneous aliasing leading to cutpoints. It can also be used by tool designers to understand the behavior of existing programs. Finally, it can be used for more effective checking of cases where the class (object) invariant is violated.

The main algorithm in the tool uses a runtime garbage collector to reduce the cost of scanning the entire global heap. Specifically, our algorithm is based on the solution presented in [2] for the cycle detection problem in reference counting based garbage collection.

We make two observations concerning [2] solution. The first observation is that the cycle collection algorithm divides the global heap into two regions: The potential roots of cyclic garbage and their transitive closure, and the rest. The second observation is that cycles, which are not garbage, are detected by finding references from the second region to the first.

By changing the potential roots to be the method's formal parameters, the first region becomes the local heap. Applying the second observation to this modification adjusts [2] solution to solve the cutpoint detection problem.

The tool is implemented on top of Jikes RVM [16] which is a Java virtual machine written in Java. Jikes RVM already implements the algorithm of [2] and is freely available.

The contributions of this thesis can be summarized as follows:

- We develop a novel algorithm for computing cutpoints using a cycle collection algorithm. The cost of the algorithm is linear in the size of local heap.
- We define the notion of *live* cutpoint objects, which are cutpoints that are referred by the program after the procedure returns via an access path bypassing the local heap. The main idea is that cutpoint objects which are not live (dead) represent harmless sharing.
- We develop an algorithm for computing live cutpoints.
- We develop a new algorithm for checking class invariants. The main idea is to use cutpoints for checking violations due to mutations of shared objects.
- We applied the algorithm to several benchmarks.

We limit our work to programming languages that pass objects to procedures by reference only, not by value. For example, The C++ programming language can pass objects on the call stack.

### **1.3 Thesis organization**

The rest of the thesis is organized as follows: Chapter 2 defines cutpoints and local heaps. Chapter 3 presents the cutpoint detection algorithm. Chapter 4 defines live and dead cutpoints and presents two new algorithms: live cutpoints detection and external sources computation. Chapter 5 introduces the sharing problem in design by contract and presents a cutpoint-based algorithm for early detection of class invariants violation. Chapter 6 shows our empirical results. Chapter 7 discusses related work and Chapter 8 concludes this thesis with ideas for future work. The appendices include prototype implementation details in Appendix A and details about results processing in Appendix B



## Chapter 2

# Local heaps and cutpoints

This chapter defines the local heap and cutpoint notions.

### 2.1 Local heap

**Definition 2.1.1 (Local Heap)** *The **local heap** for an invocation of a procedure  $p$  is the part of the heap which is accessible to the procedure. The objects that belong to the local-heap are those reachable from the procedure's formal parameters and local variables.*

A local heap exists only in the context of a procedure's execution and during that execution only. The *this* pointer in instance methods is considered a formal parameter too.

**Definition 2.1.2 (Global Heap)** *The **global heap** is the whole heap*

This definition is used to prevent confusion with the local heap.

**Observation 2.1.3 Object stack continuous reachability** - *An object is reachable from the program's call stack continuously.*

If an object becomes unreachable from the stack at depth  $i + 1$ , not because it has become garbage, and the stack depth grows, then the object will not be reachable again until the stack depth returns to  $i$ . If an object is unreachable, there is no possibility for deeper stack procedures to reach the object (excluding objects reachable from static fields). Therefore the stack reachability of an object is continuous.

The object stack continuous reachability property is used throughout the paper as a basis for computations, appearing in Section 4.3 and Section 5.2.3.



## 2.3 Usage

We suggest using the local heap, instead of the global heap, to understand a program's memory behavior. The global heap can contain a great number of objects while a procedure may access only a very small fraction of them. Therefore the local heap perspective assists in gaining a better understanding of the effect of a procedure.

Using cutpoints complements the local heap perspective. Together they provide a novel way of investigating the behavior of programs and their use of memory. The following chapters will provide ways to utilize the two to gain interesting information about programs.

## Chapter 3

# Computing cutpoints

This chapter defines a new algorithm for computing cutpoints. The chapter presents two naive solutions and shows why the new algorithm is better.

### 3.1 Preliminaries

Recall that cutpoints are defined for a method at the time of the invocation.

In order to identify a cutpoint, the algorithm has to determine which objects belong to the local heap of the invoked method and are referred from outside without passing through a formal parameter.

An *object* denotes a class object or an array object. A *field* in a class object is a class member variable. An array element is referred to as a *field* in an array.

### 3.2 Naive attempts

#### 3.2.1 Scanning the global heap

A simple method to compute cutpoints is to scan the local heap and then to scan the global heap. This is performed in two stages:

1. The local heap is scanned and each object is marked as local
2. The global heap is scanned and cutpoints are identified. Notice that here references between local heap objects are not traversed.

The cost of the first stage is  $O(n + e)$  where  $n$  is the number of objects and  $e$  is the number of references in the local heap. The cost of the second stage is  $O(N + E)$  where  $N$  is the number of objects and  $E$  is the number of references in the global heap. Therefore, since usually  $n \ll N$  and  $e \ll E$ , the dominant cost is  $O(N + E)$ .

### 3.2.2 Using a source list

Scanning the global heap on each method is expensive. One approach to reduce this cost is to maintain a list of objects which refer to a given object (inverse reference fields). This allows to check if an object in the local heap is referred from outside without scanning the global heap. This list is referred to as a *Source List*. An object  $o$ , which has a reference field pointing to an object  $o'$ , is referred to as a *source* of  $o'$  and mentioned as  *$o$  refers to  $o'$* .

This is performed in two stages:

1. The local heap is scanned. Every object in the local heap is marked as local in the list of each of the objects it refers-to.
2. The local heap is scanned and each object's source list is checked. If the list has objects not marked as local, the object is a cutpoint.

The cost of the first stage is  $O(n + e \times s)$  where  $s$  is the cost of searching the source list for an object. The search cost is implementation dependent. The cost of the second stage is  $O(n \times d + e)$  where  $d$  is the cost of finding if there is at least one unmarked object in the list ( $d$  can be done in constant time, reducing the cost of this stage to  $O(n + e)$ ). Therefore the dominant cost is  $O(n + e \times s)$ .

The cost of maintaining the source lists for the objects in the global heap is an additional cost, which does not appear in the above. This cost has to be taken into account when comparing the total cost of different solutions. Nevertheless, for the sake of brevity we do not add it here.

## 3.3 Our solution

Our algorithm is based on the solution presented in [2] for the cycle detection problem in reference counting based garbage collection. Specifically, the synchronous cycle collection algorithm, which is single-threaded. Nevertheless, other than the following observations and their application, understanding of the aforementioned work is not mandatory.

We make two observations concerning [2] solution. The first observation is that the cycle collection algorithm divides the global heap into two regions: The potential roots of cyclic garbage and their transitive closure, and the rest. The second observation is that cycles, which are not garbage, are detected by finding references from the second region to the first.

By changing the potential roots to be the method's formal parameters, the first region becomes the local heap and the second the global heap, excluding the local heap. Applying the second observation to this modification adjusts [2] solution to solve the cutpoint detection problem.

This solution obliges a reference count garbage collection or, in the case of other garbage collection, a mechanism which maintains a reference count for all objects.

The algorithm proceeds in three stages:

1. The local heap is scanned. Reference counts are decremented for internal references.
2. The local heap is scanned. An object with a positive reference count is a cutpoint.
3. The local heap is scanned. Reference counts are incremented for internal references.

The third stage restores the reference counts to their original value.

The cost of each stage is  $O(n + e)$  and, as a result, it is the dominant cost too. The linear-in-the-local-heap cost is achieved by using a single counter instead of scanning the actual referencing objects.

Maintaining the reference counts adds another cost, which can be ignored if using a reference count garbage collector.

### 3.4 The cutpoint detection algorithm

Each object  $T$  has a color and a reference count, denoted as  $\text{color}(T)$  and  $\text{RC}(T)$  respectively. The colors used are shown in Table 3.1.  $\text{children}(S)$  is a multi-set of objects that object  $S$  references, including duplicates, as  $S$  may reference an object more than once. The algorithm is shown in Fig. 3.1. `ComputeCutpoints` is invoked at the beginning of each relevant method. The rest of the procedures are internal to the algorithm. `MarkGray` and `ScanRoots` are identical to their version in [2].

**ComputeCutpoints( $f$ )** Whenever a cutpoint computation is needed on method  $f$  this procedure is invoked. There are three parts: `GetRoots`, which gathers the roots for the algorithm, `MarkRoots`, which decrements the internal references, and `Scan`, which finds cutpoints and restores the internal references to their original values.

**GetRoots( $f$ )** The formal reference parameters of the method  $f$  are extracted and inserted into the `Roots` set.

**MarkRoots(`Roots`)** The first stage removes internal references in the local heap by running `MarkGray` on each reference collected in `Roots`.

**MarkGray( $S$ )** This procedure performs a simple depth-first traversal of the graph beginning at  $S$ , marking visited nodes gray and removing internal reference counts as it goes.

**ScanRoots(`Roots`)** For each object in `Roots` that was considered by `MarkGray( $S$ )`, this procedure invokes `Scan( $S$ , Roots)` to detect cutpoints and restore reference counts.

Color	Meaning
Gray	Reference count decremented
Black	Initial color/Checked for cutpoint

Table 3.1: Colors in use

**Scan(S,Roots)** The second and third stages are optimized and implemented as one stage, reducing one local heap scan. This procedure scans the local heap, detecting cutpoints and restoring reference counts to their original value. Object reference count is restored by performing a depth first search and incrementing references as it goes. References are restored after the object is checked for being a cutpoint. If the object *S* belongs to the *Roots* set, it is not reported as a cutpoint, since the formal parameters are not cutpoints.

### 3.4.1 Acyclic data types

[2] implements a scheme to determine acyclic classes. The authors hypothesize that this kind of objects compromise the majority of objects in many applications. Therefore the cutpoint detection algorithm includes acyclic data types, as this is interesting information, which may help support this hypothesis.

### 3.4.2 A running example

**Example 3.4.1** *Fig. 3.2 shows the initial memory status. The graphics conventions used here are used throughout the rest of the thesis. The numbered ellipses on each object represent the number of references an object has. The color of the ellipse is the current `color(T)` of that object. References from the stack are not counted. The call stack is labeled with the invoked methods when there is a program with the example. The heap outside the local heap is printed as translucent.*

*Object B is passed as an actual parameter to an invoked method. The resulting local heap is shown in Fig. 3.3 inside the cloud. The objects reachable from B are C, E and F. Therefore they are part of the local heap.*

*Roots = {B}. The result of running `MarkGray(B)` is shown in Fig. 3.4. A surrounding cloud is added to help locate the current local heap. The rest of the heap is printed as translucent. Object C is referenced by objects B and F, which are inside the local heap. Therefore the reference count of object C is down to zero. Objects B, E and F are referenced from outside the local heap; B by A, E by D and G by F. The objects' reference count indicates this fact and hence E and F are cutpoints. Object B is not a cutpoint because it is a formal parameter.*

*Running `Scan(B,Roots)` returns the reference count to their original values and colors the objects from gray to black. The result is the same as in Fig. 3.3.*

```

ComputeCutpoints(f)
    Roots = GetRoots(f)
    MarkRoots(Roots)
    ScanRoots(Roots)

GetRoots(f)
    Roots = {}
    for each AP formal parameter of method f
        if (AP is an object reference)
            add AP to Roots
    return Roots

MarkRoots(Roots)
    For each S in Roots
        MarkGray(S)

MarkGray(S)
    if (color(S) != gray)
        color(S) = gray
    for each T in children(S)
        RC(T) = RC(T) - 1
        MarkGray(T)

ScanRoots(Roots)
    for each S in Roots
        Scan(S,Roots)

Scan(S,Roots)
    if (color(S) == gray)
        if((RC(S) > 0) and (S not in Roots))
            S is a cutpoint
        color(S) = black
    for each T in children(S)
        Scan(T,Roots)
        RC(T) = RC(T) + 1

```

Figure 3.1: Cutpoints detection algorithm



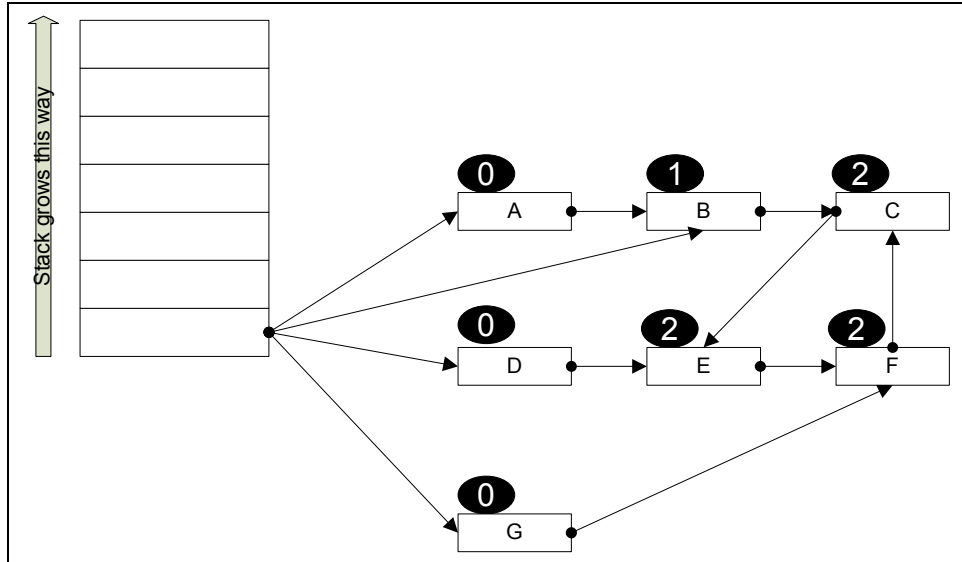


Figure 3.2: Detecting cutpoints example initial memory

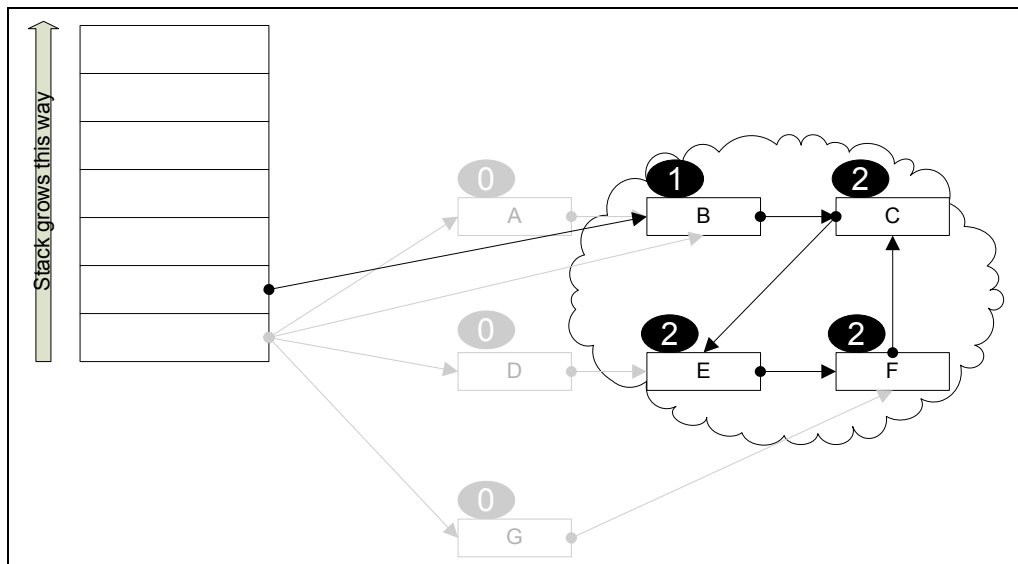


Figure 3.3: Detecting cutpoints example method call

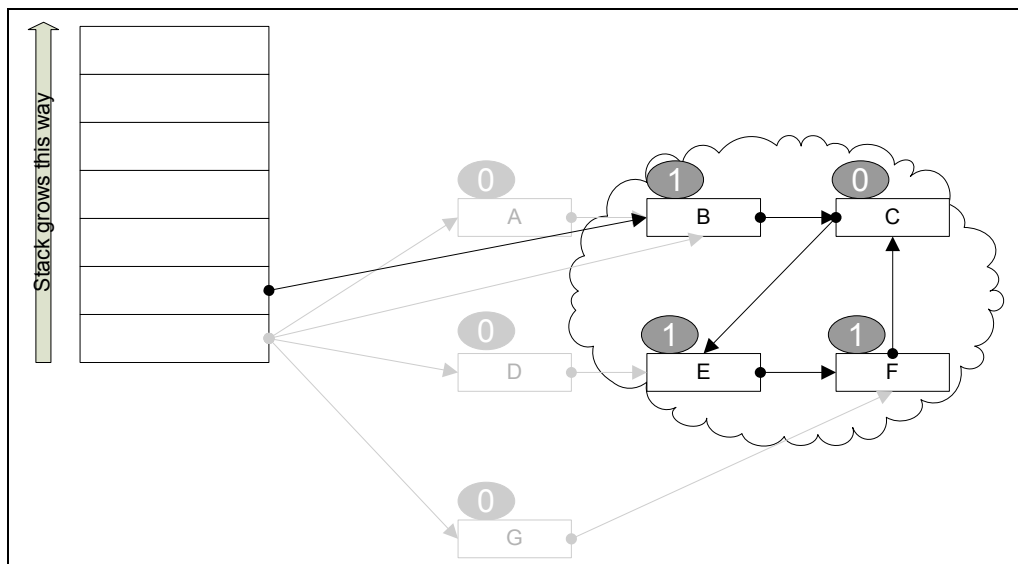


Figure 3.4: Detecting cutpoints example MarkGray result

## Chapter 4

# Computing live and dead cutpoints

This chapter introduces live and dead cutpoints and presents an algorithm for their detection.

### 4.1 Live and dead cutpoints

The cutpoints in a local heap provide a description of a method's external sharing. Nevertheless, the reported numbers may present an inflated image of the actual impact of these cutpoints. A reference causing a cutpoint may never be used, due to being overwritten or because the reference owning object is released. In this case, the cutpoint does not have any effect. This information can be used to refine the results from Section 3.4. Therefore the cutpoint definition is refined here. A *cutpoint referencing field* is an object's field referencing a cutpoint, where the object does not belong to the local heap at the time of the referenced cutpoint detection. A *live cutpoint field* is a cutpoint referencing field, which was read after the cutpoint was detected and before it was overwritten, or before the referencing object was released. Otherwise the field is a *dead cutpoint field*. A *live cutpoint* is a cutpoint where at least one of its cutpoint referencing fields is a live cutpoint field. Otherwise it is a *dead cutpoint*. The term *liveness* is used to describe the process of finding live cutpoints. This should not be confused with other usages of the term, such as variable liveness used in optimizing compilers.

**Example 4.1.1** *The following exemplifies the aforementioned terms. The example is shown in Fig. 4.2. The program uses a singly linked list class, Node, which is shown in Fig. 4.1. The program initializes its data structure in lines 1-4. The result of this initialization is shown in Fig. 4.3. Object L is referenced by objects A, B and C and has a reference count of three. The latter are referenced by srcArray and have a reference count of one each.*

```

class Node {
    private Node mNext = null;
    private int mData = 0;

    public Node(int _data, Node _next) {
        mNext = _next;
        mData = _data;
    }

    public void setNext(Node _next) {
        mNext = _next;
    }

    public Node getNext() {
        return mNext;
    }

    public int getData() {
        return mData;
    }
}

```

Figure 4.1: Liveness example node class

*After initializing, the program calls print at line 5. The actual parameter passed is object A. The resulting local heap is shown in Fig. 4.4. Running the cutpoint detection algorithm at the beginning of print detects object L as a cutpoint (The result of the MarkGray stage is shown in Fig. 4.5). Object A is not a cutpoint because it passes through a formal parameter, itself.*

*There are two cutpoint referencing fields (Object A's field referencing B is irrelevant as A is part of the local heap): Object B's mNext and object C's mNext. The next call at line 6 assigns null to object B cutpoint referencing field. Therefore object B's field is a dead cutpoint field. In lines 7 and 8 object C's cutpoint referencing field is read. Thus object C's field is a live cutpoint field. As a result the cutpoint detected in print, object L, is a live cutpoint.*

## 4.2 Computing live cutpoints

Note: This computation handles heap references and does not handle stack references.

Finding live or dead cutpoints is carried out in three stages:

1. Collecting cutpoint referencing fields into a list

```

    public static void main(String[] args) {
1:      Node tgt = new Node(0,null);

2:      Node[] srcArray = new Node[3];
3:      for(int i=0;i<3;++i)
4:      {
5:          srcArray[i] = new Node(i+1,tgt);
6:      }
7:      print(srcArray[0]);
8:      srcArray[1].setNext(null);
9:      if(srcArray[2].getNext() != null)
10:     {
11:         print(srcArray[2].getNext());
12:     }
13: }

    public static void print(Node _toPrint)
    {
14:     if(_toPrint != null)
15:     {
16:         System.out.println(_toPrint.getData());
17:     }
18: }

```

Figure 4.2: Liveness example program

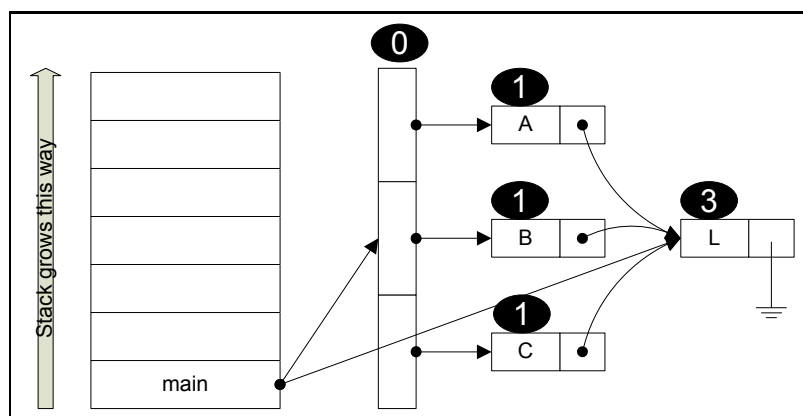


Figure 4.3: Liveness example memory status of the program in Fig. 4.2 before line 5

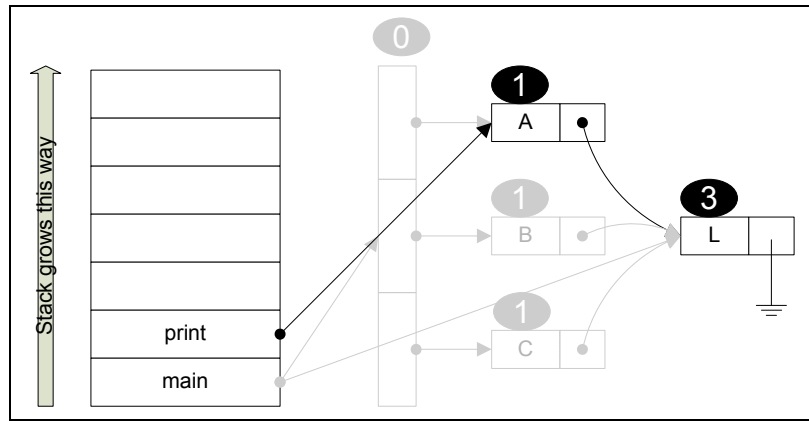


Figure 4.4: Liveness example memory status of the program in Fig. 4.2 before line 9

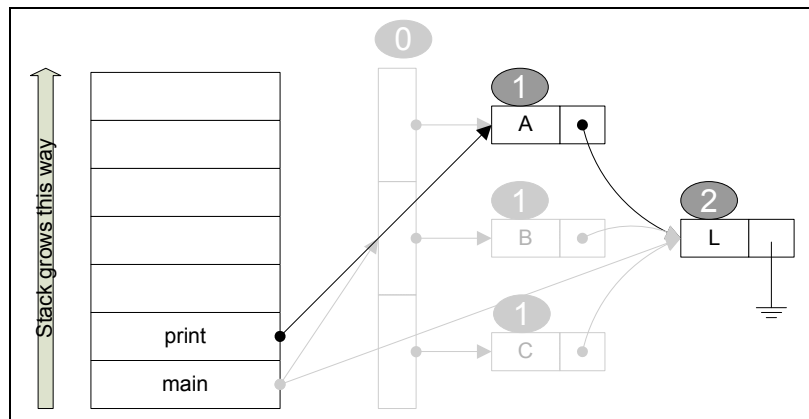


Figure 4.5: Liveness example memory status of the program in Fig. 4.2 before line 9 after MarkGray

```

OnDetectedCutpoint(CP)
  for each object Src referencing CP
    if (Src is external to the local heap)
      for each field Fld in Src referencing CP
        add (Src,Fld) to TestedCutpoints

```

Figure 4.6: Collecting cutpoints for liveness

2. Finding if a cutpoint referencing field on the list is live or dead
3. Aggregating cutpoint referencing fields results into live or dead cutpoints

The third stage can be carried out post or in-processing. The following computation performs the first two stages.

#### 4.2.1 Collecting cutpoint referencing fields

Cutpoints are discovered on each method entry and their sources are collected there. The collection is described in Fig. 4.6.

**TestedCutpoints** is a liveness candidate list. The list holds cutpoint referencing fields as pairs of owning object and field. The cutpoint object can be obtained by dereferencing the object and field.

**OnDetectedCutpoint(CP)** When a cutpoint is detected its referencing fields and their owning objects are added to **TestedCutpoints** for tracking. The added objects are outside the local heap. Selecting which objects to add is explained in Section 4.3

In order to compute live or dead cutpoints, the computation tracks the objects referencing a cutpoint. References are a one way addressing mechanism. Therefore an object lacks any knowledge as to which objects reference it. Scanning the global heap each time is one solution. Another possibility is to maintain a source list for each object and to find the external sources. Section 4.3 shows how to find the external sources in the source list.

A cutpoint referencing field may be detected again and again before it is accessed and tested for liveness. This is because the object owning the field is inaccessible until it becomes internal to the local heap (Observation 2.1.3). When the field is accessed, it is removed by the computation from the **TestedCutpoints** list. Therefore there is no need to check for an existing entry when adding a field and source pair to the list. Nevertheless, adding a source and its fields to **TestedCutpoints** requires some work, which can be reduced, as shown in Section 4.3

```

WriteBarrier(Obj,Fld)
    if((Obj,Fld) is in TestedCutpoints)
        remove (Obj,Fld) from TestedCutpoints
        CP = Obj.Fld
        report CP from (Obj,Fld) as dead cutpoint referencing field

ReadBarrier(Obj,Fld)
    if((Obj,Fld) is in TestedCutpoints)
        remove (Obj,Fld) from TestedCutpoints
        CP = Obj.Fld
        report CP from (Obj,Fld) as live cutpoint referencing field

OnObjectRelease(Obj)
    for each object reference field Fld in Obj
        WriteBarrier(Obj,Fld)

```

Figure 4.7: Finding object liveness

#### 4.2.2 Finding liveness

There is one procedure for each case in the live and dead cutpoint reference field definitions.

**WriteBarrier(Obj,Fld)** This procedure is called on every assignment to an object reference field. First `TestedCutpoints` is searched for the written source and field. If found, then this field is reported as a dead cutpoint referencing field.

**ReadBarrier(Obj,Fld)** This procedure is called on every read from an object reference field. The procedure is the same as `WriteBarrier(Obj,Fld)` but reports live cutpoint referencing field.

**OnObjectRelease(Obj)** When an object is released, its fields are not read anymore. If the fields are tracked for liveness, then they are removed from `TestedCutpoints` and declared as dead. This is conducted by iterating over the released object's object reference fields and calling `WriteBarrier` for each object-field pair.

### 4.3 Computing external sources using source lists

This section presents a new algorithm for determining local heap external sources in source lists.



The first part of the liveness process, appearing in Fig. 4.6, collects only external sources. When using a global heap scan for cutpoint detection, cutpoints are found through external sources. If utilizing source lists, detection of external sources can be carried out as follows:

1. The local heap is scanned. Sources scanned are marked as internal.
2. The local heap is scanned. The external sources are those not marked as internal. All internal markings are cleared.

Using this algorithm, the same external source can be detected over and over again. As mentioned in Section 4.2.1, overwriting does not present a computation error. Nevertheless it introduces additional work that can be prevented. The following algorithm adds discovery information, providing the ability to distinguish when a source has become external.

The algorithm takes advantage of the object stack continuous reachability property (Observation 2.1.3) to mark objects in the source list with the stack depth in which they have become external. Comparing the referencing object's external depth to the current method's stack depth will provide the answer to whether the object is internal or external and when it has become external.

**Definition 4.3.1 (External depth flag)** *An **external depth flag** is a numeric value  $e$ , where  $e \in \mathbb{N}$ , marking the depth in which a source became unreachable from procedure  $p$ 's formal parameters and local variables. The flag is stored in a source  $s$ 's entry in the source list of object  $o$ , which implies that  $s$  refers-to  $o$ . There are two reserved values, `Internal` and `Scan internal`.*

An object's reachability in the current procedure, not during the external sources computation, is determined using the external depth flag as follows:

**Internal** If the flag is marked as `Internal` or has a numeric value larger than the current procedure's depth.

**External** If the flag has a numeric value equal or smaller than the current procedure's depth.

The possible values for the external depth flag appear in Table 4.1. The `Scan Internal` value is used only during the computation. This value allows the algorithm to differentiate between previously internal sources and the current algorithm's internal objects. `Internal` indicates a referencing object, which is internal to the local heap. A natural number value  $e$  indicates a referencing object which has become external at the depth of  $e$ .

Every object in the source list is added an external depth flag. The flag has to be maintained on each method call to be up-to-date. Referencing objects may have more than one cutpoint referencing field for the same cutpoint. Therefore the referencing object's fields are also kept in the source list. Even though, the external

Value	Meaning
Scan Internal	Temporary scan value
Internal	The referencing object is in the local heap
1-Maximal stack depth	The value is the stack depth where the object has become external. The referencing object is either inside or outside the local heap, relatively to the current method stack depth

Table 4.1: External depth flag possible values

depth flag is saved at source object and not for each field, as the external depth flag has the same value for all the object fields.

The algorithm is shown in Fig. 4.8. `source_list(T)` is the group of objects referencing `T`. The initial external depth flag value is `Internal`. The algorithm uses similar procedures as the cutpoint detection algorithm (appears in Fig. 3.1) and naturally integrates with it. The first stage of the algorithm runs with the `MarkGrey(S)` procedure. The second stage runs with the `Scan(S, Roots)` procedure, but does not use the `Roots` group, as if a root object is reachable from another root, the object should be marked too. Nevertheless, we present here a stand-alone algorithm.

$(S, T)$  is an entry in object `T`'s source list when object `S` refers-to object `T`. `External(S, T)` is the external depth flag of source `S` in `T`'s source list. The algorithm uses the colors in Table 3.1.

The following procedures are identical to the ones in the cutpoint detection algorithm, Fig. 3.1: `GetRoots`, `MarkScanInternalRoots` and `MarkRoots`, `MarkExternalRoots` and `ScanRoots`.

**MarkScanInternalRoots(Roots)** The first stage scans the reachable objects from `Roots`, marking them as belonging to the local heap.

**MarkScanInternal(S)** Object `S` is marked as `Scan Internal` on each source list of the objects refers-to.

**MarkExternalRoots(Roots)** The second stage scans the `Roots` reachable objects, finding external sources.

**Scan(S)** Object `S`'s source list objects are marked as internal or external according to their external depth flag value. After that the objects `S` refers-to are scanned.

**MarkExternals(T)** Finds object `T`'s external sources and marks them with the current depth. If a source `S` was marked by `MarkScanInternalRoots(Roots)` as `Scan Internal`, it is internal and marked as `Internal`. The rest of the sources are external. `S` is marked with the current stack depth in two cases:

- If the current value is `Internal`, then the source has just become external.
- If the current value is equal or higher than the current stack depth, then the source was external, became internal again and now has become external.

If `S` is marked with a lower stack depth, then it has become external on an earlier method in the call stack and hence the flag is left unchanged. The source can not be internal while its external depth flag has a lower value than the current stack depth, because otherwise the source would have been scanned and found as internal.

**Example 4.3.2** *The example program is shown in Fig. 4.9. The program uses a singly linked list class, `Node`, which is shown in Fig. 4.1. Fig. 4.10 shows the example's initial status, after initialization in lines 1-4. Objects `A`, `B`, `C` and `D` are referenced by an array, `refArray`, one in each cell. Each of them references object `S`. The list above object `S` is its source list. Each source is represented in the list with its external depth flag. The flag's initial value is `Internal`.*

*Fig. 4.11 shows the local heap after the call to method `printFirst` in line 5, with objects `A` and `B` as the actual parameters. The result of `MarkScanInternalRoots` is shown in Fig. 4.12. Objects `A` and `B` have reference fields to object `S` and therefore are marked as `Scan Internal` in object `S`'s source list. Fig. 4.13 shows the result of `MarkExternalRoots`. Objects `A` and `B` are marked as `Internal` in object `S`'s source list because they were marked as `Scan Internal`. Objects `C` and `D` are marked with the current stack depth, because they were not scanned in `MarkScanInternalRoots` and they were found as `Internal` by `MarkExternalRoots`. Therefore objects `C` and `D` are external.*

*Method `printFirst` calls to method `print` in line 7. The resulting local heap is shown in Fig. 4.14. The actual parameter is object `A`. The result of `MarkScanInternalRoots` is shown in Fig. 4.15. Object `A` is marked as `Scan Internal` in object `S`'s source list. Fig. 4.13 shows the result of `MarkExternalRoots`. Object `A` is marked as `Internal` in object `S`'s source list. Objects `C` and `D` are not changed since their external depth flag is lower than the current stack depth. They are already external. On the other hand, Object `B` has become external and is marked with the current stack depth.*

*In line 6 method `printTwo` is called. The actual parameters are objects `A` and `C`. The resulting local heap is shown in Fig. 4.17. The result of `MarkScanInternalRoots` is shown in Fig. 4.18. Objects `A` and `C` are marked as `Scan Internal` in object `S`'s source list. Fig. 4.19 shows the result of `MarkExternalRoots`. Objects `A` and `C` are marked as `Internal` in object `S`'s source list. Object `D` is not changed since it is still external. Object `B` was external at a deeper stack depth, became internal when `print` returned and now*

```

ComputeExternalSources(f)
    Roots = GetRoots(f)
    MarkScanInternalRoots(Roots)
    MarkExternalRoots(Roots)

MarkScanInternalRoots(Roots)
    For each S in Roots
        MarkScanInternal(S)

MarkScanInternal(S)
    if (color(S) != gray)
        color(S) = gray
        for each T in children(S)
            External(S,T) = Scan Internal
            MarkScanInternal(T)

MarkExternalRoots(Roots)
    for each S in Roots
        Scan(S)

Scan(S)
    if (color(S) == gray)
        color(S) = black
        MarkExternals(S)
        for each T in children(S)
            Scan(T)

MarkExternals(T)
    for each S in source_list(T)
        if(External(S,T) == Scan Internal)
            External(S,T) = Internal
        else
            if(External(S,T) == Internal or
               External(S,T) > current stack depth)
                External(S,T) = current stack depth

```

Figure 4.8: Computing external sources algorithm

*is external again at a shallower depth. Hence the external depth flag of object  $B$  is larger than the current stack depth and marked now with the current depth.*

```

public static void main(String[] args)
{
1:     Node tgt = new Node(0,null);

2:     Node[] refArray = new Node[4];
3:     for(int i=0;i<4;++i)
        {
4:         refArray[i] = new Node(i+1,tgt);
        }
5:     printFirst(refArray[0],refArray[1]);
6:     printTwo(refArray[0],refArray[2]);
    }

    public static void printFirst(Node _first, Node _second)
    {
7:         print(_first);
    }

    public static void print(Node _node)
    {
8:         System.out.println(_node.getData());
    }

    public static void printTwo(Node _first, Node _second)
    {
9:         print(_first);
10:        print(_second);
    }

```

Figure 4.9: Computing external sources example program

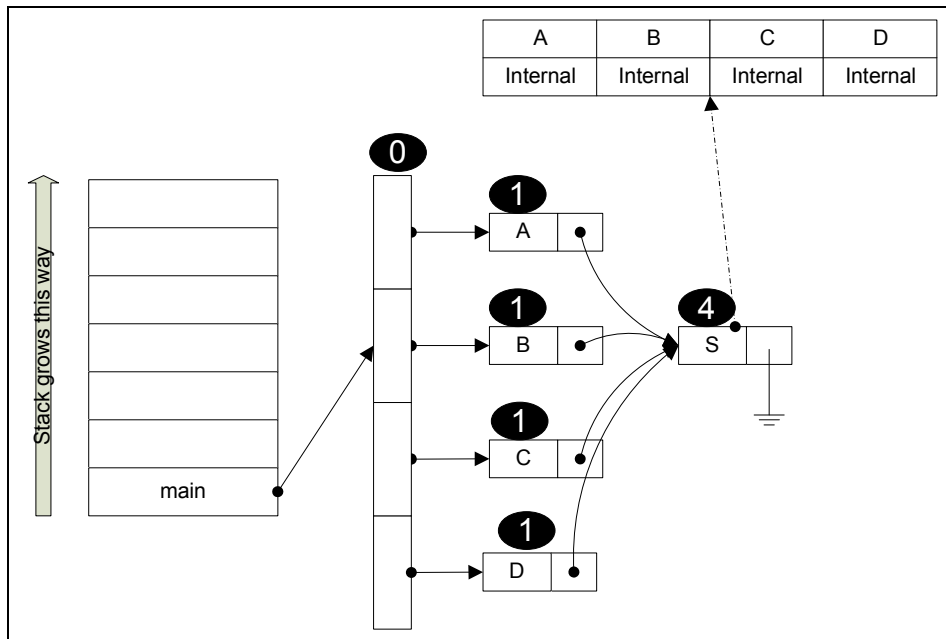


Figure 4.10: Computing external sources example initial state (Fig. 4.9 before line 5)

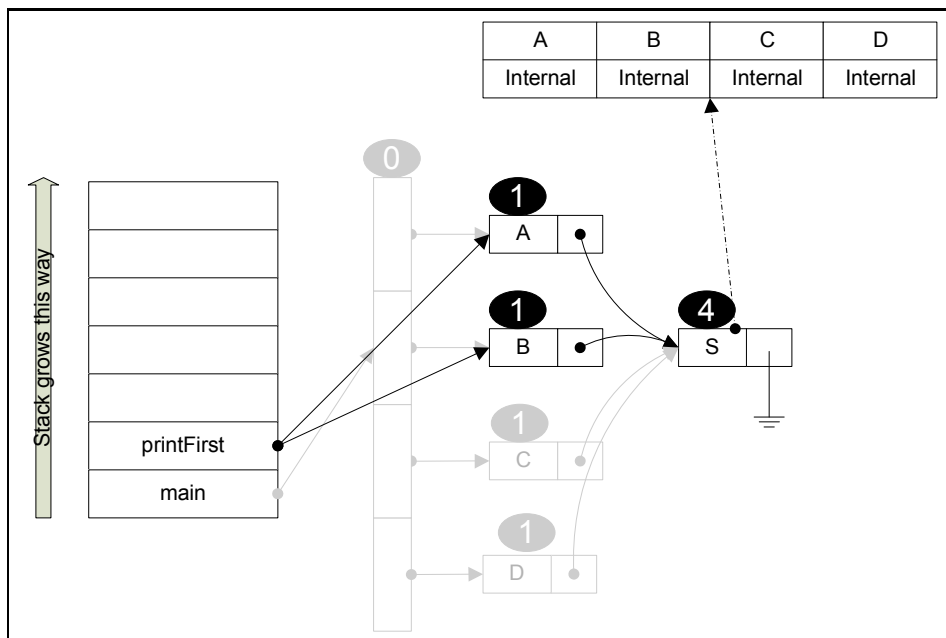
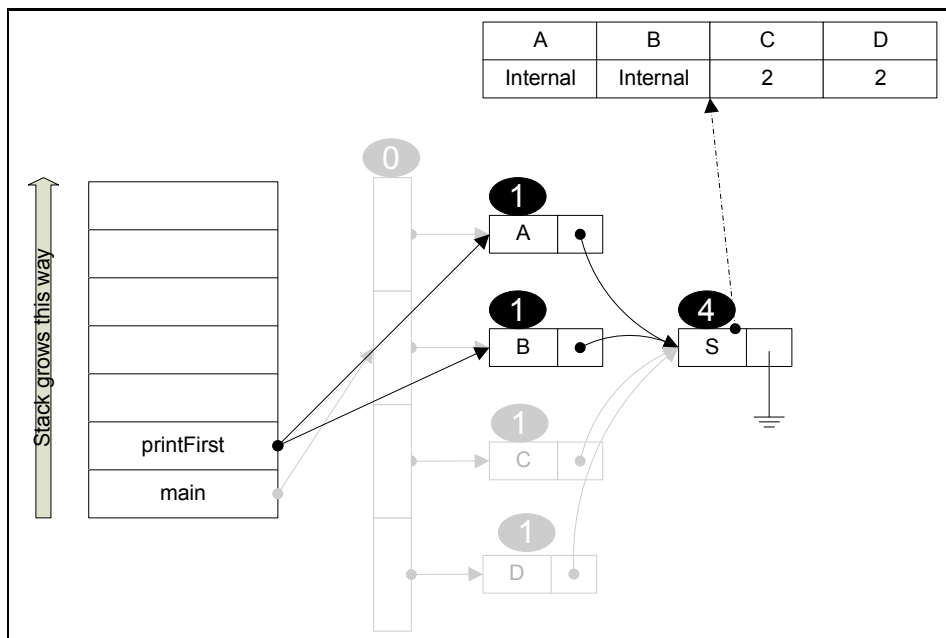
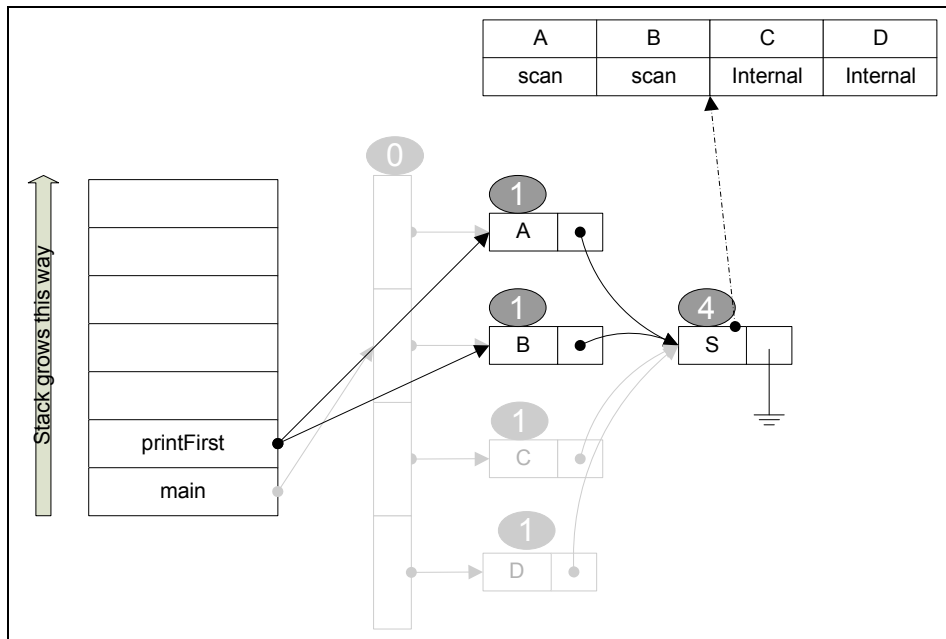


Figure 4.11: Computing external sources example in call to printFirst (Fig. 4.9 before line 7)





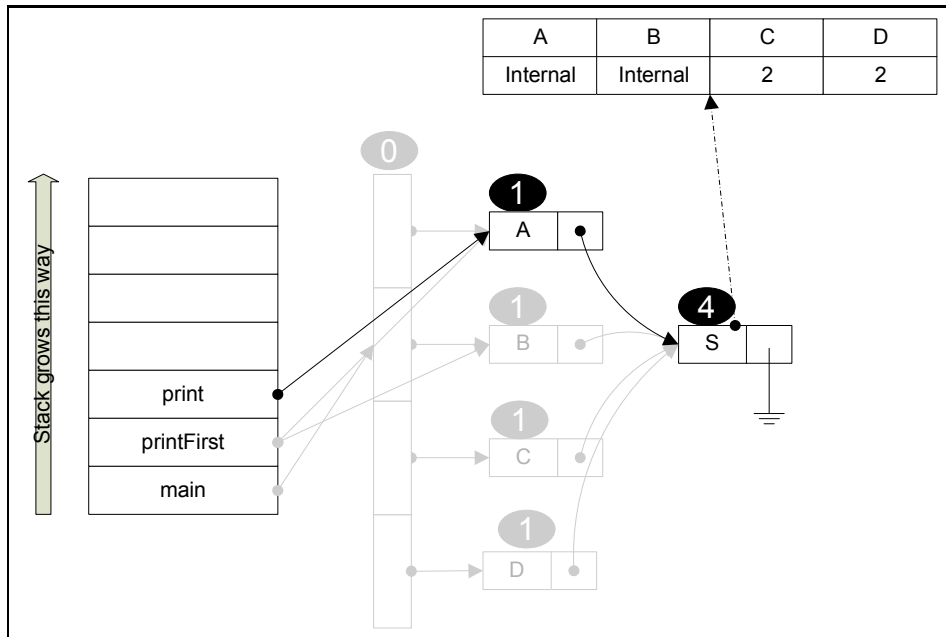


Figure 4.14: Computing external sources example in call to print (Fig. 4.9 before line 8)

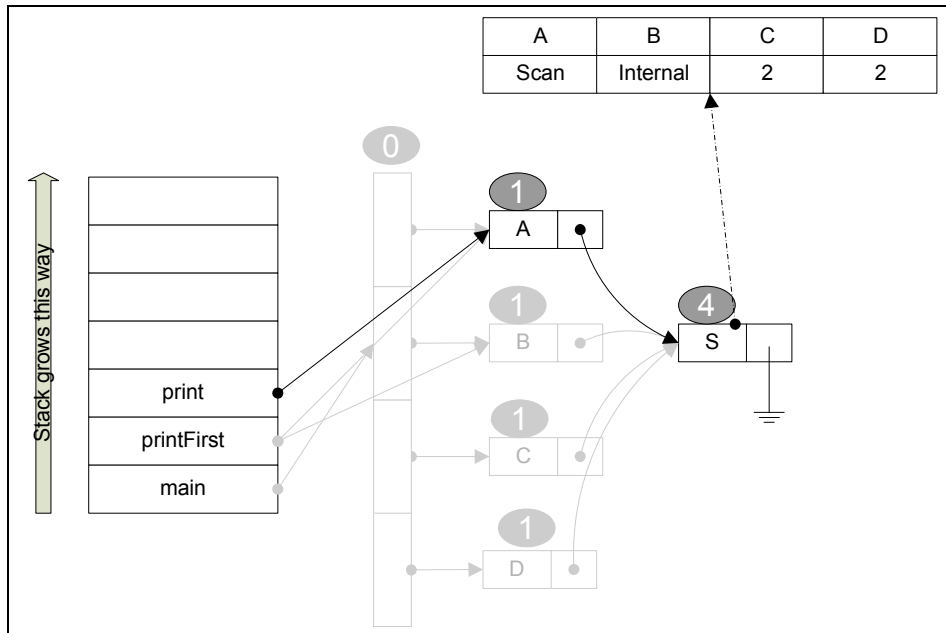


Figure 4.15: Computing external sources example in call to print (Fig. 4.9 before line 8) after MarkScanInternalRoots

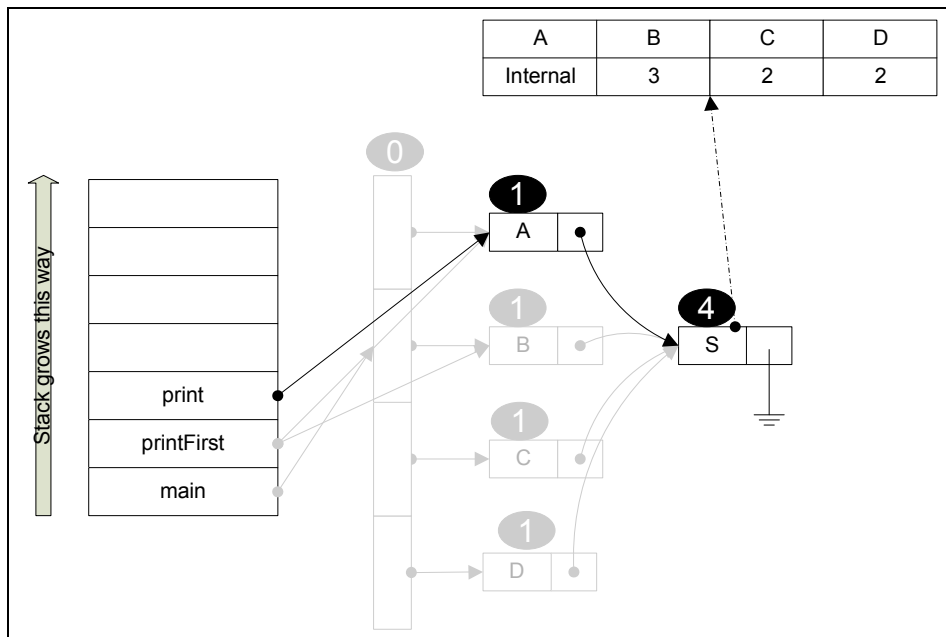


Figure 4.16: Computing external sources example in call to print (Fig. 4.9 before line 8) after MarkExternalRoots

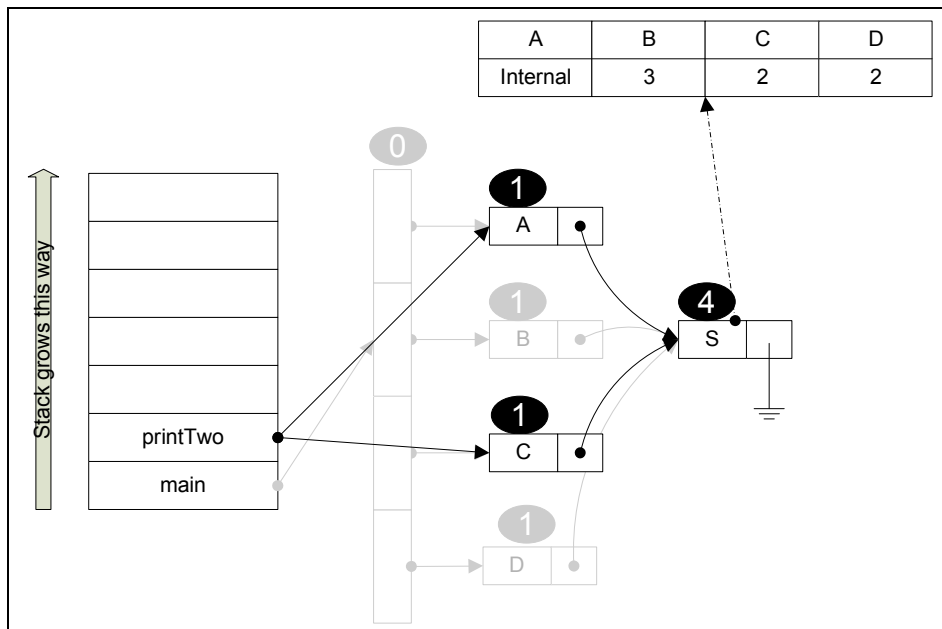


Figure 4.17: Computing external sources example in call to printTwo (Fig. 4.9 before line 9)

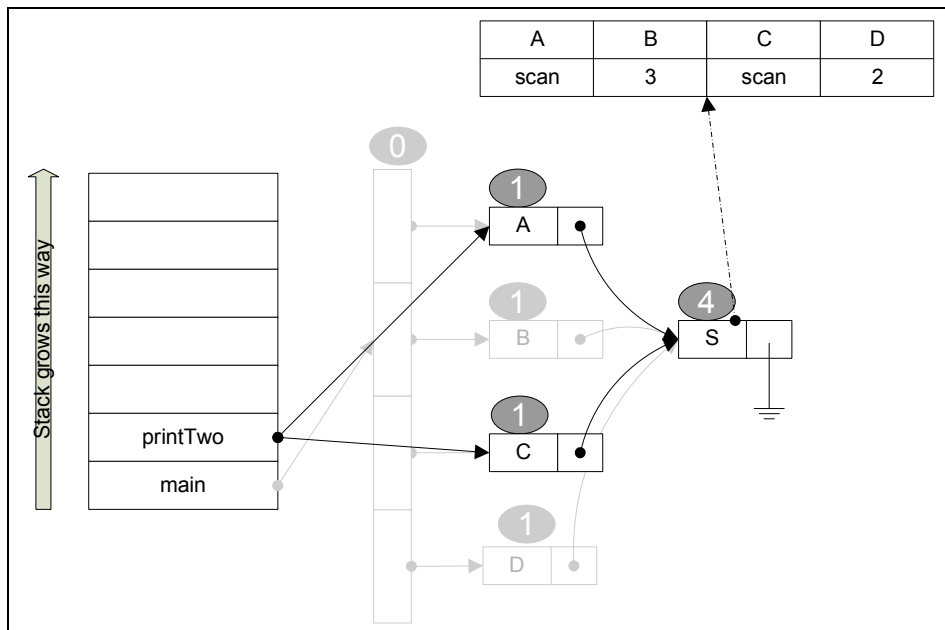


Figure 4.18: Computing external sources example in call to printTwo (Fig. 4.9 before line 9) after MarkScanInternalRoots

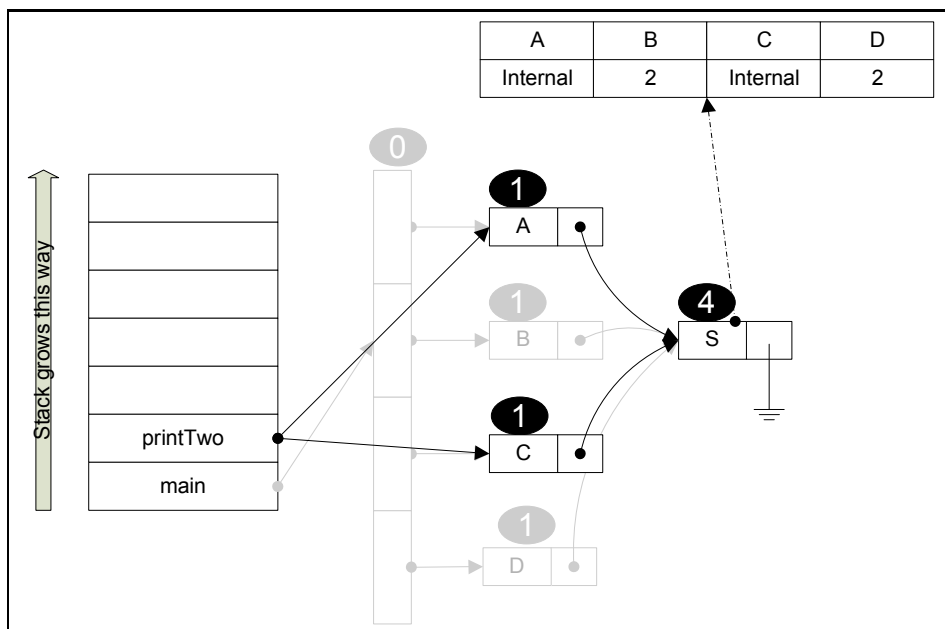


Figure 4.19: Computing external sources example in call to printTwo (Fig. 4.9 before line 9) after MarkExternalRoots

## Chapter 5

# Early detection of class invariant violations

This chapter explains the design by contract sharing problem and shows an early detection algorithm of class invariants violation.

### 5.1 Design by contract

Also known as “Programming by Contract”.

A major component of quality in software is reliability: a system’s ability to perform its job according to the specification (correctness) and to handle abnormal situations (robustness). Put more simply, reliability is the absence of bugs. In order to guarantee reliability, a systematic approach to specifying and implementing object-oriented software elements and their relations in a software system is required.

The central idea of *Design by Contract* is that software entities have obligations to other entities based upon formalized rules between them. A functional specification, or ‘contract’, is created for each module in the system before and during its implementation. Program execution is then viewed as the interaction between the various modules as bound by these contracts.

In general, routines have explicit *preconditions* that the caller must satisfy before calling the routine, and explicit *postconditions* that describe the conditions that the routine will guarantee to be true after the routine finishes. Thus, a contract takes the following general form: “If you, the caller, set up certain preconditions, then I will establish certain other results when I return to you. If you violate the preconditions, then I promise nothing.” Each module’s implementation can then be written assuming the correctness of the modules it uses (its subcontractors), as long as it satisfies their preconditions.

Contracts are also made for each class, ensuring the class is in a valid state. A *class invariant*, or *invariant*, is a set of conditions used to constrain objects of a class. Methods of the class should preserve the invariant. Class invariants are

established during construction and constantly maintained between calls to public methods. Temporary breaking of class invariance between private method calls is possible, although not encouraged.

(The text is based on [36, 35, 32].)

### 5.1.1 The sharing problem

The concept of invariants, as presented earlier, and object references are apparently two unrelated programming tools. Combining them together may result in undesirable behavior. The problem is caused by dynamic aliasing.

If  $x$  and  $y$  are of reference types and  $y$  is not void, the assignment  $x = y$  causes  $x$  and  $y$  to be attached to the same object. This is called *dynamic aliasing* or *aliasing*. The consequence of this assignment is that modifying the object through  $x$  affects any access through  $y$  too.

Therefore dynamic aliasing prevents checking the correctness of a class on the basis of that class alone. Object  $A$ 's attributes may be modified by an operation on another object,  $B$ . During this modification,  $A$ 's invariants are not tested and may be violated, because the modified object is  $B$ .

(The text is based on [22].)

**Example 5.1.1** *Fig. 5.1 and Fig. 5.2 show an example<sup>1</sup> of the sharing problem as presented in [22] (Class invariants and reference semantics, pages 403-406).*

*Class A has a reference to class B named forward (line 1). Class B has a reference to class A called backward (line 10). A has a method called attach (line 4), which assigns parameter `_b1` to forward (line 5) and calls B's attach (line 11) on `_b1` with itself as a parameter (line 7). B has a method called attach (line 11) which assigns its parameter, `_a1`, to backward (line 12). Unlike A's attach (line 4), it doesn't make a call to A's method. Class A has an invariant which requires forward to be empty (null, line 2) or to point to an object which points back to itself (line 3). This is carried out by A's forward pointing to a B instance whose backward points back to A.*

*The main method (line 13) creates two class instances, one of type A, referenced by `a1` (line 14), and one of type B, referenced by `b1` (line 15). Calling `a1.attach` (line 16) automatically creates two references, one for each formal parameter. They are named `this` and `_b1`. Their scope is the method so they will cease to exist when the method returns. Nevertheless, assigning `_b1` to forward (line 5) creates a new reference shared with `b1`. Sending `this` to `_b1.attach` creates another new reference by assigning `_a1` to backward (line 12). This reference is shared with `a1`. In this call the class invariant is checked and found to be true, as `forward.backward` does point to the instance owning forward, `a1`. The next call from the main (line 17) invalidates this invariant by removing the shared instance of A by `b1` (line 12). Despite that, the invariant is part of class A and as such is not tested in class B. Next, when a completely unrelated method of class*

---

<sup>1</sup>The example is written in Java and the class invariants are implemented using JML ([18]).

A, `doSomethingElse` (line 8), is called on instance `a1` (line 18), the class invariants are tested and found not to hold true. The JML result is seen in figure Fig. 5.3.

The program ends with a violated class invariant, but without any information as to where this violation has happened. On the next section we present our solution to help pinpoint where this kind of violations occurs.

**Example 5.1.2** This example shows the importance of early detection of class invariants violation. In this example, as opposed to the previous example, Example 5.1.1, the objects tested for class invariants do not have a reference to each other. Therefore verifying the other object's invariants is harder.

Fig. 5.4 shows two objects, `List 1` and `List 2`. Both objects are of the same class, `List`. The class has two fields: `head`, a reference to a list composed of a single linked list nodes and `size`, a counter indicating the size of the list. In Fig. 5.4 two rectangles composing each `List` class: the upper rectangle is `head` and the lower rectangle is `size`. Both lists have a `size` value of 3. The `List` class invariant verifies that `size` and the actual length of the list is the same, thus assuring the consistency of the object's state. Verifying the invariant is performed by traversing the list from `head`, counting the nodes and comparing the result to `size`.

The program creates two lists, `List 1` and `List 2`, with a common tail node, `n5`. `List 1` is made out of the nodes `n1`, `n2` and `n5`, in this order. `List 2` is made out of the nodes `n3`, `n4` and `n5`, in this order. Next, the program runs a reverse procedure, which is part of the `List` class, on `List 2`. The reverse procedure traverses the list and reverses the references it encounters. Then `head` is updated to reference the former tail of the list. The resulting list is the reverse of the input list. The result of executing reverse on `List 2` is shown in Fig. 5.5. `List 2` is of size 3 and contains three nodes, `n5`, `n4` and `n3`, in this order. The reverse side effect is that `List 1` now contains 5 nodes, `n1`, `n2`, `n5`, `n4` and `n3`, in this order.

The reverse procedure tests for `List` class invariants when it ends. `List 2` invariants are tested as the procedure ran on it. The class invariant is found to be correct as `List 2` is of size 3 and contains three nodes. Therefore the program seems to be in a consistent condition. Nevertheless, the next time the program will execute any instance method of `List 1`, the class invariants test will fail. This is because `List 1` now contains 5 nodes, while its `size` field is of value 3. `List 1` is not in a consistent state anymore. Unfortunately, this can be detected much later than when the violation actually occurred.

The following suggested solution detects the class invariants violation in the method where it occurred.

```
public class A {
1:     B forward = null;

2:     /*@ invariant forward == null ||
3:         @     forward.backward == this;
         @*/

4:     public void attach(B _b1)
        {
5:         forward = _b1;

6:         if(_b1 != null)
            {
7:             _b1.attach(this);
            }
        }

8:     public void doSomethingElse()
        {
9:         System.out.println("Doing something else");
        }
    }

    public class B {
10:        A backward = null;

11:        public void attach(A _a1)
            {
12:            backward = _a1;
            }
        }
    }
```

Figure 5.1: Classical sharing example

```

public class Main {

13:   public static void main(String[] args)
    {
14:       A a1 = new A();
15:       B b1 = new B();

16:       a1.attach(b1);
17:       b1.attach(null);
18:       a1.doSomethingElse();
    }
}

```

Figure 5.2: Classical sharing example continued

```

Exception in thread "main"
org.jmlspecs.jmlrac.runtime.JMLInvariantError: by method
A.doSomethingElse@pre<File "A.java", line 32, character 15>
regarding specifications at File "A.java", line 13, character 34
when
    'forward' is B@1fee6fc
    'this' is A@1eed786
at A.checkInv$instance$A(A.java:126)
at A.doSomethingElse(A.java:465)
at meyer.main(meyer.java:17)

```

Figure 5.3: Classical sharing example class invariant violation

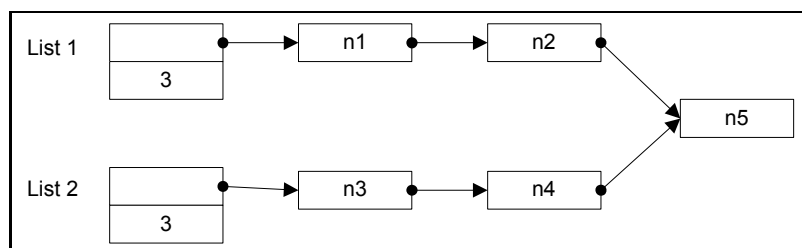


Figure 5.4: List tail sharing example



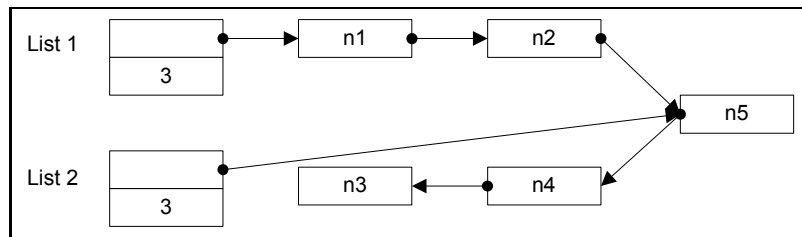


Figure 5.5: List tail sharing example after list reverse

## 5.2 Computing invalid class invariants

### 5.2.1 Using cutpoints

Cutpoints create a sharing between the local heap and the global heap. When a method modifies a cutpoint, or objects reachable from that cutpoint, it may invalidate class invariants of objects in the global heap that can reach the cutpoint. Furthermore, the method does not test them, as, in Example 5.1.2, sometimes the affected objects are not reachable at all by the method and as the method tests only the class invariants of the class it belongs to. Therefore cutpoints are a useful property for verifying the validity of class invariants of objects outside the local heap.

The cutpoints are used as the roots of a backward scan of the global heap. Each object scanned is tested for its class invariants. As a consequence the computation finds the class invariants violated by a modification in the local heap.

We believe testing for invalidated class invariants by the end of each method results in the best tradeoff between performing a long computation and providing enough information for locating invalidated invariants cause.

### 5.2.2 Holding cutpoints

The computation is carried out in two stages:

1. Detecting cutpoints
2. Using cutpoints to detect class invariants invalidations

Cutpoint detection is performed at the beginning of a method while class invariants violation computation is performed at the end of a method. During the method's execution parts of the local heap may become unreachable from the method's formal parameters. Therefore the detected cutpoints have to be kept until the method ends.

A naive solution is to use a list for this purpose. Because a method can call other methods during its execution, a list must be held for each method call. This is highly inefficient in space. Section 5.2.3 presents a space efficient solution.

### 5.2.3 The cutpoint list

The cutpoint list has to hold each cutpoint once and match the cutpoints with the method they have been detected in.

Once a local heap object becomes external, it will not become internal again until the method, in which it has become external, returns (Observation 2.1.3). Hence if this object is a cutpoint, then it is used by the class invariants violation computation from the method the cutpoint has appeared first until the method before the method where the object has become external. Therefore each cutpoint in the list has two stored values:

- Discovered stack depth ( $DSD$ )
- Maximum stack depth ( $MSD$ ) - The last stack depth in which this cutpoint was detected

$MSD \geq DSD$ . A cutpoint is removed from the list when  $MSD = DSD$ .

When detecting cutpoints, new cutpoints are added with an initial  $MSD$  value of the current stack depth; Cutpoints detected, that are already in the list, are updated by setting their  $MSD$  to the current stack depth. Cutpoints used by the class invariants computation, their  $MSD$  is decremented by one. This way the cutpoints with the highest  $MSD$  equal the current method's stack depth. Therefore the cutpoints that the class invariants violation computation uses are those with an  $MSD$  equals to the current stack depth.

**Observation 5.2.1** *The cutpoints list produces at each method's exit the exact cutpoints that have been detected at the method's entry.*

A simple optimization to prevent searching when matching cutpoints with the methods they have been detected in, is to hold the list sorted according to the  $MSD$ . Using a linked list, for example, makes this optimization easy. Sorting is performed by moving a rediscovered cutpoint entry to the head of the list. Updating the  $MSD$  while traversing the list for the computation (Section 5.3) guarantees that the list is left sorted.

### 5.2.4 Backward scan

In order to perform a backward scan, the computation has to be familiar with the refer-to objects of each object  $o$  in the heap. This information can be achieved, for example, by using source lists (Section 3.2.2). The scan is a depth first search of the global heap, starting at each cutpoint detected by the current method, and going backward.

As in the cutpoint detection computation (See Table 3.1) there has to be a way to limit the scan. A problem arises as there is only one scan each time and no way to clear a flag. This can be remedied by adding a second scan. Unfortunately each backward scan is time consuming (The cost is  $O(N + E)$ ). Another solution is to

add a counter to each object and increment it on each scan. This counter has to be large enough not to repeat itself too soon (The current implementation aborts the computation when the flag overflows).

### 5.3 The computation algorithm

The algorithm is explained using a simple linked list, which is kept sorted according to the *MSD*. The list entry fields appear in figure Fig. 5.6. The `discovered` field is the *DSD* and the `maximum` field is the *MSD*. The backward scan is handled by a counter flag, `scanFlag`, initially zero. `ScanFlag(T)` is the scanning flag at object *T*, initially zero. The scan flag ensures that an object's class invariants are not tested more than once, even if more than one cutpoint is reachable from this object (The current implementation aborts the computation if the flag overflows). The computation algorithm appears in figure Fig. 5.7 and Fig. 5.8.

**OnCutpointDetection(CP)** When a cutpoint *CP* is detected it is added to the cutpoint list.

**AddToList(CP,currentDepth)** If a cutpoint *CP* is not on the list, this procedure adds an entry to the cutpoint list and assigns the *DSD* to the current stack depth, `currentDepth`. The *MSD* is assigned the current stack depth whether the cutpoint is new or not.

**OnMethodExit(currentDepth)** Called when a method exits, normally or exceptionally. In order to start a new backward scan session the `scanFlag` is incremented. This procedure runs a backward scan only on the detected cutpoints for the current method. The *MSD* is maintained by decrementing its value for each cutpoint backward scanned. If the cutpoint's *DSD* is the current method's stack depth, `currentDepth`, then the cutpoint is removed from the list.

**BackScanStart(CP)** This procedure starts the backward scan from the cutpoint *CP*'s sources since the cutpoint itself is not scanned.

**BackScan(T)** Object *T*'s class invariants are tested. Then *T* is marked with the current `scanFlag` and its sources are backward scanned too.

**OnObjectRelease(Object)** This procedure removes the cutpoint from the list and does not perform a backward scan. The reason is that because a cutpoint is referenced by an object outside the local heap, it can not be released between its detection at the beginning of a method, and its usage as input by the class invariants computation, when the method exits. The exception is when all the objects referencing the cutpoint are already garbage. In this case, testing for class invariants violation is meaningless.

```

ListEntry
    Cutpoint cutpoint
    integer discovered
    integer maximum
    ListEntry next

```

Figure 5.6: Cutpoints list entry

```

OnCutpointDetection(CP)
    AddToList(CP,method current stack depth)

AddToList(CP,currentDepth)
    if(CP is in list)
        le = ListEntry for CP
        move le to list head
    else
        le = new ListEntry
        le.cutpoint = CP
        le.discovered = currentDepth
        add le to list head
    le.maximum = currentDepth

```

Figure 5.7: Class invariants violation computation using cutpoints

**TestInvariants(T)** Tests an object class invariants for a violation and reports (We assume there is a way for the computation to tests an object’s class invariants).

**Example 5.3.1** *This example demonstrates Observation 5.2.1.*

*Fig. 5.9 shows the cutpoint list (Section 5.2.3) contents and usage along three method calls, method\_a, method\_b and method\_c. The leftmost column shows the cutpoints detected at the beginning of each method and the cutpoints used as roots for the backward scan (Section 5.2.4) when a method exits. The next column to the right shows the current stack. The last column shows the contents of the cutpoint list. Each item on the list has three fields, from left to right: An object identifier, MSD and DSD.*

*method\_a is called and cutpoints A and B are detected. They are added to the list with the current stack depth, 1, as their MSD and DSD. When method\_b is called, cutpoints A, C and D are detected. Cutpoint A is already on the list. Therefore A is forwarded to the list’s head and its MSD is updated to the current stack depth. C and D are new to the list and are added with their MSD and DSD values equal to the current stack depth, 2. At method\_c the cutpoints detected*

```

OnMethodExit(currentDepth)
    scanFlag = scanFlag + 1
    le = list head
    while(le != null)
        if(le.maximum < currentDepth)
            return
        BackScanStart(le.cutpoint)
        if(le.discovered == currentDepth)
            remove le from list
        else
            le.maximum = le.maximum - 1
        le = le.next

BackScanStart(CP)
    for each S in source_list(CP)
        BackScan(S)

BackScan(T)
    if (ScanFlag(T) != scanFlag)
        TestInvariants(T)
        ScanFlag(T) = scanFlag
    for each S in source_list(T)
        BackScan(S)

OnObjectRelease(Object)
    if(Object is in list)
        le = ListEntry for Object
        remove le from list

TestInvariants(T)
    Test T class invariants
    Report if invalid

```

Figure 5.8: Class invariants violation computation using cutpoints continued

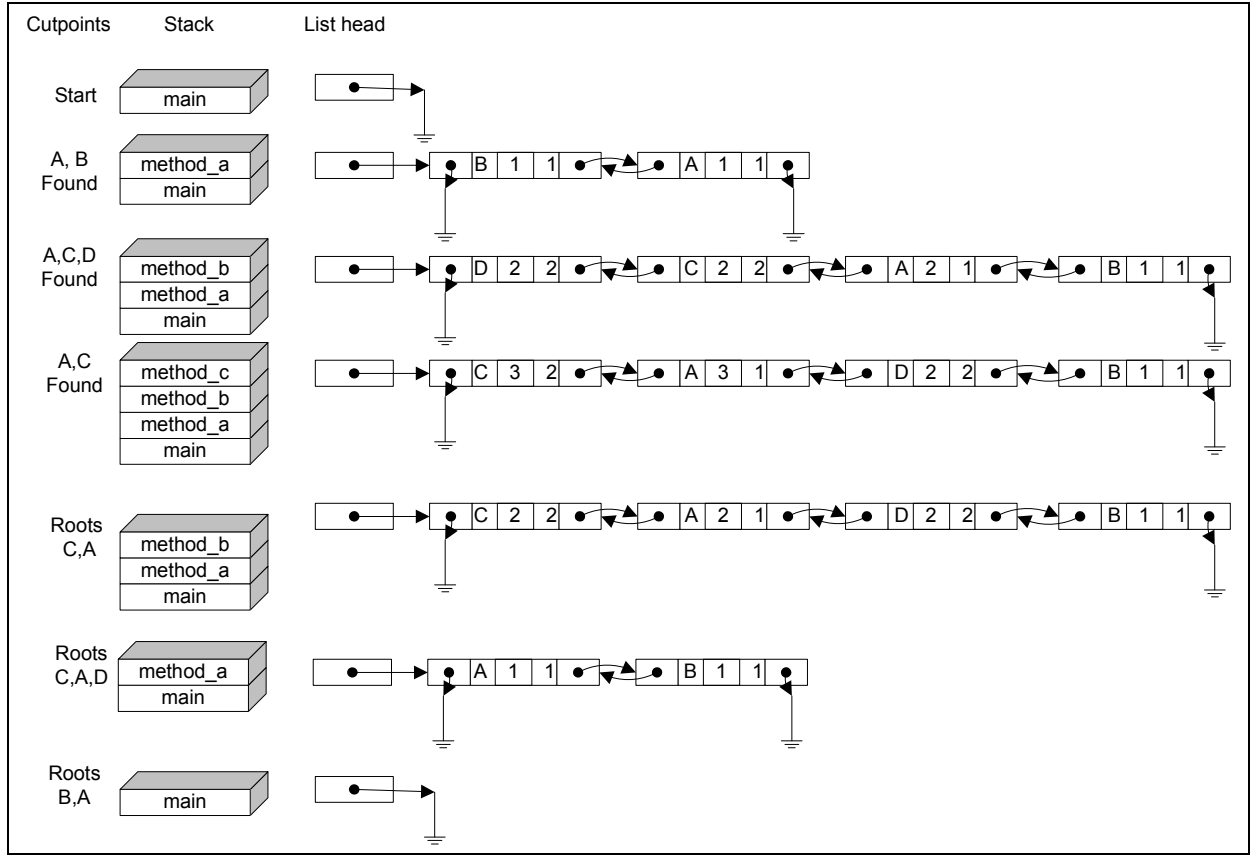


Figure 5.9: Cutpoint list example

are objects *A* and *C*. Both already exist on the cutpoint list. Therefore *A* and *C* are forwarded to the list head and their MSD is updated to the current stack depth, 3.

When a method exits, the class invariants violation computation runs. The roots for the computation are the cutpoints at the beginning of the cutpoint list, whose MSD equals to the current stack depth. For `method_c` these are cutpoints *A* and *C*. The MSD is decremented by one for each cutpoint used as root. When `method_b` exits, the cutpoints with the current stack depth are *A*, *C* and *D*. While traversing the cutpoints list, *C* and *D*'s MSD is found to be equal to their DSD. Hence *C* and *D* have been detected at these stack depth and are not needed anymore. Therefore *C* and *D* are removed from the cutpoint list. Cutpoint *A*'s MSD is decremented. Once `method_a` exits, cutpoint *A* and *B* are the roots for the computation. *A* and *B*'s DSD equals to their MSD and they are removed.

# Chapter 6

## Results

### 6.1 Motivation

The motivations for investigating the results are:

- Finding common traits for cutpoints in programs
- Pinpointing highly shared data patterns, which also might be design bugs

### 6.2 Measurements

All the measurements here are done for cutpoints referenced from the heap only and neither from the stack nor from static variables. More about result processing in Appendix B.

#### 6.2.1 Top cutpoint producing methods

First, the maximum number of cutpoints per method invocation is measured for all methods. The top ten most cutpoint causing methods are inspected further. A list of causing cutpoints is measured for each method in the top ten list. With this information, each program is examined in order to find a reason for the cutpoints.

#### 6.2.2 Well-known classes effect

A list is used to separate cutpoints of specific classes. The list is characterized by the following properties:

- Immutability
- We assume that highly shared
- Less interesting for understanding a program's sharing and therefore can be processed separately

<b>Class identifier</b>
java.lang.String
java.lang.Integer
java.lang.Boolean
java.lang.Byte
java.lang.Character
java.lang.Double
java.lang.Float
java.lang.Long
java.lang.Short

Table 6.1: The well-known classes list

The well-known classes list appears in Table 6.1.

The results of cutpoint detection are separated to two different parts according to the well-known classes list: One part contains the results where cutpoints are of the classes in the list and the second part contains the results where cutpoints are not of the classes in the list.

For each program the total number of cutpoints of classes in the well known classes list are measured throughout the program's execution. The purpose is to get an indication whether the Java language designers' decision to make these classes immutable and shared is justified.

### 6.2.3 Methods' maximum cutpoints disparity

The purpose of this measurement is to provide an initial view of the amount of cutpoints in a program. Each method is placed in a base two logarithmic scale according to the maximum number of cutpoints it had during the program's execution. If a method is placed in the location of value  $x$ , it means the method had between  $2^{(\log_2 x)-1} + 1$  and  $x$ , inclusive, maximum number of cutpoints during the program's execution. For example, if a method is placed at the value of 256, it means the maximum number of cutpoints this method had during the program's execution is between 129 and 256, inclusive. Subsequently the percentage of methods, from the total number of methods, is calculated for each entry in the scale.

The measurements are conducted once for all classes of cutpoints and once for cutpoints not belonging to the well-known classes list. This is done in order to see the effect the well-known classes have on cutpoints in methods.



## 6.3 The benchmarks

### 6.3.1 Soot: a Java optimization framework

Soot ([17]) is a Java optimization framework. It provides four intermediate representations for analyzing and transforming Java bytecode: Baf: a streamlined representation of bytecode which is simple to manipulate. Jimple: a typed 3-address intermediate representation suitable for optimization. Shimple: an SSA variation of Jimple. Grimp: an aggregated version of Jimple suitable for decompilation and code inspection. Soot can be used as a stand alone tool to optimize or inspect class files, as well as a framework to develop optimizations or transformations on Java bytecode.

### 6.3.2 The Kawa language framework

Kawa ([4]) is:

A framework written in Java for implementing high-level and dynamic languages, compiling them into Java bytecodes.

An implementation of Scheme, which is in the Lisp family of programming languages. Kawa is a featureful dialect in its own right, and additionally provides very useful integration with Java. It can be used as a “scripting language”, but includes a compiler and all the benefits of a “real” programming language, including optional static typing.

Implementations of other programming languages, including XQuery (Qexo) and Emacs Lisp (JEmacs).

### 6.3.3 SPEC JVM98 benchmarks

JVM98 ([7]) features:

- Measures performance of Java Virtual Machines
- Applicable to networked and standalone Java client computers, either with disk (e.g., PC, workstation) or without disk (e.g., network computer) executing programs in an ordinary Java platform environment.
- Requires Java Virtual Machine compatible with JDK 1.1 API, or later

### 6.3.4 TVLA: 3-valued logic analysis engine

TVLA ([20]) is an evolving research vehicle for abstract interpretation, featuring:

- A powerful language for expressing concrete semantics
- Automatic generation of abstract interpreters from concrete semantics
- Tunable abstractions
- Naturally suited for checking properties of heap allocated data

Package	Class	Method	Maximum value
soot/util/	ArrayNumberer	add (Ljava/lang/Object;)V	1523
soot/util/	HashChain\$Link	unlinkSelf ()V	1464
soot/util/	HashChain\$Link	getItem ()Ljava/lang/Object;	1464
soot/util/	HashChain\$Link	bind (Lsoot/util/HashChain\$Link;... ...Lsoot/util/HashChain\$Link;)V	1464
soot/util/	HashChain\$Link	setPrevious (Lsoot/util/HashChain\$Link;)V	1464
soot/util/	HashChain\$LinkIterator	next ()Ljava/lang/Object;	1464
soot/util/	HashChain\$Link	setNext (Lsoot/util/HashChain\$Link;)V	1464
soot/util/	HashChain\$Link	getNext ()Lsoot/util/HashChain\$Link;	1464
soot/util/	HashChain	access\$300 (Lsoot/util/HashChain;)J	1464
soot/util/	HashChain	size ()I	1464

Table 6.2: Soot run on null example top ten cutpoint causing methods

## 6.4 Results

### 6.4.1 Shared immutable objects

An example of the top methods appears in Table 6.2. In the table appear the methods with the maximum number of cutpoints throughout Soot’s execution on null example input.

By looking at the code of two of the benchmarks, Soot and TVLA, according to their top cutpoint producing methods, a common property is discovered. Both programs load their input into a data structure in memory. This data carries out two properties:

- The data is immutable
- The data is shared, as it either goes through more than one processing, requiring different views of it or, to improve its read time, it can be accessed by more than one manner

As a result, when accessing the shared data structure through one of its access objects, the other manners of access cause the appearance of many cutpoints.

Soot reads Java bytecode files and loads them into memory by creating a description of the class structure. The various objects in the class description structure are also accessible by a number. For this purpose, another object holds a mapping from numbers to objects (*soot.util.ArrayNumberer*). When Soot accesses the class structure, the class responsible for numbering causes numerous cutpoints.

TVLA loads a list of formulas and stores them in memory. Several lists of constraints on the formula are loaded and described by referencing the formulas. Whenever a constraint is processed, the other constraints cause numerous cutpoints on the formulas.

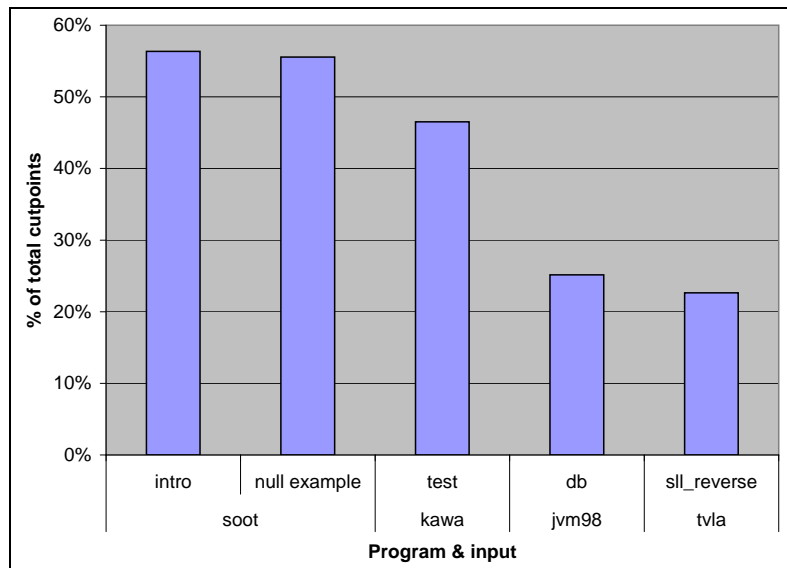


Figure 6.1: String percentage out of the total cutpoints

## 6.4.2 String

The results in Fig. 6.1 show that *java.lang.String* is a major player in causing cutpoints. String's effect is much more apparent in the first three bars. Soot and Kawa load Java classes into memory. The classes are loaded by reading files, which is performed by reading many strings. As seen from the results, strings are highly shared. From this we conclude that making *java.lang.String* immutable and shared was a good design decision.

## 6.4.3 Methods' maximum cutpoints disparity

The maximum cutpoints disparity graph shown in Fig. 6.2 illustrates how much sharing exists in each program. Other than Kawa, the programs have more than half their methods with less than 128 cutpoints per method call.

When looking at the maximum disparity graph without the well-known class cutpoints in Fig. 6.3 and comparing it to the previous graph, the well-known class cutpoints indeed contribute a fair amount. As seen in Fig. 6.1 this contribution is mostly due to *java.lang.String*.

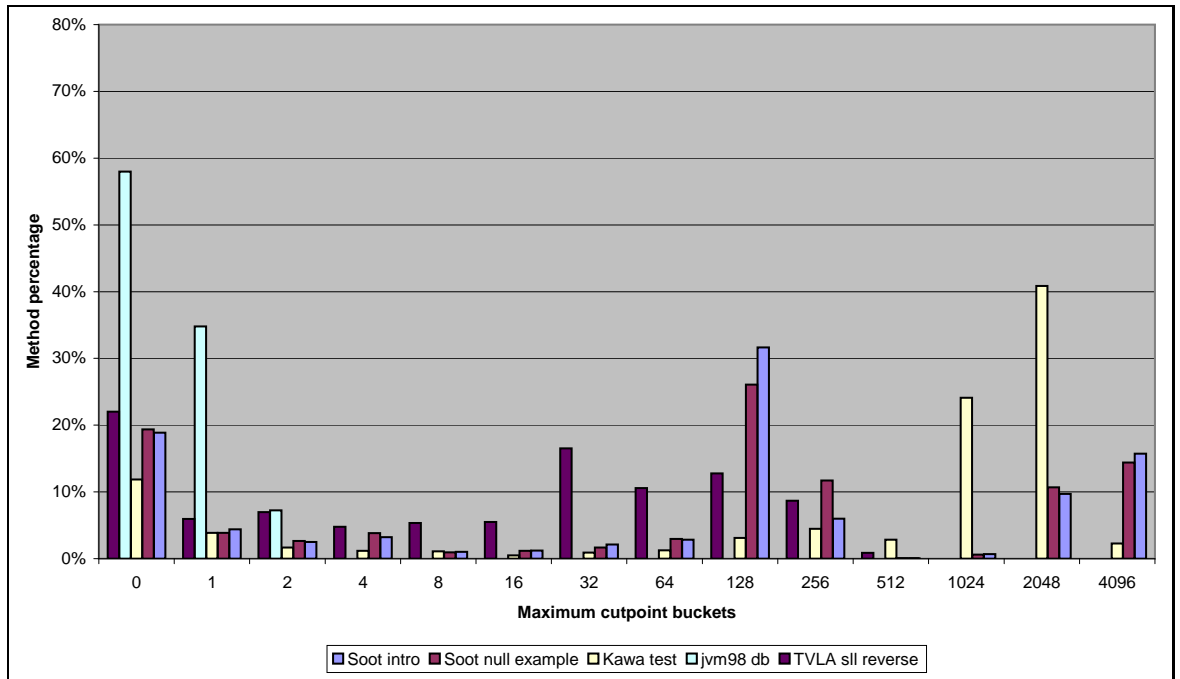


Figure 6.2: Disparity of method maximum cutpoints in total

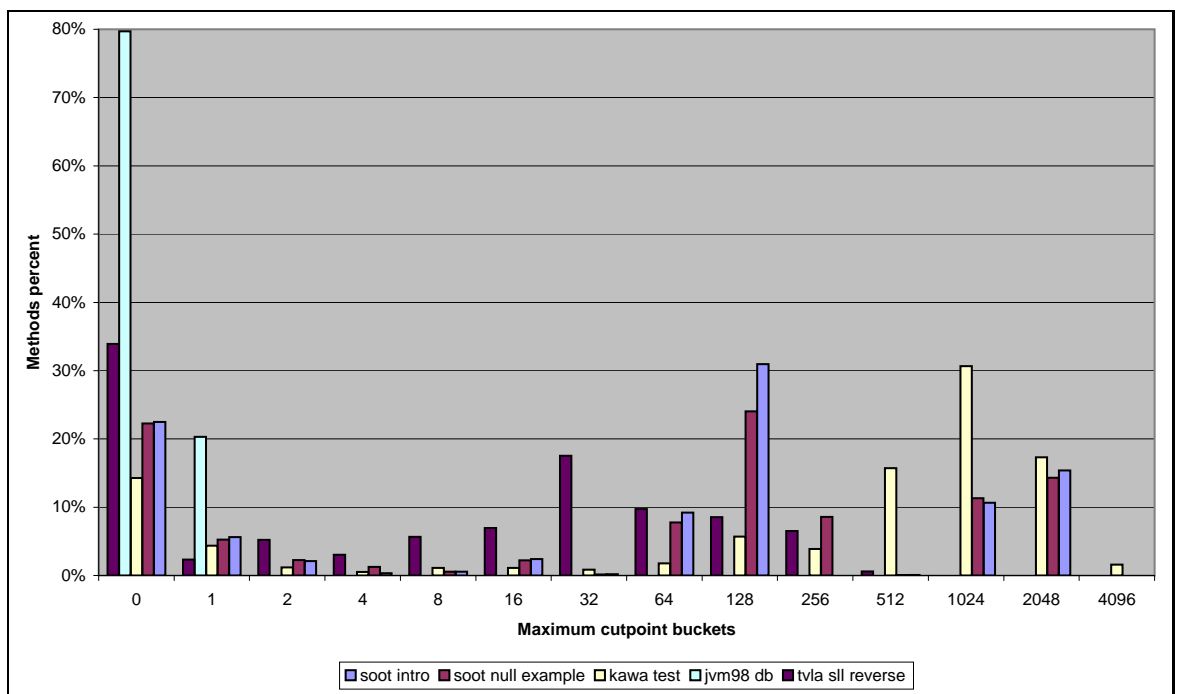


Figure 6.3: Disparity of method maximum unknown cutpoints in total

## Chapter 7

# Related work

There are several works and tools dealing with heap profiling. Other related works use cutpoints with static shape analysis. (As far as we know, there is no existing work dealing with dynamic computation of cutpoints)

**Interprocedural shape analysis** The importance of cutpoints was first identified in works regarding static interprocedural shape analysis. In [29], Rinetzky et. al. developed compile-time algorithms for automatically verifying properties of imperative programs that manipulate dynamically allocated storage. Cutpoints are used in the analysis to characterize a procedure's behavior. The work in [30] takes advantage of the absence of cutpoints to develop a procedural abstraction, which is used in a framework for interprocedural shape analysis. An interprocedural shape analysis that supports a bounded number of cutpoints in the local heap is presented in [13] by Gotsman et. al..

**Heap profilers** Modern development languages use dynamically allocated memory extensively, using complex data structures. Heap profiling is used to isolate performance problems involving memory usage and inefficient code.

**General information** General tools provide statistical information about the heap during and at the end of a program's execution. HPROF ([27]) is part of Sun's JVM library and provides heap and garbage collection statistics. HAT ([11]) is a tool for analyzing the results of HPROF. More sophisticated tools provide information and specific advice on different ways to improve the profiled program. For example, on memory corruption and leaks, application performance bottlenecks and code coverage. OptimizeIt<sup>TM</sup> ([5]), JProbe<sup>TM</sup> Memory Debugger ([33]) and Rational PurifyPlus<sup>TM</sup> ([6]) are some well known commercial tools. More heap profilers are The NetBeans Profiler project [25] and Cougaar memory profiler [8].

**Garbage collection behavior** The importance of garbage collection performance has led to the creation of tools that investigate its behavior. Sun

provides a Garbage Collector Spy Tool ([23]), which visualizes a large range of memory systems. Shaham et. al. ([31]) developed a tool, which measures the difference between the actual collection time and the actual object death time. The output of the tool is used to direct the rewriting of an application's source code in a way that allows more timely garbage collection of objects, thus saving space. Hertz et. al. ([15], [14]) present a theoretical framework for analyzing garbage collection and a tracing algorithm (called "Merlin"), which determines the exact point an object in the heap has become unreachable. Merlin is implemented as part of the Jikes RVM.

**Object ownership** The general heap profiling tools usually provide manual browsing and flat summaries, making it hard to understand today's programs. Mitchell ([26]) creates a hierarchical summary of the heap using object's ownership. Jackson et. al. ([28]) facilitate program understanding by revealing objects ownership and sharing using a visualization tool.

The sharing problem (Section 5.1.1) is tackled by Barnett et. al. in [24] and [3]. Barnett et. al. present a *friendship system*. Friendship describes a formal protocol for a *granting class* to grant a friend class permission to express its invariant over fields in the granting class. The protocol permits the safe update of the granter's fields without violating the friend's invariant. Rustan et. al. ([19]) deal with static class invariants, which describe the consistency of static fields. Static fields usually hold data that is shared among objects. The authors present a methodology for specifying and verifying static class invariants in object-oriented programs.

## Chapter 8

# Future Work

### 8.1 Suggestions for future work

We believe cutpoints may be used as an indicator for program behavior. The following topics should be investigated to find out whether a relation exists.

- **Confinement** - Cutpoints indicate externally shared data. Classes should be written in such a way that their data is confined and handled by the class or by its package only. Therefore cutpoints may indicate whether classes actually confine their data.
  - **Method access modifiers** - Private, protected and package methods share more data than public to public method calls, because they are internal to the class. Therefore these methods should have more cutpoints than public to public method calls.
  - **Cross Package** - Packages are independent execution modules and thus should confine their own data. As a result, cross package method calls should have more cutpoints than inner package method calls.
  - **Source Origin** - Looking at the type of the sources for cutpoints can help find explanations for cutpoints. Comparing the cutpoint's package and its sources' package shows how much data is confined within packages or how much of it is shared.
- **JDK** - Focus on the JDK as it is used by all Java programs. For this reason results discovered here have a large impact.

The cutpoint detection computation runs on the local heap. Taking advantage of this can provide more information, such as:

- The amount of acyclic objects (Section 3.4.1) in the local heap.
- Information that may help find common limits for cutpoints, such as distance from the formal parameters, stack depth when detected.

- Properties of the local heap, such as
  - Dimensions, for example maximum length, number of objects
  - Usability - How much of the local heap is actually accessed
  - Internal sharing - How many objects are shared in the local heap

Works [28] and [26] present heap profiling results hierarchically using object ownership. An integration of these works with the computations presented here should be examined in order to provide more interesting results.

Finding live or dead cutpoints is performed only for heap references. The process can be extended to support stack and static fields references.

The Jikes RVM has an implementation of the *Merlin* algorithm ([15]). Merlin is a trace generation algorithm, which determines when an object becomes unreachable. As a result, using the liveness computation with Merlin for detecting dead cutpoint fields should be examined.

Using Merlin and a heap modeler should be examined as another platform for detecting cutpoints.

### 8.1.1 Prototype

The following are additions and modifications to the prototype.

- Optimizing for execution speed.
- Creating a relation between the cutpoint detection computation and the liveness computation such that only live cutpoints are reported.
- Since the prototype was written, new versions of the Jikes RVM have been released. The MMTk, the memory management module, has been redesigned. Therefore the prototype should be adapted to the new design.

## 8.2 Limitations

The current cutpoints detection algorithm is single-threaded, hence it does not handle a large group of programs.

The use of Address type (Section A.2.2) forces the use of non-moving garbage collectors only. In order to support all garbage collectors, the following should be done:

- Adding an option to perform reference counting.
- Supporting moving garbage collectors.

The prototype has a large runtime overhead, as it runs at least on each method entry. Nevertheless, some of the overhead can be reduced. Currently the main reason for the slowdown is the results printing overhead. Improvements have already



been made, such as displaying a summary of the number of cutpoint and types for each method call, instead of typing each cutpoint separately. These improvements have resulted in lower overhead, but further work should be done.

# Bibliography

- [1] S.M. Blackburn M. Butrico A. Cocchi P Cheng J. Dolby S. Fink D. Grove M. Hind K.S. McKinley M. Mergen J.E.B. Moss T. Ngo V. Sarkar B. Alpern, S. Augart and M. Trapp. The jikes research virtual machine project: Buliding an open-source research community. *IBM Systems Journal*, 44(2):399–417, 2005.
- [2] D.F. Bacon and V.T. Rajan. Concurrent cycle collection in reference counted systems. In *Proceedings of the Fifteenth European Conference on Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 207–235, Budapest, Hungary, June 2001. Springer-Verlag.
- [3] M. Barnett and D.A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *MPC*, pages 54–84, 2004.
- [4] P. Bothner. The kawa language framework. <http://www.gnu.org/software/kawa/>.
- [5] Borland Software Corporation. Optimizeit<sup>TM</sup> enterprise suite, 2006.
- [6] IBM Corporation. Rational purifyplus, 2006.
- [7] Standard Performance Evaluation Corporation. Spec jvm98 benchmarks. <http://www.spec.org/jvm98/>.
- [8] Cougaar. Cougaar memory profiler, 2006.
- [9] L. P. Deutsch and D. G. Bobrow. An efficient, incremental, automatic garbage collector. *Commun. ACM*, 19(9):522–526, 1976.
- [10] Jikes RVM development team. *The Jikes<sup>TM</sup> Research Virtual Machine User's Guide*, 2.3.5 edition, 2005.
- [11] B. Foote. Hat: The java heap analysis tool, 2006.
- [12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley Professional Computing Series, 2005.
- [13] A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In *SAS*, 2006.

- [14] M. Hertz, S.M. Blackburn, J.E.B. Moss, K.S. McKinley, and D. Stefanović. Generating object lifetime traces with merlin. *ACM Trans. Program. Lang. Syst.*, 28(3):476–516, 2006.
- [15] M. Hertz, N. Immerman, and J.E.B. Moss. Framework for analyzing garbage collection.
- [16] Authors in <http://jikesrvm.sourceforge.net/info/core.shtml>. Jikes<sup>TM</sup> rvm home page. <http://jikesrvm.sourceforge.net/>.
- [17] Authors in <http://www.sable.mcgill.ca/soot/credits>. Soot: a java optimization framework. <http://www.sable.mcgill.ca/soot/>.
- [18] G.T. Leavens and Y. Cheon. Design by contract with jml. January 2006.
- [19] K. Rustan M. Leino and P. Müller. Modular verification of static class invariants. In *FM*, pages 26–42, 2005.
- [20] T. Lev-Ami, R. Manevich, and more. Tvla: 3-valued logic analysis engine. <http://www.cs.tau.ac.il/~tvla/>.
- [21] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, Second Edition*. Addison-Wesley, 1999.
- [22] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, 800 East 96th Street Indianapolis, Indiana, 2nd edition, 1997.
- [23] Sun Microsystems. Garbage collector spy tool, 2006.
- [24] D.A. Naumann and M. Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state. In *LICS '04: Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science (LICS'04)*, pages 313–323, Washington, DC, USA, 2004. IEEE Computer Society.
- [25] NetBeans. The netbeans profiler project, 2006.
- [26] M. Nick. The runtime structure of object ownership. In *European Conference on Object-Oriented Computing (ECOOP)*, 2006.
- [27] K. O’Hair. Hprof: A heap/cpu profiling tool in j2se 5.0, November 2004.
- [28] D. Rayside, L. Mendel, and D. Jackson. A dynamic analysis for revealing object ownership and sharing. In *WODA '06: Proceedings of the 2006 international workshop on Dynamic systems analysis*, pages 57–64, New York, NY, USA, 2006. ACM Press.
- [29] N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *32nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05)*, 2005.

- [30] N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. In *SAS*, pages 284–302, 2005.
- [31] R. Shaham, E.K. Kolodner, and S. Sagiv. Heap profiling for space-efficient java. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 104–113, 2001.
- [32] Eiffel Software. Building bug-free o-o software: An introduction to design by contract(tm), 2004.
- [33] Quest Software. Jprobe<sup>TM</sup> memory debugger, 2006.
- [34] Open Source. Gnu classpath project.
- [35] Wikipedia. Class invariant, 2006.
- [36] Wikipedia. Design by contract, 2006.

# List of Tables

3.1 Colors in use . . . . .	14
4.1 External depth flag possible values . . . . .	25
6.1 The well-known classes list . . . . .	47
6.2 Soot run on null example top ten cutpoint causing methods . . . . .	49
A.1 Classifying array objects. true - user object. false - otherwise. n/p - not possible. check - call stack check required . . . . .	65
B.1 Database and summary file fields . . . . .	76

# List of Figures

2.1 An illustration of the cutpoints for an invocation of the method <code>zoo</code> . . .	9
3.1 Cutpoints detection algorithm . . . . .	15
3.2 Detecting cutpoints example initial memory . . . . .	16
3.3 Detecting cutpoints example method call . . . . .	16
3.4 Detecting cutpoints example MarkGray result . . . . .	17
4.1 Liveness example node class . . . . .	19
4.2 Liveness example program . . . . .	20
4.3 Liveness example memory status of the program in Fig. 4.2 before line 5 . . . . .	20
4.4 Liveness example memory status of the program in Fig. 4.2 before line 9 . . . . .	21
4.5 Liveness example memory status of the program in Fig. 4.2 before line 9 after MarkGray . . . . .	21
4.6 Collecting cutpoints for liveness . . . . .	22
4.7 Finding object liveness . . . . .	23
4.8 Computing external sources algorithm . . . . .	27
4.9 Computing external sources example program . . . . .	29
4.10 Computing external sources example initial state (Fig. 4.9 before line 5)	30
4.11 Computing external sources example in call to <code>printFirst</code> (Fig. 4.9 before line 7) . . . . .	30
4.12 Computing external sources example in call to <code>printFirst</code> (Fig. 4.9 before line 7) after <code>MarkScanInternalRoots</code> . . . . .	31
4.13 Computing external sources example in call to <code>printFirst</code> (Fig. 4.9 before line 7) after <code>MarkExternalRoots</code> . . . . .	31
4.14 Computing external sources example in call to <code>print</code> (Fig. 4.9 before line 8) . . . . .	32
4.15 Computing external sources example in call to <code>print</code> (Fig. 4.9 before line 8) after <code>MarkScanInternalRoots</code> . . . . .	32
4.16 Computing external sources example in call to <code>print</code> (Fig. 4.9 before line 8) after <code>MarkExternalRoots</code> . . . . .	33
4.17 Computing external sources example in call to <code>printTwo</code> (Fig. 4.9 before line 9) . . . . .	33

4.18 Computing external sources example in call to printTwo (Fig. 4.9 before line 9) after MarkScanInternalRoots . . . . .	34
4.19 Computing external sources example in call to printTwo (Fig. 4.9 before line 9) after MarkExternalRoots . . . . .	34
5.1 Classical sharing example . . . . .	38
5.2 Classical sharing example continued . . . . .	39
5.3 Classical sharing example class invariant violation . . . . .	39
5.4 List tail sharing example . . . . .	39
5.5 List tail sharing example after list reverse . . . . .	40
5.6 Cutpoints list entry . . . . .	43
5.7 Class invariants violation computation using cutpoints . . . . .	43
5.8 Class invariants violation computation using cutpoints continued . . . .	44
5.9 Cutpoint list example . . . . .	45
6.1 String percentage out of the total cutpoints . . . . .	50
6.2 Disparity of method maximum cutpoints in total . . . . .	51
6.3 Disparity of method maximum unknown cutpoints in total . . . . .	51
A.1 Classifying objects . . . . .	66
A.2 GC enabled scan . . . . .	70
A.3 Modified cutpoint detection computation procedures for GC enabled scan . . . . .	71
B.1 Raw file entry . . . . .	73
B.2 Method heap only summary file example . . . . .	75
B.3 Program method heap only cutpoint type cutpoints to local heap ratio query result example . . . . .	76

## Appendix A

# Prototype implementation

This chapter discusses the prototype's implementation issues.

### A.1 Picking a platform

Bacon and Rajan [2] implemented their algorithms using the Jalapeño Java VM developed by IBM. Since then this VM has become an open source project named *Jikes<sup>TM</sup> Research Virtual Machine* ([16, 1]). Bacon and Rajan's algorithm has been implemented and has become a part of this VM. Therefore Jikes RVM was the natural choice for a platform.

Jikes RVM is written in Java<sup>TM</sup>. As a result modifying and adding new features is relatively easy. A good portion of Jikes RVM is platform independent due to the use of Java.

#### A.1.1 Limitations

Limitations implied from using Jikes RVM:

- The implementation is only in Java.
- Uninterruptible mode (Section A.4.1).
- Uses GNU's implementation of the Java libraries, GNU classpath ([34]).

This implementation is limited to:

- Single threaded applications.
- Non-copying garbage collectors.



### **A.1.2 The build process**

The Jikes RVM has an initial “bootstrap” build process in which a VM boot image is compiled and saved. A boot image builder process uses another VM to run the Jikes RVM compiler to compile itself. The resulting boot image is used to bootstrap the Jikes RVM whenever it is run. The image is loaded to memory and the RVM starts to run from there. Hence, the Jikes RVM is a VM written in Java, which runs on the host platform without a VM mediator. This fact makes the Jikes RVM a more efficient solution than other research platforms.

## **A.2 Common preliminaries**

### **A.2.1 Working on the user program**

Most RVM services, like memory allocation, run on all objects and methods, without distinction. Therefore the cutpoint detection computation has to distinguish user methods and objects from those belonging to the RVM.

The first step is classifying each class according to its package. This step is carried out when the class is loaded and, hence, only once per class. The classification is saved in the RVM class description object. The classes are classified as RVM classes, JDK classes and user classes.

#### **Object classification**

Objects are classified for the following reasons:

- Some user accessible objects reference internal VM data structures. For example, `java.lang.Class` references internal VM representation of a class in order to provide class information. As a result, the cutpoint detection computation can reach these objects too.
- Source lists, used on several occasions, should be maintained for user objects and hold only user objects.
- We were not interested in running other computations on RVM objects.

Class instances are classified according to their class. Arrays are classified according to their creator and their most inner element type. Array classification appears in Table A.1.

In some cases static information is not enough and objects are classified dynamically. Both the user program and the RVM use JDK class instances and arrays. Their classification is conducted by traversing the call stack and searching for the creator. There are occurrences where JDK objects are created by the RVM and returned to the user through the JDK. For example, when reading a file. The user calls the JDK. The JDK uses JNI to access operating system specific code. The native code handles the call, and uses the RVM to allocate memory for the returned

Creator	RVM	User	JDK
Created type			
RVM	false	n/p	false
User	true	true	true
JDK	check	true	check
Primitive	check	true	check

Table A.1: Classifying array objects. true - user object. false - otherwise. n/p - not possible. check - call stack check required

data. Therefore, to simplify matters, if the call stack scan finds a user frame, the object is classified as a user object.

Objects are classified when created. Classification starts only after the VM is fully booted. The result is saved in a flag at the object's header.

### The object classification computation

The object classification computation appears in Fig. A.1. `IsClass(Object)` indicates whether an object is a class instance or not. `IsArray(Object)` does the same for arrays. `GetObjectClassType(Object)` returns the class describing the object's class. Each description class has a classification flag, `ClassFlag(Type)`. The possible values are: User, JDK and RVM. `GetArrayMostInnerElementType(Type)` returns the most inner element type of the given array type, whether single or multi-dimensional array.

**IsUserObject(Object, Creator)** Classifies the created object `Object` according to its type, a class instance or an array.

**IsUserClass(Object)** Classifies the class instance `Object` according to its type. If the class belongs to the JDK, it is classified using a runtime test, `CheckForUser`.

**IsUserArray(Object, Creator)** Classifies the array object `Object` according to its creator type `Creator` and the most inner element type. The array is classified according to Table A.1.

**CheckForUser()** This procedure searches for a user frame on the call stack, starting from the frame where the current object was created.

### Limiting methods

The cutpoint detection computation is inserted into the beginning of user methods after they are loaded and before they are compiled to machine code. The instrumented methods are those belonging to user classes, according to the classification

```

IsUserObject(Object, Creator)
    if(IsClass(Object))
        return IsUserClass(Object)
    else if(IsArray(Object))
        return IsUserArray(Object, Creator)
    else
        return false

IsUserClass(Object)
    classType = GetObjectClassType(Object)
    if(ClassFlag(classType) == User)
        return true
    else if(ClassFlag(classType) == RVM)
        return false
    else if(ClassFlag(classType) == JDK)
        return CheckForUser()

IsUserArray(Object, Creator)
    if(ClassFlag(Creator) == User)
        return true
    innerElementType = GetArrayMostInnerElementTable(Object)
    if(IsClass(innerElementType))
        if(ClassFlag(innerElementType) == User)
            return true
        else if(ClassFlag(innerElementType) == RVM)
            return false

    return CheckForUser()

CheckForUser()
    Traverse the call stack looking for
    a frame whose method belongs to a class
    where IsUserClassType(stack-frame class) == true
    if found return true
    else return false

```

Figure A.1: Classifying objects

presented in Section A.2.1. RVM methods are not instrumented. Methods belonging to the JDK are optionally instrumented, because they can add a considerable number of cutpoints. Furthermore, before the cutpoint detection computation runs on a JDK method, it checks who asked for the JDK service, a User or an RVM class, and performs the computation or not accordingly.

In addition, in the following occasions, methods are not instrumented:

- Before the RVM is fully booted.
- Static methods without parameters.
- Methods without reference parameter types.
- The *main* method.

### A.2.2 Holding sources

The source lists can not hold references to objects when using a reference counting garbage collection due to the creation of reference cycles. There are some possible solutions:

- Reference count special case - Handles specifically the source list references. This solution complicates the garbage collection code that should be kept simple and fast.
- WeakReference - WeakReference solves the problem caused by using a regular reference. However WeakReference may pose a problem with RVM uninterruptible code (see Section A.4.1), since it is meant to use by user programs and not by internal RVM code.
- The Address type - *Address* is an RVM internal type, which appears as a class but which is replaced by the RVM compiler with an actual number. Hence no object is created. The advantage of using Address is being a fundamental part of the RVM memory module and, for this reason, is efficient. Because Address is not treated as a reference, it is not updated by moving garbage collections (such as Mark & Sweep) and hence can not be used with them.

The source list is updated at four locations:

- PutField write barrier - Object field assignment
- ArrayStore write barrier - Array element assignment
- Copying array write barrier for reference arrays
- When an object is released

The references of a released object are not cleared. Consequently the referenced objects are explicitly taken care of. When the object is released, it is removed from the source lists of all the objects it still references.

Each entry in the source list holds the following properties:

- The address of the source that the entry represents.
- A list of the fields referencing the object.
- External depth flag (See Section 4.3).

## **A.3 Computations specific**

### **A.3.1 Computing cutpoints preparations**

Coalescing (or deferred) Reference Counting ([9]) delays reference count updates to the actual garbage collection in order to reduce the write barrier cost. As a result, objects' reference counters do not hold their real value between collections. This results in inaccurate results when computing cutpoints. As a result the prototype runs a garbage collection without time quanta limit before each cutpoint detection computation.

The cutpoint detection computation's input is the actual parameters passed to the method. The parameters are read from the call stack and only object references are used. If all the references are null, the computation is not executed.

### **A.3.2 Live and dead cutpoints**

When detecting cutpoints the liveness computation stores source objects as candidates for liveness. For each cutpoint detected, the external sources, and their cutpoint referencing fields, are stored. The liveness computation is required to be quick when inside the read and write barriers. Therefore the computation uses a hash table for storing the candidates. The hash table keys are the sources. Each source's value holds a hash table of its targets, since the same source may be a candidate for several targets. Furthermore, the hash tables allows finding whether the source and the target exist in  $O(1)$  in average.

The target's value in the targets hash table for each source is a list of the cutpoint referencing fields for that source and target. The list is created easily by duplicating the list of fields for the source in the target's source list.

### **A.3.3 Early detection of class invariants violation**

In order to test the class invariants in the user program, the computation has to be able to run the invariants test. For this purpose there is a Java interface with a single method, which the computation uses to test the invariants. The user has to implement the interface such that when the computation calls the method the class

invariants are tested and the result of that test is returned. If the user has a single way for testing class invariants, this implementation can be done only once.

The computation implementation uses an index for quick access to the cutpoint objects in the cutpoint list (Section 5.2.3 and `AddToList(CP, currentDepth)` in Fig. 5.7).

A method may exit normally or exceptionally. Therefore in both cases `OnMethodExit(currentDepth)` (Fig. 5.7) has to be called.

## A.4 Other implementation notes

### A.4.1 Uninterruptible code

Uninterruptible code (see "What are the Semantics of Uninterruptible Code?" subsection in the "Magic" section of [10]) prevents "losing control" of execution to other threads. Hence in this code, more delicate operations are done. The cutpoint detection computation, source lists maintenance and the external source computation are implemented as uninterruptible code. Uninterruptible code allows the usage of only a subset of the Java language, for example, using the *new* operator or the cast operator is not allowed. Therefore uninterruptible code must be short and simple.

Adding new candidates for tracking in the liveness detection and class invariants violation detection is done in interruptible code in order to keep memory allocation simple. However the input to these computations is cutpoints, which are detected in uninterruptible code. Therefore some mechanism is needed to connect the two.

Our solution is to add a scan, which runs in interruptible code. The scan runs right after the scans of the cutpoint detection computation. The cutpoint detection computation marks all the objects detected as a cutpoint with a special flag, a *cutpoint flag*. The interruptible scan runs on the same objects and looks for the objects marked with the cutpoint flag. Each cutpoint is then passed to the computations as their input.

The scan can be interrupted by, among others, the garbage collection thread. Therefore the scan is called *GC enabled scan*. The potential problem is that the garbage collection might run and collect objects. Even so, the objects the GC enabled scan is scanning are not collected because they are reachable from the method's formal parameters, which are on the call stack.

The GC enabled scan is a depth first scan. As such, it has to mark scanned objects. Because the GC enabled scan scans the same objects the cutpoint detection computation does, a cooperation between the two scans is established. The GC enabled scan uses a scanning flag, called *GC enabled scan flag*. The cutpoint detection computation first scan, `MarkRoots(Roots)`, clears the GC enabled scan flag for all objects, ensuring, as the GC enabled scan scans the same objects, that the GC enabled scan has a clear slate. The GC enabled scan share is to clear the cutpoint flag for the scanned objects.

```

GCEnabledScan(Roots)
  For each S in Roots
    Scan(S)

Scan(S)
  if (GCESFlag(S) == false)
    GCESFlag(S) = true
    if(CutpointFlag(S) == true)
      use S in relevant computations
      CutpointFlag(S) = false
      for each T in children(S)
        Scan(T)

```

Figure A.2: GC enabled scan

$GCESFlag(S)$  is the GC enabled scan boolean flag for object  $S$ .  $CutpointFlag(S)$  is the cutpoint boolean flag for object  $S$ . Fig. A.2 shows the GC enabled scan algorithm.

**GCEnabledScan(Roots)** Runs the GC enabled scan for the same roots as in Section 3.4.

**Scan(S)** Perform a depth first scan starting at object  $S$ . Only objects not marked with the GC enabled scan flag are scanned. If an object  $S$  is found with a marked cutpoint flag,  $S$  is passed to the relevant computations and the cutpoint flag is cleared. Clearing is necessary in order not to mislead the next computation.

Fig. A.3 shows the modifications to the cutpoint detection algorithm (Section 3.4), which provides cutpoint information and initializes the scanning flag for the GC enabled scan.

The modified procedures (the new lines have the word *added* at their beginning) are:

**MarkGray(S)** This procedure clears the GC enabled scan flag for each object it scans. Hence this scan ensures that the objects are initialized for the GC enabled scan.

**Scan(S)** This procedure marks the cutpoints found using the cutpoint flag. By this marking the detected cutpoints are passed to the GC enabled scan.

```

MarkGray(S)
    if (color(S) != gray)
        color(S) = gray
added    GCESFlag(S) = cleared
        for each T in children(S)
            RC(T) = RC(T) - 1
            MarkGray(T)

Scan(S)
    if (color(S) == gray)
        if(RC(S) > 0)
            S is a cutpoint
added    CutpointFlag(S) = true
        color(S) = black
        for each T in children(S)
            Scan(T)
            RC(T) = RC(T) + 1

```

Figure A.3: Modified cutpoint detection computation procedures for GC enabled scan

## A.4.2 Summary of object header changes

The list of changes to the object header

- Cutpoint source list reference (Section A.2.2)
- User flag (Section A.2.1)
- Cut point flag (Section A.4.1)
- GC scan flag (Section A.4.1)
- Backward scan counter (Section 5.2.4)



## Appendix B

# Results processing

The results output by the prototype are processed until they become comprehensible tables. The processing stages are explained here.

### B.1 The prototype raw file

The prototype output is the the data file. The raw file has a record for each invocation of a method with cutpoints detection potential. Such a method has at least one object reference and, hence, a local heap. Therefore records appear even if a cutpoint was not detected.

There are four reported cutpoint types. The types indicate the kind of source that created the cutpoint. The cutpoint type values are exclusive. The possible cutpoint types are:

**Heap only (HO)** The source is an object in the global heap (and not in the local heap).

**Root only (RO)** The source is a local variable on the execution's stack or a static field.

**Heap and root (HR)** At least two sources, one from the heap and the second a local variable on the execution's stack or a static variable.

**Parameter (P)** An object, which is also a formal parameter, is not considered a cutpoint (Definition 2.2.1), even if it is referenced from an object in the global heap (A formal parameter is always referenced from the stack).

The record includes the following parts:

1. Method's fully qualified name in the JVM descriptor format (see [21]). For example: *java.util.HashMap Object put(Object key, Object value)* appears as *Ljava/util/HashMap;.put(Ljava/lang/Object;Ljava/lang/Object;)Ljava/lang/Object;*
2. (Optional) Method's local heap size measured in a number of objects

```
Method Ltest/TestHashEntry;.detectCutPoints(Ljava/util/HashMap;I)V
Heap size=64
Ljava/util/HashMap$HashEntry;=HO=20=
Ljava/util/HashMap;=Prm=1=
```

Figure B.1: Raw file entry

3. (Optional) A cutpoint description made out of the following:

- (a) A JVM descriptor of a class which appeared as a cutpoint on this call.  
For example: *java.lang.Class* appears as *Ljava/lang/Class;*
- (b) For each cutpoint type this class has appeared as
  - i. The cutpoint type
  - ii. The number of cutpoints of this type

**Example B.1.1** *The entry in Fig. B.1 shows a typical raw file entry. The detectCutPoints method was called and the cutpoint detection computation discovered 20 cutpoints of type java.util.HashMap.HashEntry originating from the heap (HO). java.util.HashMap was the actual parameter to the method. There were 64 objects in the local heap on this call.*

## B.2 The summary file

The summary file is a summarized version of the raw file. The file contains abbreviated data, arranged hierarchically according to methods and the classes that appeared as cutpoints in a method.

The summary file adds a level of distinction to the cutpoint types. Cutpoint types are separated by the cutpoint class, according to a list of well known classes. This list contains common classes, which usually appear as cutpoints and therefore may obscure other interesting results. The list appears in Table 6.1. The new well-known cutpoint types are *WKHO*, *WKRO*, *WKHR*, which are the same as the cutpoint types in Section B.1, but for cutpoints of well-known classes only. The original cutpoint types, *HO*, *RO*, *HR* now stand for cutpoints from all classes except those in the well-known list.

The summary file contains only one record for each method which appeared in the raw file, as opposed to one record for each method **invocation** in the raw file. The record is made out of the following sections:

1. Method summary

- (a) Number of method calls, including calls without any cutpoints.
- (b) Summarized cutpoint information for each cutpoint type, which appeared in this method throughout the program. In addition, the total of

well-known types, the total for not well-known types and the total of all types. Parameter cutpoint type does not appear in any of the totals.

- i. Number of cutpoints of this type
  - ii. Size of local heap when this type occurred
2. For each class, which appeared as a cutpoint in this method throughout the program
  - (a) Summarized cutpoint information for each cutpoint type, which appeared in this method throughout the program. In addition, the total of well-known types, the total for not well-known types and the total of all types. Parameter cutpoint type does not appear in any of the totals.
    - i. Number of cutpoints of this type
    - ii. Size of local heap when this type occurred

For each item in the list above, except the number of method calls, the following statistical information is calculated:

- Total - Sum of this item
- Call count - The number of method invocations in which this cutpoint type appeared
- Average - Equals to the total divided by the call count
- Minimum - Minimum value of this item
- Maximum - Maximum value of this item
- Variance - Variance compared to the average of this item
- Standard deviation - Standard deviation (square root) of this item

The structure of the summary file line is the same as in the database line (Section B.3) and appears in Table B.1.

**Example B.2.1** *Table B.2 shows a partial summary of the method summary. The name of the method is omitted for brevity. The method is the same as in Example B.1.1. The summary concerns all the heap only cutpoints detected for this method, along the program's execution. The first half shows the heap only (HO) cutpoint statistics and the second half shows the statistics for the local heap when heap only cutpoints were detected.*

Summary File line
*Method Call count Heap only 20
*Method Total Heap only 397
*Method Average Heap only 19.85
*Method Minimum Heap only 18
*Method Maximum Heap only 20
*Method Variance Heap only 0.2275
*Method Standard deviation Heap only 0.476969601
*Method Call count Heap only local heap 20
*Method Total Heap only local heap 1279
*Method Average Heap only local heap 63.95
*Method Minimum Heap only local heap 63
*Method Maximum Heap only local heap 64
*Method Variance Heap only local heap 0.0475
*Method Standard deviation Heap only local heap 0.217944947

Figure B.2: Method heap only summary file example

### B.3 Database processing

Due to the size of the summary file, a third stage is necessary. The summary file is loaded into a database where it is further processed. Processing is done by SQL queries, which produce summarized information according to the following divisions:

- Program, package or individual entries
- Methods or cutpoints. The method division summarizes cutpoints according to where they occurred. The cutpoints division shows the cutpoints themselves
- Well-known cutpoints and the rest
- Cutpoint types

The database table fields appear in Table B.1.

**Example B.3.1** *Table B.3 shows the results of a database query. The query shows cutpoints to local heap ratio. The data spans the whole program according to methods for all heap only cutpoints, not including well-known cutpoint classes.*

Field Name	Meaning
method_package	Package identifier of the method's class
method_class	Method's class identifier
method	Method identifier
type_package	Cutpoint class package identifier
type	Cutpoint class identifier
value_type	One of the statistical values, call count, total, etc.
cutpoint_type	Cutpoint type, such as HO, RO
value	Numerical value

Table B.1: Database and summary file fields

Query result	Value
AVG(cutpoints.value / local_heap.value)	0.310398751
COUNT(cutpoints.value / local_heap.value)	1
MIN(cutpoints.value / local_heap.value)	0.310398751
MAX(cutpoints.value / local_heap.value)	0.310398751
STDDEV(cutpoints.value / local_heap.value)	8.11E-10

Figure B.3: Program method heap only cutpoint type cutpoints to local heap ratio query result example