Tel-Aviv University
Raymond and Beverly Sackler Faculty of Exact Sciences
School of Computer Science

Heap-Liveness-based Memory Management: Potential, Tools, and Algorithms

by

Ran Shaham

under the supervision of Dr. Mooly Sagiv
and the consultation of Dr. Elliot Kolodner

A thesis submitted
for the degree of Doctor of Philosophy

Submitted to the Senate of Tel-Aviv University
September 2003

To my parents, Rivka and Joshua.

To my dearest ones, Yael, Nadav and Avishai.

Abstract

Heap-Liveness-based Memory Management:
Potential, Tools, and Algorithms

Ran Shaham
Doctor of Philosophy
School of Computer Science
Tel-Aviv University

Deallocating memory in a timely manner is a hard (and in general undecidable) problem. Programs (and programmers) face the problem of determining the exact lifetime of an allocated piece of memory. Failing to do so may lead either to errors as memory leaks or to poor performance due to consumption of extra space, or worse errors due to incorrect reuse of needed memory. Automatic garbage collectors mitigate some of these problems, but even they cannot prevent memory leaks and may be unacceptable in some environments.

In OO programs, such as Java programs, most of the allocated memory resides in the heap. We develop automatic techniques to determine when memory allocated in the heap is live at certain point in the execution, i.e., can be used further along some execution path from this point. We developed dynamic and static algorithms to determine heap liveness. This information is used to safely deallocate space and determine when manual memory deallocation is unsafe. The information may also be used to allow a garbage collector (GC) to reclaim more space.

Our dynamic algorithms profile the memory in order to collect liveness information (including liveness of heap-allocated elements) in a feasible manner. The dynamic algorithms were implemented in the Java Virtual Machine and used to estimate the potential memory savings beyond the ones obtained by existing GC algorithms. We also show that simple program transformations can be used to achieve some of these savings.

Our static algorithms identify program points at which memory can be deallocated and when heap references will not be used further. These algorithms establish temporal properties of the heap, which is challenging because the addresses of the memory elements concerned are not statically known. The algorithms were implemented and applied to a set of small interesting Java programs including JavaCard programs, to show we can actually deallocate memory at compile time.

Acknowledgements

I am deeply indebted to my advisor, Mooly Sagiv, for his dedicated supervision, guidance, great patience and for his endless optimism throughout this work. The imprint of his vast knowledge, scientific method and enthusiasm on my academic development and on this thesis is deep.

I have been fortunate to have Elliot Kolodner as my second unofficial advisor. I am grateful to him not only for his dedicated guidance, ideas, and significant contribution, but also for his unflagging support.

I would like to thank to Michael Pan for contributing to the research presented in in Chapter 4. Thanks to David Detlefs and Eliot Moss for their help in validating our empirical results for stack liveness in Chapter 5. Special thanks to Eran Yahav for joining in the research presented in Chapter 6, and for providing useful comments on earlier drafts of this thesis. I would like to thank to Roman Manevich and Thomas Stocker for many insights contributing to the research presented in Chapter 6. Special thanks to IBM Haifa Research Laboratory for its generous financial and technical support. Also, I would like to thank Giesecke & Devrient, Munich for their assistance and financial support. Finally, I would like to thank Nurit Dor, Viktor Kuncak, Tom Reps, Noam Rinetzky and Reinhard Wilhelm for reading earlier drafts of papers, on which this thesis is based.

This research was supported in part by the Israeli Academy of Science and by IBM through the Faculty Partnership Award.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Deallocating memory in a timely manner is a hard (and in general undecidable) problem. Programs (and programmers) face the problem of determining the exact lifetime of an allocated piece of memory. Failing to do so may lead either to errors as memory leaks or to poor performance due to consumption of extra space, or worse errors due to incorrect reuse of needed memory.

Automatic garbage collectors mitigate some of these memory problems, but even they cannot prevent memory leaks and may be unacceptable in some environments. Also, in some environments the cost of garbage collection can be significant. Typically, a garbage collector (GC) collects objects that are no longer reachable from a set of *root* references. However, there are some objects that the program never accesses again, even though they are reachable. This is illustrated pictorially in Fig. 1.1. Allocated but not reachable objects can be collected by GC. In-use objects have a future use and must be retained. Thus, an additional potential for space savings are the reachable objects that are not in-use.

Programmers, compiler writers, and memory manager developers are already aware of this problem. For example, in *live-precise GC* [5, 65, 1] the GC is enhanced with local variable liveness information to reclaim some of the reachable but not in-use objects; in particular, objects solely reachable from dead stack slots may be collected. Also, some of the JDK implementations are written in a GC-aware fashion,

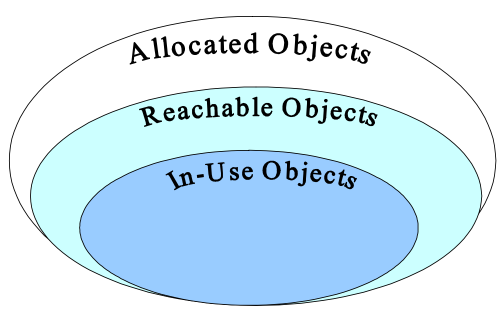


Figure 1.1: The heap.

although this is not a good programming practice. For example, in Sun's implementation of the class `java.lang.Vector` a null assignment statement is introduced for the sole purpose of making an object, which is not in-use object, unreachable [55, Section 1.1].

In this thesis we investigate the potential for improving current memory management techniques and develop algorithms for that purpose. In particular, we study the gap between the reachable objects and the reachable but not in-use objects (which we call *gap*). Closing this gap will allow more space-efficient programs, consuming space just for objects needed in the run. In addition, we would like to develop static analysis techniques to reduce GC cost. Thus, we study the following questions:

1. How big is the gap? A small gap indicates that current GCs are good enough to obtain space-efficient programs, while a large one leaves room for further improvements.

2. There exist techniques allowing GC to partially close the gap (e.g., live-precise GC). We would like to know how far these techniques are from closing the gap.

3. What kind of static information do we need to further close the gap beyond existing techniques?

4. What effective static algorithms could be developed to further close the gap beyond existing techniques?

5. Can static analysis be used to enable GC to be more effective (i.e., reclaim more space when invoked), and to eliminate the need for GC in memory constrained environments?

In order to estimate the gap between the reachable objects and the reachable but not in-use objects we develop dynamic algorithms to profile the memory behavior of a program in a feasible manner. The dynamic algorithms were implemented in a Java Virtual Machine (JVM). Our empirical results show a large potential for space savings beyond GC — 42% on the average for a set of Java programs.

Our measurements also give a kind of an upper bound for space savings beyond GC using existing techniques. We show a small potential (2% on the average) for space savings if static information on local variables is used, and medium potential if this information is combined with static information on global variables (9% on the average). The fact that local static information yields small benefits indicate that such algorithms are inadequate. This is in line with [1] that concludes that the main benefit of such algorithms is "preventing bad surprises".

For static information regarding heap paths (of arbitrary length) our measurements show a large potential for space savings (39% on the average), and when restricting the interface to GC such that the program is instrumented with null assignments to heap references we obtain a potential of 15% on the average beyond current GCs. These measurements motivate the development of static algorithms that reason rather precisely on arbitrary heap paths.

Based on our dynamic algorithms we developed a prototype heap profiling tool that attributes the potential space savings to source locations. The tool is used to guide a user in performing simple space-saving source modifications. Our experience with the tool shows an actual reduction of 14% on the average for a set of Java programs.

Finally, we developed a framework for proving temporal properties of heap manipulating programs. We instantiate the framework with two new static algorithms allowing the automation of the process of program space savings. These algorithms fall in the domain of compile-time garbage collection (see Chapter 7), which was studied in the past mostly for functional languages. Our algorithms are shown to be precise for heap manipulating programming languages allowing destructive updates (e.g., Java). Furthermore, our algorithms allow space saving beyond GC, i.e., they allow the deallocation of reachable objects not needed further in the run. For some of our example programs they even eliminate the need for GC.

The first algorithm instantiated from the framework detects places in code where `free` statements can be safely inserted. The second algorithm detects places in code where statements assigning null to heap references can be safely inserted, thereby allowing GC to reclaim more space. A prototype implementation of the static algorithms is applied to a set of Java programs, including JavaCard programs, to show we can actually insert code to deallocate memory at compile time. Moreover, we show that using points-to based heap abstraction [3, 63] is insufficient for deallocating memory in our example programs. Interestingly, our techniques could also be used for languages like C to either deallocate memory or to find errors by detecting a misplaced call to `free` that prematurely deallocates an object.

1.1 Thesis Contribution

The contributions of this thesis can be summarized as follows. We also cite for each contribution, where we first reported it publicly.

1. New algorithms for dynamic measurements of space, which cannot be saved by existing GC [56, 59].

2. Measurements for potential space savings beyond GC showing large potential for heap-liveness-based memory management, and showing small potential for existing techniques (e.g., live-precise GC) [59].

3. We show that the information collected by our dynamic algorithms can be utilized to reduce space in programs, and present a tool allowing a programmer to find places in code to save space [57].

4. We develop a framework for proving temporal properties of heap manipulating programs [61, 60].

5. We instantiate the framework for providing new static algorithms for finding space, which cannot be saved by existing GC [61, 60].

1.2 Outline

Chapter 2 give some preliminaries; we define the terms used throughout the thesis and discuss our experimental methodology. In Chapter 3 we study the effectiveness of GC, by measuring the potential space savings if an object is collected as soon it is no longer accessed in the run. In Chapter 4 we present a heap profiling tool based on the measurements of Chapter 3 for guiding the user in simple space-saving source modifications. In Chapter 5 we present further dynamic measurements for showing upper bounds for space savings for existing techniques, and for investigating the potential savings due to the consideration of static information of arbitrary heap paths. Chapter 6 reports a framework for verifying temporal heap properties, along with two instances of the framework for providing static algorithms allowing space savings beyond current GCs. Throughout the thesis we briefly mention related work. In addition in Chapter 7 we give a more in depth view of the related work. Finally, in Chapter 8 we conclude and discuss further research.

Chapter 2

Preliminaries

This chapter lays the groundwork for later chapters of this thesis. First, in Section 2.1 we define the terms used in this thesis. Then, in Section 2.2 we discuss the experimental methodology used for obtaining empirical results.

2.1 Technical Definitions

We assume a Java-like programming model. Throughout this chapter we use the simple program shown in Fig. 2.1 as our running example. A *program state* $\sigma_i = \langle \text{store}_i, \text{pt}_i \rangle$ represents the state of the program, which consists of the store (store_i) and the current program point (pt_i)[1]. The store consists of an execution stack and a heap of objects. A heap object is viewed as a record of fields, where an object field is associated with a memory location. Thus, an object field has an l-value [43]. An array is a special kind of an object. In the following definitions we omit the treatment of reference array expressions (e.g., of the form $a[i]$) since it is similar to the treatment of other reference expressions (e.g., of the form $e.f$).

We use a pictorial representation of program states shown in Fig. 2.2. For example, Fig. 2.2(d) shows σ_4, the program state where the current program point is 4, i.e., just before executing the statement `y = x` at line 4. The global store in σ_4 consists of an execution stack and two heap objects. The stack variable `x` references the object o_1, and the stack variable y references the object o_2. Both objects o_1, o_2 consist of a single field named `f`. The field `f` of o_1 is used to reference o_2. The field `f` of o_2 has a `null` value, thus does not reference any object.

A *trace* $\pi = \sigma_1, \sigma_2, \ldots$ is a (possibly infinite) sequence of program states σ_i. A trace reflects a program execution. For example, Fig. 2.2(a)-(i) shows a trace $\pi = \sigma_1, \ldots, \sigma_9$ of our example program. Even for this simple straight-line code program, there are other possible traces, for example, the program can terminate after the allocation at 2 due to memory overflow. For a trace π, and for some i, we define

[1] We make a simplifying assumption, and do not discuss multithreaded execution in this chapter. We discuss multithreaded execution in later chapters where applicable.

```
    class C {
      public C   f;
    }
    class Main {
     public static void main(String args[]) {
 [1]     x = new C();
 [2]     y = new C();
 [3]     x.f = y;
 [4]     y = x;
 [5]     x = x.f;
 [6]     if (x == y)
 [7]       System.out.println{"internal error");
 [8]     x = null;
 [9]     y = null;
[10]
     }
    }
```

Figure 2.1: Reachability, drag and liveness example.

the *trace prefix* π_i as the program states $\sigma_1, \ldots, \sigma_i$; we also define the *trace suffix* π^i as the program states $\sigma_i, \sigma_{i+1}, \ldots$.

2.1.1 Drag Definitions

As stated in Chapter 1, our main motivation is to improve current memory management techniques by reducing the gap between the reachable objects and the reachable but not in-use objects. We now define the terms that characterize this gap.

Definition 2.1.1 (Reachable Object) An object is a reachable object in a program state $\sigma_\mathbf{i} = \langle \mathit{store}_\mathbf{i}, \mathit{pt}_\mathbf{i} \rangle$, *if it is reachable from the set of the root references in $store_i$; root references typically include stack reference variables and global reference variables*[2].

Garbage collectors typically reclaim objects based on reachability information. Objects that are not reachable are considered as garbage and thus collected. For example, o_1 is reachable in $\sigma_2, \ldots, \sigma_8$ shown in Fig. 2.2(b)-(h). It is not reachable in σ_9 shown in Fig. 2.2(i), thus it can be collected by a reachability-based GC, after the program executes the statement at 9.

Definition 2.1.2 (Object Dereference) An object o is dereferenced in a program state $\sigma_\mathbf{i} = \langle \mathit{store}_\mathbf{i}, \mathit{pt}_\mathbf{i} \rangle$, *if the statement at pt_i contains an expression $e.f$ and the r-value of e at $store_i$ is o.*

[2]Later in the text, in Table 3.1, we list the root kinds for Java.

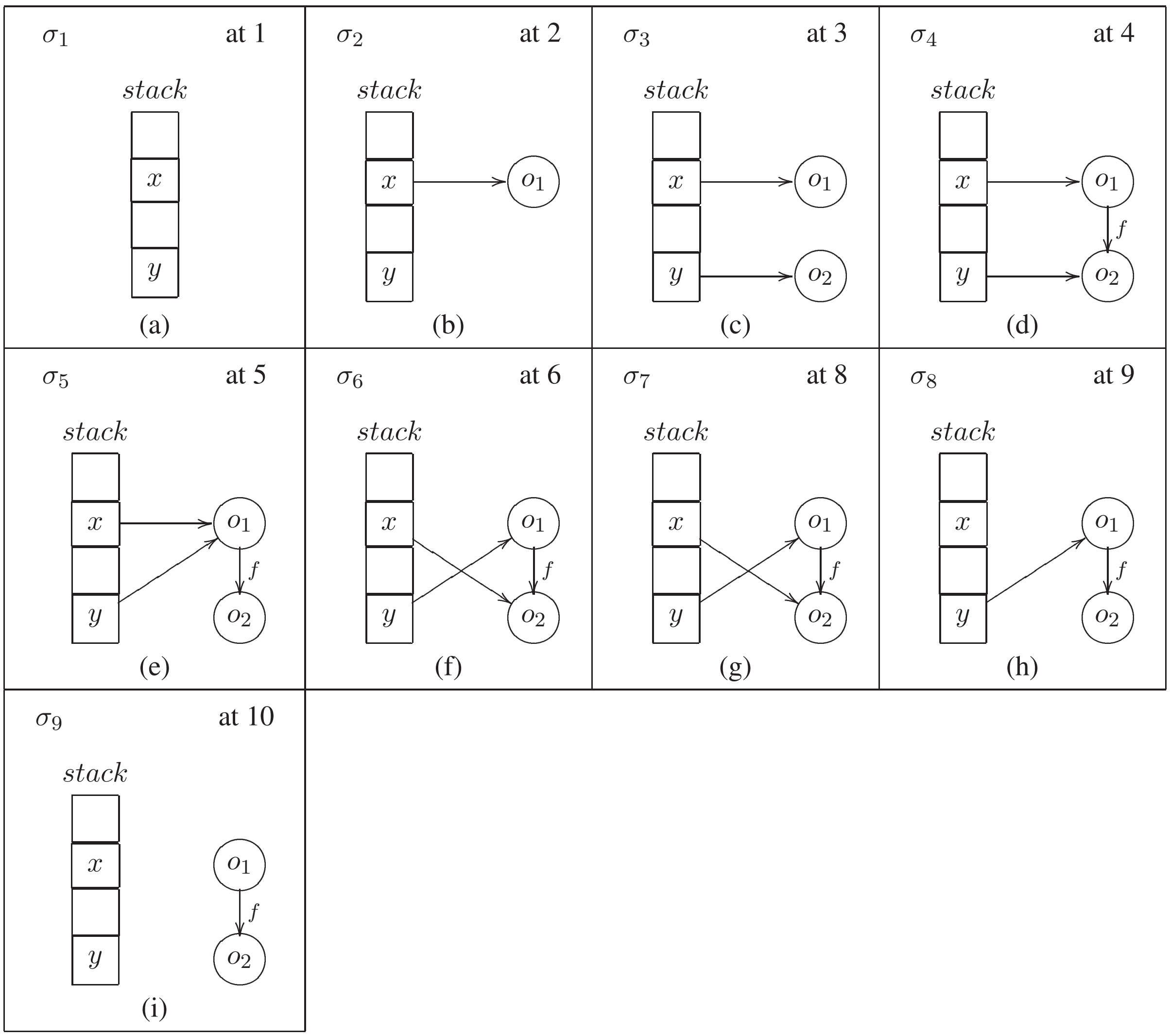


Figure 2.2: A trace of the program in Fig. 2.1.

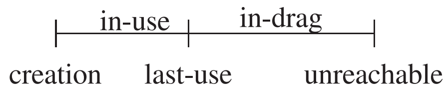


Figure 2.3: The lifetime of an object.

For example, o_1 is dereferenced in σ_3, since the statement `x.f = y` at 3 contains the expression `x.f` and the r-value of `x` in store$_3$ is o_1. Later in the text the terms *object use* and *object access* are also used to denote an object dereference. The last object dereference determines when an object is no longer needed in a program execution, as captured in the following definition.

Definition 2.1.3 (In-Use Object) **Given a trace π, an object is an in-use object in program state σ_i,** *if it is dereferenced in program state σ_j, for some $j \geq i$.*

In other words, an in-use object is an object which is still accessed in the program; thus still needed. As shown in Fig. 2.3, an object is in-use from the time it is created until the time of its last use, i.e., the time it is last accessed. Clearly, an in-use object is a reachable object; however, a reachable object may not be in-use. For example, o_1 is in-use (and reachable) in $\sigma_2, \sigma_3, \sigma_4, \sigma_5$. It is not in-use (but reachable) in $\sigma_6, \sigma_7, \sigma_8$.

Definition 2.1.4 (In-Drag Object) **Given a trace π, an object is an in-drag object in program state** σ_i*, if it is not dereferenced in program state σ_j, for all $j \geq i$.*

For example, o_1 is in-drag in $\sigma_6, \sigma_7, \sigma_8$. The term drag is due to Röjemo and Runciman [50]. We refer to the time interval from the last use of an object until it becomes unreachable as the object's *drag time*. An object with a drag time greater than zero is considered a *dragged object*. Drag time measures a potential for space savings independent of the actual GC method or GC implementation.

Drag information allows determining the earliest possible time an object could be collected. There is one subtle issue, though, with collecting in-drag objects as demonstrated in σ_6 (shown in Fig. 2.2(f)). The object o_1 is in-drag in σ_6, thus can be collected. However, the reference to o_1 from y is still used in the condition `x == y` at 7, thus the value of y cannot be discarded by the memory manager. This situation would require the memory manager to handle "valid dangling references", in which an object is reclaimed but references to that object cannot yet be ignored.

The main burden in maintaining "valid dangling references" stems from the fact that due to reuse of space, a valid dangling reference and a regular reference may have the same r-value, although the meaning of these references is different. The valid dangling reference value denotes a dangling reference to an object that have been deallocated, while the regular reference value denotes a reference to an allocated object. Moreover, due to multiple reuse of the same space, the memory manager may need some mechanism for distinguishing valid dangling references that have the same r-value but refer to different deallocated objects. Motivated by the above, in the next section we discuss our remedies for this problem.

2.1.2 Liveness Definitions

One way to alleviate the problem of "valid dangling references" is to delay the reclamation of an object until its references are no longer used. Indeed, this is the approach we develop in Chapter 5 and Chapter 6. More technically, we focus on the liveness of references (to be defined shortly), and in particular the liveness of heap references, to determine when an object can be collected. Interestingly, our experimental results (see Section 5.4) show that the potential for space savings if an object is collected as soon as its references are dead (i.e., all its references are no longer used) is close to the potential space savings if an object is collected as soon it is in-drag (i.e., after the object is last used).

Before presenting the definitions for reference liveness, recall (e.g., [43]), that a scalar variable `var` is *live* at a program point *pt*, if there exists a trace π including a program state $\sigma_i = \langle \text{store}_i, \text{pt} \rangle$ and a use of `var` in σ_j such that (i) $i \leq j$ and (ii) `var` is not assigned in $\sigma_i \ldots \sigma_{j-1}$. For example `x` is live at 4, due to the use of `x` at the statement `y = x` at 4.

We now present a two-fold extension of this definition. First, we extend the definition to allow reasoning about "complete liveness information", i.e., the most precise liveness information given a program execution. We use this complete liveness information to give theoretical upper bounds on the impact of liveness information on the space consumption of a program. Second, following our previous work [55] we extend the liveness definition to express the liveness of arbitrary expressions.

Definition 2.1.5 (Dynamic Location Liveness) A memory location l is dynamically live in a program state σ_i along a trace π *if (i) l is used in σ_j, for some $j \geq i$, and (ii) l is not assigned in all* $\sigma_i, \ldots, \sigma_{j-1}$.

For example, the location of the field `f` of object o_1 is live in σ_5 due to its use in σ_5 (to obtain the r-value of `x.f` in the statement `x = x.f` at 5). Dynamic location liveness allows reasoning about complete liveness information given a program execution, as it defines the liveness of every location in every program state along the trace. As usual in liveness definitions, a location, which is not dynamically live is considered a *dynamically dead location*. In the same manner, the next definitions in this section also implicitly define their corresponding negated (dead) notions,, e.g., Definition 2.1.6 also implicitly defines what is a *dynamically dead object*.

Later in the text, we use the term *heap reference liveness* to denote dynamic heap location liveness, when the heap location stores information of reference type. We do the same for locations used to store stack and global variables, i.e., we use the terms *stack reference liveness* and *global reference liveness*, to denote dynamic liveness of stack or global variable locations of type reference.

Using Definition 2.1.5 we are now able to determine the time an object can potentially be collected.

Definition 2.1.6 (Dynamic Object Liveness) An object is dynamically live in a program state $\sigma_\mathbf{i}$ along a trace π *if at least one location among the locations of its references in store$_i$ is dynamically live in σ_i.*

In other words, an object can be collected as soon as its references are no longer used. For example, o_1 is dynamically live in $\sigma_2, \sigma_3, \sigma_4, \sigma_5, \sigma_6$, and is dynamically dead in $\sigma_7, \sigma_8, \sigma_9$. Thus, o_1 may be collected in σ_7 assuming dynamic liveness information is available.

Our second extension to the liveness definition concerns the liveness of an expression. The following definitions allow the relating of liveness information to the program code.

Definition 2.1.7 (Dynamic Expression Liveness) An expression e is dynamically live in a program state $\sigma_\mathbf{i}$ along a trace π, *if (i) e denotes a location l in σ_i, and (ii) l is used in σ_j, for some $j \geq i$, and (iii) l is not assigned in all $\sigma_i \ldots \sigma_{j-1}$.*

For example, the expression `x.f` is dynamically live in σ_5 in our example trace, since `x.f` denotes the location of the field `f` of object o_1 in σ_5; this location is used (without prior assignment) in σ_5 (to obtain the r-value of `x.f` in the statement `x = x.f`).

Definition 2.1.8 (Dynamic Expression Liveness at a Program Point) An expression e is dynamically live at a program point *pt* in a trace π, *if π includes a program state $\sigma_i = \langle store_i, pt \rangle$ such that (i) e denotes a location l in σ_i, and (ii) l is used in σ_j, for some $j \geq i$, and (iii) l is not assigned in all $\sigma_i \ldots \sigma_{j-1}$.*

For example, the expression `x.f` is dynamically live at 5 in our example trace, since `x.f` denotes the location of the field `f` of object o_1 in σ_5; this location is used (without prior assignment) in σ_5 (to obtain the r-value of `x.f` in the statement `x = x.f`).

Definition 2.1.9 (Static Expression Liveness) An expression e is statically live at a program point *pt*, *if there exists a trace π including a program state $\sigma_i = \langle store_i, pt \rangle$ such that (i) e denotes a location l in σ_i, and (ii) l is used in σ_j, where $j \geq i$, and (iii) l is not assigned in all $\sigma_i \ldots \sigma_{j-1}$.*

Clearly, if e is dynamically live at pt in a trace π then it is also statically live. For example, the expression `x.f` is statically live at 5. Also, note the distinction we make between *static liveness* and *dynamic liveness* definitions. Static liveness existentially quantifies over traces, while dynamic liveness reasons about a given trace. Finally, note that static expression liveness and the classic definition of variable liveness coincide when e is a simple variable expression (e.g., $e \equiv x$).

2.1.3 Summary of Drag and Liveness Definitions

In Section 2.1.1 and in Section 2.1.2 we have characterized three dynamic approaches for determining when an object can be collected given a trace:

Reachability Information An object may be collected as soon it is not reachable from the root set. In our running example the earliest program state at which o_1 is not reachable is σ_9.

Drag Information An object may be collected as soon it is in-drag. In our running example the earliest program state at which o_1 is in-drag is σ_6. This approach provides the earliest potential collection time for an object at the price of handling "valid dangling references".

Object Liveness Information An object may be collected as soon it is dynamically dead. In our running example the earliest program state at which o_1 is dynamically dead is σ_7. The potential collection time using object liveness information is between the drag-information-based collection time and the reachability-information-based collection time. Indeed, object liveness information provides collection time potentially earlier than the collection time based on reachability information, while avoiding the problem of valid dangling references arising with drag information.

Dynamic drag and liveness information require a full trace, thus these techniques are applicable only for a *post-mortem* analysis, allowing theoretical upper bounds on space savings beyond the ones obtained by reachability information. However, both drag and liveness information may be beneficial for improving a reachability-based GC. In particular, in Chapter 3 and Chapter 4 we show that drag information provides insights on space-saving code transformations. In Chapter 5 and Chapter 6 we (i) show that dynamic object liveness information provides potential space benefits similar to the ones of drag information, and (ii) provide heap-liveness-based static algorithms to automate the process of locating space-saving code transformations.

2.1.4 Program Transformation Definitions

One way to allow space savings in a program is by transforming the program code, while preserving the correctness of the original program. Throughout the thesis we explore simple program transformations that allow space savings. Some of the transformations assume a GC facility, while others may work either with or without a GC facility. We now define what is a correctness preserving transformation and define the program transformations used. For simplicity, in the following we consider terminating programs.

Definition 2.1.10 (Correctness-Preserving Transformation) A program transformation is correctness-preserving *if it preserves the observable properties of the original program, i.e., on a given input the* *original and transformed program either terminate and produce the same output, or they both do not terminate.*

This definition assumes no resource limits when running the program, unless the program code explicitly handles the case of an unavailable resource. For example, a transformation may still be correctness-preserving even if the original program abnormally terminates due to lack of memory, and the transformed program completes normally due to some space savings. The idea is that if the original program completes normally given enough space, then both the original and the transformed program would produce the same output. Having said that, special care still needs to be taken when the program code explicitly handles the case of an unavailable resource. For simplicity, we assume that space saving transformations are not allowed for Java programs that handle a `java.lang.OutOfMemoryError` exception raised by an allocation statement.

Let us now look at two important space saving transformations we investigate throughout this thesis. The first transformation allows us to free an object, by issuing a free statement in the program. The second transformation allows the garbage collector to reclaim more space by issuing a statement assigning null to a reference in the heap graph.

In Java, free statements are not supported, thus we assume the Java language is extended with a `free x`[3] statement. In Section 6.2 we discuss how free statements could be implemented in Java.

For that purpose of giving semantics to a free statement, we assume a constant domain semantics [20], where Loc is the set of program locations. When the program begins, all program locations are inactive (i.e., not allocated). As usual, an allocation request is satisfied by taking an inactive location l and making it active. Garbage collection frees locations by making active locations inactive. Our free statement also free a location by making it inactive, thus candidate for reuse.

Definition 2.1.11 (Free Property $\langle pt, \mathtt{x}\rangle$) The property free $\langle \mathbf{pt}, \mathtt{x}\rangle$ holds *if there exist no trace π with a program state $\sigma_i = \langle store_i, pt\rangle$ such that there exists a reference to the object referenced by* x *in σ_{i+1}, which is dynamically live in σ_{i+1} in π.*

The free property allows us to free an object that is referenced solely by dead references. In particular, if a free property $\langle pt, \mathtt{x}\rangle$ holds, then it is safe to issue a `free x` statement immediately after pt. For example, the free property $\langle 6, \mathtt{y}\rangle$ holds for our example program, since the reference(s) to o_1 at 6 are dynamically dead on all possible traces. Thus, `free y` may be safely inserted after 6, allowing the reclamation of o_1 in before σ_7.

For expository purposes, we only present the free property for an object referenced by a program variable. However, this free property can easily handle the free for an object referenced through an

[3]for simplicity we assume only a variable expression is included in a free statement

arbitrary reference expression `exp`, by introducing a new program variable `z`, assigned with `exp`, and verifying that `free z` may be issued just after the statement `z = exp`.

We now prove that issuing a free statement where the free property holds is a correctness-preserving transformation. We assume the transformed program is deterministic. In the following, P denotes the original program, and P' denotes the transformed program. For convenience, we assume the original program P includes a "placeholder" program point for inserting the free statement, i.e., we assume a program point pt_f that immediately follows pt, which includes a `skip` statement in P, and that P' transforms the `skip` at pt_f to a `free x` statement. $[\![e]\!](\sigma)$ denotes the r-value of an expression e at program state σ. It can be defined by structural induction on e. Finally, a *store mapping function* S : Loc $\cup$ Atoms $\rightarrow$ Loc $\cup$ Atoms maps locations of one store to locations of another store. In addition such store mapping function serve as the identity function for atomic values.

Lemma 2.1.12 *If a free property $\langle pt, \mathtt{x}\rangle$ holds, and pt_f is the program point immediately after pt and P is transformed to a program P' so the* `skip` *statement in P at pt_f is replaced by* `free x` *in P' at pt_f, and $\pi = \sigma_1 \ldots, \pi' = \sigma'_1 \ldots$ are traces in P, P', respectively obtained for the same input, and $\sigma_i = \langle store_i, p\rangle, \sigma'_i = \langle store'_i, p\rangle$ are program states in π, π', respectively, then there exist store mapping functions S_i : $store_i \rightarrow store'_i$ and S'_i : $store'_i \rightarrow store_i$, such that for every dynamically live expression e in either σ_i or σ'_i it holds that $[\![e]\!](\sigma_i) = S_i([\![e]\!](\sigma'_i))$ and $S'_i[\![e]\!](\sigma_i) = [\![e]\!](\sigma'_i)$.*

Proof: In the spirit of Lacey et. al [39] we prove the Lemma by induction on the length k of the trace.
Basis: For the base case ($k = 1$) we set S_1, S'_1 to the identity function. The claim trivially holds for that case, since both the original and transformed programs start with uninitialized values for all program locations.
Induction hypothesis: If π_i, π'_i are the trace prefixes of the original and transformed program respectively, then there exist store mapping functions S_i : $\mathrm{store}_i \rightarrow \mathrm{store}'_i$ and S'_i : $\mathrm{store}'_i \rightarrow \mathrm{store}_i$ such that for every dynamically live expression e in either store_i or store'_i it holds that $[\![e]\!](\sigma_i) = S_i([\![e]\!](\sigma'_i))$ and $S'_i[\![e]\!](\sigma_i) = [\![e]\!](\sigma'_i)$.
Induction step: In every step we set $S_{i+1}(v) = S_i(v)$ and set $S'_{i+1}(v) = S'_i(v)$ for every location or atomic value v. The only changes to the store mapping functions take place during allocation. These changes are described in the first subcase of case 1 below. We note that during deallocation there are no changes to the store mapping functions. This is since the free property holds for the deallocated locations, thus deallocation takes place only for dynamically dead locations, which have no effect on our claims for dynamically live expressions.

For brevity, we show in the following that for S_{i+1} and every dynamically live expression e, $[\![e]\!](\sigma_{i+1}) = S_{i+1}([\![e]\!](\sigma'_{i+1}))$. Showing that for S'_{i+1} and for every dynamically live expression e, it holds that $S'_i[\![e]\!](\sigma_i) = [\![e]\!](\sigma'_i)$ is done in a similar manner.

For the step case ($k = i + 1$) we distinguish between two cases:
case 1: $\sigma_i = \langle \text{store}_i, p\rangle, \sigma'_i = \langle \text{store}'_i, p\rangle$ and $p \neq pt_f$. We now consider the possible assignments at p and their effect on $\text{store}_i, \text{store}'_i$, and also consider the effect of an allocation statement. We distinguish between a variable expression (of the form x) and a field expression (of the form $x.f_1.f_2.\ldots.f_j$), and assume assignment statements are normalized so a field expression may not appear on both the left hand and right hand sides of the statement. In the following subcases, `y` denotes a program variable.
subcase $\mathbf{x} = \mathbf{new}\ \mathbf{C}()$: Assuming the allocation in σ_i is satisfied by a location l, and the allocation in σ'_i is satisfied by a location l', we update S_{i+1} to map l to l', and update S'_{i+1} to map l' to l (for all other locations and atomic values $S_{i+1}(v) = S_i(v)$, and $S'_{i+1}(v) = S'_i(v)$). In addition we assume allocation returns an uninitialized location, thus $[\![x]\!](\sigma_i) = S_{i+1}([\![x]\!](\sigma'_i))$.
subcase $\mathbf{x} = \mathbf{y}$: Because y is used at p, it is live in store_i, therefore by the induction hypothesis $[\![y]\!](\sigma_i) = S_i([\![y]\!](\sigma'_i))$. S_{i+1} is set to S_i, leading to $[\![x]\!](\sigma_{i+1}) = S_{i+1}([\![x]\!](\sigma'_{i+1}))$.
subcase $\mathbf{x} = \mathbf{y.f_1.f_2.\ldots.f_j}$: Because $y.f_1.f_2.\ldots.f_j$ is used at p, it is live is store_i and therefore by the induction hypothesis $[\![y.f_1.f_2.\ldots.f_j]\!](\sigma_i) = S_i([\![y.f_1.f_2.\ldots.f_j]\!](\sigma'_i))$. S_{i+1} is set to S_i, leading to $[\![x]\!](\sigma_{i+1}) = S_{i+1}([\![x]\!](\sigma'_{i+1}))$.
subcase $\mathbf{x.f_1.f_2.\ldots.f_j} = \mathbf{y}$: Because $x.f_1.f_2.\ldots.f_{j-1}$ is used at p, the expressions $x, x.f_1, x.f_1.f_2, \ldots, x.f_1.f_2.\ldots.f_{j-1}$ are dynamically live in store_i, therefore by the induction hypothesis $[\![x]\!](\sigma_i) = S_i([\![x]\!](\sigma'_i))$, $[\![x.f_1]\!](\sigma_i) = S_i([\![x.f_1]\!](\sigma'_i))$, $[\![x.f_1.f_2]\!](\sigma_i) = S_i([\![x.f_1.f_2]\!](\sigma'_i))$, $\ldots$, $[\![x.f_1.f_2.\ldots.f_{j-1}]\!](\sigma_i) = S_i([\![x.f_1.f_2.\ldots.f_{j-1}]\!](\sigma'_i))$, Also, y is used at p, thus y is dynamically live in store_i, therefore by the induction hypothesis $[\![y]\!](\sigma_i) = S_i([\![y]\!](\sigma'_i))$. S_{i+1} is set to S_i, leading to $[\![x.f_1.f_2.\ldots.f_j]\!](\sigma_{i+1}) = S_{i+1}([\![x.f_1.f_2.\ldots.f_j]\!](\sigma'_{i+1}))$.
case 2: $\sigma_i = \langle \text{store}_i, \text{pt}_f\rangle, \sigma'_i = \langle \text{store}'_i, \text{pt}_f\rangle$. We now apply a `free x` to σ'_i and a `skip` to σ_i to get $\sigma'_{i+1}, \sigma_{i+1}$, respectively. We assume l'_x denotes the location of x in store'_i. Thus, `free x` makes the location l'_x inactive in store'_{i+1}. From Definition 2.1.11, l'_x is solely referenced by dynamically dead references, thus no dynamically live expression e in σ'_i may reference l'_x. Thus, making l_x inactive in store'_i does not affect the evaluation of dynamically live expressions in either σ'_i (or σ_i), thus the induction hypothesis still holds.

Lemma 2.1.13 *If a free property $\langle pt, \mathtt{x}\rangle$ holds, then issuing a `free x` statement immediately after pt is a correctness-preserving transformation.*

Proof: We assume the program can output the r-value of non-reference expressions (as in Java). Thus if e is used as output, then e is dynamically live, therefore from Lemma 2.1.12 e has the same value in both P, P' and the programs will produce the same output.

Definition 2.1.14 (Assign-Null Property $\langle pt, \mathtt{x}, \mathtt{f}\rangle$) The property assign null $\langle pt, \mathtt{x}, \mathtt{f}\rangle$ holds *if there exist no trace π that includes a program state $\sigma_i = \langle store_i, pt\rangle$ such that the location denoted by $x.f$ in σ_{i+1} is dynamically live in σ_{i+1} in π.*

The assign-null property allows us to assign null to a dead heap reference (recall a live heap reference is a dynamically live heap location of type reference, see Definition 2.1.5). Adding an assign-null statement potentially makes objects unreachable, thus subject to GC. In particular, if an assign-null property $\langle pt, \mathtt{x}, \mathtt{f}\rangle$ holds, then it is safe to add a `x.f = null` statement immediately after pt. For example, the assign-null property $\langle 5, \mathtt{x}, \mathtt{f}\rangle$ holds for our example program, since the location denoted by `x.f` (the location of field `f` emanating from o_1 in our example trace) is dynamically dead in all traces in the program state following a program state at 5. Inserting `x.f = null` after 5 removes the reference from o_1 to o_2; thus it allows the reclamation of o_2 in σ_7, which is earlier than collecting it in σ_9, the original earliest time to collect o_2 based on reachability information. As in the free property case, our assign-null property can also handle arbitrary reference expressions (e.g., of the form `exp.f`), by introducing a new program variable `z`, assigned with `exp`, and verifying the `z.f` may be issued just after the statement `z = exp`.

Lemma 2.1.15 *If an assign null property $\langle pt, \mathtt{x}, \mathtt{f}\rangle$ holds, and pt_f is the program point immediately after pt and P is transformed to a program P' so the `skip` statement in P at pt_f is replaced by `x.f = null` in P' at pt_f, and $\pi = \sigma_1 \ldots, \pi' = \sigma'_1 \ldots$ are traces in P, P', respectively obtained for the same input, then there exist store mapping functions $S_i : store_i \rightarrow store'_i$ and $S'_i : store'_i \rightarrow store_i$, such that for every dynamically live expression e in either σ_i or σ'_i it holds that $[\![e]\!](\sigma_i) = S_i([\![e]\!](\sigma'_i))$ and $S'_i[\![e]\!](\sigma_i) = [\![e]\!](\sigma'_i)$.*

Proof: The proof is similar to the proof of Lemma 2.1.12 with the difference that in case 2 we apply a `x.f = null` to σ'_i and a `skip` to σ_i to get $\sigma'_{i+1}, \sigma_{i+1}$, respectively. The induction hypothesis holds in case 2 since from Definition 2.1.14, `x.f = null` assigns a null value to a dynamically dead location, and this is the only change taking place in $\mathrm{store}_i, \mathrm{store}'_i$.

Lemma 2.1.16 *If an assign null property $\langle pt, \mathtt{x}, \mathtt{f}\rangle$ holds, then issuing a `x.f = null` statement immediately after pt is a correctness-persevering transformation.*

Proof: Similar to the proof of Lemma 2.1.13, with the difference that now Lemma 2.1.15 is used (instead of Lemma 2.1.12).

2.1.5 Space Measurement Definitions

This section defines the terms we use when estimating the potential space savings achievable beyond current GCs. The main idea here is to compute the space consumption of a program (and its potential reduced space consumption) as independently as possible of the actual GC implementation and the underlying machine used to run the benchmarks. We now define the related terms.

Definition 2.1.17 (Allocation Time) **Allocation time** *is the number of bytes allocated so far in the program.*

We use allocation time rather than wall-clock time. This is since: (i) GC is typically triggered upon allocation events, and (ii) it is less machine dependent, e.g., gives the same results on machines identical up to their CPU clock rate. Such byte-time measurements are standard for the purpose of evaluating a memory manager (e.g., [64, 1]).

Definition 2.1.18 (Reachability Integral) **Reachability Integral** *is the area under the curve plotted by the size of the reachable objects over (allocation) time*

Definition 2.1.19 (Reachability Footprint) **Reachability footprint** *is the maximum of the curve plotted by the size of the reachable objects over (allocation) time*

There are two approaches we take when computing the space consumption of the program. One is to plot the size of the heap at regular intervals and then measure the time-space product (Definition 2.1.18). The second is to compute the maximum heap size over time (Definition 2.1.19). Both approaches are standard for the purpose of evaluating a memory manager (e.g., [1]).

The reachability integral captures an "average space behavior", while the reachability footprint captures "extreme space behavior". In a similar way, we define other space-related integrals and footprints. For example, the *in-use integral* is the area under the curve plotted by the size of the in-use objects over allocation time, and *in-use footprint* is the maximum of the same curve.

Finally, in order to compute the potential space savings, we compute the ratio between two integrals (or two footprints) reflecting the (potential) space consumption of an application. For example, in Chapter 3 we divide the in-use integral by the reachability integral to compute the potential savings due to drag information. Similarly, we also show there the result of dividing the in-use footprint by the reachability footprint.

2.2 Experimental Methodology

We now discuss the methodology used for obtaining results in Chapter 3, Chapter 4 and Chapter 5, in which we perform dynamic measurements, i.e., profile the run of a program on an input, and then analyze the profiling data. In Section 2.2.1 the benchmark programs we use for the experiments are described, and in Section 2.2.2 we discuss the implementation used to collect the profiling information. The methodology for obtaining other results that do not involve profiling information is discussed later in the thesis (see Chapter 6).

2.2.1 The Benchmark Programs

We report results for the 10 benchmarks shown in Table 2.1 in Chapter 3, Chapter 4, and Chapter 5. In Chapter 6 we consider a different set of benchmarks, due to scalability issues. The second and third columns of Table 2.1 show the total number of application classes and the total number of application source code statements, respectively. As shown in Table 2.1 we concentrate on sequential programs, although our suggested program transformations are applicable also for multi-threaded programs. We run the benchmarks with a "typical" input, and in some cases where an investigation of alternative execution paths of the benchmark program is necessary we run the benchmarks with an "alternative" input. The number of JDK classes and general SPEC classes shared by all SPECjvm98 benchmarks are not included in the numbers shown in Table 2.1. There are 32 general SPEC classes having total of 3173 source code statements.

We employed 5 of the benchmarks from the SPECjvm98 benchmark suite [62] and excluded two of the benchmarks. We use the original SPECjvm98 input (*size 100*) to run the benchmarks. `javac` is a java compiler compiling a large file multiple times. `jack` is a parser generator generating multiple copies of itself from an input file describing grammar and actions of `jack`. Interestingly, this periodical behavior of `javac` and `jack` is reflected in their memory behavior shown in Fig. 3.2. `raytrace` is a raytracer that works on a scene depicting a dinosaur. The input model is 340KB in size. `db` performs multiple database functions on memory resident database of names, addresses and phone numbers. `jess` is the Java Expert Shell System based on NASA's CLIPS expert shell system. The benchmark workload solves a set of puzzles commonly used with CLIPS. We did not consider `compress` or `mpegaudio` because they do not use significant amounts of heap memory.

Turning to the other benchmarks, `juru` and `analyzer` are internal IBM tools. `euler` and `mc` were taken from Java Grande benchmark suite [34]. Finally, `tvla` is a three-valued-logic analysis engine framework for static analysis [41]. We now describe these benchmarks and their input used to run the benchmarks.

`juru` performs web indexing. We run it to index several hundred input files. `analyzer` performs mutability analysis of Java files. We run it on the (transitive closure of the) `java.lang` source files of the JDK. `euler` is a Euler equations solver. It solves time-dependent Euler equations for flow in a channel with a "bump" on one of the walls. The solution is iterated for 200 time steps. `mc` performs financial simulation using Monte Carlo techniques to price products derived from the price of an underlying asset. The code generates N sample time series with the same mean and fluctuation as a series of historical data. `tvla` performs static analysis of programs. We use `tvla` to apply shape analysis [52] to analyze a program that merges two singly-linked lists. The analysis results prove that the merge program indeed outputs a singly-linked list as expected.

2.2.2 JVM Instrumentation

Our profiling information is gathered by instrumenting the JVM to record events of interest. An alternative approach to JVM instrumentation is bytecode instrumentation, where the program code is instrumented to record the events of interest. In Chapter 4 we describe our experience with bytecode in-

Benchmark	Class	Stmts	Short Description
javac	176	12345	java compiler
jack	56	5106	parser generator
raytrace	25	1479	raytracer of a picture
db	3	512	database simulation
jess	151	4567	expert system shell
euler	5	726	Euler equations solver
mc	15	880	financial simulation
analyzer	258	35489	mutability analyzer
juru	38	2505	web indexing
tvla	218	25264	static analysis framework

Table 2.1: The benchmark programs.

strumentation (see Section 4.5). The benefit of bytecode rewriting is portability, while the main benefits of using JVM instrumentation are (i) it is easier to profile native code, i.e., code of libraries not written in Java, and (ii) it gives more control on the execution of the program, e.g., it allows the forcing of GC at regular intervals.

We use JVM instrumentation to gather profiling information for several algorithms presented in later chapters. In the following section, we describe the principles of the JVM instrumentation we use. Some details, which are algorithm-specific are described later in the text, when the algorithm is explained.

Recorded Information

The instrumented JVM is based on Sun's JVM 1.2 (aka classic JVM) [36]. Its memory system uses indirect pointers to objects ("handles"); thus, objects can be relocated easily during GC.

We attach a trailer to every object to keep track of our profiling information. We do not count the space taken for this trailer in our data. The information in an object's trailer is written to a log file upon reclamation of the object or upon program termination. An object's trailer fields include its creation time, its last use time, its length in bytes, its allocation site, and mark bits to keep track of root reachability. The length includes the handle, the header and the alignment (i.e., the bytes that were skipped in order to allocate the object on an 8 byte boundary), but excludes the trailer.

Profiling Information

Here is the set of events we profile. In later chapters we describe for every profiling-based algorithm, which of the events in this set it actually needs.

Object Creation Occurs upon allocation of an object (e.g., via `new` bytecode).

Object Use The following events constitute an object *use*: (1) getting field information (e.g., via `getfield` bytecode), (2) setting field information (e.g., via `putfield` bytecode), (3) invoking a method on that object (e.g., via `invokevirtual` bytecode) (4) entering or exiting a monitor on that object (via `monitorenter`, `monitorexit` bytecodes) and (5) derefencing a handle to that object.

Reference Use The following events constitute a reference *use*: (1) getting reference field information (e.g., via `getfield` bytecode). (2) getting local variable information (e.g., via `aload` bytecode). (3) getting global variable information (e.g., via `getstatic` bytecode).

GC An invocation of a full garbage collection. When an object is freed, we log all of the information collected in its trailer.

Reporting Information

After every 100 KB of allocation we trigger a *deep GC* (a larger interval yields less precise results). A deep GC consists of the following steps: (1) GC, (2) run finalizers for all objects waiting for finalization, (3) GC. Forcing finalization ensures instant reclamation of all unreachable objects and removes a source of non-determinism (since finalization would otherwise occur in a separate thread). When an object is freed, we log all of the information collected in its trailer. When the program terminates, we perform a last deep GC and then we log information for all objects that still remain in the heap.

The rules for the collection of `Class` objects are not the same as for regular objects. Thus, we exclude them and the special objects reachable from them (e.g., *constant pool* strings and per-class security-model objects) from our reports.

Process Reported Information

Depending on the profiling-based algorithm, an analyzer processes the log file to produce the results. The details of the specific analyzers are described together with the description of the profiling-based algorithms.

Chapter 3

Drag Potential

We study the effectiveness of GC algorithms by measuring drag time for objects, i.e., the time difference between the actual collection time of an object and the potential earliest collection time for that object. Specifically, we compare the objects reachable from the root set to the ones that are actually used again. The idea is that GC could reclaim unused objects even if they are reachable from the root set. Thus, we conduct experiments to indicate a kind of upper bound on storage savings that could be achieved. We also try to characterize these objects in order to understand the potential benefits of various static analysis algorithms.

The Java Virtual Machine (JVM) was instrumented to measure objects that are reachable, but not used again, and to characterize these objects. Experimental results are shown for our set of 10 benchmarks including 5 SPECjvm98 benchmarks. The potential memory savings for these benchmarks is 42% on average, ranging from 4% to 72%.

The remainder of this chapter is as follows. Section 3.1 discusses drag memory. In Section 3.2 we describe the framework of our experiment. Finally, Section 3.3 gives our experimental results.

3.1 Introduction

We study the effectiveness of GC algorithms by measuring the drag time for objects. Our measurements provide an upper bound on the storage savings over current GC algorithms that could be achieved. The bound is not tight since it is not clear that all such dragged objects could be identified by automatic means. However, a small value for the upper bound would indicate that the GC reclaims unused memory in a timely fashion.

3.1.1 Measuring Dragged Objects

We instrumented Sun's JDK 1.2 in order to measure dragged objects. Specifically, we record at regular intervals (1) the reachable objects, i.e., the objects reachable at the end of the interval and (2) the in-use objects, i.e., the objects that are also used subsequent to the interval. As discussed in Section 2.1.5 we compare the reachability integral and the in-use integral, and we also compare the maximum reachable heap size and the maximum in-use heap size. The ratio between the reachability integral and the in-use integral captures the average space savings, and the ratio between the reachability footprint and the in-use footprint captures the potential for savings in the maximum heap size.

The measurements were conducted on our set of 10 benchmarks. The differences between the integrals for the reachable and the in-use objects is 42% on average, and ranges from 4% to 72%. The differences between the reachable footprint and the in-use footprint is 36% on average, and ranges from 4% to 62%.

Our drag measurements can also be used to understand memory allocation behavior (as done in [50]). We can track memory leaks and tune performance by inspecting the dragged objects (see Chapter 4). Other tools for memory profiling [54, 46] show the heap configuration and allocation frequency; they also help in tracking memory leaks by allowing the heap to be inspected at points during the execution of the program. As noted in [54], tracking memory leaks by inspecting dragged objects is orthogonal to tracking leaks by inspecting the heap during the course of execution. This is since measurements of drag are based on future information, i.e., the future use of objects, while heap inspection requires history information, i.e., the current heap paths in the run.

3.1.2 Characterizing Dragged Objects

Our experiments indicate that for some benchmarks, such as `jack`, most dragged objects are also reachable from roots, which are not local variables. For these benchmarks, techniques such as live-precise GC [5, 65, 1] which are based on liveness analysis of local variables, should have a small impact. Our experiments also indicate that most of these objects are reachable from non-private fields; dealing with this case would require analysis of the entire program.

Another interesting fact that we found is that dragged objects exist along very deep heap paths (e.g., a path length of 1231 for the `javac` benchmark). Motivated by that, in Chapter 6 we analyze heap paths, e.g., through shape analysis [17, 52], to reclaim more memory.

Finally, it should be noted that in some programs it might be hard if not impossible for any static analysis algorithm to identify all dragged objects and to realize the space savings potential. For example, in the `db` benchmark a random access database is maintained. Thus, static analysis of such a database is expected to conservatively determine that all parts of the database are in use. In Chapter 5 we perform dynamic measurements that allow tighter bounds by considering a feasible GC interface (see Section 5.3). Indeed, our results there indicate a negligible potential for space savings for the `db` benchmark.

Root Kind	Short Description
Java stack root	The Java stack is used for maintaining the execution frames of Java methods. A Java stack root is a memory slot of reference type.
native stack root	The native stack is used for maintaining the execution frames of native (usually C) code. A native stack root is a memory slot of reference type.
static variable root	A static reference variable.
other roots	For example, Java Native Interface (JNI) global references (used by native code to access Java objects).

Table 3.1: The major kinds of roots recorded.

3.2 The Scene

3.2.1 Concept

At a high level we associate a time field with every object in order to record drag information. On every use of an object, we record the current time in its field. Thus, the field always holds the time at which the object was last used. In order to approximate the time at which an object becomes unreachable, we frequently trigger a full GC; thus when an object is collected, its drag time is calculated as the difference between the approximated time it becomes unreachable (i.e., its collection time) and the time it was last used. We measure time in bytes allocated since the beginning of program execution, assuming that all uses of an object in the interval between consecutive garbage collection cycles are performed at the beginning of the interval.

For every dragged object we also report the kind of its roots. The major kinds of roots are shown in Table 3.1. This experiment may indicate the potential benefits of performing liveness analysis for some or all of these root kinds, specifically the benefit of performing local variable analysis (e.g., [1]). In Chapter 5 we further explore the potential for space savings using liveness information for some or all the root kinds.

In another experiment we measure the minimal distance of a dragged object from a root in the Java stack. This may indicate whether it is important to statically analyze deep heap paths.

3.2.2 Implementation

We follow the implementation described in Section 2.2.2. In order to compute root kind reachability information, we keep special mark bits for an object. These mark bits contain one bit for each kind of root. Object information is updated upon the following events:

Object Creation The creation time, length and allocation site are set.

benchmark	Integral			Footprint		
	in-use	reach	potential savings(%)	in-use	reach	potential savings(%)
javac	1058.19	1644.62	35.66	8.35	9.25	9.74
jack	86.33	216.59	60.14	0.57	1.39	58.81
raytrace	253.73	656.89	61.37	2.39	4.35	45.10
db	497.63	805.31	38.21	7.60	9.60	20.90
jess	108.53	392.97	72.38	0.61	1.41	56.70
euler	1936.73	2148.82	9.87	6.72	7.76	13.40
mc	11423.94	11923.18	4.19	78.93	82.27	4.07
analyzer	276.08	674.09	59.06	1.38	3.59	61.64
juru	55.31	88.25	37.33	0.46	0.88	47.61
tvla	318.12	569.29	44.12	1.51	2.40	37.07
average			42.23			35.50

Table 3.2: Reachability integrals (in MB^2) and maximum reachable heap size (in MB).

Object Use The last use time is set.

GC Before GC we clear the special markbits for all objects[1]. During the phase of GC that marks roots, we set the special markbits for the objects directly reachable from the roots according to the kind of root. At the completion of the trace phase, we perform a special trace phase per root kind in order to propagate the marks according to reachability. During the special trace phase for the Java stack root kind, we also measure the (minimal) distance from the Java stack for each object, which is reachable from the Java stack.

3.3 Experimental Results

We now present our empirical results. In Section 3.3.1 we explain how dragged objects have a significant impact on the space consumption despite the fact that most allocated objects are not dragged objects. Then, in Section 3.3.2 we give the ”high-level” results. We compare the size of the reachable heap to the size of the in-use objects over time, and show a large potential for space savings attributed to dragged objects.

[1]Clearing the special markbits before every GC means that we report only the kind of roots that kept the object reachable in the interval just before it became unreachable. We have found that considering all possible root kinds from the last moment the object is used adds “white noise” to the results. For example, if an object dies at a young age, the native stack slot used to hold the address of the object during allocation may still hold the object’s address.

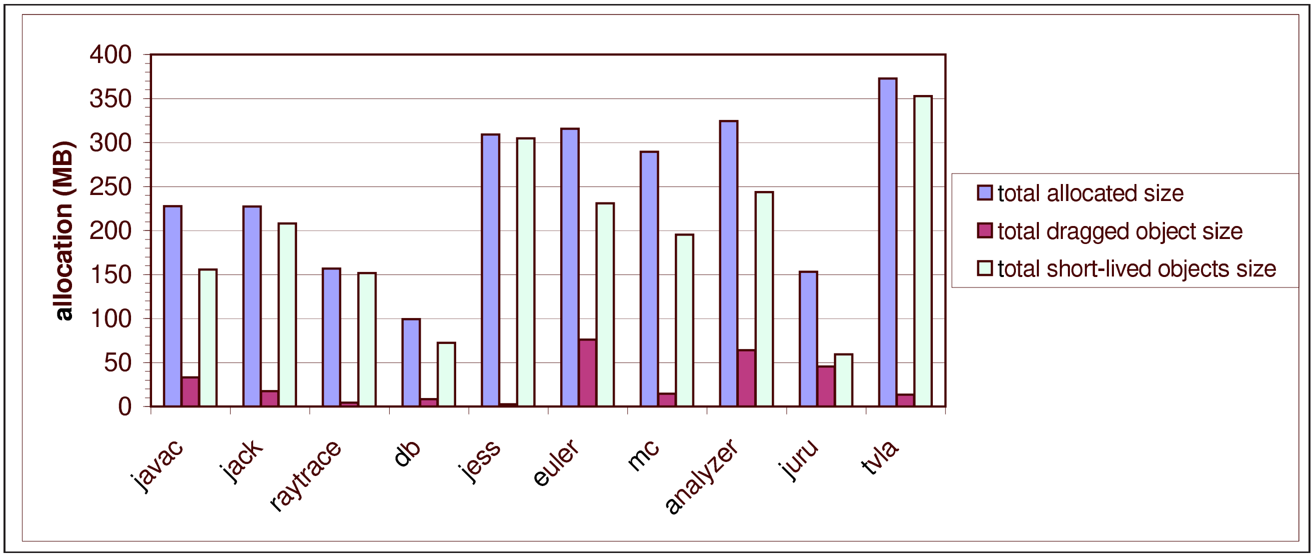


Figure 3.1: Total allocation size, total dragged object size and total short-lived object size.

In order to understand what strategies and information is needed to reduce the drag time for object, we further analyze dragged objects as follows: Section 3.3.3 analyzes dragged objects with respect to their drag time, as for example it may be beneficial to focus on dragged objects with large drag drag. Section 3.3.4 and Section 3.3.5 classify dragged objects by their root kind and their distance from the stack, respectively. This information may hint on the kind of static analysis needed to reduce drag time for objects. For example, a dragged object on a long heap path may require static analysis techniques such as shape analysis.

3.3.1 Dragged Object Size

Fig. 3.1 shows total number of bytes allocated and total number of bytes attributed to dragged objects for each benchmark. We define a *short-lived object* to be an object that is allocated and becomes unreachable in the same measurement interval; thus, there cannot be any memory savings for theses objects. Short-lived objects are excluded from the subsequent results, since these objects are not reachable at the sampling points.

Although in four benchmark less than 10% of the total object allocation are dragged objects, the total size of the dragged objects is large compared to the total size of the reachable objects, (as shown in Section 3.3.2), so there is a potential for a large savings in memory.

3.3.2 Reachable vs. In-Use Objects

Table 3.2 compares the reachable object size with the in-use integral, and the maximum reachable heap size with the maximum heap size when considering just the in-use objects. The potential for savings is

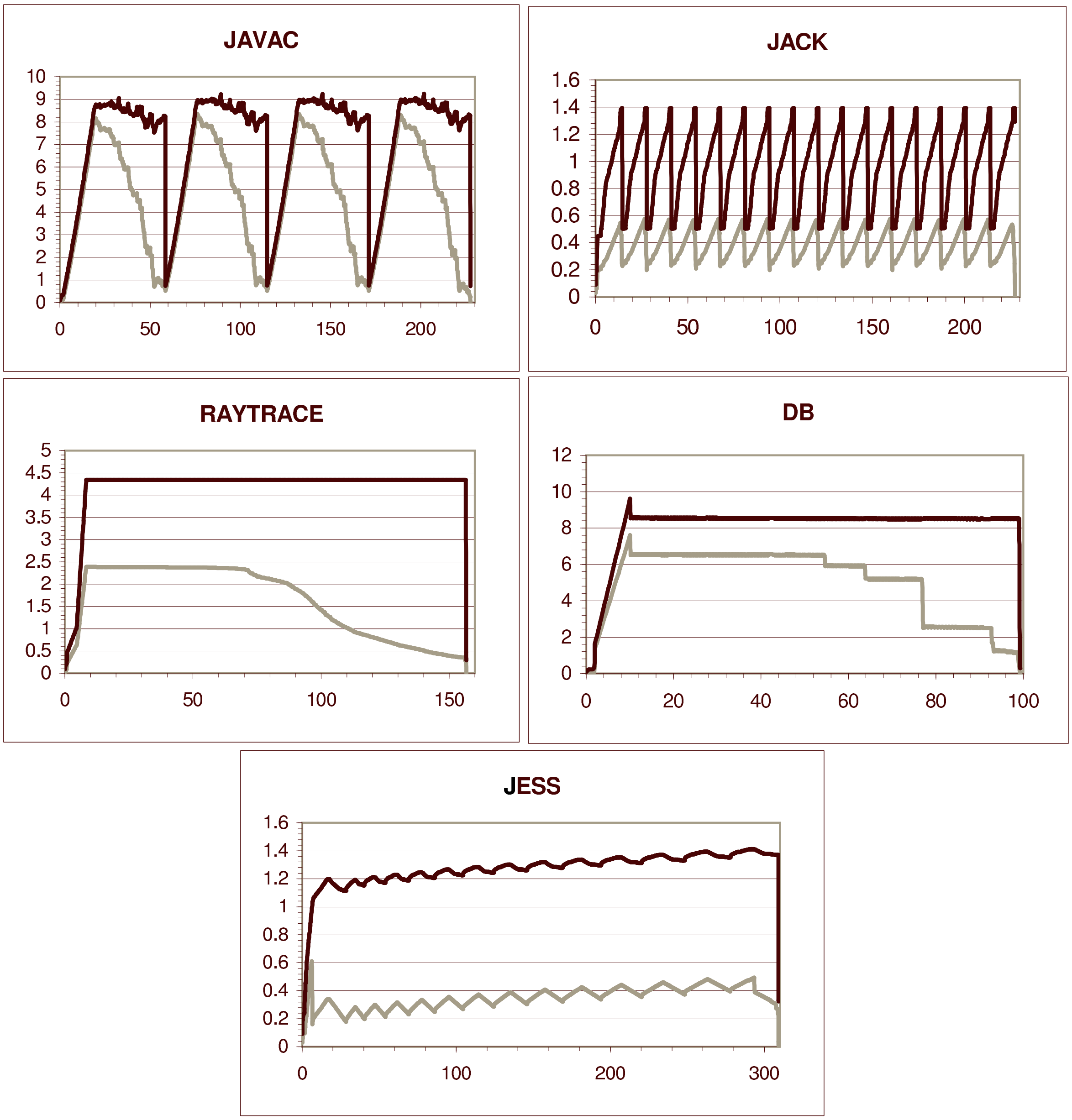


Figure 3.2: Reachable object size vs. in-use object size for SPECjvm98 benchmarks.

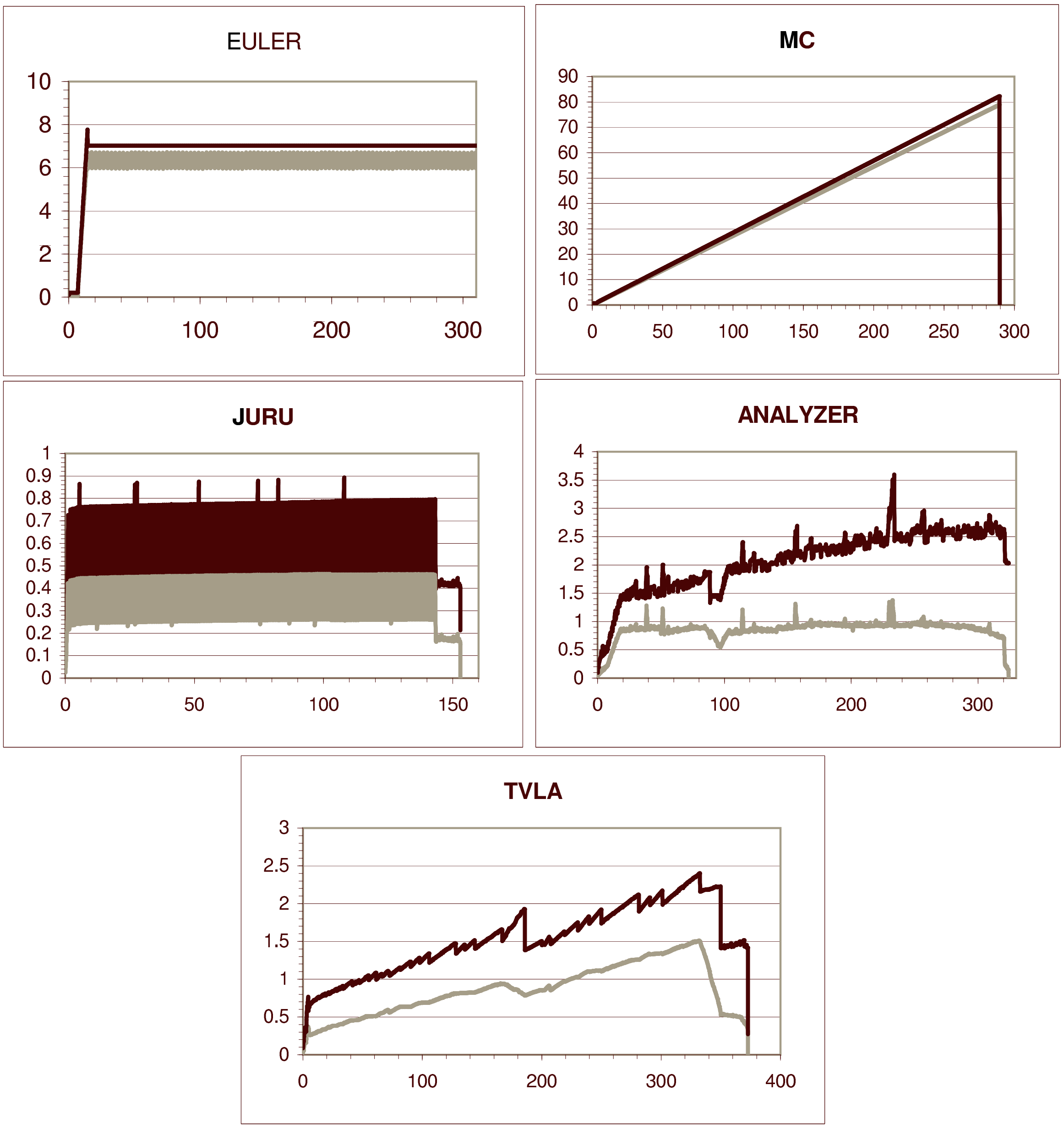


Figure 3.3: Reachable object size vs. in-use object size for non-SPECjvm98 benchmarks.

Pct.	javac	db	jack	raytrace	jess	euler	mc	analyzer	juru	tvla
90%	0.9	0.1	0.3	0.9	0.1	0.1	0.1	0.1	0.1	0.1
80%	6.5	0.2	33.1	6.3	0.1	0.3	0.1	0.1	0.1	0.1
70%	8.7	0.4	51.9	22.0	1.5	0.7	0.1	0.1	0.1	0.1
60%	11.6	0.9	60.5	22.0	6.4	0.9	0.1	0.1	0.1	0.2
50%	13.4	1.2	75.3	22.0	12.8	1.3	0.1	0.2	0.1	0.2
40%	18.1	4.7	148.5	35.2	18.4	1.5	0.1	0.2	0.1	0.5
30%	21.1	8.8	149.6	44.5	302.5	1.9	0.1	0.3	0.1	1.6
20%	26.0	9.7	150.7	90.5	302.5	2.1	0.1	0.9	0.1	5.6
10%	35.9	10.6	151.7	94.4	307.5	2.5	0.1	3.2	0.1	13.5

Table 3.3: Distribution of total dragged object size with respect to drag time (expressed in MB).

42% on the average for the reachable integral, and 36% on the average for the maximum heap size.

Fig. 3.2 and Fig. 3.3 compare the reachable object size and the in-use object size over time. We now provide details on the memory behavior of the benchmarks. We see that the `javac` benchmark operates in several cycles. These cycles correspond to the fact that `javac` benchmark compiles a large file several times (see Section 2.2.1). For every cycle there is an initial phase of allocating memory, in which almost every reachable object is in-use. As the program continues, the difference in sizes between the reachable and the in-use objects increases until the end of the cycle. At that time the memory consumption of the program drops at once, and another cycle begins.

For the `db` benchmark, after an initial phase of allocating memory, the difference between the reachable and the in-use objects is constant until the program allocates 54.52 MB. After that point the size of the reachable objects remains more or less constant, but fewer objects are used, so that the difference between the reachable and in-use object sizes increases. `jack` behaves similarly to `javac`, and `raytrace` behaves similarly to `db`. The cycles in `jack` correspond to the fact the `jack` is a parser generator generating multiple copies of itself (see Section 2.2.1). For `jess` the difference between the reachable and in-use object sizes is constant at around 0.92 MB. This is further explained in Section 3.3.3.

For `euler` the size of the reachable objects behaves similarly to `db`. However, the size of the in-use objects grows and shrinks in cycles of small width and small height. This is further explained in Section 3.3.3. For `mc` the size of the reachable heap keep growing until the end of the program, and the size of the in-use objects behaves similarly. This suggests that almost all program objects are traversed when the program is about to terminate. `juru` operates in cycles and behaves similarly to `jack`. `analyzer` and `tvla` behaves quite similarly to `jess`.

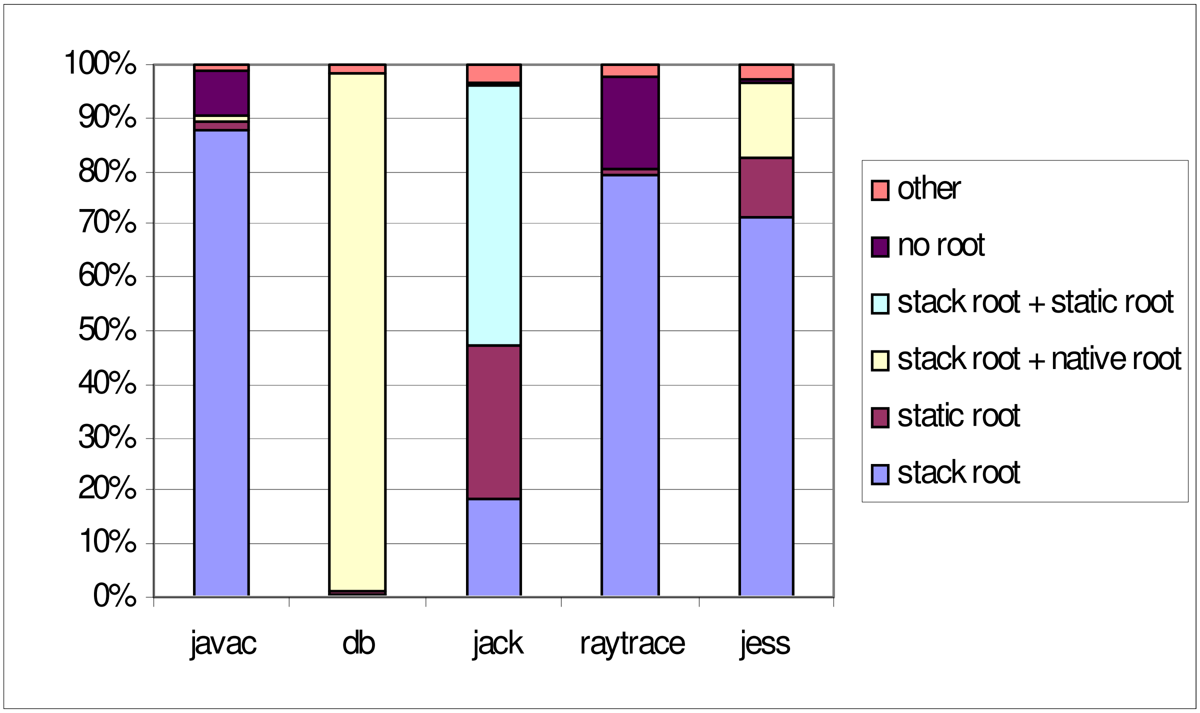


Figure 3.4: Distribution of dragged objects by root kind.

3.3.3 Drag Time

Table 3.3 shows the distribution of dragged object size with respect to drag time. We further discuss some of the results. For `jess` 30% of the dragged object size (which is 0.72 MB) has a drag time of at least 302.5 MB; this is nearly the lifetime of the program. Thus, these objects are used for their last time near the beginning of the program, yet they are not collected until the program terminates. As noted before, there is also a constant difference of 0.92 MB between the reachable and in-use objects. These two facts explain how a seemingly insignificant amount of dragged objects (less than 1% of the total allocation size) can result in a 72% savings in the reachable integral (i.e., 72% of average space savings). Similarly, for `raytrace` we see that close to 40% of the dragged objects have drag time, which is nearly the lifetime of the program. In Chapter 4 we focus on these objects to obtain an actual 33% space savings in `raytrace`.

`euler`, `mc`, `analyzer`, and `juru` have objects with small drag time. In `euler` and `mc` it leads to small potential in space savings. For `analyzer` and `juru` the potential is still large due to relatively small reachable heap size.

Benchmark Program	Average Distance	Maximum Distance
javac	20.47	1231
jack	6.78	25
raytrace	22.19	1412
db	6.24	20
jess	6.72	28

Table 3.4: Distance of dragged objects from Java stack.

3.3.4 Distribution of Dragged Objects by Root Kind

Fig. 3.4 shows for the SPECjvm98 benchmarks the distribution of the dragged objects by the types of the roots that keep them reachable. In most of the SPECjvm98 benchmarks (3 out of 5), most of the dragged objects have just one kind of root, the Java stack. For example, in `javac` close to 88% of the dragged objects are reachable solely through the Java stack.

For `db`, close to 97% of the dragged objects are reachable from the Java stack, as well as from the native stack. `db` maintains a database; all objects in the database are reachable from the database root object of type `spec.benchmarks._209_db.Database`. This database root object is directly referenced from the native stack for the entire course of the program; thus, all database objects are considered as native stack reachable. In this case, live-precise GC should have little impact.

For the `jack` benchmark, close to 30% of the dragged objects are reachable only from static variables. Another 50% of the dragged objects are reachable from both the Java stack and from static variables. Investigating the bytecode we found that the relevant static fields have `public` access; thus, in order to perform liveness analysis for these global variables the whole program would need to be analyzed. This could be very expensive and is difficult to apply to Java.

3.3.5 Distance from the Stack

Table 3.4 shows the average distance and maximum distance of dragged objects from the Java stack for the SPECjvm98 benchmarks. For 3 out of 5 benchmarks the average distance for dragged objects is around 6.5. This seems like a reasonable number for object-oriented languages. `javac` and `raytrace` benchmarks have a much larger average distance from the Java stack. This is due to the presence of long linked lists (of length 1231 and length 1412, respectively) containing dragged objects. In `javac` this linked list represents the bytecodes of a method being compiled, while in `raytrace` this linked list represents the vertices of a complex polygon. Using shape analysis to analyze these paths may provide a memory savings. In Chapter 6 we rely on shape analysis to save memory along arbitrary heap paths.

Chapter 4

A Drag Tool

In Chapter 3 we measured the drag time for objects and tried to characterize the dragged objects. The results showed a potential for space savings from 4% to 72% for our set of benchmarks if objects could be collected at their last use. We now describe a prototype tool based on drag measurements that is aimed to help in actual reduction of space.

Our tool measures the drag time of objects for a Java application. Drag time indicates potential savings. Then the tool sorts the allocation sites in the application source code according to the accumulated potential space saving for the objects allocated at the sites. A programmer can investigate the source code surrounding the sites with the highest potential savings to find opportunities for code rewriting that could save space. Our experience shows that in many cases simple code rewriting leads to actual space savings and in some cases also to improvements in program runtime.

Experimental results using the tool and manually rewriting code show average space savings of 14% for our set of Java benchmarks. We have also classified the program transformations that we have used and argue that in many cases improvements can be achieved by an optimizing compiler assuming whole code availability.

In follow-up research done in collaboration with M. Pan as part of his Master's thesis [45] a further drag tool named *HUP* was built. HUP is a portable tool based on bytecode instrumentation and on a standard profiling interface (JVMPI), while the tool reported here is based on JVM instrumentation. HUP is available at *hup.sourceforge.net*. In Section 4.5 we report our experience with HUP for measuring *lag* information, the difference between the time an object is allocated and the time the object is first used.

The remainder of this chapter is as follows. Section 4.2 describes our prototype tool and explains how drag information could be used for space savings. Section 4.3 presents the framework of the experiment. In Section 4.4 empirical results are discussed. Finally, Section 4.5 discusses our experience with the HUP tool.

4.1 Introduction

Our goal is reducing the size of the reachable heap at every sampling point, by reclaiming reachable objects that are not in-use. Thus, our prototype tool obtains drag information for objects as described in Chapter 3. Then, in an offline phase, dragged objects are partitioned according to their allocation site. For each allocation site we sum the *drag space-time product* of every dragged object. The drag space-time product, or *drag* for short, is the product of the size of the object and the time the object is reachable but not in-use. Allocation sites having a large drag suggest a potential for significant space savings. Therefore, our tool displays allocation sites according to their drag.

Our profiler tool can be used either directly by a programmer or to produce input for a profile-based optimizer. Currently, we use the tool to direct manual code inspection of the allocation sites having a large drag. Examining these sites we have found that three kinds of simple program transformations have been effective in achieving space savings: (i) explicitly assigning null to a dead reference to an object, (ii) removal of dead code, and (iii) lazy allocation of objects. These program transformations cannot harm the space consumption of a program, and in most cases save space. In Chapter 6 we take a further step in the process of automating the program transformations, by developing a static algorithm for assigning null to dead heap references.

The tool was applied to our set of benchmarks. Although we were not familiar with the code of these benchmarks, it took us just a few hours to understand the results produced by the tool and to rewrite the code. Also, only few lines of code had to be rewritten. Of course, we believe that an application programmer using our tool on his own code could do a better job, both in terms of speed (of rewriting) and more importantly in terms of space (reduction of drag).

Code rewriting for the benchmarks we considered reduces the total drag by 45% on average, leading to an average space saving of 14%. In some cases the runtime cost is also reduced by 1%-2%. This is mainly due to: (i) GC is invoked less frequently and (ii) allocation and initialization are avoided for objects that are never used.

4.2 The Tool

The heap-profiling tool consists of two phases: (i) an online phase, in which we measure the dragged objects, and (ii) an offline phase, in which we analyze the results and produce a list of allocation sites of dragged objects sorted by their potential space savings.

4.2.1 Drag Profiling

In the first online phase, an instrumented JVM runs a Java application and outputs information on dragged objects to a file. This is the same instrumentation described in Section 2.2.2. For the purpose of dragged object profiling, object information is updated upon the following events:

Object Creation The creation time, length and allocation site are set.

Object Use The last use time and last line of code at which the object is used (hereafter, called *last use site*) are set.

4.2.2 Drag Reporting

The second offline phase analyzes the file, producing a list of allocation sites sorted by their potential for drag reduction. In particular, we partition the dragged objects according to their allocation site, and for each allocation site we sum the drag of every dragged object. Allocation sites having a large drag suggest a potential for significant space savings.

We found that it is best to have information regarding the *nested allocation site* of a dragged object, i.e., the call chain leading to its allocation. Thus, we partition the dragged objects into groups according to their nested allocation site, associating with each group the sum of the drag for all objects allocated at its site.

Sometimes an allocation site is used in many contexts and a large drag may be distributed among several smaller drag groups when partitioning solely according to nested allocation site. Thus, we also do a coarse-grained partition according to the allocation site.

Sometimes the last-use site, i.e., the site at which a dragged object was last used, may provide a hint at the code rewriting strategy best-suited for reducing its drag. For example, if a dragged object remains reachable due to dead references, the last-use site may hint at the program point where a reference to that object becomes dead. Thus, we also partition dragged objects according to nested allocation site and last-use site. In the `Euler` benchmark (see Table 2.1), the last-use site is used to determine which reference variable prevents an object from being reclaimed.

A special case for a dragged object is an object that is never used. We partition these *never-used* objects according to nested allocation site and according to allocation site. Our experience so far suggests that a large drag caused by never-used objects is a “sure bet” for code rewriting.

4.3 Applying the Tool

We applied the drag reduction tool to our set of Java programs to determine the allocation sites in those programs contributing the most to the drag. Then we carefully analyzed the source code around those sites in order to find program transformations that reduce the drag. Interestingly, we found that a very shallow understanding of the program suffices to reduce a significant part of the drag. Moreover, we believe that future optimizing compilers may be able to automatically conduct these transformations.

In this section we describe the drag reducing program transformations and how we used the tool to direct the choice of the proper transformations.

4.3.1 Reducing the Drag

The code was manually investigated in order to determine the reasons for drag. Since we were unfamiliar with the code of the sample programs, we employed JAN [47], a tool for analyzing and understanding Java programs. We used two kinds of information provided by JAN: (i) the *class hierarchy graph*, and (ii) the *program call graph*. This kind of information is also provided by many other Java compilers and analysis tools.

We used the class hierarchy graph for accelerating source browsing, e.g., locating overloaded methods. We used the program call graph to check the applicability and validity of program transformations. For example, assigning null to a dead reference variable requires inspection of every possible use of that variable. Using the program call graph, we can check that some uses of the variable, which appear in the source code, do not actually occur at runtime, e.g., if the graph shows that the method is never invoked.

Source transformations were applied only after a thorough inspection of the source code and validation that the transformation was applicable using the program call graph. We also checked that the original and revised benchmarks produce identical results on several inputs. In order to measure the effect of code rewriting on the running times of the programs, the original and revised versions were invoked and their running times also compared.

The code of the benchmarks as well as selected classes of the JDK itself were rewritten using the code rewriting strategies discussed below in Section 4.3.2. The tool was reapplied to the revised code in order to measure the resulting drag, and determine the actual space savings. Sometimes, the results revealed more opportunities for drag reduction; in that case, another cycle of code rewriting and applying the tool took place.

4.3.2 Code Rewriting Strategies

Surprisingly enough (at least for us), it appears that a few simple code rewriting techniques suffice for reducing much of the drag. The first technique is simply assigning the null value to a reference that is no longer in use, i.e., a dead reference (see Definition 2.1.14). This reference may be a local variable, an instance field, a static field, or an element within an array of references. The second technique is removing dead code, i.e., code that has no effect on the result of the program. Our main interest is in dead code that produces dragged objects. The third technique is delaying the allocation of an object until its first use; thus, avoiding memory consumption for objects that are never used.

Assigning Null

In many cases a dragged object remains reachable after its last use due to a dead reference. We inspect the code for all possible uses, identify the places where the reference stops being used, and insert a statement assigning null to that reference. The details depend on the kind of variable containing the reference.

- For a local reference variable we inspect the containing method for possible uses.
- For an instance field we inspect the code according to visibility modifiers of the field, e.g., for a private field we inspect only the code of the containing class. Sometimes the program call graph is used to determine the context for a possible use. As noted earlier, we use program call graph information to invalidate possible uses in unreachable methods.
- For arrays of objects we have found cases where an element becomes dead. For almost all cases the array is being used to implement a data type similar to a Java vector and an element is being removed from the array [55].

Our tool provides information about the last line of code at which an object is used. This helps to find the place in the code where the object is no longer in use.

Dead Code Removal

Dead code [43] does not affect the result of the program. Using a feature of the tool showing objects that are allocated but never used, we find allocation sites where all objects are never-used. These never-used objects may be referenced by local variables, instance variables and array elements of type reference. We eliminate the allocation of these objects. This optimization is not always possible as it also removes the invocation of the constructors for these objects. We must guarantee that the constructor is the only code that references the object and that the constructor has no influence on the rest of the program, e.g., it does not update other objects or static variables and it cannot throw an exception for which there may be a handler in the surrounding code.

Lazy Allocation

Using the same feature of the tool showing objects that are allocated but never used, we find allocation sites that produce many never-used objects. If these objects are big contributors to the drag, then we change the code to allocate them lazily. In particular, we eliminate the original allocation of the object and the variable that would have referenced the object remains null or is assigned null. Then, at every possible first use of the object, there is a test to check whether the variable is null. If so, the object is allocated. We find possible first uses in the source code by employing the program call graph.

In general determining that postponing the allocation of an object does not change program semantics is hard. For example, the object could be shared by several threads. Thus, we only employ this transformation for the easy cases. In particular, the constructor may not depend on program state, e.g., it must have no parameters or parameters that are constant, and it may not read program state (for example, access a static variable) in its body. Also, the constructor may not throw exceptions for which there may be handlers in the surrounding code. For all of the objects for which we applied this transformation, the only possible exception was `java.lang.OutOfMemoryError` for the allocation itself; thus, we only had to check that there were no handlers for `java.lang.OutOfMemoryError` in the program.

Lazy allocation may add runtime overhead for the checks at every possible first use. Thus, there is a risk that it could be detrimental for some program inputs, and care needs to be taken to apply it only when most of the objects allocated at the dragged allocation site are never-used. Nevertheless, this transformation has a potential for reducing space and drag. We used it for just one of the benchmarks, `jack`. Interestingly, later versions of `jack` (now called `javacc` [33]) employ a similar rewriting for lazy allocation of objects.

4.3.3 Putting It All Together

Given the output of the drag-profiling tool we need to determine which program transformation to apply. Our experience shows that the first step is to find the method and the reference variable on which attention should be focused. We choose a nested allocation site with high drag. The bottom level is likely to be an allocation site in JDK or other library code, e.g., allocating a character array in `java.util.String`. We follow the call chain upwards looking for the first place in application code where a reference to the allocated object (or to an object containing the allocated object) is stored in a variable. We call this place the *anchor allocation site*. We call the variable the *anchor variable*.

Our experience suggests that the second step is to employ the output of the tool to investigate the lifetime characteristics of dragged objects at the anchor allocation site. In particular, the tool outputs the drag size due to objects allocated at a given anchor allocation site without any recorded use (the last use time is zero). The tool also partitions the dragged objects at that anchor allocation site according to their drag time, in-use time, and collection time. We have identified the following patterns of behavior:

1. All of the drag at the site is due to objects that are never-used. In some cases the only use of an object may be in its constructor and its in-use time is very short; we also consider these as objects that were never used.

2. Most of dragged objects at the site are never-used.

3. Most of dragged objects at the site have a large drag.

4. The variance of the drag for the objects at the site is high.

Based on the lifetime pattern at the anchor allocation site, one of the drag reducing program transformations might be applicable. The first pattern suggests the dead code removal transformation. The second pattern suggests the lazy allocation transformation. The third pattern suggests that assigning null (to a dead reference) is the most applicable transformation.

The fourth pattern suggests that there may be no program transformation that might help. For example there may be a large repository of objects as in the `db` benchmark. A query on the repository leads to a use of an object. However, each query accesses only a small number of objects and the queries are spread out over the whole application. Nevertheless the repository and all objects in it need to be kept as the exact queries cannot be predicted in advance.

These patterns do not cover all of the possibilities. However, most repeated more than once and we found them to be useful for finding applicable program transformations. Below we provide examples from the benchmark programs for each of the transformations and try to relate them to the patterns. Sometimes, when applying these transformations, the value of the anchor variable is assigned to other variables. These variables also have to be considered when applying the transformation.

Assigning Null

In `juru` the largest drag for an allocation site is 12.4MB^2. Character arrays of 100K elements are allocated at this site and assigned to a local variable. Each of these arrays is in-use for 200KB of allocation and then in-drag for another 200KB until it becomes unreachable. Assigning null to this local variable after its last use eliminates this drag and leads to a 38% reduction in total drag for `juru`. The drag time of these objects is short, but the objects are large so these objects contribute significantly to the drag space-time product. This fits the third pattern.

Dead Code Removal

In the `raytrace` benchmark there are 17 allocation sites with the same behavior: an object is allocated and assigned to an array element; the object's last use occurs during its initialization, which is done in its constructor. Thus, all objects allocated at these sites are considered never-used. Each of these allocation sites contributes 13.5MB^2 to the drag. This behavior fits the first pattern and we apply the dead code removal transformation.

With the help of the program call graph, we verify that these objects referenced by the array elements are never accessed outside their constructors. We also verify that the constructors do not have any side effects. Thus, the code for the allocation of these objects can be removed. This leads to a 44% reduction in total drag.

Lazy Allocation

In the `jack` benchmark, the three allocation sites producing the largest drag are all in the same constructor. More than 97% of the drag for these three allocation sites is due to objects that are never-used. This behavior fits the second pattern and we apply the lazy allocation transformation.

We turn to the constructor's code. One `Vector` and two `HashTable` objects are allocated at the allocation sites. References to each of these data structures are assigned to instance fields. These instance fields have package visibility, i.e., they are visible to all package members. Thus, we scan the package for possible uses. It turns out that all uses of these instance fields occur in their containing class. We eliminate the allocations and before every possible first use of one of the instance fields, we add a test to check whether the allocation has already been done. If not, the allocation is performed.

4.4 Results

First, we discuss our rewritings to the benchmark programs. Then, we present the results of applying our heap profiling tool to the benchmark applications: the savings in space and the savings in time.

4.4.1 Rewritings

Table 4.1 summarizes the rewritings applied to our set of benchmarks. For every benchmark we show the applicable rewriting techniques (*Rewriting Strategy* column), and the drag saving due a rewriting (*Drag Saving* column). Column *Reference Kinds* shows the kind of references for which a rewriting strategy was applied. Reference kind information gives initial insight to the code that needs to be inspected before applying a rewriting.

Code removal is applied to three benchmarks (`raytrace, jess`, and `mc`). For example, in `raytrace` we notice that some of the coordinates in an octagon face class are created but never used. This code inefficiency is due to modularity. Thus, we remove the code allocating these coordinates; this removal leads to 44% drag savings.

Lazy allocation is applied to just one benchmark (`jack`). In a class maintaining an automaton state, two hashtables, and one vector are lazily allocated, so their allocation takes place only when these are first accessed. As noted in Section 4.3.2, later versions of `jack` (now called `javacc`) employ a similar rewriting for lazy allocation of objects.

Finally, assign null is applied to all benchmarks except `jack`. This fact motivates our work described in Chapter 6 where we further explore how to automate the process of assigning null to dead references. For example, in `javac` we find for a parser class that assigns null to a statically dead reference field that maintains a large string yields 10% of drag savings. In Chapter 6 we statically analyze a slice of the code of `javac` to infer the same assign null automatically.

Benchmark Program	**Rewriting Strategy**	**Reference Kinds**	**Drag Saving (%)**
`javac`	assign null	protected	9.95
`jack`	lazy allocation	package	54.49
`raytrace`	code removal	private array	44.39
	assigning null	private	8.63
`jess`	assigning null	private array	1.60
	code removal (JDK rewrite)	public static final	1.00
	code removal	private static	6.57
`euler`	assigning null	package array	75.26
`mc`	code removal	local variable + private	65.07
	assigning null	private array	26.51
`analyzer`	assigning null	local variable + private static	27.81
`juru`	assigning null	local variable	37.65

Table 4.1: Summary of Rewritings.

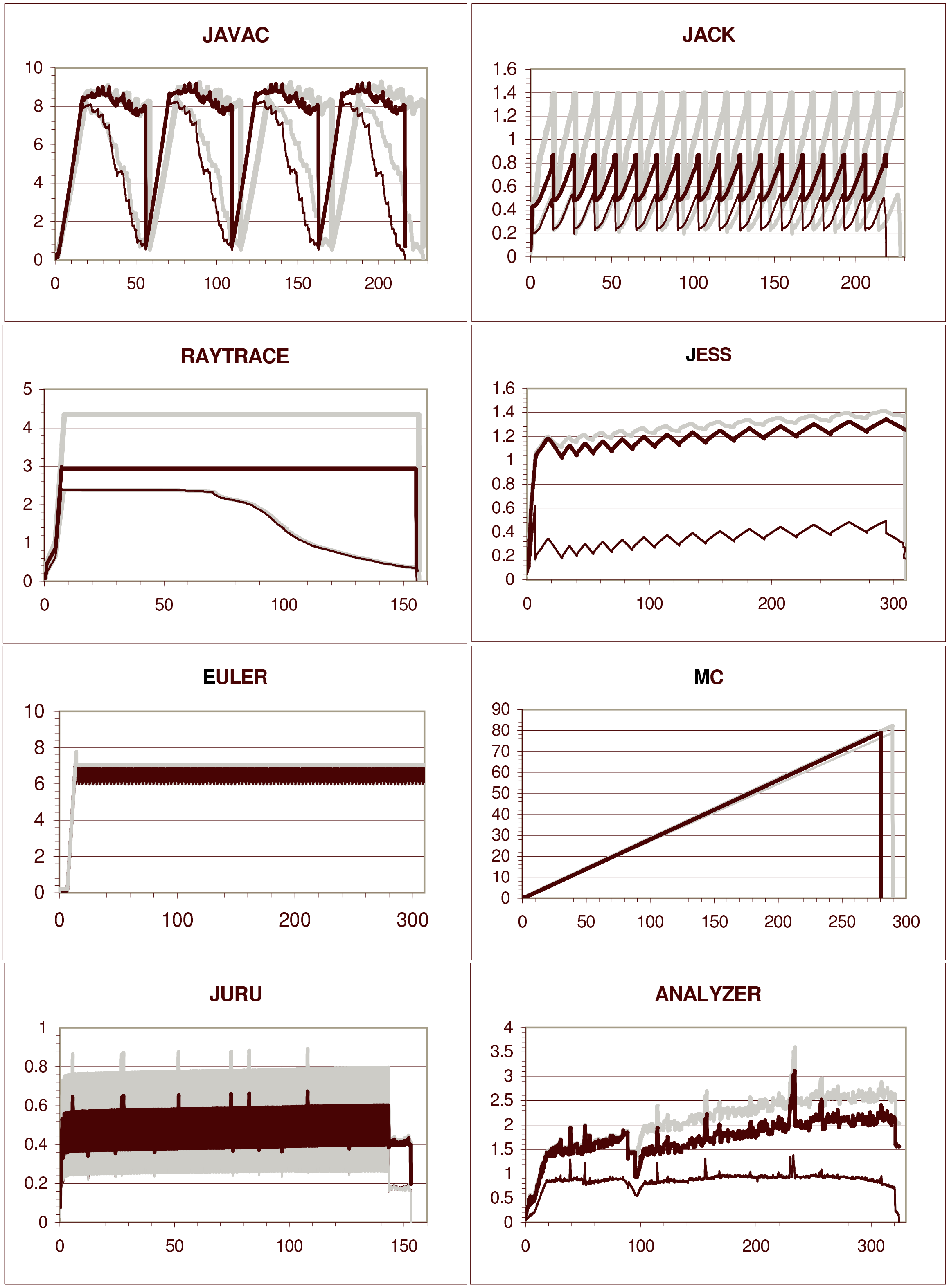


Figure 4.1: Original reachable/in-use heap size vs. revised reachable/in-use heap size. X-axis denotes allocation time in MB. Y-axis denotes size in MB. The thick gray line shows the reachable size and the thin gray line the in-use size before rewriting. The thick black line shows the reachable size and the thin black line the in-use size after rewriting.

4.4.2 Space Savings

For each of the benchmarks Fig. 4.1 shows graphs of the reachable heap size (sum of the sizes of the reachable objects) and the in-use heap size (sum of the sizes of the in-use objects) before and after the rewriting of the code. The thick gray line shows the reachable size and the thin gray line the in-use size before rewriting. The area between these two lines is the initial drag. The thick black line shows the reachable size and the thin black line the in-use size after rewriting. In most cases the thin gray line and the thin black line coincide and cannot be differentiated. The area between the thick gray line and the thick black line is the space that is saved by the source rewriting optimizations.

Looking at the graphs for SPECjvm98 benchmarks, we see that the size of the reachable heap is reduced for `jack` and there is a rather small reduction in the reachable heap size for `javac`. In addition, for both benchmarks, the reachable and in-use object size lines for the optimized version occur "earlier" in the graph than for the original run. This is due to the elimination of some unnecessary allocation.

For `raytrace` the size of the reachable heap is reduced by a constant size of 1.4MB, and the in-use object size remains the same. This is due to the fact that a similar amount allocation of long-lived never-used objects has been eliminated. However, there is practically no shift in the graph since 1.4MB is less than 1% of the total number of bytes allocated during a run.

There is a potential for drag reduction by rewriting some of the JDK code. We demonstrate drag reduction due to JDK rewriting in `jess` benchmark. In particular, we rewrite the code of a JDK converter class to eliminate the references to an alias table after this table is last used, thus enabling earlier collection of that table. In principal, JDK rewriting is applicable for all of the benchmarks. The size of the reachable heap for `jess` is reduced by a constant size, similarly to `raytrace`.

The graph for `db` is not shown. There is no space savings for this benchmark.

Turning to the other benchmarks, for `euler` the size of the reachable heap for the original run has a constant size, because all allocations are done in advance. By assigning null to dead references we were able to reduce most of the drag (75% of it), and the optimized heap size almost coincides with the in-use object size.

In `mc` the size of the reduced reachable heap is even below the size of original in-use object size. This is due to the fact that many allocations are eliminated.

In `juru` there is a constant reduction of 200KB in the reachable heap size. `juru` acts in cycles, with the same reduction on every cycle.

Lastly, for the `analyzer` benchmark the size of the reachable heap is reduced only after allocating the first 98MB in the program. This occurs because objects used for the first part of computation (first 98MB of allocation) are not needed later in the computation.

In Table 4.2, we show measurements of the drag reduction and the total space savings. The original

benchmark	Integral in-use	Integral reach	Integral drag savings(%)	Integral space savings(%)	Footprint in-use	Footprint reach	Footprint footprint savings(%)
javac	1033.58	1561.64	9.95	5.05	8.35	9.20	0.52
jack	77.93	137.21	54.49	36.65	0.53	0.87	37.57
raytrace	252.03	441.40	53.03	32.80	2.41	2.99	31.17
db	497.63	805.31	0	0	7.60	9.60	0
jess	108.54	366.90	9.17	6.63	0.62	1.34	4.94
euler	1932.08	1984.56	75.26	7.64	6.76	6.94	10.61
mc	11022.10	11064.12	91.58	7.20	78.93	79.07	3.89
analyzer	275.64	563.19	27.81	16.49	1.38	3.11	13.55
juru	55.81	76.35	37.65	13.49	0.46	0.67	24.17
average			44.87	14.00			14.05

Table 4.2: Reachability integrals (in MB^2) and maximum reachable heap size (in MB) after code rewriting.

drag is the difference between original reachable and in-use integrals. The amount of drag reduction is the difference between the original reachable integral and reduced reachable integral. Drag savings is computed as the ratio between the amount of reduced drag and the original drag.

The average space savings for all the benchmarks (including `db`) is 14% and the average drag savings is 51%. In `mc` the size of the reduced reachable heap is close to size of the original in-use objects. This leads to 92% savings of drag, since we saved most of original drag.

We also ran each benchmark on an input other than the one initially analyzed by the tool. Results are shown in Table 4.3. For `raytrace`, `euler`, `mc`, `juru` and `analyzer` space saving results were similar to the ones reported for the initial input. For `javac`, `jack` and `jess` some space is saved, although less than the amount of space saved for the initial input. We believe that this shows that the transformations work for multiple inputs, noting that the approximation of the amount of space savings in general should be based on measurements for a set of inputs.

We informed the developers of `juru` and `analyzer` of our results. These developers will be integrating our suggested rewritings into future versions of their code. Interestingly, we also noted that later versions of `jack` application, which is now called `javacc`, use similar rewritings to the ones we suggest.

Finally, the drag information for `tvla` was investigated using HUP (see Section 4.5). Applying assign-null transformations to the code, 10% space savings and 21% footprint savings are reported

	Integral			**Footprint**		
benchmark	**reduced reach**	**original reach**	**space savings(%)**	**reduced reach**	**original reach**	**footprint savings(%)**
javac	546.00	565.09	3.38	7.03	7.06	0.41
jack	74.64	91.52	18.45	0.80	1.13	29.65
raytrace	1057.20	1529.39	30.87	3.14	4.54	30.79
db	N/A	N/A	0	N/A	N/A	0
jess	128.41	129.88	1.14	2.01	2.03	0.84
euler	9790.88	10552.93	7.22	15.08	17.00	11.32
mc	7088.99	7638.18	7.19	63.29	65.85	3.89
analyzer	1183.40	1421.45	16.75	3.85	4.50	14.54
juru	107.96	120.73	10.58	0.71	0.90	21.96
average			10.62			12.6

Table 4.3: Space savings and footprint savings for alternate inputs.

in [45].

4.4.3 Runtime Savings

Our motivation here is to show that the rewritings aimed at space savings have no negative impact on the runtime of the benchmarks. The measurements were done on a 400MHz Pentium-II CPU with 128MB main memory running Windows NT 4.0 on Sun HotSpot Client 1.3 [37]. For the SPECjvm98 benchmarks, `juru` and `analyzer` we used a 32MB initial heap size and a 48MB maximum heap size. For `euler` and `mc` we used a 64MB initial heap size and a 96MB maximum heap size. Each reported result is the average of 10 runs.

Table 4.4 shows the runtime savings for the optimized benchmarks. The average runtime for all of the benchmarks is reduced by 1.2%. This indicates that on the original and alternate input our space savings transformation have a small positive impact on the performance of the benchmarks.

4.5 Extensions

This section describes a follow-up research done in collaboration with M. Pan as part of his Master's thesis. In this section we elaborate on a more portable drag tool named *HUP* we built and used in order to extend our research. In Section 4.5.1 we discuss the HUP tool. Then, in Section 4.5.2 we describe *lag* information measurements, allowing to reason on when objects can be lazily allocated.

Benchmark Program	Sun HotSpot 1.3 Client		
	Reduced Runtime (sec.)	Original Runtime (sec.)	Runtime Saving (%)
javac	26.569	26.538	-0.12
jack	14.961	15.11	0.99
raytrace	16.07	16.452	2.32
jess	11.868	12.116	2.05
euler	42.772	43.605	1.91
mc	34.154	34.884	2.09
juru	23.51	23.69	0.76
analyzer	15.958	15.898	-0.38
average			1.2

Table 4.4: Runtime Savings.

4.5.1 HUP tool

HUP (**H**eap **U**sage **P**rofiler) is a profiling tool for measuring drag and lag information. The motivation for HUP is having a portable tool. While the drag tool reported so far in this chapter is based on JVM instrumentation, and thus not portable, HUP is made portable by basing it on bytecode instrumentation and on a generic JVM profiling interface, named JVMPI [69]. JVMPI provides a (limited) predefined set of profiling events. Example events are object allocation and object deallocation by GC. The idea is that JVMPI will be supported by many JVMs. Bytecode instrumentation is used to profile events not supported by JVMPI. For example, an object use event is profiled using bytecode instrumentation. One benefit of HUP's approach is that an application can be profiled on several different Java Virtual Machine implementations, in order to better tune its performance.

As in our prototype drag tool, HUP consists of two phases. The first online phase profiles information using JVMPI and bytecode instrumentation. The information is logged to a file. The second offline phase analyses the log file, and allows a user to perform queries regarding the lag and the drag of the profiled application.

4.5.2 Lag Information

The lag is defined as the difference between the time an object is allocated and the time the object is first used [50]. This is shown in Fig. 4.2(a). The motivation for lag measurements is to understand whether objects can be lazily allocated so these are allocated just before their first use. We would like to

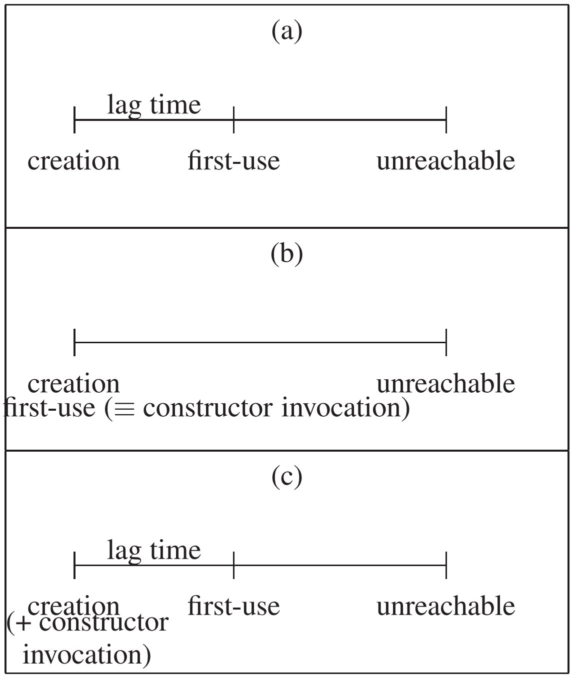


Figure 4.2: The lag of an object.

understand: (i) what is the potential space savings due to lag information, and (ii) the feasibility of lazy allocation of lagged objects.

Röjemo and Runciman [50] conducted lag measurements for Haskell, a lazy functional language. They note that "Under a call-by-need regime, we should not be surprised that many cells are used as soon they are created – zero lag". Not surprisingly, our experience with Java, an object-oriented imperative language, shows similar findings. Thus, we slightly modify the definition of lag (to be described shortly) and then use HUP to measure lag of objects in several benchmarks.

The zero lag problem in Java stems from the fact that the constructor of an object is invoked immediately upon the allocation of an object. A method invocation for an object is considered as a use of an object, thus all objects have zero lag as shown in Fig. 4.2(b). Other programming languages (e.g., C) avoid the zero lag problem since there is no coupling between the allocation of an object and the initialization of the object.

Fig. 4.2(c) shows a simple remedy to avoid the zero lag problem, by ignoring the first use event of an object, i.e., not considering the invocation of the object constructor as a use event. As a result, the next use event, either taking place in the code of the constructor (e.g., initialization of a field of that object), or taking place after the constructor method returned, is considered as the first use of the object. In [45] we discuss other remedies to the zero lag problem, e.g., ignoring the constructor invocation as well as ignoring the uses of the object occurring in the code of the constructor. We do not detail the alternative remedies here, since the empirical results for these alternatives (given in [45]) are quite similar to the

Benchmark	Lag/Reachable
javac	13.77%
jack	5.01%
raytrace	7.76%
db	29.76%
jess	0.12%

Table 4.5: Lag measurements.

results reported here.

Table 4.5 shows lag measurements for 5 SPECjvm98 benchmarks. As explained, we consider the lag shown in Fig. 4.2(c), i.e., the time from the constructor invocation of an object to the time the object is first used. The second column shows the ratio between the lag space integral and the reachability integral. For lag space integral we plot the size of objects allocated but not used yet as a function over allocation time, and compute the integral under the curve. We see that the lag ranges from 0% to 30%, i.e., up to 30% of the total space is due to objects allocated but not yet used.

The only benchmark we obtain space savings due to rewritings aimed at lag reduction is `Raytrace`. We apply *lazy allocation* (see Section 4.3.2) to objects, which were not found to be used immediately after their allocation. These transformations lead to a total of 2.5% space savings due to reduction in lag for objects in `Raytrace`. For other benchmarks, e.g., *db* the lag is due objects in a random-access repository, which are accessed long after allocated and initialized. Due to the undetermined nature of access to objects it is not clear how to rewrite the code to reduce the lag.

Chapter 5

Memory Liveness Potential

We study the potential impact of different kinds of liveness information on the space consumption of a program in a garbage collected environment, specifically for Java. The idea is to measure the time difference between the actual time an object is collected by the garbage collector and the potential earliest time an object could be collected assuming dynamic location liveness information were available. We focus on the following kinds of liveness information: (i) stack reference liveness (local reference variable liveness in Java), (ii) global reference liveness (static reference variable liveness in Java), (iii) heap reference liveness (instance reference variable liveness or array reference liveness in Java), and (vi) any combination of (i)-(iii). We also provide some insights on the kind of interface between a compiler and GC that could achieve these potential savings.

We use our instrumented JVM to measure dynamic location liveness information. Experimental results are given for our set of 10 benchmarks. We show that in general stack reference liveness may yield small benefits, global reference liveness combined with stack reference liveness may yield medium benefits, and heap reference liveness yields the largest potential benefit. Specifically, for heap reference liveness we measure an average potential savings of 39% using an interface with complete liveness information, and an average savings of 15% using a more restricted interface.

The remainder of the chapter is as follows. Section 5.1 gives motivation for our experiments. Section 5.2 describes our algorithm for measuring the impact of liveness information and its implementation, assuming an idealized GC interface. Section 5.3 presents the algorithm and implementation for measuring the impact of liveness in the presence of a restricted GC interface. In Section 5.4 we discuss the benchmark programs and compare the empirical results of both experiments.

5.1 Motivation

In Chapter 3 we measure the potential space savings if an object is collected as soon as it is in-drag, i.e., immediately after it is last dereferenced. However, as explained in Chapter 2 collecting in-drag objects requires the memory manager to handle the burden of "valid dangling references". One way to alleviate this burden, is to delay the reclamation of an object until its references are (dynamically) dead; thus, we measure the potential space savings if an object is collected as soon its references are dynamically dead. Interestingly, our experimental results indicate that dynamic location liveness information provides potential space benefits similar to the ones obtained for drag information.

It is well known that liveness information may aid in earlier reclamation of objects, by reducing the set of root references, or by removing dead references (i.e., references that are not subsequently used prior to redefinition) from the heap graph [43]. For example, in [1] liveness information for local reference variables is used to remove dead local reference variables from the root set; thus, some objects could be identified as unreachable and garbage collected earlier.

However, the impact of general liveness on the space collected by GC is as yet unknown. Moreover, the overhead of having GC consult liveness information may be significant. In fact, as noted in [28], it is not clear how to represent heap liveness information in a feasible manner.

In this chapter we study the above questions for Java programs. The information is gathered dynamically for a given run for different kinds of liveness information. Thus, these experiments indicate a kind of upper bound on storage savings that could be achieved assuming static liveness information is available.

5.1.1 Main Results

We estimate the effect of general liveness information on space consumption of Java applications. We measure the impact of following kinds of liveness information: (i) stack reference liveness (local reference variable liveness in Java), (ii) global reference liveness (static reference variable liveness in Java), (iii) heap reference liveness (instance reference variable liveness or array reference liveness in Java), and (vi) any combination of (i)-(iii).

Our instrumented Sun JVM is used in order to measure dynamic location liveness information. Experimental results are given for our 10 benchmarks, including 5 of the SPECjvm98 benchmark suite. The information for our measurements is gathered dynamically during a program run. Specifically, we compare the objects reachable from the root set to the ones that are reachable from the root set when ignoring dead references. Thus, these experiments indicate a kind of upper bound on storage savings that could be achieved assuming static liveness information is available.

We measure the impact of liveness assuming two possible interfaces for communicating liveness information to the garbage collector: (a) an idealized interface, where the dynamic location liveness is recorded for individual heap references (b) a more restricted interface, in which dead heap references are assigned `null` immediately after their last use.

In a first experiment we consider the idealized interface to GC, which gives an upper bound on any static liveness analysis algorithm combined with any interface to GC. While in some cases this upper bound may turn out to be too loose, we found that for stack reference and global reference liveness analysis, this upper bound suggests small to medium benefits. On average stack reference liveness information buys 2% savings, and combining it with global reference liveness information the savings increases to 9%. Thus, although these liveness schemes suggest known practical solutions both with respect to the static analysis algorithm (at least for stack reference liveness), and with respect to the GC interface, the importance of having these kinds of liveness information is not clear considering their limited impact on space savings.

For heap reference liveness, the idealized interface yields an average 39% potential savings. In this case the question is what part of the potential is achievable considering a "realistic" heap reference liveness analysis and a "reasonable" interface to GC. Using the more restricted but more realistic assign to null interface we measured a potential savings of 15% on average. This leads us to believe that a practical algorithm achieving significant space savings may be possible.

The results of these experiments with the restricted interface also give insights for the kind of information required by heap reference liveness analysis. For example, interprocedural information seems to have major effect on the expected impact. On average we get 6% potential savings assuming only intraprocedural information, and this increases to 15% when interprocedural information is available. In addition, combining heap reference liveness and global reference liveness information using the same restricted GC interface gives negligible additional benefits (less than 0.5% on average) to the potential benefits obtained just by heap liveness.

5.2 Liveness Measurements

In this section we present a method for measuring the impact of general liveness schemes on space consumption in a garbage collected environment. Our major observation for approximating the impact of liveness information on space consumption is that there is no need to directly compute dynamic location liveness information. In a nutshell, we found that instead it is sufficient to identify the last use of any of the references reaching an object. In Section 5.2.1 we show that this can be done with a single run of the program by keeping a constant overhead per object (around 60 bytes in our current implementation), and by extending the tracing phase in GC to compute the necessary information[1].

In [28, 29], the impact of stack reference liveness and global reference liveness on space consumption is computed by running the program once to track uses and definitions of references, writing them to a log file. The resulting log file is analyzed in a backward direction to directly compute dynamic liveness information for references, and finally the information is communicated to a second run of the program. Our technique has the following advantages over the above technique. First, it alleviates

[1] For a non-tracing GC, this may require a separate tracing phase which is specialized to compute just the necessary information for the liveness measurements.

Property	**Intended Meaning**	**Phase**	**Event**	**Update**
$heap_L(obj)$	*obj* is referenced by a live heap reference at time $heap_L(obj)$	M	$\widehat{use}$ *x.f*	$heap_L(env(x.f)) =$ *Current Time*
			$\widehat{use}$ *a[i]*	$heap_L(env(a[i])) =$ *Current Time*
$stack_L(obj)$	*obj* is referenced by a live stack root at time $stack_L(obj)$	M	*use y*	$stack_L(env(y)) =$ *Current Time*
$stack_{L^*}(obj)$	*obj* is reachable along a path starting from a live stack root at time $stack_{L^*}(obj)$	C	init liveness	$stack_{L^*}(obj) = \max(stack_L(obj), stack_{L^*}(obj))$
			trace children	$stack_{L^*}(obj) = \max\begin{pmatrix} stack_{L^*}(father), \\ stack_{L^*}(obj) \end{pmatrix}$
$stack_R(obj)$	*obj* is referenced by a stack root at time $stack_R(obj)$	C	trace roots	$stack_R(obj) =$ *Current Time* $stack_{R^*}(obj) =$ *Current Time*
$stack_{R^*}(obj)$	*obj* is reachable along a path starting from a stack root at time $stack_{R^*}(obj)$	C	trace children	$stack_{R^*}(obj) =$ *Current Time*
$static_L(obj)$	*obj* is referenced by a live static root at time $static_L(obj)$	M	*use g*	$static_L(env(g)) =$ *Current Time*
$static_{L^*}(obj)$	*obj* is reachable along a path starting from a live static root at time $static_{L^*}(obj)$	C	init liveness	$static_{L^*}(obj) = \max(static_L(obj), static_{L^*}(obj))$
			trace children	$static_{L^*}(obj) = \max\begin{pmatrix} static_{L^*}(father), \\ static_{L^*}(obj) \end{pmatrix}$
$static_R(obj)$	*obj* is referenced by a static root at time $static_R(obj)$	C	trace roots	$static_R(obj) =$ *Current Time* $static_{R^*}(obj) =$ *Current Time*
$static_{R^*}(obj)$	*obj* is reachable along a path starting from a static root at time $static_{R^*}(obj)$	C	trace children	$static_{R^*}(obj) =$ *Current Time*
$other_R(obj)$	*obj* is referenced by an other root at time $other_R(obj)$	C	trace roots	$other_R(obj) =$ *Current Time* $other_{R^*}(obj) =$ *Current Time*
$other_{R^*}(obj)$	*obj* is reachable along a path starting from an other root at time $other_{R^*}(obj)$	C	trace children	$other_{R^*}(obj) =$ *Current Time*

Table 5.1: Liveness information gathered during the run. `y` is a stack variable and `g` is a static variable. *use y, use g* denote a use of the variables `y,g`, respectively. $env(x)$ gives the object referenced by `x`. We treat a dereference as two consecutive events. Thus, *use x.f* is split into *use x* and $\widehat{use}$ *x.f*, where $\widehat{use}$ is a special operation that only uses the r-value of `x.f`. The mutator phase is denoted by *M*, and the collector phase is denoted by C.

the problem of the infeasible amount of dynamic liveness information (as reported in [28, 29]). As an immediate result our measurements can handle local variable references, global variable references and heap references without further approximations. Second, the implementation is simpler as there is no need for a mechanism to match a reference in the first run to its corresponding reference in the second run for communicating liveness information. Finally, running the program once allows us to handle non-deterministic (e.g., multithreaded) applications.

5.2.1 Algorithm

Our algorithm operates on the program states. It computes liveness information while the program (mutator) executes: (i) *when the mutator uses the store*, we record local information for objects, (ii) *when the garbage collector traverses the reachable heap* it propagates global information, and (iii) *when an unreachable object is collected* we compute liveness information.

Table 5.1 shows the information gathered during the run of the program. As usual, time is measured in bytes allocated so far in the program. All the information is maintained at the level of an object. Columns *property* and *intended meaning* describe the properties we maintain for every allocated object. We classify references as follows: (i) stack references correspond to references on the Java stack, (ii) static references correspond to references in static variables, (iii) other references correspond to references on the native stack, and other special root references, and (iv) heap references correspond to references in instance variables and references in arrays. For each type of reference there are four kinds of properties: (i) liveness (the last time a reference of this type to the object is used), (ii) direct reachability (the last time the object was directly reachable from a variable of this reference type), (iii) path liveness (the last time the object was reachable through a path starting at a live reference of this type), and (iv) reachability (the last time the object was reachable through a path starting at this reference type). Table 5.1 includes only reference properties needed for computing the liveness impact of stack, static and heap references. For example, $stack_{L^*}(obj)$ keeps the time an object is still reachable along a path starting from a live stack root. Such information is used to determine when obj could be collected assuming stack reference liveness information were available. Interestingly, associating the necessary information with objects avoids the need to keep track of all the references in a run and is one of the keys to keeping the amount of information required for the analysis feasible.

The *phase* column indicates which properties are updated during GC, and which properties are updated as result of mutator execution. Specifically, the liveness properties are updated during mutator execution, and the direct reachability, reachability and path liveness properties during GC. A property is updated upon events shown in *event* column. The *update* column shows the new value for the corresponding property when the event occurs. In the following, we provide details for property update.

We trigger GC after every 100 KB of allocation in order to propagate the liveness and reachability information at regular intervals. In Section 7 we discuss alternative ways we could obtain reachability information and liveness information along heap paths.

Actions in the Mutator

When an object reference is used, we update the local information associated with the object. In particular, for a *use r* event, where r is either a heap, stack or a static reference, we set the corresponding liveness property of the object referenced by r, $heap_L$, $stack_L$, or $static_L$ to be the current time. We treat a dereference as two consecutive events. Thus, *use x.f* is split into *use x* and $\widehat{use}$ *x.f*, where $\widehat{use}$ is a special operation that only uses the r-value of `x.f`. Also, *def x.f* is split into *use x* and $\widehat{def}$ *x.f*, where $\widehat{def}$ is a special operation that only uses the l-value of `x.f`. Array reference expressions are handled similarly.

Interestingly, since we associate the liveness properties with the objects, we only need to update the properties for the *use x* and $\widehat{use}$ *x.f* events. These events are sufficient to determine the last time a reference to the object of a particular type (stack, static, or heap) was used. Notice, that after the last use through a particular reference type, the corresponding property will hold the time of last use and the property will not be updated further.

Actions in the Collector

When the collector runs, we update the direct reachability, reachability and path liveness properties of the objects. Specifically, the collector establishes the invariants that these properties correctly describe the status of the heap. For example, when the collection is complete, $stack_{L^*}(obj)$ is set to the last time that obj is reachable along a path starting from a live stack root.

Here are the details. We update the path liveness properties for each kind of root. We begin by setting the path liveness properties of each object to the maximum of its current value and the object's liveness property. Next, we propagate path liveness information from each root to its children. In particular, if object A references object B, and the path liveness property of A is greater than that of B, then we set B's value to A's and continue propagating from B. To keep the cost of the propagation proportional to the number of references, we scan stack roots in decreasing order of the path liveness property value of their referenced object. We do the same for static roots. Note that the above implies we visit an object at most twice.

We also update the direct reachability and reachability properties for each kind of root. First we scan the roots and for the objects referenced by the roots, set the directly reachable and reachable properties to the current time according to the kind of the root. For example, if obj is referenced by a stack variable, we set $stack_R(obj), stack_{R^*}(obj)$ to the current time. Then, we propagate reachability information from each root to its children. For example, if obj is along a path starting from a stack variable, we set $stack_{R^*}(obj)$ to the current time. Notice that the propagation of reachability and liveness information

Information	**Collection Time**
none	$max(stack_{R^*}(obj), static_{R^*}(obj), other_{R^*}(obj))$
heap liveness	$max(heap_L(obj), stack_R(obj), static_R(obj), other_R(obj))$
stack liveness	$max(stack_{L^*}(obj), static_{R^*}(obj), other_{R^*}(obj))$
static liveness	$max(static_{L^*}(obj), stack_{R^*}(obj), other_{R^*}(obj))$
stack + static liveness	$max(stack_{L^*}(obj), static_{L^*}(obj), other_{R^*}(obj))$
heap + stack liveness	$max(heap_L(obj), stack_L(obj), static_R(obj), other_R(obj))$
heap + static liveness	$max(heap_L(obj), static_L(obj), stack_R(obj), other_R(obj))$
heap + stack + static liveness	$max(heap_L(obj), stack_L(obj), static_L(obj), other_R(obj))$

Table 5.2: Computation of the earliest collection time for an object.

can be combined; thus, in total we visit an object at most twice, scanning the roots once.

Computing Impact of Liveness

When an object is collected, we evaluate the earliest time it could have been collected assuming each of the liveness schemes. Table 5.2 shows how to compute these times from the object properties. We now demonstrate two cases using Fig. 5.1, which shows a snapshot of the heap during the run at time t before GC is invoked.

Assuming stack reference liveness information were available, an object could be collected when it is no longer reachable along a path from a live stack root, and also not reachable from other roots (e.g., a static root). For example, in Fig. 5.1, if $stack_{L^*}(o_3) < t$, then o_3 could be collected since it is not reachable from other roots. However, o_4 could not be collected at time t since it is still reachable from a static variable (i.e., $static_{R^*}(o_4) \geq t$).

An object's heap liveness property provides information as to whether there is a live reference to it from another object. However, to determine whether the object can be collected, we also need to make sure that it is not directly reachable from a root. For example, o_3 could be collected at time t if $heap_L(o_3) < t$. However, o_4 could not be collected at time t since it is still directly reachable from a static variable (i.e., $static_R(o_4) \geq t$).

We compute the average space savings for a particular liveness scheme using the ratio of two reachability integrals. The numerator is the integral for the liveness scheme: we plot the size of the reachable objects assuming the scheme as a function of time and compute the integral under the curve. The denominator is the reachability integral assuming no liveness information.

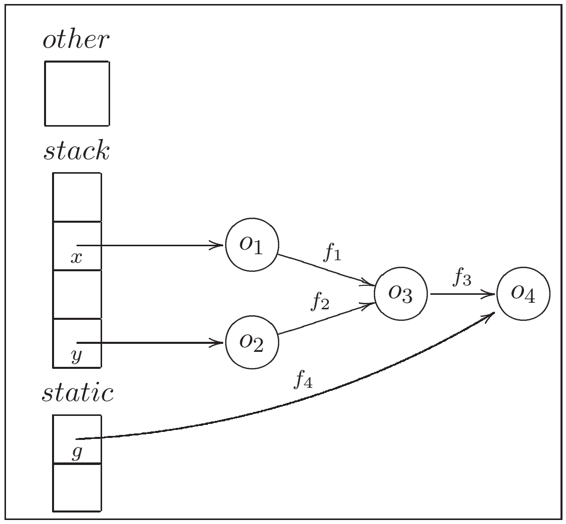


Figure 5.1: A heap snapshot at time t.

5.2.2 Implementation

We use the instrumented JVM described in Section 2.2.2. For the purpose of dynamic location liveness profiling, object information is updated upon the following events:

Object Creation Creation time and length fields are set.

Reference Use Information is updated according to the reference kind: (1) getting reference field information (e.g., via `getfield` bytecode) updates the $heap_L$ property of the referenced object. (2) getting local variable information (e.g., via `aload` bytecode) updates the $stack_L$ property of the referenced object. (3) getting global variable information (e.g., via `getstatic` bytecode) updates the $static_L$ property of the referenced object.

GC Our implementation follows Section 5.2.1.

When the program terminates our analyzer reads the output log file, and then follows Table 5.2 to compute the earliest time an object could be collected assuming a particular liveness scheme. Then, the reachability integral for each of the liveness schemes is computed.

5.3 A Feasible Heap Liveness GC Interface

In this section we consider a feasible interface to communicate heap reference liveness information to GC, in which dead references are assigned null immediately after their last use. This interface does not require any changes to GC. As noted in [1, 55] such an interface may not be practical. However, we believe that it allows us to estimate the potential savings expected with a reasonable interface.

Event	Action
`p: use x.f`	$SNULL = SNULL \setminus P(lval(x.f))$; $P(lval(x.f)) = p$
`p: def x.f`	$P(lval(x.f)) = p$
`p: use a[i]`	$SNULL = SNULL \setminus P(lval(a[i]))$; $P(lval(a[i])) = p$
`p: def a[i]`	$P(lval(a[i])) = p$

Table 5.3: Detection of null assignable program points. The set $SNULL$ holds the null assignable program points. For a heap reference h in the run, $P(h)$ holds the last program point that used the l-value of h, i.e., either h itself was used as an r-value, or h was assigned. When the program starts, $SNULL$ is initialized with all program points manipulating the heap.

The algorithm operates in two runs. In the first run we detect the places in the code where assignments to null potentially reduce the space, while preserving program semantics, and the program is modified accordingly. Then, the modified program is executed on the same input, in order to evaluate the space savings. Unlike the algorithm in Section 5.2, this technique is limited to applications with deterministic behavior, due to the second execution of the program.

5.3.1 Algorithm

The idea is to run the program once to identify dynamically dead reference expressions. Clearly, if an expression e is dynamically dead after pt in a trace π, then e may be assigned null every time pt is executed, since the value e is not used after pt. In order to simplify the presentation, we assume the code is normalized so each program point manipulates at most one heap reference expression. The Java bytecode satisfies this requirement. In addition, our algorithm guarantees that e is assigned null after pt is executed, only if e is included in the statement in pt. Thus, further on we use the term *null assignable program points*, since it is clear to which expression a null value is assigned. Finally, the algorithm could be refined to assign null to a heap reference expression occurring in a specific calling context. For example, in our implementation explained in Section 5.3.2, program points are actually sequences of calling contexts.

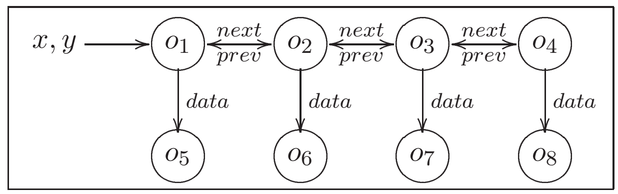
At the outset our algorithm assumes that all program points that manipulate the heap are candidates for null assignment. As the program runs, it determines the points where null assignment is impossible, and eliminates them from consideration. At program termination, the remaining points are the null assignable ones. This algorithm is summarized in Table 5.3.

In particular, the algorithm starts by inserting all candidate program points in $SNULL$. Then it runs

```
  void foo(DL x) {
     DL y,t;

     y = x;
     t = null;
p_1: while (y != null) {
       t = y.prev;
       // y.prev = null;
       if (t != null) {
p_2:     process(t.data, y.data);
         // t.data = null;
       }
p_3:   t = y.next;
       // y.next = null;
       y = t;
     }
  }
```

(a)



(b)

Figure 5.2: Assign null example.

the program. Upon a *use x.f* event at program point p, the algorithm concludes that the previous point that used the location denoted by `x.f` cannot be assigned null, since it is currently used. Therefore, we remove the previous program point from the set of null assignable program points $SNULL$. In addition, the last program point that used the location denoted by `x.f` (i.e., $P(lval(x.f))$) is set to the current program point p. Upon a *def x.f* event at program point p, we simply set $P(lval(x.f))$ to the current program point p. Usage or a definition of an array reference expression is handled similarly. When the program terminates, $SNULL$ contains the null assignable program points with respect to this run. Finally, the program is modified to assign null to the heap reference expressions in program points included in $SNULL$.

The modified program is executed a second time on the same input and its space consumption is measured. Finally, we compare the space consumption with that in the first run to evaluate the savings.

We demonstrate the algorithm using the code fragment in Fig. 5.2(a) that processes a doubly linked list elements in pairs. We assume the heap at program point `p_1` shown in Fig. 5.2(b). The code is annotated with null assignment for null assignable reference expressions. For example, at `p_3` the expression `y.next` is null assignable. This is since all `next` fields used at `p_3` (i.e., those emanating from $o_1, \ldots, o_4$) have no subsequent use in this code fragment. Assuming no further uses of this list in the program, `y.next` is null assignable at `p_3`. Similarly, `t.data` is null assignable at `p_2`. Lastly, `y.data` is not null assignable at `p_2` since the `data` field emanating from $o_1, \ldots, o_4$ will be subsequently used in the next iteration through the `t.data` expression.

5.3.2 Implementation

In our instrumented JVM, we attach a trailer to every object to keep track of calling contexts. For every reference instance variable, the trailer contains an entry for the last calling context that used the location of this reference. For feasibility we limit the calling context length to at most k calls. For longer contexts, we only record the suffix of the k last calls (we ran our experiments for $k = 0, 1, 2, 3$.)

Our implementation does not modify the program, but instead the JVM is modified to record the fact that null could be assigned at the proper program points/calling contexts. Since the label of a program point could change across runs, we represent a program point p by the method m containing p, the offset of p in m, and also the opcode in p. Opcode is also used to represent a program point, since our underlying JVM uses byte code rewriting (e.g., it replaces `getfield` bytecode with a `getfield_quick` non-standard bytecode for faster code execution). We also track information about executed calling contexts, and only these contexts are candidates for a null assignment. Finally, program points that manipulate references subsequently manipulated by native program points are not candidates for null assignment, since a static analysis algorithm is usually not expected to analyze native code.

Object information is updated as described in Table 5.3 upon the following events:

Heap Reference Use getting reference field information (e.g., via `getfield` or `aaload` bytecode),

Heap Reference Def setting reference field information (e.g., via `putfield` or `aastore` bytecodes)

When the program terminates, we write all null assignable calling contexts to a file. Also, we output the size of the reachable integral.

For the second run, the modified JVM could execute the program and assign null whenever it encounters a null assignable calling context. However, our implementation makes an optimization to allow estimating in a single run the impact of null assignment for several context lengths, and for breaking down the space benefits due to assigning null to field references, and due to assigning null to elements in an array of references, as explained in Section 5.4. Therefore, in the second run we record for every reference the fact that null could be assigned in the trailer of the object containing that reference. Then, every 100KB of allocation we invoke a mark phase per *null assign scheme*, i.e., per a combination of context length and the kind of references being assigned null (e.g., only elements of an array of references). During such a mark phase the information recorded in the object trailer is used to determine whether a reference is considered as assigned null according to the corresponding null assign scheme.

bench.	drag	Liveness Information: heap + stack + static	heap + static	heap + stack	heap	stack + static	static	stack	w/out liveness
jess	108.53	118.64	118.81	119.64	119.80	359.84	365.86	391.05	392.97
raytrace	253.73	258.44	258.51	258.96	259.01	640.03	642.99	656.16	656.89
db	497.63	500.57	500.59	500.85	500.87	795.12	796.43	805.27	805.31
javac	1058.19	1065.08	1065.17	1065.90	1065.98	1623.14	1630.31	1640.73	1644.62
jack	86.33	93.33	93.41	93.95	94.03	187.11	194.97	215.99	216.59
analyzer	276.08	284.98	285.15	287.04	287.21	493.91	615.67	657.54	674.39
juru	55.31	58.64	68.54	59.19	69.09	68.01	78.13	77.97	88.25
euler	1936.73	1942.03	1942.06	1946.41	1946.43	2116.45	2125.61	2139.69	2148.82
mc	11423.94	11429.30	11429.32	11433.51	11433.52	11900.48	11901.69	11921.98	11923.18
tvla	318.14	324.95	325.14	332.40	332.58	525.53	528.78	566.08	569.30

Table 5.4: Reachability integrals (in MB^2) for different liveness kinds.

When the program ends, we output the size of the reachable integral per every null assign scheme. The ratio of these integrals and the integral from the first run yields the potential savings. Finally, the output of the original and modified JVM executions of the program are compared in order to provide a sanity check for the correctness of the implementation.

5.4 Experimental Results

5.4.1 The Space Savings due to Liveness

Our experiments were applied to our set of benchmark programs. We ran the algorithm described in Section 5.2 on the benchmarks. Table 5.4 shows for every benchmark the size of the reachability integral assuming different kinds of liveness information (see Section 5.2.1). We show information for stack, static, heap liveness information and for every combination of these. Column *w/out liveness* shows the original reachability integral. Column *drag* repeats the information reported in Section 3.3 by showing the reachability integral if an object is collected immediately after it is last dereferenced in the program.

In a similar manner, Table 5.5 shows for every benchmark the maximum size of the reachable heap in the run for different kinds of liveness information. The integrals indicate an overall view of the space consumption of the program, whereas the maximum heap size is a kind of a feasibility criteria.

Fig. 5.3(a) shows the ratio between the integrals and the original reachability integral, and Fig. 5.3(b) shows the ratio between the maximum heap size and the original maximum heap size. In both figures we also show the average across the benchmarks.

We first discuss the resulting integrals. For stack reference liveness the average potential savings is

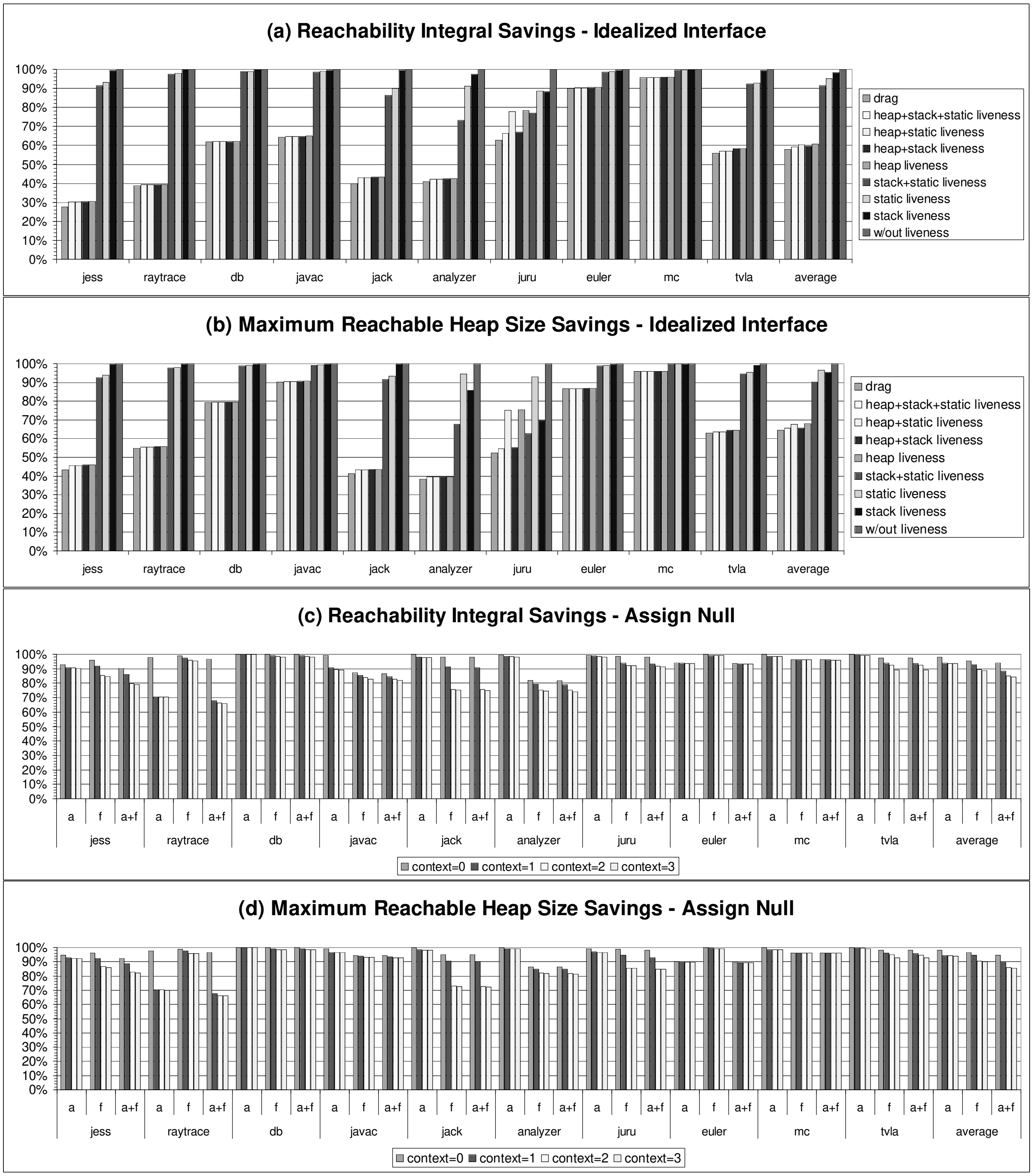


Figure 5.3: Potential space savings results.

benchmark	drag	Liveness Information: heap + stack + static	heap + static	heap + stack	heap	stack + static	static	stack	w/out liveness
jess	0.61	0.64	0.64	0.65	0.65	1.30	1.33	1.41	1.41
raytrace	2.39	2.42	2.42	2.42	2.42	4.24	4.26	4.34	4.35
db	7.60	7.63	7.63	7.63	7.63	9.50	9.52	9.60	9.60
javac	8.35	8.38	8.38	8.38	8.38	9.16	9.19	9.24	9.25
jack	0.57	0.61	0.61	0.61	0.61	1.28	1.30	1.39	1.39
analyzer	1.38	1.42	1.42	1.43	1.43	2.43	3.40	3.08	3.59
juru	0.46	0.48	0.66	0.49	0.66	0.55	0.82	0.61	0.88
euler	6.72	6.74	6.74	6.75	6.75	7.67	7.70	7.73	7.76
mc	78.93	78.95	78.95	78.96	78.96	82.17	82.20	82.25	82.27
tvla	1.51	1.53	1.53	1.55	1.55	2.27	2.29	2.38	2.40

Table 5.5: Maximum reachable heap size (in MB) for different liveness kinds.

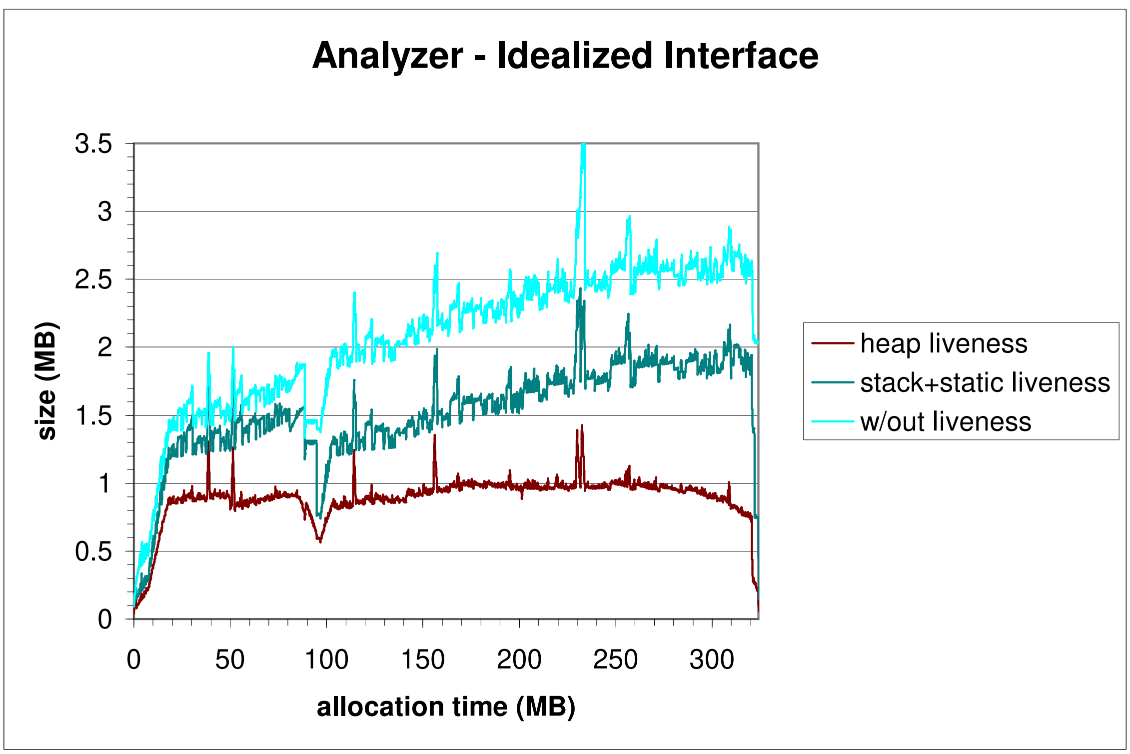


Figure 5.4: Potential space savings for *Analyzer*.

2%, and in the best case (*juru*) 12%. In Agesen et. al [1] the actual savings obtained by implementing stack reference liveness analysis for Java are shown. The trend of our stack reference liveness results match [1], and also for a simple artificial benchmark (EllisGC) we get duplicate results. Together with Detlefs and Moss [16] we have investigated the differences for other benchmarks, where our dynamic measurements show less potential than the actual savings obtained in [1]. We concluded that our results are not directly comparable due to differences in the experimental environment (i.e., different versions of JDK/JVM, different versions of benchmarks, and different input size used to run the benchmarks). However, the final outcome of both experiments is the same: "the main benefit of stack reference liveness analysis is preventing bad surprises". Another difference in our measurements is that our underlying GC is type-conservative; thus, potential savings are shown w.r.t. a conservative GC, where in [1] the base GC is type-accurate. According to [28], this latter difference is not expected to have a major effect on the results.

To the best of our knowledge, the rest of the results are new for object-oriented programs. Moreover, we provide the first study of the effect of heap reference liveness. For static variable liveness, considering the cost of obtaining the information statically, which requires whole program analysis, our experimental results (average of 5% savings) indicate that this optimization may not be profitable. Also for the combination of static variable liveness and stack reference liveness for most of the programs, we get medium savings (average of 9%), while for *juru* and *analyzer* the potential is quite large. This leads us to the same conclusion of "preventing bad surprises".

The comparison of the maximum heap sizes assuming liveness information to the original maximum heap size yields similar results for the benchmarks excluding *juru* and *analyzer*. In *juru* and *analyzer* the profit in maximum heap size is much larger than the overall benefit expressed by the integral size. From our experience discussed in Section 4.4, the reason is that for both benchmarks, stack liveness and/or static variable liveness aid in a few places in the code where a large object, or a group of objects referenced by a single root are kept after their last use.

For heap liveness, our results show an average of 39% potential savings. Moreover, combining heap liveness with other liveness information has negligible effect. This may not come as a surprise, since most of the objects in the heap are referenced solely by heap references. Finally, assuming drag information (i.e., tracking the last access to an object) we get 42% potential savings, which is very close to heap liveness results. Thus, we conclude that heap liveness information potentially brings most of the space benefits achievable beyond reachability-based garbage collectors.

The comparison of the maximum heap sizes assuming liveness information to the original maximum heap size shows that for most of the benchmarks the overall benefit (i.e., the savings in the integral size) is larger (around 7% more savings) than the benefit at the peak (i.e., maximum heap size comparison).

It is interesting to note that our results provide non-trivial upper bounds for potential savings in the integral and the memory footprint between the current GC and an idealized one which has complete liveness information. For example, for *analyzer*, which is a benchmark that allocates memory extensively, the maximum heap size results indicates that in the worst case, the current GC consumes 2.5 times space than the one obtained by an idealized GC.

Fig. 5.4 shows the reachable heap graph over time for *analyzer* considering the following liveness schemes: (i) heap liveness, (ii) stack combined with static variable liveness, and (iii) no liveness information. We see that assuming complete heap liveness the memory footprint of this program remains around 920KB. In contrast, assuming complete stack and static variable liveness information, heap consumption increases as the program executes. As expected, the peaks in memory footprint occur simultaneously for these three liveness schemes.

5.4.2 The Space Savings due to Null Assignments

Fig. 5.3(c) shows the ratio between the integrals of the modified programs assigning `null` to dynamically dead instance variables and the reachability integral in the original program. Similarly, Fig. 5.3(d) shows the ratio for the maximum heap size. We measure potential savings for calling contexts of length 0, 1, 2, and 3. Heap references consist of two kinds, instance fields and elements of an array of references. Since algorithms for approximating the liveness of instance fields may be different in nature than the ones for approximating the liveness of elements in an array of references, we show for each context a breakdown of heap liveness according to these reference kinds. The *a* column shows potential savings when `null` is assigned only to an element in array of references (i.e., immediately after `aaload` or `aastore`), the *f* column shows potential savings when `null` is assigned only to an instance field (i.e., immediately after `getfield` or `putfield`), and the *a+f* column shows overall potential savings when null is assigned to both kinds of heap references.

The average overall space savings for intraprocedural information is 6% while for contexts of length 2, we get 15% potential savings. Breaking down the overall space savings according to heap reference kind, we see that assigning null to instance fields assuming intraprocedural information saves 4.5%, while for contexts of length 2 we get on average 10.5% potential savings. Assigning null to elements in an array of references yields 2% on average assuming intraprocedural information, and 10% potential savings assuming contexts of length 2. In all cases, the added value of contexts of length 3 is insignificant.

On the negative side, we see that this interface provides significantly less potential saving than the one by the idealized interface. For example, in *jack* we get 25% potential savings in contrast to the idealized GC interface in which the potential savings is 57%. However, we believe that the upper bound obtained here is actually more tight than the one reported for the idealized interface since it resembles more closely the effects of static analysis. For example on *db* we get 2% saving here versus 38% with the idealized interface. This is due to the fact that this program randomly accesses a database of objects and thus we believe that the 36% extra saving are due to dynamically dead references in the database which

benchmark	ref kind	context=0	context=1	context=2	context=3
jess	a	5.38	5.41	5.44	5.44
	f	0.05	0.06	0.06	0.07
	a+f	5.43	5.43	5.45	5.45
raytrace	a	0	0	0	0
	f	0	0	0	0
	a+f	0	0	0	0
javac	a	0.34	0.35	0.41	0.43
	f	0.10	0.13	0.15	0.67
	a+f	0.23	0.26	0.25	0.68
tvla	a	0	0	0	0
	f	0.17	0.23	0.34	0.56
	a+f	0.17	0.22	0.34	0.57

Table 5.6: The difference (in %) in assign null reachable integral results when considering null assignable program points computed for two runs with different inputs.

cannot be assigned null. Since the input provided in SPECjvm98 is large enough, our algorithm yields that for every point pt, there exists at least one database reference h used in pt which is subsequently used and thus pt is not null assignable.

In another experiment we tried assigning null to static variables. On average, assigning null just to static variables yields less than 1% potential savings, and assigning null both to heap references and static variables yields 15.3% potential space savings, which is less than 0.5% additional benefits comparing to assigning null just to heap references.

Validity of Assign Null Results

Our assign null experiment detects null assignable program points with respect to the current run. However, an optimizer may instrument a program with null assignments, only if a program point is null assignable on *all* execution paths (i.e., an assign-null property holds). In order to validate our assign null results, we ran 4 of our benchmarks with another input, and computed null assignable program points. Then, each of these benchmarks was run again with the original input, considering only null assignable program points detected in the run for both inputs.

Here are the alternate inputs we use. For `jess` we supply the input `hard.clp`, which is supplied in the `jess` distribution. `javac` is given a different set of files to compile. For `raytrace` we still apply the scene depicting a dinosaur, but this time the raytracer frame's width and height is changed.

Finally, `tvla` is applied with an input aimed at verifying the partial correctness of an implementation of the bubble sort algorithm.

Table 5.6 shows the percentage difference in reachable integral potential savings comparing the results of running a benchmark with the original input considering the original set of null assignable program points, and then running the benchmark with the original input considering only the set of program points that are null assignable for both inputs. *ref kind* column denotes the kind of heap reference being assigned null (see Section 5.4.2). We see that except for the case of assigning null to array elements in *jess* benchmark, the difference is insignificant for all context lengths, and for all reference kinds. This fact gives hope that a precise static analysis algorithm will be able to achieve most of the potential savings shown here.

Chapter 6

A Framework for Static Analysis of Local Temporal Heap Safety Properties

In previous chapter, we estimated the potential for space savings given a trace if (i) an object is collected as soon its references are dynamically dead, and (ii) a dynamically dead reference expression is assigned null. Motivated by that, in this chapter we present new static algorithms for saving space in applications for all traces by (i) issuing a free statement to reclaim objects, which are dynamically dead on all traces, and (ii) issuing an assign-null statement to nullify a heap reference expression which is dynamically dead on all traces. This is done by investigating a more general problem, which is static reasoning of temporal heap safety properties.

We focus on *local temporal heap safety properties*, in which the verification process may be performed for a program object independently of other program objects. We present a framework for statically reasoning about local temporal heap safety properties, and apply it to produce new conservative static algorithms for compile-time memory management, which prove for certain program points that a memory object or a heap reference is dynamically dead on all traces. We have implemented a prototype of our framework, and used it to verify compile-time memory management properties for several small, but interesting example programs, including JavaCard programs.

The rest of this chapter is organized as follows. Section 6.1 introduces the problem of verifying local temporal heap safety properties and gives a motivating example. In Section 6.2, we describe heap safety properties in general, and formulate the free property (see Definition 2.1.11), as a heap safety propery. Then, in Section 6.3, we give our instrumented concrete semantics, which maintains an automaton state for every program object. Section 6.4 describes our property-guided abstraction and provides an abstract semantics. In Section 6.5, we formulate the assign-null property (see Definition 2.1.14) as a heap safety property, and discuss efficient verification of multiple properties. Finally, Section 6.6 describes our implementation and empirical results.

6.1 Introduction

In this chapter we show how static analysis can be used to reduce space consumption, by giving static algorithms for verifying the free and assign-null properties discussed in Section 2.1.4. Once such property is verified the program may be transformed to directly free dead objects, or aid a runtime garbage collector collect objects earlier in the run by nullifying dead heap references.

In order to verify free and assign-null properties we determine for a source location whether a heap-allocated object or a heap reference is dynamically dead on all traces. The problem of statically determining for a source location whether a heap-allocated object or a heap reference is dead can be formulated as a local temporal heap safety property — a temporal safety property specified for each heap-allocated object independently of other objects.

In this chapter we present the following contributions:

1. A framework for verifying local temporal heap safety properties of Java programs is given.
2. Using this framework, we formulate two important compile-time memory management properties that identify when a heap-allocated object or a heap reference is dead, allowing space savings in Java programs.
3. We have implemented a prototype of our framework, and used it as a proof of concept to verify compile-time memory management properties for several small but interesting example programs, including JavaCard programs.

6.1.1 Local Temporal Heap Safety Properties

We develop a framework for automatically verifying *local temporal heap safety properties*, i.e., temporal safety properties that could be specified for a program object independently of other program objects. We assume that a safety property is specified using a *heap safety automaton* (HSA), which is a deterministic finite state automaton. The HSA defines the valid sequences of events that are permissable on a given program object.

It is important to note that our framework implicitly allows infinite state machines, since a state is maintained for every object and the number of objects is unbounded. Furthermore, during the analysis an event is triggered for a state machine associated with an object. Thus, precise information on heap paths to disambiguate program objects is crucial for the precise association of an event and its corresponding program object's state machine. In fact, in Section 6.6 we show that using a less-precise points-to based heap abstraction is insufficient for proving the properties of interest for our set of benchmark programs.

We develop static analysis algorithms that verify that on all traces, all objects are in an HSA accepting state. In particular, we show how the framework is used to verify properties that identify when a heap-allocated object or heap reference is dynamically dead in all traces. This information could be used by an optimizing compiler or communicated to the runtime garbage collector to reduce the space consumption of an application. Our techniques could also be used for languages like C to find a misplaced call to `free` that prematurely deallocates an object.

6.1.2 Compile-Time Memory Management Properties

In Chapter 2 we defined the free property (see Definition 2.1.11) and the assign-null property (see Definition 2.1.14). In this chapter we formulate these properties as local temporal heap safety properties, allowing us to instantiate our framework with two new static algorithms for statically detecting and deallocating garbage objects:

free analysis Statically identify source locations at which it is safe to insert a free statement in order to deallocate an object, which is dynamically dead on all traces.

assign-null analysis Statically identify source locations at which it is safe to assign null to a heap reference, which is dynamically dead in all traces.

The free analysis may be used with or without a garbage collector. In Chapter 5 we showed that the potential for space savings beyond GC is on average 39% if an object is freed as soon it is dynamically dead in a trace.

The assign-null analysis leads to space saving by allowing the GC to collect more space. In Chapter 5 we showed that assigning null to heap references immediately after their last use has an average space-saving potential of 15% beyond existing GCs. Free analysis could be used with runtime GC in standard Java environments (see Section 6.2) and without GC for JavaCard.

Both of these algorithms handle heap references and destructive updates. They employ both forward (history) and backward (future) information on the behavior of the program. This allows us to free more objects than reachability based compile-time garbage collection mechanisms (e.g., [7, 32]), which only consider the history.

6.1.3 A Motivating Example

Fig. 6.1 shows a program that creates a singly-linked list and then traverses it. We would like to verify that the free property $\langle 10, y \rangle$ holds, i.e., a `free y` statement can be added immediately after line 10. The free property $\langle 10, y \rangle$ holds because once a list element is traversed in a program state $\sigma_i = \langle store_i, 10 \rangle$ along any trace π, it cannot be accessed in the suffix of π (i.e., $\pi^{i+1} = \sigma_{i+1}, \sigma_{i+2}, \ldots$). In the sequel, we show how to formulate this property as a heap safety property and how our framework is used to successfully verify it.

It is interesting to note that even in this simple example, standard compile-time garbage collection techniques (e.g., [7, 32]) will not issue such a free statement, since the element referenced by `y` is

```
   class L { // L is a singly linked list
     public L   n;   // next field
     public int val; // data field
   }
   class Main { // Creation and traversal of a singly-linked list
    public static void main(String args[]) {
      L x, y, t;
[1]   x = null;
[2]   while (...) { // list creation
[3]     y = new L();
[4]     y.val = ...;
[5]     y.n = x;
[6]     x = y;
      }
[7]   y = x;
[8]   while (y != null) {         // list traversal
[9]     System.out.print(y.val);
[10]    t = y.n;
[11]    y = t;
      }
    }
   }
```

Figure 6.1: A program for creating and traversing a singly linked list.

reachable via a heap path starting from x. Furthermore, integrating limited information on the future of the computation such as liveness of local reference variables (e.g., [1]) is insufficient for issuing such free statement. Nevertheless, our analysis is able to verify that the list element referenced by y is no longer needed, by investigating the trace suffixes starting at a program state $\sigma_i = \langle store_i, 10 \rangle$ for all traces.

6.1.4 A Framework for Verifying Heap Safety Properties

Our framework is conservative, i.e., if a heap safety property is verified, it is never violated on any trace of the program. As usual for a conservative framework, we might fail to verify a safety property which holds on all traces of the program.

Assuming the safety property is described by an HSA, we instrument the program semantics to record the automaton state for every program object. First-order logical structures are used to represent a global state of the program. We augment this representation to incorporate information about the automaton state of every heap-allocated object.

Our abstract domain uses first-order 3-valued logical structures to represent an abstract global state of the program, which represents several (possibly an infinite number of) concrete logical structures [52].

We use *canonic abstraction* that maps concrete program objects (i.e., individuals in a logical structure) to abstract program objects based on the properties associated with a program object. In particular, the abstraction is refined by the automaton state associated with every program object.

For the purpose of our analyses one needs to: (i) consider information on the history of the computation, to approximate the heap paths, and (ii) consider information on the future of the computation, to approximate the future use of references. Our approach here uses a forward analysis, where the automaton maintains the temporal information needed to reason about the future of the computation.

In principle we could have used a forward analysis identifying heap-paths integrated into a backward analysis identifying future uses of heap references [58]. However, we find the cost of merging forward and backward information too expensive for a heap analysis as precise as ours.

6.2 Specifying Compile-Time Memory Management Properties via Heap Safety Properties

In this section, we introduce heap safety properties in general, and a specific heap safety property that allows us to identify source locations at which heap-allocated objects may be safely freed.

Informally, a heap safety property may be specified via a heap safety automaton (HSA), which is a deterministic finite state automaton that defines the valid sequences of events for a single object in the program. An HSA defines a prefix-closed language, i.e., every prefix of a valid sequence of events is also valid. This is formally defined as follows:

Definition 6.2.1 (Heap Safety Automaton (HSA)) *A heap safety automaton $A = \langle \Sigma, Q, \delta, \mathrm{init}, F \rangle$ is a deterministic finite state automaton, where Σ is the automaton alphabet which consists of observable events, Q is the set of automaton states, $\delta : Q \times \Sigma \to Q$ is the deterministic transition function mapping a state and an event to a single successor state,* init $\in Q$ *is the* initial state*,* err $\in Q$ *is a distinguished* violation state *(the sink state), for which for all $a \in \Sigma$, $\delta(err, a) = err$, and $F = Q \setminus \{\mathrm{err}\}$ is the set of accepting states.*

In our framework, we associate an HSA state with every object in the program, and verify that on all program traces, all objects are in an accepting state. The HSA is used to define an instrumented semantics, which maintains the state of the automaton for each object. The automaton state is *independently* maintained for every program object. However, the same automaton A is used for all program objects.

When an object o is allocated, it is assigned the initial automaton state. The state of an object o is then updated by automaton transitions corresponding to events associated with o, triggered by program statements.

We now show how the free property is formulated using an HSA, and begin with an example showing how the free property $\langle 10, \mathrm{y} \rangle$ is formulated using an HSA $A^{free}_{10,y}$. In the sequel, we make a simpli-

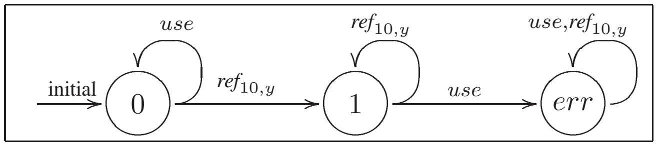


Figure 6.2: A heap safety automaton $A^{free}_{10,y}$ for free y at line 10.

fying assumption and focus on verification of the property for a single program point. In Section 6.5.2 we discuss a technique for verification for a set of program points.

Example 6.2.2 Consider the example program of Fig. 6.1. We would like to verify that a `free y` statement can be added immediately after line 10, i.e., a list element can be freed as soon as it has been traversed in the loop. The HSA $A^{free}_{10,y}$ shown in Fig. 6.2 represents the free property $\langle 10, y \rangle$. States 0 and 1 are accepting, while the state labelled *err* is the violation state. The alphabet of this automaton consists of two events associated with a program object o: (i) *use*, which corresponds to a use of a reference to o, and (ii) $ref_{10,y}$, which is triggered when program execution is immediately after execution of the statement at line 10 and `y` references o.

The HSA is in an accepting state along a trace π iff o can be freed in the program after line 10. Thus, when on all traces, for all program objects o, only accepting states are associated with o, we conclude that `free y` can be added immediately after line 10.

First, when an object is allocated, it is assigned the initial state of $A^{free}_{10,y}$ (state 0). Then, a use of a reference to an object o (a *use* event) does not change the state of $A^{free}_{10,y}$ for o (a self-loop on state 0). When the program is immediately after line 10 and y references an object o ($ref_{10,y}$ event), o's automaton state is set to 1. If a reference to o is used further, (i.e., in the subsequent program configurations along the trace a reference to o is used), and o's automaton state is 1 the automaton state for o reaches the violation state of the automaton. In that case the property is violated, and it is not possible to add a `free y` statement immediately after line 10 since it will free an object that is needed later in the program. However, in the program of Fig. 6.1, references to objects referenced by `y` at line 10 are not used further, hence the property is not violated, and it is safe to add a `free y` statement at this program point. Indeed, in Section 6.4 we show how the *free* $\langle 10, y \rangle$ property is verified.

An arbitrary free property is formulated as a heap safety property using an HSA similar to the one shown in Fig. 6.2 where the program point and program variable name are set accordingly.

It should be noted that in Java, free statements are not supported. Therefore, we assume that equivalent free annotations are issued, and could be exploited by the runtime environment. For example, a Java Virtual Machine (JVM) may include an internal free function, and the Just-In-Time (JIT) compiler (which is a runtime compiler included in the JVM) computes where calls to the function can be added. The internal free function can then indicate to GC not to trace the freed object.

Predicates	Intended Meaning
$\mathit{after}[pt]()$	The program execution is immediately after program point pt
$x(o)$	The program variable x references the object o
$f(o_1, o_2)$	The field f of the object o_1 points to the object o_2
$s[q](o)$	The the current state of o's automaton is q

Table 6.1: Predicates for partial Java semantics.

6.3 Instrumented Concrete Semantics

We define an instrumented concrete semantics that maintains an automaton state for each heap-allocated object. In Section 6.3.1, we use first-order logical structures to represent a global state of the program and augment this representation to incorporate information about the automaton state of every heap-allocated object. Then in Section 6.3.2, we describe an operational semantics manipulating instrumented configurations.

6.3.1 Representing Program Configurations using First-Order Logical Structures

A program state $\sigma_i = \langle store_i, \mathrm{pt}_i \rangle$ along a trace π can be naturally expressed as a first-order logical structure in which each individual corresponds to a heap-allocated object and predicates of the structure correspond to properties of heap-allocated objects. In the rest of this chapter, we work with a fixed set of predicates denoted by P.

Definition 6.3.1 (Program Configuration) *A* program configuration *is a* 2*-valued first-order logical structure* $C^\natural = \langle U^\natural, \iota^\natural \rangle$ *where:*

- $U^\natural$ *is the universe of the* 2*-valued structure. Each individual in* $U^\natural$ *represents an allocated heap object.*
- $\iota^\natural$ *is the interpretation function mapping predicates to their truth-value in the structure, i.e., for every predicate* $p \in P$ *of arity* k*,* $\iota^\natural(p) \colon {U^\natural}^k \to \{0, 1\}$.

We use the predicates of Table 6.1 to record information used by the properties discussed in this chapter. The nullary predicate $\mathit{after}[pt]$ records the program location in a configuration and holds in configurations in which the program is immediately after line pt. The unary predicate x records the value of a reference variable `x` and holds for the individual referenced by `x`. The binary predicate f records the value of a field reference; $f(o_1, o_2)$ holds when the field `f` of o_1 points to the object o_2.

Unary predicates of the form $s[q]$ (referred to as *automaton state predicates*) maintain temporal information by maintaining the automaton state for each object. Such predicates record history informa-

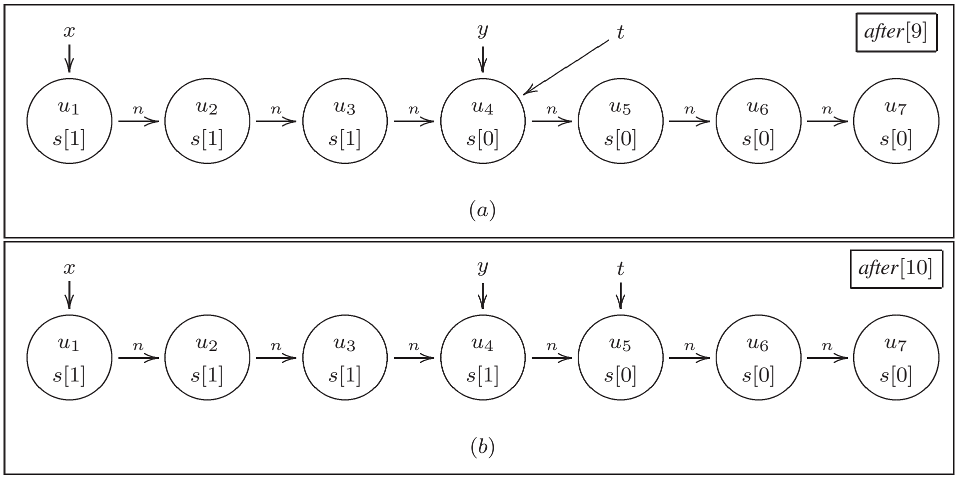


Figure 6.3: Concrete program configurations (a) before — and (b) immediately after execution of `t = y.n` at line 10.

tion that is used to refine the abstraction. In Section 6.3.2 we describe how these predicates are updated. The abstraction is refined further by predicates that record spatial information, such as *reachability* and *sharing* (referred to as *instrumentation predicates* in [52]).

Program configurations are depicted as directed graphs. Each individual of the universe is displayed as a node. Unary predicates which hold for an individual (node) are drawn inside the node. Unary predicates of the form x, where `x` is a reference variable, are shown as an edge from the predicate symbol to the node in which it holds since they can only hold for a single individual. The name of a node is written inside the node using an *italic* face. Node names are only used for ease of presentation and do not affect the analysis. A binary predicate $p(u_1, u_2)$ which evaluates to 1 is drawn as directed edge from u_1 to u_2 labelled with the predicate symbol. Finally, a nullary predicate $p()$ is drawn inside a box.

Example 6.3.2 The configuration shown in Fig. 6.3(a) corresponds to a program state in which execution is immediately after line 9. In this configuration, a singly-linked list of 7 elements has been traversed up to the 4-th element (labelled u_4) by the reference variable y, and the reference variable t still points to the same element as y. This is shown in the configuration by the fact that both predicates $y(o)$ and $t(o)$ hold for the individual u_4. Directed edges labelled by n correspond to values of the n field. The nullary predicate *after*$[9]()$ shown in a box in the upper-right corner of the figure records the fact that the program is immediately after line 9. The predicates $s[0](o)$ and $s[1](o)$ record which objects are in state 0 of the automaton and which are in state 1. For example, the individual u_3 is in automaton state 1 and the individual u_4 is in automaton state 0.

6.3.2 Operational Semantics

Program statements are modelled by generating the logical structure representing the program state after execution of the statement. In [52] it was shown that first-order logical formulae can be used to formally define the effect of every statement. In particular, first-order logical formulae can be used to model the change of the automaton state of every affected individual, reflecting transition-updates in an ordered sequential manner.

In general, the operational semantics associates a program statement with a set of HSA events that update the automaton state of program objects. The translation from the set of HSA events to first-order logical formulae reflecting the change of the automaton state of every affected individual is automatic. We now show how program statements are associated with $A^{free}_{pt,x}$ events. For expository purposes, and without loss of generality, we assume the program is normalized to a 3-address form. In particular, a program statement may manipulate reference expressions of the form `x` or `x.f`.

Object Allocation

For a program statement `x = new C()` for allocating an object, a new object o_{new} is allocated, which is assigned the initial state of the HSA, i.e., we set the predicate $s[init](o_{new})$ to 1.

Example 6.3.3 Consider the HSA $A^{free}_{10,y}$ shown in Fig. 6.2. For this HSA we define a set of predicates $\{s[0](o), s[1](o), s[err](o)\}$ to record the state of the HSA individually for every heap-allocated object. Initially, when an object o is allocated at line 3 of the example program, we set $s[0](o)$ to 1, and other state predicates 0 on o.

Use Events

Table 6.2 shows the use events fired by each kind of a program statement. These events correspond to the r-values, which occur in these statements. In particular, (i) a use of `x` in a program statement updates the automaton state of the object referenced by `x` with a *use* event, and (ii) a use of the field `f` of the object referenced by `x` in a program statement updates the automaton state of the object referenced by `x.f` with a *use* event. For example, the statement `x = y.f` triggers use events for y and $y.f$, which update the automaton state of the object referenced by `y` with a *use* event, and update the automaton state of the object referenced by `y.f` with a *use* event. The order in which use events are triggered does not matter. However, in general, the order should be consistent with the HSA.

ref$_{pt,x}$ Events

For a free property $\langle pt, \mathtt{x}\rangle$, the corresponding automaton $A^{free}_{pt,x}$ employs *ref*$_{pt,x}$ events in addition to *use* events. A *ref*$_{pt,x}$ event is triggered to update the automaton state of the object referenced by `x` when the

statement	**use events are triggered for an object referenced by**
x = y	y
x = y.f	$y, y.f$
x.f = null	x
x.f = y	x, y
x binop y	x, y

Table 6.2: Use events triggered by r-values of expressions in program statements.

current program point is pt. A statement at pt may trigger both *use* events and a $\mathit{ref}_{pt,x}$ event. In this case the $\mathit{ref}_{pt,x}$ event is triggered only after the *use* events corresponding to the program statement at pt are triggered.

Example 6.3.4 Fig. 6.3 shows the effect of the `t = y.n` statement at line 10, where the statement is applied to the configuration labelled by (a). First, this statement updates the predicate t to reflect the assignment by setting it to 1 for u_5, and setting it to 0 for u_4. In addition, it updates the program point by setting $\mathit{after}[10]()$ to 1 and $\mathit{after}[9]()$ to 0. Then, two *use* events followed by a $\mathit{ref}_{10,y}$ event are triggered: (i) use of the object referenced by `y`, causing the object u_4 to remain at automaton state 0, i.e., $s[0](u_4)$ remains 1; (ii) use of the object referenced by `y.n`, causing the object u_5 to remain at automaton state 0, i.e., $s[0](u_5)$ remains 1; and (iii) the event $\mathit{ref}_{10,y}$ for the object referenced by `y`, causing the object u_4 to change its automaton state to 1, i.e., setting the predicate $s[1]$ to 1 for u_4, and setting the predicate $s[0]$ to 00 for u_4. After applying the above updates we end up with the logical structure shown in Fig. 6.3(b), reflecting both the changes in the store, and the transitions in the automaton state for program objects.

6.4 An Abstract Semantics

In this section, we present a conservative abstract semantics [14] abstracting the concrete semantics of Section 6.3. In Section 6.4.1, we describe how abstract configurations are used to finitely represent multiple concrete configurations. In Section 6.4.2, we describe an abstract semantics manipulating abstract configurations.

6.4.1 Abstract Program Configurations

We conservatively represent multiple concrete program configurations using a single logical structure with an extra truth-value $1/2$ which denotes values which may be 1 and may be 0.

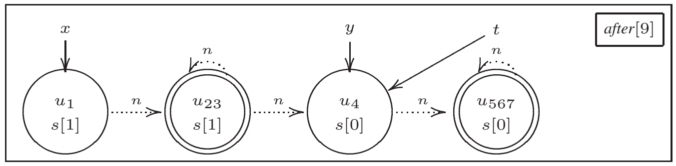


Figure 6.4: An abstract program configuration representing the concrete configuration of Fig. 6.3(a).

Definition 6.4.1 (Abstract Configuration) *An* abstract configuration *is a* 3*-valued logical structure* $C = \langle U, \iota \rangle$ *where:*

- U *is the universe of the* 3*-valued structure. Each individual in* U *represents possibly many allocated heap objects.*

- ι *is the interpretation function mapping predicates to their truth-value in the structure, i.e., for every predicate* $p \in P$ *of arity* k*,* $\iota(p) \colon U^k \to \{0, 1/2, 1\}$*. For example,* $\iota(p)(u) = 1/2$ *indicates no restriction on the truth value of* p*, i.e., it may be* 1 *for some of the objects represented by* u *and may be* 0 *for some of the objects represented by* u*.*

We allow an abstract configuration to include a *summary node*, i.e., an individual which corresponds to one or more individuals in a concrete configuration represented by that abstract configuration. Technically, we use a designated unary predicate *sm* to maintain summary-node information. A summary node u has $sm(u) = 1/2$, indicating that it may represent more than one node.

Abstract program configurations are depicted by enhancing the directed graphs from Section 6.3 with a graphical representation for $1/2$ values: a binary predicate value which evaluates to $1/2$ is drawn as dashed directed edge labelled with the predicate symbol, and a summary node is drawn as circle with double-line boundaries.

Example 6.4.2 The abstract configuration shown in Fig. 6.4 represents the concrete configuration of Fig. 6.3(a). The summary node labelled by u_{23} represents the linked-list items u_2 and u_3, both having the same values for their unary predicates. Similarly, the summary node u_{567} represents the nodes u_5, u_6, and u_7.

Note that this abstract configuration represents many configurations. For example, it represents any configuration in which program execution is immediately after line 9 and a linked-list with at least 5 items has been traversed up to some item after the third item.

Embedding

We now formally define how configurations are represented using abstract configurations. The idea is that each individual from the (concrete) configuration is mapped into an individual in the abstract configuration. More generally, it is possible to map individuals from an abstract configuration into an individual in another less precise abstract configuration. The latter fact is important for our abstract transformer.

Formally, let $C = \langle U, \iota \rangle$ and $C' = \langle U', \iota' \rangle$ be abstract configurations. A function $f\colon U \to U'$ such that f is surjective is said to *embed C into C'* if for each predicate p of arity k, and for each $u_1, \ldots, u_k \in U$ the following holds:

$$\iota(p(u_1, \ldots, u_k)) = \iota'(p(f(u_1), \ldots, f(u_k))) \text{ or } \iota'(p(f(u_1), \ldots, f(u_k))) = 1/2$$
$$and$$
$$\text{for all } u' \in U' \text{ s.t. } |\{u \mid f(u) = u'\}| > 1 : \iota'(sm)(u') = 1/2$$

One way of creating an embedding function f is by using *canonical abstraction*. Canonical abstraction maps concrete individuals to an abstract individual based on the values of the individuals' unary predicates. All individuals having the same values for unary predicate symbols are mapped by f to the same abstract individual. Only summary nodes (i.e., nodes with $sm(u) = 1/2$) can have more than one node mapped to them by the embedding function.

Note that since automaton states are represented using unary predicates, the abstraction is refined by the automaton state of each object. This provides a simple property-guided abstraction since individuals at different automaton states are not summarized together. Indeed, adding unary predicates to the abstraction increases the worst-case cost of the analysis. In particular, when considering the effect of the statement, the cost of updating the automaton state predicates is proportional to $|Q|$. Our initial experiments indicate this overhead is tolerable in our case.

6.4.2 Abstract Semantics

Implementing an abstract semantics directly manipulating abstract configurations is non-trivial since one has to consider all possible relations on the (possibly infinite) set of represented concrete configurations.

The *best* conservative effect of a program statement [14] is defined by the following 3-stage semantics: (i) a concretization of the abstract configuration is performed, resulting in all possible configurations *represented* by the abstract configuration; (ii) the program statement is applied to each resulting concrete configuration; (iii) abstraction of the resulting configurations is performed, resulting with a set of abstract configurations *representing* the results of the program statement.

Example 6.4.3 Fig. 6.5 shows the stages of an abstract action: first, concretization is applied to the abstract configuration resulting with an infinite set of concrete configuration represented by it. The program statement update is then applied to each of these concrete configurations. Following the program statement update, automaton transition updates are applied as described in Section 6.3.2. That is, at first *use* events are triggered to update the automaton states of the objects referenced by `y` and `y.n`. Then,

Figure 6.5: Concretization, predicate-update, automaton transition updates, and abstraction for the statement `t = y.n` in line 10.

a $\mathit{ref}_{10,y}$ event is triggered to update the automaton state of the object referenced by y. Finally, after all transition updates have been applied, the resulting concrete configurations are abstracted resulting with a finite representation.

Our prototype implementation described in Section 6.6.1 operates directly on abstract configurations using *abstract transformers*, thereby obtaining actions which are more conservative than the ones obtained by the best transformers. Interestingly, since temporal information is encoded as part of the concrete configuration via automaton state predicates, the soundness of the abstract transformers is guaranteed by the *Embedding Theorem* of [52]. Our experience shows that the abstract transformers used in the implementation are still precise enough to allow verification of our heap safety properties.

When the analysis terminates, we verify that in all abstract configurations, all individuals are associated with an accepting automaton state, i.e., in all abstract configurations, for every individual o, $s[err](o)$ evaluates to 0.

The soundness of our abstraction guarantees that in all concrete configurations, all individuals are associated with an accepting automaton state, and we conclude that the property holds.

6.5 Extensions

In this section, we extend the applicability of our framework by: (i) formulating an additional compile-time memory management property — the assign-null property; and (ii) extending the framework to simultaneously verify multiple properties.

6.5.1 Assign-Null Analysis

The assign-null problem determines source locations at which statements assigning null to heap references can be safely added. Such null assignments lead to objects being unreachable earlier in the program, and thus may help a runtime garbage collector reclaim objects earlier, thus saving space. As in Section 6.2, we show how to verify the assign-null property for a single program point and discuss verification for a set of program points in Section 6.5.2.

The assign-null property $\langle pt, \mathtt{x}, \mathtt{f}\rangle$ (see Definition 2.1.14) allows assigning null to a dead heap reference. We now show how this property is formulated using an HSA, and begin with an example showing how the assign-null property $\langle 10, \mathtt{y}, \mathtt{n}\rangle$ is formulated using an HSA $A^{an}_{10,y,n}$.

Example 6.5.1 Consider again the example program shown in Fig. 6.1. We would like to verify that a `y.n = null` statement can be added immediately after line 10, i.e., a reference connecting consecutive list elements can be assigned null as soon as it is traversed in the loop. The HSA $A^{an}_{10,y,n}$ shown in Fig. 6.6 represents the assign-null $\langle 10, \mathtt{y}, \mathtt{n}\rangle$ property. The alphabet of this automaton consists of the following events for an object o: (i) use_n, which corresponds to a use of the field n of the object; (ii) def_n,

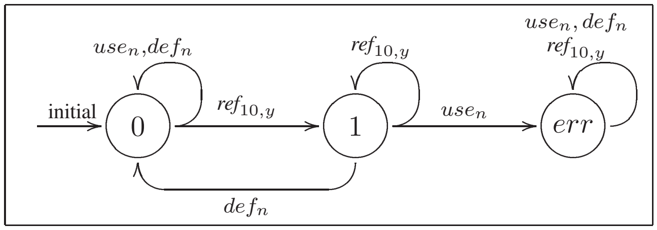


Figure 6.6: A heap safety automaton $A^{an}_{10,y,n}$ for assign null to $y.n$ at 10.

which corresponds to a definition of the field n of the object; (iii) *ref*$_{10,y}$, which is triggered when program execution is immediately after execution of the statement in 10 and y references the object o. Our implementation verifies assign-null $\langle 10, \mathtt{y}, \mathtt{n} \rangle$ property, by applying the framework with $A^{an}_{10,y,n}$ to the example program. Notice that this automaton contains a back arc and thus is more complex than the one for the free property.

First, when an object o is allocated it is assigned the initial state of $A^{an}_{10,y,n}$. Then, uses or definitions of the n field of an object o (a *use*$_n$ event or a *def*$_n$ event, respectively) do not change the state of $A^{an}_{10,y,n}$ for o (the self-loop in state 0). When the program is immediately after line 10 and y references an object o (*ref*$_{10,y}$ event), o's automaton state is set to 1. Now, if the n field of o is further defined (i.e., a *def*$_n$ event occurs for o in the subsequent program configurations along the trace), and o's automaton state is 1, the automaton state for o gets back to the initial state (state 0). However, if the n field of o is further used (i.e., a *use*$_n$ event occurs for o in the subsequent program configurations along the trace) before this field is redefined, and o's automaton state is 1 the automaton state for o reaches the *violation state* of the automaton. However, in the program of Fig. 6.1, the n-field references emanating from objects referenced by y at line 10 are not further used before redefined, hence the property is not violated, and it is safe to add a `y.n = null` statement at this program point.

An arbitrary assign-null property is formulated as a heap safety property using an HSA similar to the one shown in Fig. 6.6 where the program point, variable and field names are set accordingly.

6.5.2 Simultaneous Verification of Multiple Properties

So far we showed how to verify the free and assign-null properties for a single program point. Clearly, in practice one wishes to verify these properties for a set of program points without repeating the verification procedure for each program point. Our framework supports simultaneous verification of multiple properties, and in particular verification of properties for multiple program points. Assuming HSA$_1, \ldots,$ HSA$_k$ describe k verification properties, then k automaton states $s_1, \ldots, s_k$ are maintained for every program object, where s_i maintains an automaton state for HSA$_i$. Technically, as described in Section 6.3, a state s_i is represented by automaton state predicates $s_i[q]$, where q ranges over the states of HSA_i. The events associated with the automata $\mathrm{HSA}_1, \ldots, \mathrm{HSA}_k$ at a program point are triggered simultaneously, updating the corresponding automaton state predicates of individuals.

The worst-case cost of simultaneous verification of properties is higher than the worst-case cost of verifying the same properties one by one. However, verifying properties one by one ignores the potential of computing overlapping heap information just once, where in simultaneous verification of properties this overlap is taken into consideration. Thus, we believe that in practice simultaneous verification of properties may achieve a lower cost than verifying the properties one by one.

Interestingly, if we limit our verification of free $\langle pt, \mathtt{x}\rangle$ properties to ones where `x` is used at pt (i.e., `x` is used in the statement at pt), then the following features are obtained: (i) an object is freed just after it is referenced last, i.e., exactly at the earliest time possible, and (ii) an object is freed "exactly once", i.e., there are no redundant frees of variables referencing the same object.

A similar choice for assign-null properties assigns null to a heap reference immediately after its last use. The motivation for this choice of verification properties comes from our results in Chapter 5, showing an average of 15% potential space savings beyond a run-time garbage collector if a heap reference is assigned null just after its last use. However, we note that our framework allows verification of arbitrary free and assign-null properties, which may yield further space reduction.

6.6 Empirical Results

We implemented the static analysis algorithms for verifying free and assign-null properties, and applied it to several programs, including JavaCard programs.

Our benchmark programs were used as a proof of concept. Due to scalability issues our benchmarks only provide a way to verify that our analysis is able to locate the static information at points of interest, and we do not measure the total savings. In particular the benchmarks provide three kinds of proof of concept: (i) we use small programs manipulating a linked-list to demonstrate the precision of our technique; moreover, we show that less precise analyses as points-to analysis is insufficient for proving free and assign-null properties for these programs. (ii) we demonstrate how our techniques could be used to verify/automate manual space-saving rewritings. In particular, in Chapter 4 the code of the `javac` Java compiler was manually rewritten in order to save space. Here, we verify the manual rewritings in `javac`, which assign null to heap references, by applying our prototype implementation to a Java code fragment emulating part of the Parser facility of `javac`; (iii) we demonstrate how our techniques could play an important role in the design of future JavaCard programs. This is done by rewriting existing JavaCard code in a more modular way, and showing that our techniques may be used to avoid the extra space overhead due to the modularity.

Program	Description	Free		Assign Null	
		space	time	space	time
Loop	the running example	1.71	1.93	1.37	1.76
CReverse	constructive reverse of a list	3.03	5.17	2.58	4.79
Delete	delete an element from a list	5.33	19.66	4.21	13.84
DLoop	doubly linked list variant of Loop	2.09	2.91	1.75	2.68
DPairs	processing pairs in a doubly-linked list	2.76	5.01	2.54	4.86
small javac	emulation of `javac`'s parser facility	N/A	N/A	16.02	43.84
JavaPurse' slice	a JavaCard simple electronic purse	56.3	979	56.15	991
GuessNumber' slice	a JavaCard distributed guess number game	9.99	17.3	N/A	N/A

Table 6.3: Analysis cost for the benchmark programs. Space is measured in MB, and time is measured in seconds.

6.6.1 Implementation

Following a brief description of our implementation. A more detailed technical view is provided at Appendix A. Our implementation consists of the following components: (i) a front-end, which translates a Java program (.class files) to a TVLA program [41]; (ii) an analyzer, which analyzes the TVLA program; (iii) a back-end, which answers our verification question by further processing of the analyzer output.

The front end (J2TVLA), developed by R. Manevich, is implemented using the Soot framework [68]. The analyzer, implemented using TVLA, includes the implementation of static analysis algorithms for the free and assign-null property verification. TVLA is a parametric framework that allows the heap abstractions and the abstract transformers to be easily changed. In particular, for programs manipulating lists we obtain a rather precise verification algorithm by relying on spatial instrumentation predicates, that provide sharing, reachability and cyclicity information for heap objects [52]. For other programs, allocation-site information for heap objects suffices for the verification procedure. In both abstractions interprocedural information is computed [49]. Finally, our implementation allows simultaneous verification of several free or assign-null properties, by maintaining several automaton states per program object.

The back-end, implemented using TVLA libraries, traverses the analysis results, i.e., the logical structures at every program point, and verifies that all individuals are associated with an accepting state. For a single property, we could abort the analyzer upon reaching a non-accepting state on some object and avoid the back-end component. However, in the case of simultaneous verification of multiple safety properties, this would not work and the back-end is required.

6.6.2 Benchmark Programs

Table 6.3 shows our benchmark programs. The first 4 programs involve manipulations of a singly-linked list. `DLoop, DPairs` involve a doubly-linked list manipulation. `small javac` is motivated by our experiments in Chapter 4, where we manually rewrite the code of the `javac` compiler, issuing null assignments to heap references. We can now verify our manual rewriting by applying the corresponding assign-null properties to Java code emulating part of the parser facility in `javac`.

The last two benchmarks are JavaCard programs. `JavaPurse` is a simple electronic cash application, taken from Sun JavaCard samples [35]. In `JavaPurse` a fixed set of loyalty stores is maintained, so every purchase grants loyalty points at the corresponding store. `GuessNumber` [44] is a guess number game over mobile phone SIM cards, where one player (using a mobile phone) picks a number, and other players (using other mobile phones) try to guess the number.

Due to memory constraints, JavaCard programs usually employ a static allocation regime, where all program objects are allocated when the program starts. This leads to non-modular and less reusable code, and to more limited functionality. For example, in the `GuessNumber` program, a global buffer is allocated when the program starts and is used for storing either a server address or a phone number. In `JavaPurse`, the number of stores where loyalty points are granted is fixed.

A better approach that addresses the JavaCard memory constraints is to rewrite the code using a natural object-oriented programming style, and to apply static approaches to free objects not needed further in the program. Thus, we first rewrite the JavaCard programs to allow more modular code in the case of `GuessNumber`, and to lift the limitation on the number of stores in `JavaPurse`. Then, we apply our free analysis to the rewritten code, and verify that an object allocated in the rewritten code can be freed as soon it is no longer needed. In `JavaPurse` we also apply our assign null analysis and verify that an object allocated in the rewritten code can be made unreachable as soon it is no longer needed (thus, a runtime garbage collector may collect it). Concluding, we show that in principle the enhanced code bears no space overhead compared to the original code when the free or the assign-null analysis is used.

6.6.3 Results

Our experiments were done on a 900 Mhz Pentium-III with 512 MB of memory running Windows 2000. Table 6.3 shows the space and time the analysis takes. In `Loop` we verify our free $\langle 10, \mathtt{y}\rangle$ and assign-null $\langle 10, \mathtt{y}, \mathtt{n}\rangle$ properties. For `CReverse` we verify an element of the original list can be freed as soon it is copied to the reversed list. In `Delete` we show an object can be freed as soon it is taken out of the list (even though it is still reachable from temporary variables). Turning to our doubly linked programs, we also show objects that can freed immediately after their last use, i.e., when an object is traversed in the loop (`DLoop`), and when an object in a pair is not processed further (`DPairs`). We also verify corresponding null-assignments that make an object unreachable via heap references just after their last use.

For `small javac` we verify that heap references to large objects in a parser class may be assigned null just after their last use. Finally, for scalability reasons we analyze slices of rewritten JavaCard programs. Our current implementation does not include a slicer, thus we manually slice the code. Using the sliced programs we verify that objects allocated due by our rewritings, can be freed as soon they are no longer needed.

We have also tried our benchmarks using a points-to based heap abstraction, which is considered relatively cheap and scalable. We use a flow-sensitive, field-sensitive points-to analysis with unbounded context information [19]. Our results indicate that in all cases but one (assign-null properties for `JavaPurse` benchmark), points-to analysis is insufficient for proving the free and assign-null properties of interest. For `JavaPurse` the points-to analysis is able to prove the assign-null properties of interest since (i) we try to nullify fields emanating from a singleton object, and (ii) field-sensitive information allow disambiguation of the fields emanating from the singleton object.

Chapter 7

Related Work

Work closely related to our research falls into the following categories: (i) applying drag and liveness information for the purpose of space savings, (ii) memory management techniques, and (iii) software verification of safety properties.

7.1 Drag and Liveness Information

Drag Measurements

Röjemo and Runciman [50] originally defined the terms *lag* and *drag*. Their lag and drag measurements are performed for Haskell, a lazy functional language. Our results for Java are not directly comparable with their results, since they use wall-clock time where we use allocation time. Using drag and lag information they demonstrate how to save space by rewriting the code of a Haskell compiler. Their rewritings are based on a deep understanding of the code.

There are other approaches for profiling the memory of the program. For example, [54, 46] show the heap configuration and allocation frequency; such information may help in tracking memory leaks by allowing the heap to be inspected at points during the execution of the program. As noted by Serrano and Boehm [54], tracking memory leaks by inspecting dragged objects is orthogonal to tracking leaks by inspecting the heap during the course of execution. This is since dragged objects reasoning requires future information, i.e., the future use of objects, while heap inspection requires history information, i.e., the current heap paths in the run.

Liveness Information

Liveness analysis [43] may be used in the context of a run-time to reduce the size of the root set [5, 65, 1] (i.e., ignoring dead stack variables and dead global variables) or to reduce the number of scanned references (i.e., ignoring dead heap references). For example, in Agesen et. al [1] static stack reference liveness information is used by a type-accurate GC to reduce the set of root references, thus potentially saving some space. In Chapter 5 we show a 2% upper bound for space savings achievable through any static stack reference liveness algorithm. This upper bound is close to the actual space savings reported in [1], which lead us to conclude that: (i) the static analysis algorithm reported there is precise enough, and (ii) as noted in [1]: "the main benefit of stack reference liveness analysis is preventing bad surprises".

In Hirzel et. al [28] the impact of stack reference and global reference liveness information on space consumption is studied for C programs in a garbage-collected environment. Interestingly, the trend of our results for stack reference and global reference liveness information is in line with their results. However, they do not study heap liveness information. In addition, the measurements technique reported there leads to an infeasible amount of information, which requires approximations. In Section 5.2 we explain why our measurement technique maintains a feasible amount of information, thus no further approximations are required. In particular, our technique is applicable to large applications. Finally, in [28] two runs of the program are required for obtaining the information. Our technique directly computes the impact of complete heap liveness information on space consumption using a single run. This allows us to handle non-deterministic (e.g., multithreaded) applications.

For the purpose of estimating the potential savings due to liveness (and drag) information we approximate reachability information by sampling the reachable heap graph, i.e., by frequent invocation of GC and then traversing the reachable heap. In Hertz et. al [27, 26] reachability information is measured by recording the last time a reference to an object is deleted, and propagating the information on the heap graph when the program terminates. In a private communication with Hertz [25] we verified together that reachability information could be collected in either way, i.e., using our technique or using the technique described in [27, 26]. In any case, our measurements in Chapter 5 report finer reachability information; for example, we report the last time an object is reachable along a heap path starting from a stack variable (since an object may still be reachable along a heap path starting from other root kinds), and the last time an object is reachable along a heap path starting at from live stack variable.

7.2 Memory Management Techniques

Compile-Time GC

Our free property (see Chapter 6) falls in the *compile-time garbage collection* research domain, where techniques are developed to identify and recycle garbage memory cells at compile-time. Most work has been done for functional languages [7, 32, 21, 24, 38]. In this thesis, we show a free analysis, which handles a language with destructive updates, that may reclaim an object still reachable in the heap, but not needed further in the run.

Escape Analysis

Escape analysis, which allows stack allocating heap objects, has been recently applied to Java [9, 70, 10]. In this technique an object is freed as soon as its allocating method returns to its caller. While this technique has shown to be useful, it is limited to objects that do not escape their allocating method. Our technique for providing free and assign-null information (see Chapter 6) applies to all program objects, and allows freeing objects before their allocating method returns.

Region-Based Memory Management

In region-based memory management [8, 67, 2, 23], the lifetime of an object is predicted at compile-time. An object is associated with a memory region, and the allocation and deallocation of the memory region are inferred automatically at compile time. It would be interesting to instantiate our framework presented in Chapter 6 with a static analysis algorithm for inferring earlier deallocation of memory regions.

7.3 Software Verification of Safety Properties

In Chapter 6 we present a framework for verifying local temporal heap safety properties. One of the main difficulties in verifying local temporal heap safety properties is considering the effect of aliasing in a precise-enough manner. Some of the previous work on software verification allows universally quantified specifications similar to our local heap safety properties (e.g., [6, 12]). We are the first to apply such properties to compile-time memory management and to employ a high-precision analysis of the heap.

ESP [15] uses a preceding pointer-analysis phase and uses the results of this phase to perform finite-state verification. Separating verification from pointer-analysis may generally lead to imprecise results.

The Bandera project [12] uses the Bandera specification language (BSL) [13] to specify properties of software systems. Bandera constructs a finite-state model of the program and uses existing model-checkers (e.g., SPIN [30]) to perform verification. BSL allows universally quantified specifications which are similar to our local heap safety properties. However, the abstractions currently applied by Bandera to verify these properties may generally lead to results that are less precise than ours.

The SLIC specification language [6] from MSR's SLAM project [42] is a low-level specification language which defines a (possibly infinite) state-machine for tracking temporal safety properties. Although SLIC is more powerful than our local heap safety properties (e.g., it allows counting), the abstraction applied by SLAM to verify SLIC properties may produce results that are less precise than ours.

In Yahav [71] a framework is introduced for verifying safety properties of concurrent Java programs.

The framework allows the verification of an assertion specified as a first-order formula. This is done by verifying that the assertion holds on all reachable states. However, this framework does not support temporal safety properties.

Some prior work used automata to dynamically monitor program execution and throw an exception when the property is violated (e.g.,[53, 11]). Obviously, dynamic monitoring cannot verify that the property holds for all program executions.

Recoding history information for investigating a particular local temporal heap safety property was used for example in [31, 51] (approximating flow dependencies) and [40] (verification of sorting algorithms). The framework presented here generalizes the idea of recording history information by using a heap safety automaton.

Chapter 8

Conclusion

In this thesis we study the limits of current garbage collection techniques, and show how static information regarding heap behavior can aid in closing the gap between reclaiming part of the unneeded program memory (e.g., collecting according to reachability) to retaining just the memory needed to run the program.

We present new dynamic algorithms for measuring the potential space savings beyond current GCs that allow profiling the memory behavior in a feasible manner. For a set of Java benchmarks, our dynamic measurements show potential space savings of 42% on average if an object is collected assuming drag information is available, i.e., assuming an object is collected as soon it is no longer used by the application. Moreover, our dynamic measurements show that existing techniques fail to exploit the large potential for space savings. Finally, our dynamic measurements reveal that taking advantage of static information regarding heap behavior has a large potential for space savings.

In particular, our dynamic algorithms estimate the potential benefits of enhancing GC with static liveness information including local variabl liveness (stack reference liveness), global variable liveness (global reference liveness), and instance variable liveness (heap reference liveness). The algorithms require one run of the program and can handle non-determinism. For heap reference liveness, these algorithms give the potential space savings if complete heap reference liveness is available; thus, we also measure the potential space savings due to heap liveness using a more realistic approach, by finding places in the code where null can be assigned to heap references.

For a set of Java benchmarks our dynamic measurements show that in general stack reference liveness may yield small benefits (2% potential space savings on average) and that global reference liveness combined with stack reference liveness may yield medium benefits (9% potential space savings on average). Furthermore, for static information regarding heap behavior, our dynamic measurements show that heap reference liveness yields the largest potential benefit. Specifically, for heap reference liveness we measure an average potential savings of 39% using an interface with complete liveness information, and an average savings of 15% using a more restricted interface, where dead heap references are assigned null immediately after their last use.

We present a tool allowing a programmer to find places in code to save space. The tool utilizes our dynamic algorithms for measuring potential space savings. In addition, we show that simple correctness-preserving program transformations, such as assigning null to dead references allow significant space savings. Applying the tool to our set of Java benchmarks, and manually rewriting the code using the simple program transformations lead to average space savings of 14%.

Motivated by our dynamic measurements we develop a framework for proving temporal properties of heap-manipulating programs and instantiate it with static algorithms that identify program points at which memory can be deallocated. In particular, our static algorithms are used to (i) statically identify source locations at which it is safe to insert a free statement in order to deallocate an object, which is dynamically dead on all traces, and (ii) statically identify source locations at which it is safe to assign null to a heap reference, which is dynamically dead in all traces. The algorithms were implemented and applied to Java programs including JavaCard programs to show we can actually deallocate memory at compile-time.

8.1 Further Work

A Framework for dynamic measurements of heap properties

Our dynamic algorithms for profiling the memory behavior of a program include a set of algorithms, each targeted at a (temporal) heap property. We would like to generalize the algorithms using a framework for dynamically quantifying temporal properties in heap manipulating programs. This task is challenging since our properties of interest including non-local temporal properties that involve a transitive closure in the heap graph. For example, the stack liveness property (see Section 5.2.1) combines information on the future use of a stack references together with transitive closure in the heap graph.

Scaling our Static Analysis Algorithms

Our static algorithms are rather precise, but could be quite expensive on large programs. One way to reduce the cost is analyzing part of the program. In particular, our dynamic algorithms could be used to locate source locations that exhibit a large potential for space savings (e.g., allocations sites with large drag). Then, the analysis is limited to code around these space-beneficial locations, using slicing techniques [66]. Moreover, our dynamic algorithms may also be used to provide the input for our static algorithms. For example, for a last use site *use* `x` at pt attributed with large drag, we can try to verify the free property $\langle pt, x\rangle$.

Another way to scale our static analysis algorithm is to reduce the cost of gathering interprocedural information. This could be done by considering a more modular shape analysis [48], where an attempt is made to identify the part of the heap affected by a procedure p, thus ignoring the rest of the heap when analyzing p, saving time and space.

Combining Forward and Backward Heap Information

Both free and assign-null analysis consider information on the future of computations of the program. Traditionally, such future information is approximated using a backward analysis. In particular, classical liveness analysis for scalar variables is computed using a backward analysis [43]. However, for the purpose of our analyses one needs to: (i) consider information on the history of the computation, to approximate the heap paths, and (ii) consider information on the future of the computation, to approximate the future use of references.

In Chapter 6 we use a forward analysis, where the automaton maintains the temporal information needed to reason about the future of the computation. An alternative approach to statically approximate free and assign-null information is to use a forward analysis identifying heap-paths, integrated into a backward analysis identifying future uses of heap references [58]. The cost of merging forward and backward information may be too expensive for a heap analysis as precise as ours, but this approach has the benefit of being used to perform exhaustive approximation of free properties (or assign-null properties) in a program.

Bibliography

[1] Ole Agesen, David Detlefs, and Elliot Moss. Garbage Collection and Local Variable Type-Precision and Liveness in Java Virtual Machines. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 269–279. ACM Press, June 1998.

[2] Alexander Aiken, Manuel F ahndrich, and Raph Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 174–185. ACM Press, June 1995.

[3] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, Univ. of Copenhagen, May 1994. (DIKU report 94/19).

[4] Andrew W. Appel. Runtime tags aren't necessary. *Lisp and Symbolic Computation*, 2:153–162, 1989.

[5] Andrew W. Appel. *Compiling with Continuations*, chapter 16, pages 205–214. CUP, 1992.

[6] T. Ball and S.K. Rajamani. SLIC: A Specification Language for Interface Checking (of C). Technical Report MSR-TR-2001-21, MSR, 2001.

[7] Jeffrey M. Barth. Shifting garbage collection overhead to compile time. *Communications of the ACM*, 20(7):513–518, 1977.

[8] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von neumann machines via region representation inference. In *Symp. on Princ. of Prog. Lang.*, pages 171–183. ACM Press, 1996.

[9] Bruno Blanchet. Escape analysis for object oriented languages. application to Java$^{\text{tm}}$. In *Conf. on Object-Oriented Prog. Syst., Lang. and Appl.*, pages 20–34. ACM Press, 1998.

[10] J-D. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape Analysis for Java. In *Conf. on Object-Oriented Prog. Syst., Lang. and Appl.*, pages 1–19. ACM Press, November 1999.

[11] T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *Symp. on Princ. of Prog. Lang.*, pages 54–66. ACM Press, January 2000.

[12] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, R. Shawn, and L. Hongjun. Bandera: Extracting finite-state models from Java source code. In *Int. Conf. on Soft. Eng.* ACM Press, June 2000.

[13] J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby. A language framework for expressing checkable properties of dynamic software. In *Int. Spin Workshop on Model Check. of Soft.*, volume 1885 of *Lec. Notes in Comp. Sci.*, pages 205–223. Springer-Verlag, 2000.

[14] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Symp. on Princ. of Prog. Lang.*, pages 269–282. ACM Press, 1979.

[15] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 57–68. ACM Press, June 2002.

[16] David Detlefs and Elliot Moss, February 2002. Private Communication.

[17] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 230–241. ACM Press, 1994.

[18] Amer Diwan, Eliot Moss, and Richard Hudson. Compiler support for garbage collection in a statically typed language. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 273–282. ACM Press, June 1992.

[19] M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.* ACM Press, 1994.

[20] Melvin Fitting and Richard Mendelsohn. *First-Order Modal Logic*, volume 277 of *Sythese Library,*. Kluwer Academic Publishers, 1998.

[21] Ian Foster and William Winsborough. Copy avoidance through compile-time analysis and local reuse. In *Proceedings of International Logic Programming Sympsium*, pages 455–469. MIT Press, 1991.

[22] Benjamin Goldberg. Tag-free garbage collection for strongly typed programming languages. *ACM SIGPLAN Notices*, 26(6):165–176, 1991.

[23] Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining region inference and garbage collection. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 141–152. ACM Press, 2002.

[24] G. W. Hamilton. Compile-time garbage collection for lazy functional languages. In *Memory Management, International Workshop IWMM 95*, volume 637 of *Lec. Notes in Comp. Sci.* Springer-Verlag, 1995.

[25] Matthew Hertz, July 2002. Private Communication.

[26] Matthew Hertz, Stephen Blackburn, J. Eliot B. Moss, Kathryn S. McKinley, and Darko Stefanovic. Error-free garbage collection traces: how to cheat and not get caught. In *Int. Conf. on Measurements and Modeling of Comp. Syst.*, pages 140–151. ACM Press, 2002.

[27] Matthew Hertz, Neil Immerman, and J. Eliot B. Moss. Framework for analyzing garbage collection. In *IFIP Int. Conf. on Theor. Comp. Sci.*, pages 230–242. Kluwer Academic Publishers, 2002.

[28] Martin Hirzel, Amer Diwan, and Antony L. Hosking. On the usefulness of liveness for garbage collection and leak detection. In *European Conf. on Object-Oriented Prog.*, volume 2072 of *Lec. Notes in Comp. Sci.*, pages 181–206. Springer-Verlag, 2001.

[29] Martin Hirzel, Amer Diwan, and Antony L. Hosking. On the usefulness of type and liveness accuracy for garbage collection and leak detection. In *Trans. on Prog. Lang. and Syst.* ACM Press, 2002.

[30] G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, 1997.

[31] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 28–40. ACM Press, 1989.

[32] Katsuro Inoue, Hiroyuki Seki, and Hikaru Yagi. Analysis of functional programs to detect run-time garbage cells. *Trans. on Prog. Lang. and Syst.*, 10(4):555–578, October 1988.

[33] JavaCC - The Java Parser Generator. Available at http://www.metamata.com/javacc.

[34] Java Grande Benchmark Suite. Available at http://www.epcc.ed.ac.uk/javagrande.

[35] Java card 2.2 development kit. Available at java.sun.com/products/javacard.

[36] Sun JDK 1.2. Available at http://java.sun.com/j2se.

[37] Sun HotSpot Client 1.3. Available at http://java.sun.com/products/hotspot.

[38] Richard Jones. *Garbage Collection. Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, 1999.

[39] David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *Symp. on Princ. of Prog. Lang.* ACM Press, 2002.

[40] Tal Lev-Ami, Thomas W. Reps, Reinhard Wilhelm, and Mooly Sagiv. Putting static analysis to work for verification: A case study. In *Int. Symp. on Soft. Testing and Anal.*, pages 26–38. ACM Press, 2000.

[41] Tal Lev-Ami and Mooly Sagiv. TVLA: A system for implementing static analyses. In *Static Analysis Symp.*, volume 1824 of *Lec. Notes in Comp. Sci.*, pages 280–301. Springer-Verlag, 2000.

[42] Microsoft Research. The SLAM project, 2001. http://research.microsoft.com/slam/.

[43] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[44] Oberthur card systems. www.oberthurcs.com.

[45] Michael Pan. HUP - a heap usage profiling tool for Java programs. Master's thesis, Tel-Aviv University, 2001.

[46] W. De Pauw and G. Sevitski. Visualizing reference patterns for solving memory leaks in Java. In *European Conf. on Object-Oriented Prog.*, volume 1628 of *Lec. Notes in Comp. Sci.*, pages 116–134. Springer-Verlag, 1999.

[47] Sara Porat, Bilha Mendelson, and Irina Shapira. Sharpening global static analysis to cope with Java. In *CASCON*, 1998.

[48] Noam Rinetzky. Interprocedural shape analysis. Master's thesis, The Technion, 2001.

[49] Noam Rinetzky and Mooly Sagiv. Interprocedural shape analysis for recursive programs. In *Int. Conf. on Comp. Construct.*, volume 2027 of *Lec. Notes in Comp. Sci.*, pages 133–149. Springer-Verlag, 2001.

[50] Niklas Röjemo and Colin Runciman. Lag, drag, void and use—heap profiling and space-efficient compilation revisited. In *SIGPLAN Int. Conf. on Func. Prog.*, pages 34–41. ACM Press, 1996.

[51] J.L. Ross and M. Sagiv. Building a bridge between pointer aliases and program dependences. In *European Symp. on Prog.*, volume 1381 of *Lec. Notes in Comp. Sci.*, pages 221–235. Springer-Verlag, March 1998. Available at "http://www.math.tau.ac.il/∼ sagiv".

[52] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *Trans. on Prog. Lang. and Syst.*, 24(3):217–298, 2002.

[53] F.B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.

[54] Manuel Serrano and Hans-Juergen Boehm. Understanding memory allocation of scheme programs. In *SIGPLAN Int. Conf. on Func. Prog.*, pages 245–256. ACM Press, 2000.

[55] Ran Shaham, Elliot K. Kolodner, and Mooly Sagiv. Automatic removal of array memory leaks in Java. In *Int. Conf. on Comp. Construct.*, volume 1781 of *Lec. Notes in Comp. Sci.*, pages 50–66. Springer-Verlag, April 2000.

[56] Ran Shaham, Elliot K. Kolodner, and Mooly Sagiv. On the effectiveness of GC in Java. In *Int. Symp. on Memory Management*, pages 12–17. ACM Press, October 2000.

[57] Ran Shaham, Elliot K. Kolodner, and Mooly Sagiv. Heap profiling for space-efficient Java. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 104–113. ACM Press, June 2001.

[58] Ran Shaham, Elliot K. Kolodner, and Mooly Sagiv. Backward shape analysis to statically predict heap behavior. Unpublished manuscript, 2002.

[59] Ran Shaham, Elliot K. Kolodner, and Mooly Sagiv. Estimating the impact of heap liveness information on space consumption in Java. In *Int. Symp. on Memory Management*, pages 171–182. ACM, June 2002.

[60] Ran Shaham, Eran Yahav, Elliot K. Kolodner, and Mooly Sagiv. Establishing local temporal heap safety properties with applications to compile-time memory management. Invited to a special issue of the Science of Computer Programming Journal.

[61] Ran Shaham, Eran Yahav, Elliot K. Kolodner, and Mooly Sagiv. Establishing local temporal heap safety properties with applications to compile-time memory management. In *Static Analysis Symp.*, volume 2692 of *Lec. Notes in Comp. Sci.* Springer-Verlag, 2003.

[62] SPECjvm98. Standard Performance Evaluation Corporation (SPEC), Fairfax, VA, 1998. Available at http://www.spec.org/osg/jvm98/.

[63] B. Steensgaard. Points-to analysis in almost-linear time. In *Symp. on Princ. of Prog. Lang.*, pages 32–41. ACM Press, 1996.

[64] Darko Stefanovic and J. Eliot B. Moss. Characterisation of object behaviour in standard ml of new jersey. In *Conf. on Lisp and Func. Programming*, pages 43–54. ACM Press, 1994.

[65] Stephen Thomas. Garbage collection in shared-environment closure reducers: Space-efficient depth first copying using a tailored approach. *Information Processing Letters*, 56(1):1–7, October 1995.

[66] Frank Tip. A survey of program slicing techniques. *J. Prog. Lang.*, 3:121–189, 1995.

[67] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *Symp. on Princ. of Prog. Lang.*, pages 188–201. ACM Press, January 1994.

[68] R. Vallée-Rai, L. Hendren, V. Sundaresan, E. Gagnon P. Lam, and P. Co. Soot - a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[69] D. Viswanathan and S. Liang. Java virtual machine profiler interface. *IBM Systems Journal*, 2000.

[70] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *Conf. on Object-Oriented Prog. Syst., Lang. and Appl.*, pages 187–206. ACM Press, 1999.

[71] E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *Symp. on Princ. of Prog. Lang.*, pages 27–40. ACM Press, March 2001.

Appendix A

QNF prototype

QNF is a prototype for detecting places in Java code where free or assign-null statements can be inserted safely[1]. This guide included in the appendix gives a more technical view of the tool.

A.1 Overview

QNF is a prototype for detecting places in Java code where free or assign-null statements can be inserted safely. The main idea is that a user specifies assign null or free *queries* and the prototype answers these queries. The input to QNF is a Java application (.class files) and a set of queries. For expository purposes we discuss here a query in terms of the Java source code. In Section A.2 we discuss queries more formally and relate them to Jimple code, which serves as a simple representation of the Java code.

A query specifies places in the code (e.g., a line number, a method, a class), candidates for the insertion of a free or an assign-null statement. For example, consider the code in Fig. A.1 showing a part of the `Loop` example. The query

Loop:void main(java.lang.String[])[line32] (A.1)

specifies a free query regarding line 32 in `main` method of `Loop` class. This query as explained in Section A.2 regards the insertion of a `free y` statement or `free y.n` statement immediately after line 32.

QNF answers a query by showing the valid free or assign-null statements corresponding to that query. For example, applying QNF to the `Loop` example with the query (A.1) yields the following output:

FREE=> Loop:void main(java.lang.String[])[line32][bc66][var y] (A.2)

[1]The meaning of a free statement in Java is discussed further in Chapter 6

```
[29]  y = x;
[30]  while (y != null) {         // list traversal
[31]   System.out.print(y.val);
[32]   t = y.n;
[33]   y = t;
     }
```

Figure A.1: A piece of Java code for traversing a list.

This output states that a `free y` may be safely inserted immediately after line 32 (we ignore for the moment the bytecode offset part `[bc66]`)

A.1.1 QNF Phases

Applying the prototype to a Java program consists of the following phases shown in Fig. A.2:

QNF2Jimple The application is first transformed into Jimple format [68]. Jimple provides a simple representation of the code, which is suitable for our query interface. Fig. A.3 shows the Jimple version of the Java code in Fig. A.1.

QNF2Tvla The user browses the Jimple code and selects places candidates for the insertion of a free or an assign-null statement. The queries along with the application are given as input to QNF2Tvla, which produces a TVLA program [41] that serves as an input to our free and assign-null analyzer. QNF2Tvla is based on the J2TVLA front-end developed by R. Manevich.

QNFAnalyzer The analyzer applies the free and assign-null static analysis algorithms described in Chapter 6.

QNFAnswer Finally, QNFAnswer processes the analysis results to produce messages regarding the valid free and assign-null statements that may be inserted. Our prototype gives answers at the Jimple level, and in most cases it is rather easy to interpret the answers also at the Java source code level or at the Java bytecode level.

Optimizer The optimizer should insert free and assign-null statements to the application. This part is currently not implemented.

A.2 Free and Assign Null Queries

A.2.1 Specifying a Query

Queries are expressed in terms of the Jimple code generated from the class files of an application. In order to relate the Jimple code to the corresponding Java source code/Java bytecode, every Jimple

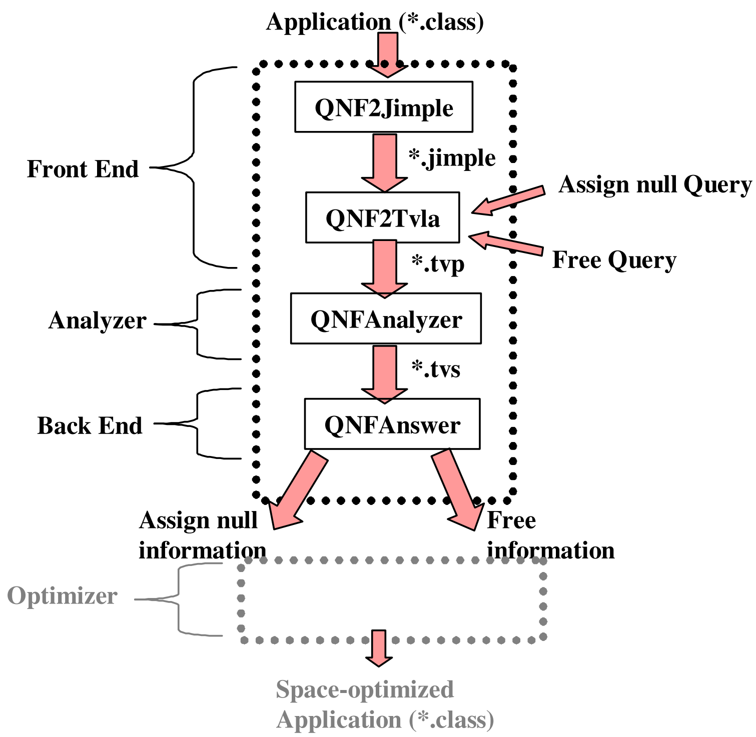


Figure A.2: QNF Architecture. The current version supplies a front-end, an analyzer and a back-end for reporting free and assign-null information. The optimizer is not implemented yet.

```
        y = x;
48
29
        goto label4;
49
30

     label3:
        $r1 = <java.lang.System: java.io.PrintStream out>;
52
31
        $i0 = y.<SLL: int val>;
56
31
        virtualinvoke $r1.<java.io.PrintStream: void println(int)>($i0);
59
31
        t = y.<SLL: SLL n>;
66
32
        y = t;
68
33

     label4:
        if y != null goto label3;
70
30
```

Figure A.3: A piece of Jimple code for traversing a list. Every Jimple statement is succeeded by two numbers, which represent a bytecode offset and a line number in the original code. These numbers appear in the .jimple file, and used to specify free and assign-null queries.

statement is succeeded by two numbers. The first number relates the Jimple statement and a bytecode offset in the corresponding Java bytecode. The second number relates the Jimple statement and the corresponding Java source code line number. Fig. A.3 shows an example of Jimple code. The numbers 66 and 32 after the Jimple statement `t = y.<SLL: SLL n>` associate this statement with bytecode offset 66 and line number 32 in the original Java bytecode/Java source code.

A query specifies places in the Jimple code candidates for the insertion of an assign-null or a free statement. The syntax of a query is as follows:

$$\begin{aligned}\text{Query} = \text{class}|\text{class: method}|\text{class: method[lineNUM]}|\\ \text{class: method[lineNUM1][bcNUM2]}\end{aligned} \tag{A.3}$$

where *class* specifies a class name, *method* specifies a method signature.*lineNUM* and *bcNUM* specify a line number and bytecode offset in the original code and the bytecode. As shown in (A.4), a query may specify: (i) the Jimple statements of a class, (ii) the Jimple statements of a method, (iii) the Jimple statements associated with line number NUM in the original Java source code, or (iv) the Jimple statements associated with line number NUM1 and bytecode offset NUM2. For example, Query (A.1) considers Jimple statements associated with line number 32 in `main` method of `Loop` class. From Fig. A.3 we see that `t = y.<SLL: SLL n>` is the only such Jimple statement.

A query is interpreted either as a *free query* or as an *assign-null* query. The mapping between a Jimple statement st specified by a query and the free or assign-null statement candidate for insertion after st is defined as follows:

- for a free query, a variable or a field *used* at *st* is considered. For example, Query (A.1) considers `free y` and `free y.n`[2] for insertion after the Jimple statement `t = y.<SLL: SLL n>`, since both `y` and `y.n` are used at that statement.

- for an assign-null query, a field *used* at *st* is considered. For example, Query (A.1) considers `y.n = null` for insertion after the Jimple statement `t = y.n`, since `y.n` is used at that statement. We do not consider assign-null to local variables in our prototype, since there are known simpler algorithms for doing that, and applying our framework for this case is like killing a fly with a cannon. . .

We note that issuing a query to consider the insertion of a free or an assign-null statement after a statement st for a variable or field not used in st requires modification of the original Java code. For example, suppose we would like to consider a `free x` after the statement `t = y.n` at line 32 in the `Loop` example. Since `x` is not used at that statement, inserting a new statement `SLL dummy = x` after line 32 and then issuing a free query regarding this new statement will do the job.

[2]for our free and assign-null statements we use Java syntax instead of Jimple syntax

Finally, we note that in principle, the user may specify a query regarding finding all valid free or assign-null statements in the program. However, there is a direct tradeoff between the number of code places candidates for free or an assign-null to the cost the analyzer. The current implementation is more suitable for issuing queries regarding few lines in a program.

A.2.2 Query Answers

The answer to a query shows the valid free or assign-null statements that can be inserted in the code places specified by that query. For example, Answer (A.2) shows that it is valid to insert a `free y` statement immediately after the Jimple statement associated with Java line 32 and bytecode offset 66, i.e., immediately after the statement `t = y.<SLL: SLL n>`.

A.3 Using the Prototype

Using the prototype consists of going through the phases described below.

Compilation In order to allow a more natural interface it is recommended that .class files are generated with all debugging info (e.g., specifying $-g$ flag in the `javac` compiler during the compilation of the .java files). Otherwise, the Jimple code will not show original variable names.

QNF2Jimple The user queries are specified at the Jimple level, thus QNF2Jimple needs to be applied to the application to produce .jimple files out of the .class files. The command line options for QNF2Jimple are as follows:

[-startMethod startClass: startMethod][soot-options] mainClass

where *startMethod* allows the specification of the starting point of the program. For example, in an applet it allows the specification of the `run` method of the main class. If the *startMethod* option is not used, the directory is set to `mainClass.main`. *soot-options* are described in `http://www.sable.mcgill.ca/soot/tutorial`. Finally, `mainClass` specifies the main class of the application. If *startMethod* option is not used, the starting point of the program is set to `mainClass.main`.

The Jimple files are produced in the directory `startClass.startMethod`. For example, applying `QNF2Jimple Loop` produces the Jimple Files `Loop.jimple, SLL.jimple` in `Loop.main` directory.

QNF2Tvla The queries along with the application are given as input to QNF2Tvla, which produces .tvp files (i.e., a TVLA program) that serve as an input to our free and assign-null analyzer. In the current implementation QNF2Tvla repeats the process of converting .class files to .jimple files, and then the .jimple files are converted to .tvp files. The command line options for QNF2Tvla are as follows:

[-qn|-qf|-qnf q_1;q_2;...;q_k|@qfile] [-keepDeadRefs][-scopedVarsField] [QNF2Jimple-options]

where *qn* interprets a query as an assign-null query, *qf* interprets a query as a free query, and *qnf* interprets a query as a free and as an assign-null query. Queries may be specified in the command line separated by a semi-colon, or by giving a file that contains queries, each in a new line. For example, applying `QNF2Jimple -qf @Loop.qf Loop` specifies for the `Loop` example the query file Loop.qf, which contains Query (A.1).

The input to TVLA is a set of .tvp files generated from the .jimple files. By default, before the translation process begin, the Jimple code is instrumented with statements assigning null to dead local variables. These null assignments help our analysis drop unnecessary information (i.e., ignoring the value of dead variables) at the cost of analyzing the null assignment statements. The user may disable this step of null assignment instrumentation using *keepDeadRefs* option. Finally, *scopedVarsField* option may be used to affect the abstraction. This option specifies longer names for variables and fields, by prefixing variables with their containing method, and by prefixing fields with their type name. This option allows disambiguating fields of the same name, but of different types, and variables of the same name but of different declaring methods.

QNFAnalyzer The command line options for QNFAnalyzer are as follows:

QFREE|QNULL SLL|DLL|ALLOC-SITE [tvla-options]

This command applies our assign-null/free TVLA-based analyzer. QFREE specifies the free analysis, and QNULL specifies the assign-null analysis. *SLL|DLL|ALLOC_SITE* determines the type of analysis being applied. Each type of analysis trades off precision and cost. In particular, *SLL* precisely handles a singly linked list, *DLL* precisely handles a doubly-linked list, and *ALLOC_SITE* abstracts objects based on their allocation sites (combined with the values of reference variables). The cost of *ALLOC_SITE* is cheaper than *SLL, DLL* analyses; however, it is less precise for recursive data structures. Finally, *tvla-options* are described in `http://www.math.tau.ac.il/~rumster/TVLA`.

The QNFAnalyzer command is applied in the `startClass.startMethod` directory. For example, in `Loop.main` directory we apply `QNFAnalyzer QFREE SLL` to execute a free analysis for the `Loop` example using *SLL* analysis kind.

QNFAnswer QNFAnswer processes the analysis results to produce messages regarding the valid free and assign-null statements that may be inserted. QNFAnswer must be applied with the same options used for the analyzer. For example, in `Loop.main` directory we apply `QNFAnswer QFREE SLL` to give as an answer the free statement shown in (A.2).

A.4 Known Limitations

- static initializers are ignored
- exceptions are ignored
- multithreading is not supported