View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Washington University St. Louis: Open Scholarship

Washington University in St. Louis

Washington University Open Scholarship

All Computer Science and Engineering Research

Computer Science and Engineering

Report Number: WUCSE-2005-24

2005-04-26

Static Determination of Allocation Rates to Support Real-Time Garbage Collection

Tobias Mann

While it is generally accepted that garbage-collected languages offer advantages over languages in which objects must be explicitly deallocated, real-time developers are leery of the adverse effects a garbage collector might have on real-time performance. Semiautomatic approaches based on regions have been proposed, but incorrect usage could cause unbounded storage leaks or program failure. Moreover, correct usage cannot be guaranteed at compile-time. Recently, real-time garbage collectors have been developed that provide a guaranteed fraction of the CPU to the application, and the correct operation of those collectors has been proven, subject only to the specification of certain statistics related... **Read complete abstract on page 2.**

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

 Part of the Computer Engineering Commons, and the Computer Sciences Commons

Recommended Citation

Mann, Tobias, "Static Determination of Allocation Rates to Support Real-Time Garbage Collection" Report Number: WUCSE-2005-24 (2005). *All Computer Science and Engineering Research.*
https://openscholarship.wustl.edu/cse_research/942

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

This technical report is available at Washington University Open Scholarship: https://openscholarship.wustl.edu/cse_research/942

Static Determination of Allocation Rates to Support Real-Time Garbage Collection

Tobias Mann

Complete Abstract:

While it is generally accepted that garbage-collected languages offer advantages over languages in which objects must be explicitly deallocated, real-time developers are leery of the adverse effects a garbage collector might have on real-time performance. Semiautomatic approaches based on regions have been proposed, but incorrect usage could cause unbounded storage leaks or program failure. Moreover, correct usage cannot be guaranteed at compile-time. Recently, real-time garbage collectors have been developed that provide a guaranteed fraction of the CPU to the application, and the correct operation of those collectors has been proven, subject only to the specification of certain statistics related to the type and rate of objects allocated by the application. However, unless those statistics are provided or estimated appropriately, the collector may fail to collect dead storage at a rate sufficient to pace the application's need. Overspecification of those statistics is safe, but leaves the application with less than its possible share of the CPU, which may prevent the application from meeting its deadlines. In this thesis, we present a dynamic and static analysis of one such statistic, namely the real-time application's memory allocation rate. The dynamic analysis highlights the variability of a program's allocation rate. It also serves to quantify the conservatism of the statically computed upper bound. The static analysis is based on a data flow framework that requires interprocedural evaluation. We present the framework and results from analyzing some Java benchmarks from the jvm98 suite. Our work is a necessary step toward making real-time garbage collectors attractive to the hard-real-time community. By guaranteeing a bound on statistics provided to a real-time collector, we can guarantee the operation of the collector for a given application.

Department of Computer Science & Engineering

Washington University in St. Louis
School of Engineering & Applied Science

2005-24

Static Determination of Allocation Rates to Support Real-Time Garbage Collection, Master's Thesis, May 2005

Authors: Mann, Tobias

April 26, 2005

Abstract: While it is generally accepted that garbage-collected languages offer advantages over languages in which objects must be
explicitly deallocated, real-time developers are leery of the adverse effects a garbage collector might have on real-time performance.

Semiautomatic approaches based on regions have been proposed, but incorrect usage could cause unbounded storage leaks or program failure. Moreover, correct usage cannot be guaranteed at compile-time.

Recently, real-time garbage collectors have been developed that provide a guaranteed fraction of the CPU to the application, and the correct operation of those collectors has been proven, subject only to the specification of certain statistics related to the type and rate of objects allocated by the application. However, unless those statistics are provided or estimated appropriately, the collector may fail to collect dead storage at a rate sufficient to pace the application's need. Overspecification of those statistics is safe, but leaves the application with less than its possible share of the CPU, which may prevent the application from meeting its deadlines.

In this thesis, we present a dynamic and static analysis of one such statistic, namely the real-time application's

WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

STATIC DETERMINATION OF ALLOCATION RATES TO SUPPORT REAL-TIME GARBAGE COLLECTION

by

Tobias Mann B.Sc

Prepared under the direction of Dr. Ron K. Cytron

A thesis presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of

Master of Science

May, 2005

Saint Louis, Missouri

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ABSTRACT

STATIC DETERMINATION OF ALLOCATION RATES TO SUPPORT REAL-TIME GARBAGE COLLECTION

by Tobias Mann

ADVISOR: Dr. Ron K. Cytron

May, 2005
Saint Louis, Missouri

While it is generally accepted that garbage-collected languages offer advantages over languages in which objects must be explicitly deallocated, real-time developers are leery of the adverse effects a garbage collector might have on real-time performance.

Semiautomatic approaches based on regions have been proposed, but incorrect usage could cause unbounded storage leaks or program failure. Moreover, correct usage cannot be guaranteed at compile-time.

Recently, real-time garbage collectors have been developed that provide a guaranteed fraction of the CPU to the application, and the correct operation of those collectors has been proven, subject only to the specification of certain statistics related to the type and rate of objects allocated by the application. However, unless those statistics are provided or estimated appropriately, the collector may fail to collect

dead storage at a rate sufficient to pace the application's need. Overspecification of those statistics is safe, but leaves the application with less than its possible share of the CPU, which may prevent the application from meeting its deadlines.

In this thesis, we present a dynamic and static analysis of one such statistic, namely the real-time application's memory *allocation rate*. The dynamic analysis highlights the variability of a program's allocation rate. It also serves to quantify the conservatism of the statically computed upper bound. The static analysis is based on a data flow framework that requires interprocedural evaluation. We present the framework and results from analyzing some Java benchmarks from the jvm98 suite.

Our work is a necessary step toward making real-time garbage collectors attractive to the hard-real-time community. By guaranteeing a bound on statistics provided to a real-time collector, we can guarantee the operation of the collector for a given application.

For Heather, my life companion and friend

Contents

List of Figures

Acknowledgments

The road that lead me to Washington University and the DOC group has been anything but straight, and I anticipate that my future academic career will continue this pattern. This thesis marks the culmination of a Computer Science journey that started at Pacific Lutheran University, when my friend Brendan Phillips convinced me to take intro to programming with him. I am thankful for his tenacity and friendship. I am also thankful for my other roommates at PLU, Ludvig, David, and Aaron who continuously prevented me from taking school too seriously.

A very special thanks to my advisor Dr. Ron K. Cytron without whom, this work would not have been possible. His humor, generosity and patience have guided my work in the lab, and will continue to be a guidance for me, as I embark on a career in medicine.

Thanks to the students in the DOC group for all the good times, both inside and outside of the lab. I would especially like to acknowledge the contributions to this work by Morgan Deters and Rob LeGrand. I also thank to DARPA and AFRL for supporting this project under contracts F33615–00–C–1697 and PC Z40779 respectively.

These two years at Washington University have been memorable for many reasons, not the least of which has been the time spent with friends outside the walls of Brian 503. I would like to say thanks to the friends I have made here in St. Louis for all the good times. A special thanks goes out to Richard for all the entertaining moments and great discussions.

I would like to thank my family in Sweden, especially my parents and brother for their love and support all throughout my youth and early adulthood. I would like to also thank my mother-in-law for all her help on the way.

Most of all, I would like to thank my wife Heather. Without your love, companionship, support, and willingness to proof read I would be lost.

Tobias Mann

Washington University in Saint Louis
May 2005

Chapter 1

Introduction

There is considerable interest in Java as a software development vehicle for real-time applications. There are several reasons for this, some of which are listed by the National Institute of Standards and Technology (NIST) [6].

- Java's high level of abstraction allows for increased programmer productivity.
- Java is relatively easier to master than C++.
- Java is relatively secure, keeping software components protected from one another.
- Java supports dynamic loading of new classes.
- Java is highly dynamic, supporting object and thread creation at runtime.
- Java is designed to support component integration and reuse.
- The Java programming language and Java platforms support application portability.
- The Java technologies support distributed applications
- Java provides well-defined execution semantics.

Standards, such as the Real-Time Specification for Java^{TM} (RTSJ) [4], have emerged that offer facilities for the specification, scheduling, and management of real-time structures, such as periodic threads, asynchronous events, and high resolution timers. There is general agreement that the efficient and predictable execution of

such structures is necessary for the acceptance of the RTSJ or any other Java implementation that claims real-time performance.

However, when it comes to storage management, there is not (yet) universal agreement as to *how* to make object allocation and (in particular) deallocation and garbage collection reasonably predictable. While the safety and software engineering aspects of garbage collected languages have earned these languages an increased popularity in the general purpose computing industry, the real-time community is hesitant to embrace garbage collection due to the adverse effects a Garbage Collector (GC) might have on real-time performance. Real-time systems are dependent on predictable program execution to guarantee that all temporal constraints are met. This is not easily merged with traditional garbage collection techniques. Any work performed by a GC takes away processor time from the real-time program, referred to as the mutator in the literature. Thus, to guarantee that deadlines can be met, the execution of the GC must be predictable. NIST states this more formally in their specification for real-time Java [6]:

> Any garbage collector that is provided shall have a bounded preemption latency. The preemption latency is the time required to preempt garbage collection activities when a higher priority thread becomes ready to run.

Essentially, a GC suitable for real-time must be able to collect sufficient storage so that the mutator does not run out, and must do so without denying the mutator reasonable use of the CPU(s). In addition, the system must be able to schedule the GC just as it would any other activity. Therefore, the overhead imposed by the GC must be quantifiable and its execution must be predictable.

Recently, real-time garbage collectors have been developed that provide a guaranteed fraction of the CPU to the mutator. The correct operation of those collectors has been proven, subject only to the specification of certain statistics related to the type and rate of objects allocated by the mutator. However, unless those statistics are provided or estimated appropriately, the collector may fail to collect dead storage at a rate sufficient to pace the mutator's need [2]. Overspecification of those statistics is safe but leaves the mutator with less than its possible share of the CPU, which may prevent the mutator from meeting its deadlines.

In this thesis, we present a dynamic and static analysis of the memory allocation rate of mutator programs. As we will show, in Section 2.3.2, this is the most

influential of the aforementioned statistical properties and it is also the one most difficult for the developer to estimate. The dynamic analysis highlights the variability of the rate, with which programs tend to allocate memory. It also allows us to quantify the degree of conservatism in our static analysis. The static determination of a program's maximum allocation rate is crucial to the correct operation of a real-time collector. Our static analysis method (a dataflow framework) makes the assumption that it has access to the whole program. While it is true that Java dynamically loads classes, real-time collectors need a whole program conservative estimate of the mutator's allocation rate. Short of a guess, the whole program must be available to either a human or to our analysis to make the estimate possible. We further assume that only classes that are known *a priori* can be instantiated using reflection.

Once properly bounded, a program's allocation rate determines the necessary fraction of CPU time that must be devoted to garbage collection. This automation will enable the GC to truly abstract memory management from the programmer and improve the accuracy of the information. The first area of concern is the maximum allocation rates of the entire mutator program. However, this technique will also enable determination of an allocation rate's upper bound at different sections of the program. The idea is that GC scheduling can be improved by increasing the amount of information available to the scheduler. There may be times during the execution of the mutator where it is more or less appropriate for the collector to execute.

This thesis is organized as follows: Chapter 2 will provide the reader with necessary background information about our problem domain, and the techniques that we use. This is followed by a detailed coverage of our dynamic analysis, in Chapter 3. Next, in Chapter 4, we present our static framework for binding the maximum allocation rate. Chapter 5 analyzes the results of the previous two chapters. We deal with multithreaded mutator programs in Chapter 6. Finally, Chapter 7 offers some concluding remarks.

Chapter 2

Background

This chapter covers general concepts that are of importance for the understanding of the remainder of this thesis. Issues, such as what it means for a system to be real-time and what challenges real-time constrains poses for the memory management of these applications, will be discussed. We will also cover general garbage collection techniques and collection techniques specific for real-time collectors. In addition, we will give a short introduction to static analysis using a dataflow framework.

2.1 Real-time Constraints

A real-time application is one where, in addition to semantic correctness, there is a notion of temporal correctness. A real-time system will attempt to schedule all real-time threads in a manner that will maximize some metric of how well the temporal constraints of the application are met. A feasibility analysis determines if a given schedule has an acceptable value for the metric used. If the feasibility analysis fails, then either some code must be rewritten or the temporal constraints must be relaxed. Our work is mostly concerned with systems, which the literature refers to as *hard-real-time* systems. The metric used for these systems is the number of missed deadlines, and the only acceptable value is 0 [4].

The addition of these temporal constraints to the semantic correctness of a program implies that any memory management scheme used in a real-time system must have a predictable execution and an upper bound on the preemption latency for any real-time thread.

2.2 Real-time Memory Management

Any approach to real-time memory management must be able to ensure that the mutator program does not run out of memory and has sufficient usage of the CPU. This type of memory management can be divided into three categories, as shown in the sections below.

2.2.1 Manual Approach Using Explicit Allocation and Deallocation of Memory

In non-garbage-collected languages, the programmer must insert explicit statements to deallocate storage. The programmer must understand ownership and lifetime issues of dynamically allocated objects. Such information is often difficult to obtain, especially where a program makes extensive use of externally authored material, such as middleware and libraries. Moreover, insertion of explicit `delete` instructions can make code difficult to reuse.

2.2.2 Semiautomatic Approach Using Scopes

Specialized storage-allocation structures can be introduced to obviate the need for traditional garbage collection. For example, the Real-Time Specification for Java^{TM} (RTSJ) introduces memory *scopes* where objects can be allocated. Hard-real-time threads are allowed to access only these objects allocated in scopes.[1] The rules for scope creation are established so that a reference count on the entire scope suffices to determine liveness of all objects in the scope. The reference count is affected by threads entering and exiting the scope.

While the task of deallocation becomes simple and predictably bounded, the burden of correct usage of scopes falls on the programmer, with the following disadvantages:

- The application is constrained as to how objects in scopes can reference each other, as depicted in Figure 2.1.
- Scopes are a specialized form of *regions* [22], and programs can leak an unbounded amount of dead storage in a region. For example, consider a doubly-linked list in an RTSJ scope. Because they reference each other, all container

[1] Access is also permitted to *immortal memory*, from which objects are never collected.

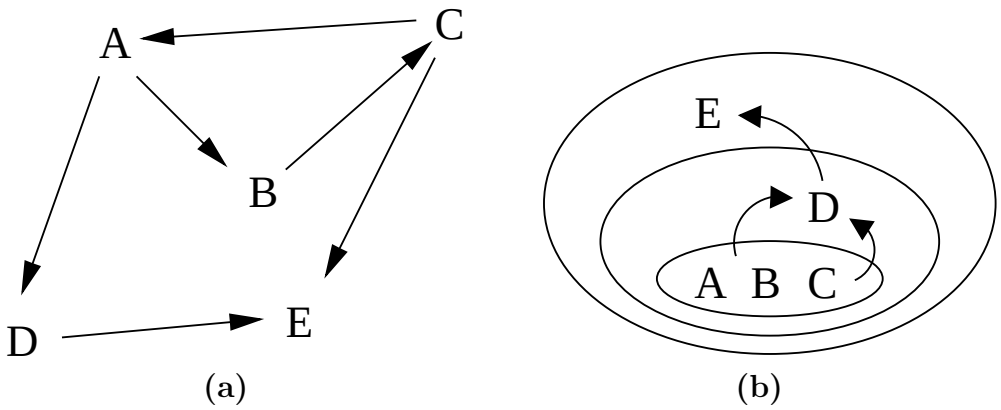

Figure 2.1: Use of scoped storage. **(a)** shows the references between objects at one point in time; **(b)** shows the resulting scope assignment. If E tried to reference D, the program would fail given this scope assignment.

cells must be allocated in the same scope. Thus, repeated deletion and insertion will leak uncollectible objects in the scope while not increasing the live-storage requirement of the program.

Traditional garbage collection can also be avoided by using techniques such as reference counting [24] and contaminated garbage collection [5], but those collectors are inexact and thus suffer from the same leakage problems as scopes.

2.2.3 Fully Automated Approach

A real-time GC, such as Metronome [2] or Perc [16], is assigned the responsibility of detecting and collecting dead storage. The mutator need not change, but its *behavior* strongly influences how the collector must operate so as to guarantee sufficient availability of storage.

Because of the burden placed on a programmer when faced with specialized storage-allocation structures, real-time garbage collection is the method of preference. The RTSJ, with its scoped memories was arguably formulated in a context that doubted the veracity of a real-time collector. More recently, research has proven [2, 1] that collectors such as Metronome operate correctly *if* the mutator's behavior is properly described. The description of the mutator's behavior is discussed in more detail in Section 2.3.2

2.3 Real-time Garbage Collection

This section will present some general ideas, concepts and techniques associated with modern garbage collection implementations. For a more in-depth coverage of this, see Paul Wilson's work [24].

2.3.1 General Ideas and Concepts

There are two general techniques used by any garbage collector to distinguish live memory from garbage, *reference counting* and *tracing*. In a reference counted system, each object keeps a count of the number of references that point to it. When this count transitions from 1 to 0 the object may be collected. One advantage of reference counting is that it is incremental by design: the work of the collector (the updating of the reference counts) is interleaved with the program's execution. This incremental property of a reference counted collector is attractive for real-time systems. However, as we mentioned in Section 2.2.2 this technique is inexact, and thus may suffer from memory leakage problems.

These problems makes reference counted systems unacceptable for deployment in real-time environments. Instead, we turn to collectors that rely on tracing to differentiate live memory from garbage. A tracing collector builds and traverses a graph, called the *reachability graph*, of the objects that are reachable by the mutator. In doing so, the collector identifies the objects that are live. To build this graph, the collector starts with the *root set*, also know as the *live roots*. The root set typically contains the pointers that reside on the stack and static pointers. It follows the pointers in the root set to look for pointers to other objects. This way the collector will eventually traverse over all objects reachable to the mutator program. Tracing collectors come in many varieties. We will look at some of these in the next couple of sections.

Mark-Sweep

As its name suggests, Mark-Sweep collection has two major phases, Mark and Sweep. During the *mark* phase the mutator's runtime heap is traced as aforementioned, using either a breadth-first or a depth-first technique. All objects that the collector touches are marked as live. When the marking phase completes the *sweep* phase takes over.

In this phase memory is examined to find all unmarked objects and reclaim the space they occupy.

There are three major problems usually identified with Mark-Sweep collection [24].

Fragmentation: Objects with varying sizes will often cause fragmentation of the heap, with the adverse effect that allocating large objects may be difficult. This problem can be made less severe by using free lists of varying sizes and allocating objects from these lists using a best fit approach.

Computational Complexity: The major cost of this technique is the mark phase. The mark phase cost is proportional to the amount of live memory that must be traversed. All live objects must be marked imposing an inherent limit on efficiency.

Locality of Reference: Because live objects retain their place in memory, when a collection cycle finishes live objects will be interspersed with the free space generated by the collected objects. New objects are then allocated in these areas. The end result is that objects of different ages will be scattered all over the heap, which in turn may adversely affect locality of reference.

Copying

A copying collector gets its name from the fact that it does not actually collect garbage. Instead, it moves all objects known to be live into a special area of memory. The remainder of the heap is then known to be garbage. The most common copy collector is the *semispace* collector [10] using the Cheney copying traversal algorithm [7]. In this scheme, the heap is partitioned into two equasized parts, called semispaces. At runtime, the executing program only has access to one semispace, called *fromspace*. At collection, fromspace is traced and the live objects are copied over into the unused semispace, called *tospace*. When collection completes the roles of tospace and fromspace are reversed. The advantage of this approach is that it avoids fragmentation of the heap. The main disadvantages associated with copying collectors is that only half of the heap memory is ever available to the application. Hence the memory footprint of the system doubles.

Incremental

The temporal constraints of real-time applications places special requirements on real-time garbage collectors. Traditional techniques, such as *stop-the-world* collectors are not suitable in these environments. A fine-grained incremental GC, which will interleave small units of collector work with small units of mutator execution, is needed. Hence, whatever garbage collection technique is used, it needs to be able to allow the mutator to access, and perhaps alter, the heap during a collection cycle.

If the mutator is allowed to alter (mutate) the heap while the GC is building the reachability graph, then the collector needs some mechanism for keeping track of those changes. To facilitate our discussion of incremental garbage collectors we will classify memory objects in accordance with the *tricolor-marking* scheme [19]. In this scheme, memory objects are classified using the three colors white, grey, and black.

White: White means that the memory object has not been visited by the GC. All objects that are white by the end of the collection cycle are reclaimed.

Grey: Grey means that the memory object has been visited but not scanned. Scanning refers to the collector examining the object for references to other memory objects. In terms of Breadth-, or Depth-First Search, grey objects are the objects in the fringe of the search tree.

Black: Black means that the memory object has been visited and scanned. All objects that are black by the end of the collection cycle are retained.

When garbage collection begins, all objects are white and when it ends all objects are either white or black. However, in an incremental collector the intermediate states are very important because of the ongoing mutator activity. For example, the mutator may change the reachability graph so that an object already marked black (live) becomes unreachable, and thus should have been marked white (dead). These objects that "die" during a collection cycle, but go uncollected, are called *floating garbage* in the literature. As we will discuss in Section 2.3.2, the floating garbage cannot be collected until, at the earliest, the subsequent collection cycle. Therefore, it may increase the applications memory footprint beyond maxlive.

Another, more serious, problem is that the mutator may either create a new object, or move references around so that a black object now has a reference to a white object. This white object is live; however, if the only references to it are from

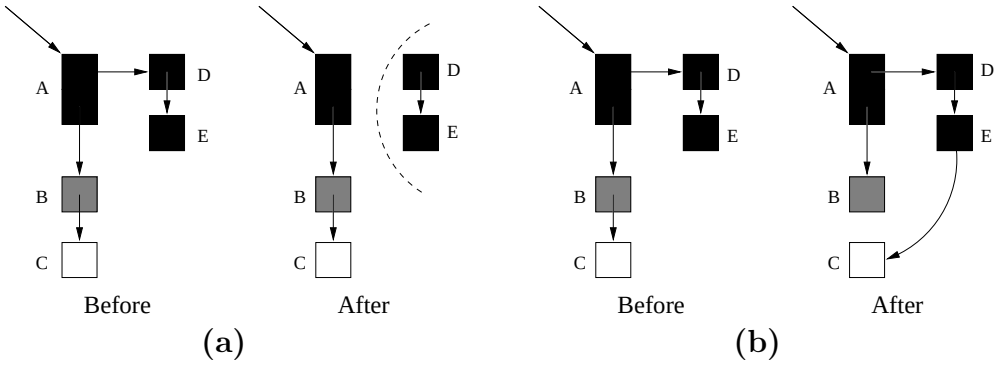

Figure 2.2: Mutating the Heap. **(a)** shows how the mutator may alter the heap to create floating garbage, objects D and E; **(b)** shows how the mutator may alter the heap so that the collector will reclaim an object that is still live, object C.

black objects then it will never be visited by the collector, and thus the collector will erroneously reclaim it at the end of the collection cycle. Figure 2.2 illustrates the problems here discussed.

Stated more formally, any correct incremental GC must maintain the invariant that no black object has a direct reference to a white object. To ensure that newly allocated objects preserve the invariant new objects are often allocated black. However, how can a collector be sure that the mutator does not move references around in a manner that violates the invariant? The obvious solution would be to recompute the reachability graph whenever the mutator changes it. However, this is an unacceptable solution since there can be no guarantee that such a GC ever completes its collection cycle. Next, we will discuss two techniques that address this problem.

Read and Write Barriers

To ensure that the tricolor invariant is maintained, either the mutator must be prevented from reading white objects or it must be prevented from writing a reference to a white object into a object colored black. The approach that prevents the mutator from reading white objects is called a *read barrier*. The read barrier examines all attempts of the mutator to access the heap. If it detects an access to a white object, it will immediately color that object grey by placing it in the fringe of the reachability graph. Consequently, during a collection cycle any references held by the mutator will be either grey or black. Thus, the mutator cannot write a reference to a white object into a black object.

The other approach is to allow the mutator to read whatever references it wants, but trap all attempts of the mutator to write a reference into an object. This technique is conveniently called a *write barrier*. Write barriers come in two flavors that differ on which aspect of the problem they address. The mutator can cause problems for the collector by both writing a reference to a white object into a black object and destroying the original path to the white object. For example, the situation shown in Figure 2.2b would not present a problem if the reference from object B to object C had been left intact. One write barrier technique addresses the problem by ensuring that no path to a white object can be broken without providing the collector with another path to that object. The other technique records references written into black objects and either color the referenced objects grey, or reverts the black object to grey.

Out of the two approaches to maintaining the tricolor invariant, the read barrier is generally considered more expensive because heap reads are much more frequent than heap writes. However, certain collectors, such as copying collectors, need to use a read barrier to ensure that the mutator only sees references to valid copies of objects.

2.3.2 Statistical Properties Needed by any Real-time GC

As we mentioned in Section 2.2.3 recent real-time collectors have proven that they operate correctly provided that the user correctly characterizes the mutator's behavior. Fortunately, a mutator's relevant behavior can be distilled into a few statistics.

Maximum live storage: We denote as *maxlive* the maximum storage live at any point during the application's execution. In other words, the program cannot run in fewer than maxlive bytes, given a perfect, continuously-operating garbage collector. Determining maxlive statically is undecidable. Even a dynamic approach to determining maxlive [20, 5] is computationally intensive, as the garbage collector must be run when any stack or heap cell is modified.

In spite of the above considerations, it is generally assumed that developers and those who execute Java applications know maxlive for a given application. This follows from the fact that all programs (including those written in languages with explicit deallocation) execute with a specified or nominal heap size.

Pointer density: The *mark* phase of a precise garbage-collection algorithm involves touching all live objects. Liveness is determined by tracing references from a program's live roots (see Section 2.3.1), such as its stack and static variables. Each object visited by the mark phase offers pointers that, if not null, point to objects now assumed to be live. The cost of the marking phase is thus dependent on the number of non-null references that can be discovered while marking live objects.

Fortunately, in languages like Java, reference fields are explicitly declared. The pointer density of each object type can thus be determined, if all object types are known *a priori*. Dynamic, worst-case pointer density can thus be bounded by assuming the object with worst-case pointer density dominates. While a better bound on pointer density can limit the work of a tracing collector, no real harm comes from overestimating this statistic, even to the point of assuming that every field of every object is a non-null reference.

Allocation rate: A real-time collector cannot be permitted to suspend a mutator indefinitely. Thus, the work of the GC must be interleaved with the mutator's execution. In rate-based collectors, such as Metronome, a predetermined fraction of the CPU is devoted to collection, so that context may switch between the mutator and the collector many times before a collection cycle is truly complete. In the span of a collection cycle, the mutator runs periodically and can continue to allocate memory. As mentioned in Section 2.3.1, some of those objects may become dead during the cycle. This floating garbage does not count toward maxlive, but the collector cannot collect it in the current cycle.

The extent of floating garbage must be known, so that a real-time collector can specify sufficient storage beyond maxlive so as not to run out of storage during a collection cycle. Ideally, a scheduler would know *a priori* the amount of floating garbage that will be created during a specific collection cycle. When this information is known, the scheduler can schedule the collector such that the impact on the mutator is minimal. Unfortunately, this would require knowing, for any given point in the program, the exact amount of memory that the mutator will allocate during the next collection cycle.

The computation of this is not feasible, however we can bind the amount of floating garbage as the product of the mutator's execution time during the entire collection cycle and the maximum rate at which the mutator can allocate

storage. That product is influenced by the mutator in terms of its allocation rate, but the fraction of time given to the mutator is the key parameter used by the collector to guarantee pacing with the mutator.

At issue is whether a programmer can reliably provide these statistics. Even if a programmer knows the application well, use of libraries or other code greatly complicates manual computation or estimation of the statistics. If the provided statistics do not bound the actual behavior of the mutator, then the collector may fail to collect dead storage at a rate sufficient to pace the application's need, with the implication that the application will exhaust its runtime heap. One could try to overspecify the statistics, but this is still an educated guess on the part of the developer. Also, while overestimating the rate is safe it will cause the program's required heap size to increase, which may be tolerable, but the fraction of time given to the mutator will decrease, which may make the real-time program unschedulable.[2]

An automated approach to estimate these statistical properties is needed. In Chapter 4 of this thesis, we present a dataflow framework which statically computes the allocation rate of a mutator program.

2.4 How Does a Real-time Garbage Collector Affect the Mutator

To quantify the effect that the real-time GC has on the mutator programs ability to meet its temporal requirements we borrow the following notation from Bacon et al. [2]:

$$\begin{aligned}
\gamma(\tau, \Delta\tau) &= \text{the allocation rate from time } \tau \text{ to time } \tau + \Delta\tau \\
T &= \text{runtime of the program} \\
\gamma(0, T) &= \text{average allocation rate for the program} \\
\gamma^*(\Delta\tau) &= \max_{\tau}(\gamma(\tau, \Delta\tau)), \text{maximum allocation rate during any } \Delta\tau
\end{aligned}$$

[2]In the sense that rate-monotonic analysis [13] cannot guarantee that all deadlines are met.

2.4.1 Minimum Mutator Utilization (MMU)

Traditionally, approaches to real-time garbage collection have quantified their impact on the execution of the mutator program by measuring the maximum *pause time* experienced by the mutator. However, as noted by Bacon et al. [2] , a mutator thread that experiences a period of low CPU utilization may fail to meet its temporal requirements even though all individual pause times are short.

Therefore, a more accurate measure of a real-time collector's effect on the mutator program is MMU. Cheng and Blelloch [8] defines MMU for a given time interval, $\Delta\tau$, as the smallest fraction of CPU utilization experienced by the mutator over all intervals of width $\Delta\tau$. Equation 2.1 [2] shows how MMU can be computed assuming a time based scheduling algorithm where Q_T and C_T are the mutator and GC time quanta respectively. A time quantum is the smallest amount of non-preemptive execution time that is guaranteed.

$$MMU(\Delta\tau) = \frac{Q_T * \lfloor \frac{\Delta\tau}{Q_T + C_T} \rfloor + x}{\Delta\tau} \tag{2.1}$$

$\Delta\tau$ is the time window for which MMU is calculated and x is the remaining partial mutator quantum, define in Equation 2.2.

$$x = max(0, \Delta\tau - (Q_T + C_T) * \lfloor \frac{\Delta\tau}{Q_T + C_T} \rfloor - C_T) \tag{2.2}$$

2.4.2 Max Allocation Rate vs. Average Allocation Rate

To quantify how well the maximum allocation rate of a mutator program estimates the average allocation rate we define the following metric, $\frac{\gamma(0,T)}{\gamma^*(\Delta\tau)}$. When this value is close to 1, the maximum rate provides a good approximation of the average. As this value gets further from 1 the validity of that assumption diminishes.

2.5 Static Analysis Using a Dataflow Framework

A common technique for analyzing a static property of a program is to formulate the problem as a data flow framework [15]. To this end a control flow graph representing the program is constructed. Below we show an example C program, and in Figure 2.3 we show the control flow graphs for the two methods.

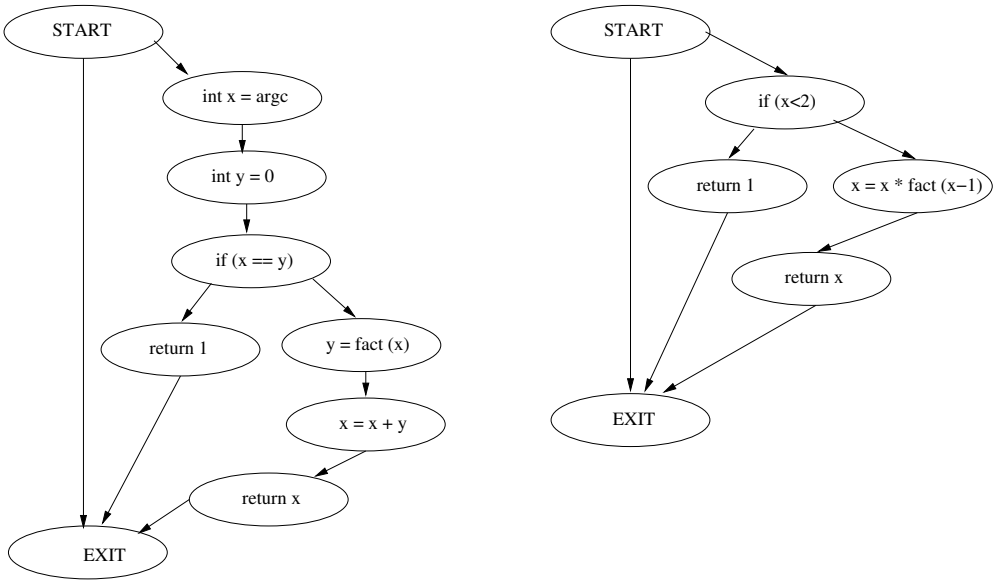

Figure 2.3: Intraprocedural dataflow framework generated from our example program

```
int fact(int x){
  if (x < 2)
    return 1;
  x = x * fact(x-1);
  return x;
}
int main(int argc, char** argv){
  int x = argc, y = 0;
  if (x == y)
    return 1
  y = fact(x);
  x = x+y;
  return x;
}
```

Formally a data flow framework is expressed as a triple $DF = (G_p, L, F)$ where G_p is the data flow graph for procedure (or method) p, L is the meet lattice, and F is the set of transfer functions.

- $G_p = (N_p, E_p, s_p, e_p)$

- $L = (A, \top, \bot, \preceq, \wedge)$
- $F \subseteq \{f : L \to L\}$

N_p is the set of nodes in the graph and E_p is the set of control-flow edges. For our purposes, each node $n \in N_p$ represents one instruction and each edge $(n_1, n_2) \in E_p$ represents a possible execution path of the procedure. In addition, G_p is augmented with start and exit nodes, s_p and e_p, and an edge (s_p, e_p).

The meet lattice, L, is a quintuple consisting of the following:

- A a set whose elements form the domain of the problem
- $\top$ (top) a special element in A representing the best possible solution to the dataflow problem
- $\bot$ (bottom) a special element in A representing the worst possible solution to the dataflow problem
- $\preceq$ is a reflexive partial order which is used to compare different solutions to each other.
- $\wedge$ the meet operator, which combines solutions from different paths. The meet operator must satisfy the following properties for any $a, b \in A$:
  1. $a \preceq b \Leftrightarrow a \wedge b = a$
  2. $a \wedge a = a$
  3. $a \wedge b \preceq a$
  4. $a \wedge b \preceq b$
  5. $a \wedge \top = a$
  6. $a \wedge \bot = \bot$

The last element of the DF triple is the set of transfer functions, F. A transfer function $f \in F$ maps the combined input to a node, $n.in$, to its output $n.out$. To be able to guarantee that a dataflow framework converges to a solution, we require that all $f \in F$ are *monotone*.

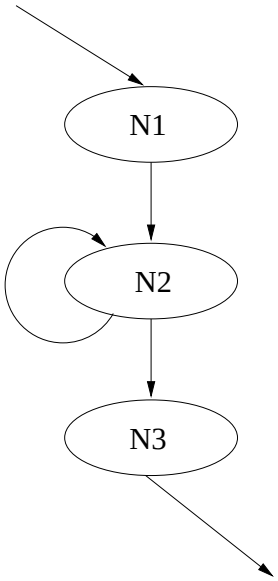

Figure 2.4: Given the transfer function $f_{N2}(IN) = \top$ if $IN = \bot, \bot$ if $IN = \top$ this dataflow graph would never converge to a solution because the output from $N2$ would oscillate between $\top$ and $\bot$

Formally monotonicity is defined as:

$$(\forall f \in F)(\forall x, y \in A) x \preceq y \rightarrow f(x) \preceq f(y) \tag{2.3}$$

Intuitively, this means that given a *worse* input a transfer function cannot produce a *better* output. Figure 2.4 shows why we must require that all $f \in F$ are monotone.

A dataflow framework that satisfies the properties discussed thus far will converge to a solution. However, we have provided no guarantees as to the quality of the solution. When a dataflow framework is *distributive* then the (intraprocedural) solution we compute is the meet-over-all-paths (MOP) solution, which is guaranteed to be the best possible static solution of the framework [15]. Distributivity is defined as:

$$(\forall f \in F)(\forall x, y \in A) \quad f(x \wedge y) = f(x) \wedge f(y) \tag{2.4}$$

The last criteria by which we will evaluate our framework is the speed of convergence. A dataflow framework is called *rapid* if the following property holds:

$$(\forall a \in A)(\forall f \in F) \quad a \wedge f(\top) \preceq f(a) \tag{2.5}$$

In a rapid dataflow framework, one iteration through the graph gathers all data necessary to reach convergence.

Chapter 3

Dynamic Measure of Mutator Memory Allocation Rate

In this chapter, we present a simple technique to measure the memory allocation rate of a mutator program at execution time. Although binding the mutator allocation rate dynamically is not directly useful to a real-time GC, the dynamic allocation rate is worthy of study for the following reasons:

1. The allocation characteristics of applications have not been widely studied. This dynamic measure will thus aid us in getting a deeper understanding of our problem domain and in getting a sense of the results that we may expect from a static prediction.

2. Static analysis is necessarily conservative, and we are interested in knowing actually observed upper bounds on a program's allocation rate.

We are interested in the allocation rate exhibited by a mutator program during a specific window of time. This time window is the total time that the mutator is permitted to execute during one garbage collection cycle. As mentioned in Section 2.3.2, the garbage collector will use the maximum allocation rate exhibited by the mutator during this time window to schedule its collection cycles in a manner that will ensure that the heap memory is not exhausted.

3.1 Hypothesis

Assumptions made in existing work speculate that the allocation rate of a mutator program during one garbage collection cycle is close to the mutator's average allocation rate taken over the whole program execution [2]. We hypothesize that this is commonly not the case. Instead, we believe that most programs will have certain regions during their execution where they exhibit a significantly elevated allocation rate. If our hypothesis is correct, this will have consequences both for the usefulness of the maximum allocation rate as a descriptor of a mutator program and for the assumption that the programmer, or user, can correctly estimate the maximum allocation rate.

3.2 Experimental Setup

We executed our experiments on a subset of the jvm98 SPEC benchmarks. All experiments were performed on a Solaris 7 machine with a Sparcv9-333 MHz processor with hardware support for floating point operations. The Java Virtual Machine (JVM) used was the jdk-1.1.8 source release. For the purpose of this research, the JVM was instrumented to record the size and time of each allocation. To ensure that the gathered data were free of noise due to other processes executing on the computer we execute the benchmarks in high priority, real-time executing mode.

The information gathered by the instrumented JVM was processed off-line to compute $\forall\tau\ \gamma(\tau, \Delta\tau)$, see Section 2.4, for a range of different values of $\Delta\tau$. The end result provided us with the maximum allocation rate, as well as the allocation rate for any given part of the executing program.

3.3 Implementation

Our allocation-rate-finding software uses a queue data structure, implemented as a linked list. One queue is created for each putative time window. The input into this program is the allocation trace generated by our instrumented JVM. As the software steps through the allocation trace, it encapsulates each allocation into an object. This allocation object is then processed by placing a reference to it in each of the queues as shown below:

```
addAllocation( A )
   Define
    |W| is the size of the window for this queue
    A is an allocation object, extracted from the JVM output
    A.time refers to the time this allocation occurred, as recorded by the JVM
    Q is a queue object
    Q.size() returns the total number of MB allocated by all allocations in Q
    Q.front returns a reference to the front element without dequeuing it

   Begin
1     Q.enqueue( A )
2     timeLimit ← A.time - |W|
3     A_old ← Q.front()

4     while A_old.time < timeLimit do
5       Q.dequeue()
6       A_old ← Q.front

8     if Q.maxSize() < Q.size() then
9       Q.setMaxSize( Q.size() )

10    printToFile( A.time )
11    printToFile( Q.size() / |W| )
```

The basic idea is that as the program processes its input, each queue will always contain a reference to all allocations that have occurred within the time window assigned to that queue. Each queue is updated both synchronously and asynchronously. The synchronized update occurs with a period specified by the user and the asynchronous occurs with each new allocation.

The synchronous update is handled by a method similar to the addAllocation method, only it accepts the current time as a parameter rather than an allocation object. This update is used to ensure that allocations that fall out of the window are removed from the queue even between allocations. In our experiments, we used a synchronous period of half the size of the smallest window considered.

When the program terminates we have one file for each queue that associates a time, from beginning of program execution, with the total number of bytes allocated by each allocation in the queue at that time. This data was then used to generate the plots shown in Section 3.4.

3.3.1 Implementation Analysis

The cost of each update to a queue is $O(k)$, where k is the number of dequeue operations. Each queue is updated $n+m$ times where n is the number of allocations of the program and m is the number of times that the queue is updated synchronously. To simplify the analysis we assume that each update has the same constant cost, meaning $\forall_{i,j}(k_i = k_j)$. This is a valid assumption because if we use a small enough synchronous period we can expect the variations of k to be small. Therefore, the cost of updating a queue can be treated as a constant. The complexity of maintaining one queue throughout the execution of the program is then the number of times that the queue is updated, $O(m + n)$. The program has x number of queues, specified by the user. Thus, the overall complexity of the program is $O(x(m + n))$.

3.4 Result

The primary results of these experiments are depicted as graphs of the mutator's allocation rate for a given time window as a function of time. What is remarkable about these results is the high variability of the allocation rate for each of the benchmarks tested. In particular, all of the benchmarks exhibited short periods of time were the allocation rate was unusually high. An example of this is shown in Figure 3.2.

This is significant because, as previously mentioned, existing work speculates that the allocation rate exhibited by the mutator during a collection cycle will be close to the mutator's average allocation rate taken over the whole program. As we have seen, the maximum allocation rate of a mutator, for the window size considered, is not representative of the allocation rate throughout the execution of the program. Furthermore, these *spikes* in allocation rate may make it increasingly difficult for a developer to correctly estimate the maximum allocation rate, especially if the spikes occur in library code not written by the developer in question.

The graph depicted in Figure 3.3 shows $\frac{\gamma(0,T)}{\gamma^*(\Delta\tau)}$ as a function of $\Delta\tau$. As expected, for all benchmarks tested this value is approaching 1 as $\Delta\tau$ increases. Figure 3.4 shows

Benchmark	Max Live	Coll. Rate	$\Delta\tau$ (ms)	MMU
`javac`	34	39.4	707	0.446
`jess`	21	53.2	325	0.441
`jack`	30	57.4	429	0.441
`mtrt`	28	45.1	509	0.446
`db`	30	36.7	670	0.441

Figure 3.1: Time Windows for the Metronome Collector $Q_T = 10$, $C_T = 12.2$

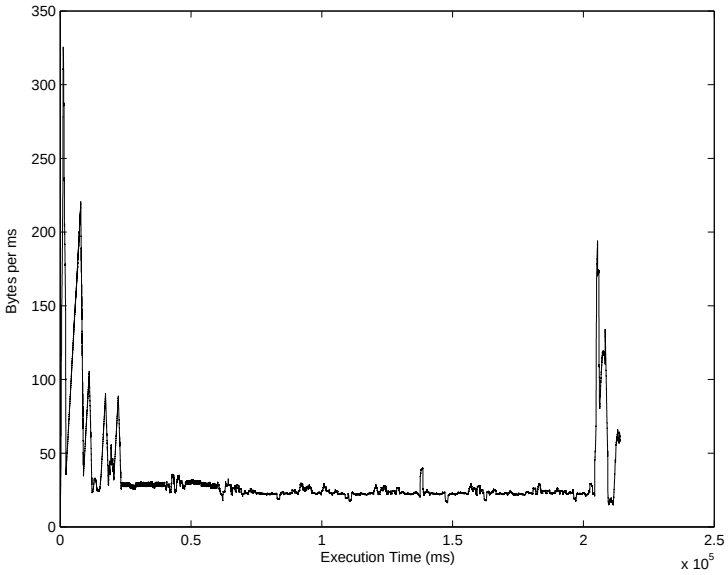

Figure 3.2: Allocation rate of jvm98 benchmark `jack`, as a function of execution time, $\Delta\tau = 1048ms$.

an indirect benefit of an increased window size. Here MMU is graphed as a function of $\Delta\tau$. MMU is formally defined in Equation 2.1. As a frame of reference the $\Delta\tau$ values used in the experiments presented by Bacon et. al [2] are shown in Figure 3.1.

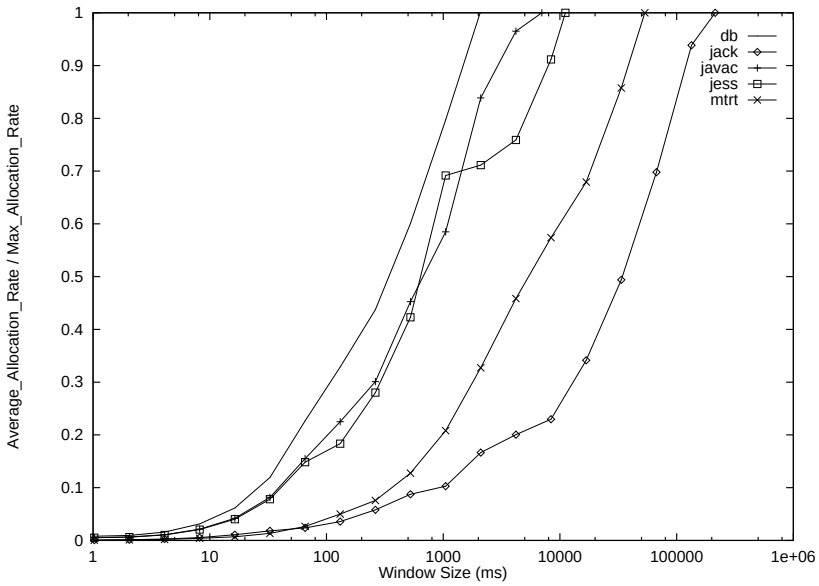

Figure 3.3: $\frac{\gamma(0,T)}{\gamma^*(\Delta\tau)}$ vs. $\Delta\tau$. This graph shows that as $\Delta\tau$ increases, assuming that the allocation rate is $\gamma^*(\Delta\tau)$ becomes less costly for any given point in the programs execution. The reason for this is that $\gamma^*(\Delta\tau)$ approaches $\gamma(0,T)$

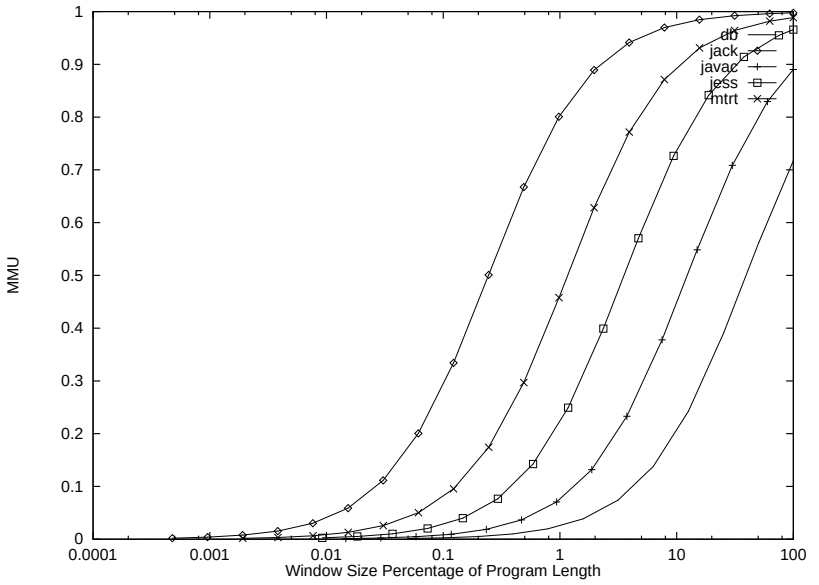

Figure 3.4: MMU vs. $\frac{\Delta\tau}{T} * 100\%$ Here we see experimental data depicting the relationship between MMU and $\Delta\tau$ described in Equation 2.1

Chapter 4

Static Measure of Mutator Memory Allocation Rate

This chapter presents the dataflow framework that we use to statically compute the maximum allocation rate of a mutator program. As in our dynamic technique, presented in Chapter 3, this framework uses a window that essentially slides over a program's instructions, and we compute the maximum allocation rate seen in that window. In Chapter 3, this window was expressed in units of time. However, time is not a convenient metric for static analysis, so instead our dataflow framework sizes the window with respect to a program's Java byte code instructions. While it is true that those instructions take varying time, conversion to time is still possible on average. For now, we will make the conservative estimate that each byte code instruction executes in one clock cycle. This assumption is safe because we are assuming that instructions execute faster then they actually do. Thus, relaxing this assumption can only increase the space between allocating instructions, which decreases computed allocation rate.

For the purposes of this framework, a program's instructions fall into two categories: those that allocate storage and those that do not. This binary categorization suggests an abstraction in which each instruction is represented by a bit: 1 for allocation and 0 for non-allocation. The relationship is slightly more complicated since we must account for the size of each allocation. However, at this point, we assume that all allocations are of unit size. We take into account actual object sizes in Section 4.3.2.

Based on the above assumptions, a window of instructions is represented by a bit-vector, where each bit represents one instruction; we adopt the convention that the most significant bit represents the most recent instruction.

4.1 Hypothesis

As mentioned in Chapter 1, the problem with the real-time collectors available today is that they do not fulfill the reason why they were introduced. Garbage collection is attractive to the real-time community because of the software engineering and security aspects it offers. In short, the intent of any GC is to make the job of the developer easier. We feel that current real-time collectors fall short of this goal because of the non-trivial statistical mutator properties they require for correct operation. We hypothesize that these properties can be computed statically, thus relieving the developer of this obligation. It is our belief that until automatic memory management can truly be abstracted from the user, the gains anticipated from a real-time GC will not be fulfilled. We hope that, if we can show that reasonable bounds on these statistics can be computed, garbage collection will become attractive to the real-time community.

4.2 Naïve Framework

We begin with a simple framework that explains our approach, but which provides unnecessarily conservative results on Java programs because of the `try...catch` idiom, as we explain below. In this naïve framework, the meet lattice L is defined as follows:

- $A = \{0, 1\}$
- $\top = \langle 0, 0, 0, \ldots, 0 \rangle$
- $\bot = \langle 1, 1, 1, \ldots, 1 \rangle$
- $\wedge$ is logical bitwise *or* of the input bit-vectors
- $a \preceq b$ holds if and only if $a \wedge b = a$

Thus, $\top$ is a window in which none of the instructions allocates memory; $\bot$ is a window in which all instructions allocate memory. The meet operator $\wedge$ summarizes the allocation windows of its inputs, and bitwise *or* is a valid meet operator for a monotone framework.

For example, the bit-vectors $\langle 0, 0, 1, 0 \rangle$ and $\langle 0, 1, 0, 0 \rangle$ inform us that on their respective paths through G_p an allocation has occurred three and two instructions

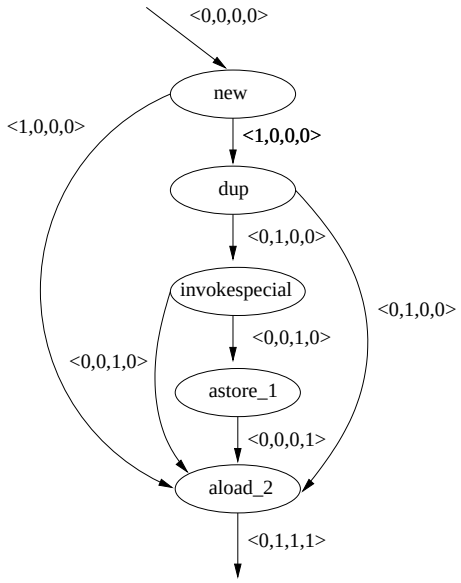

Figure 4.1: The control flow of a `try...catch` block fills a window unnecessarily with allocations in the naïve framework.

ago, respectively. Using the above meet, $\langle 0,0,1,0\rangle \wedge \langle 0,1,0,0\rangle = \langle 0,1,1,0\rangle$. Clearly all information has been retained and therefore the result can never be better than the input vectors. However, as we shall see in the next section, this meet function is overly conservative.

Each transfer function, $f \in F$, must update the solution at a given node, n, so that the output of n encompasses the instruction represented by it. This is accomplished by a simple right shift of the solution bit-vector. If n represents an allocation then a 1 is shifted in; if n is a non-allocation then a 0 is shifted in. The *least recent* bit (rightmost in the bit-vector) is shifted out.

The naïve framework works well on simple Java programs, yielding allocation rates of some 2–3 allocations per 16-instruction window. However, when we turned to real benchmarks (such as `jess`), we found overly conservative solutions from using logical bitwise *or* as the meet operator. Our framework computed some 15 allocations per 16-instruction window. We discovered that this high allocation rate was caused by blocks of code similar to the one shown in Figure 4.1.

The fact that our meet operator retains all information from its input vectors gives us an artificially high allocation rate in certain cases. The example in Figure 4.1 may seem contrived, but it is exactly what happens within a Java *try-catch* block, or within a *monitor*. We need a meet function, the result of which is no better than any

of its input vectors, without being overly conservative. By looking at the example in Figure 4.1, it is apparent that one of the problems is that the meet function, at the last node, increases the number of allocations in the solution. It seems reasonable to restrict the meet function so that its result cannot contain more allocations than any one of its input vectors.

When each incoming vector contains at most one set bit (allocation) this is simple enough. The meet will just return the incoming solution that has seen an allocation most recently. But how should the meet function react when one or more of its input vectors has more than one allocation? Clearly, the result will contain the same number of allocations as the vector with the most allocations, but where will they be placed? For example, say we need $\langle 0,1,0,0,1\rangle \wedge \langle 0,0,1,1,0\rangle$. One idea is to set the most significant bits in the result: $\langle 0,1,0,0,1\rangle \wedge \langle 0,0,1,1,0\rangle = \langle 1,1,0,0,0\rangle$. This is better than logical bitwise *or* because it does not increase the number of allocations.

Nonetheless, this meet produces a solution that reflects that the last instruction it encountered was an allocation when none of its input vectors reflected that fact. A better idea is to let the meet place allocations in the positions of the most significant set bits in its input vectors: $\langle 0,1,0,0,1\rangle \wedge \langle 0,0,1,1,0\rangle = \langle 0,1,1,0,0\rangle$. This meet will never increase the number of allocations and it will never place an allocation at a position that all of its input vectors regard as a non-allocation. Still, this meet function is overly conservative because it accounts for the most recent allocation in both input vectors when, through any given path, only one of them may occur. In the next section, we will refine this idea further.

4.3 Better Framework

A better way to compute *meet* in light of the example shown in Figure 4.1 is as follows. We scan the bit-vectors a and b from left to right (most recent to least recent). At each position i, we compute the corresponding bit of the result vector, c, by taking the bitwise *or* of a_i and b_i. If the resulting bit, c_i, is set, then we reset the leftmost, non-zero bit of a and of b. The intuition is that the resulting 1 in c covers the next allocation in a and in b, whether it comes at position i or later.

For example,

$$\langle 0,1,0,0,1\rangle \wedge \langle 0,0,1,1,0\rangle = \langle 0,1,0,1,0\rangle$$

The result reflects the fact that the most recent allocation was encountered two instructions ago, and that the second most recent was encountered four instructions ago. Our experiments were conducted using this framework, but accounting properly for object size as described in Section 4.3.2.

4.3.1 Framework Evaluation

As previously mentioned this dataflow framework requires interprocedural evaluation. The reason for this is that a program is more than a collection of procedures. Procedures interact with one another and this interaction may change the allocation rate in the procedures involved. In their work, Reps et al. [17] defined the notion of a supergraph G^*. We use a modified version of their definition to build our interprocedural framework.

Recall that in Section 2.5 we presented the intraprocedural dataflow (Figure 2.3) graph for an example C program. Building an interprocedural solution from this is conceptually trivial. As shown in Figure 4.2, the only changes that are made to the intraprocedural graph is to connect method calls to the actual flow-graph for the called method. If procedure A calls procedure B, then creating the interprocedural graph from the intraprocedural ones is simply a matter of connecting the call node in A, with the start node in B, and the exit node in B with the node directly downstream from the call node in A. However, it would be prohibitively costly for any program of size, to reevaluate procedure B every time a node anywhere in the program that calls B is encountered. Furthermore, reevaluating B implies that all procedures called by B would have to be reevaluated, and so forth. In our detailed description of the algorithm that we use to evaluate our interprocedural framework, we show how we get around this problem.

We use the following notation, based on the work by Reps et al [17], to formally specify our interprocedural dataflow framework:

- $G^* = (N^*, E^*)$
- P^* = the set of all procedures p represented in G^*
- $N^* = \bigcup_{p \in P^*} N_p$
- $E^* = E^0 \bigcup E^1$
- $E^0 = \bigcup_{p \in P^*} E_p^0$ is the collection of intraprocedural control-flow edges.

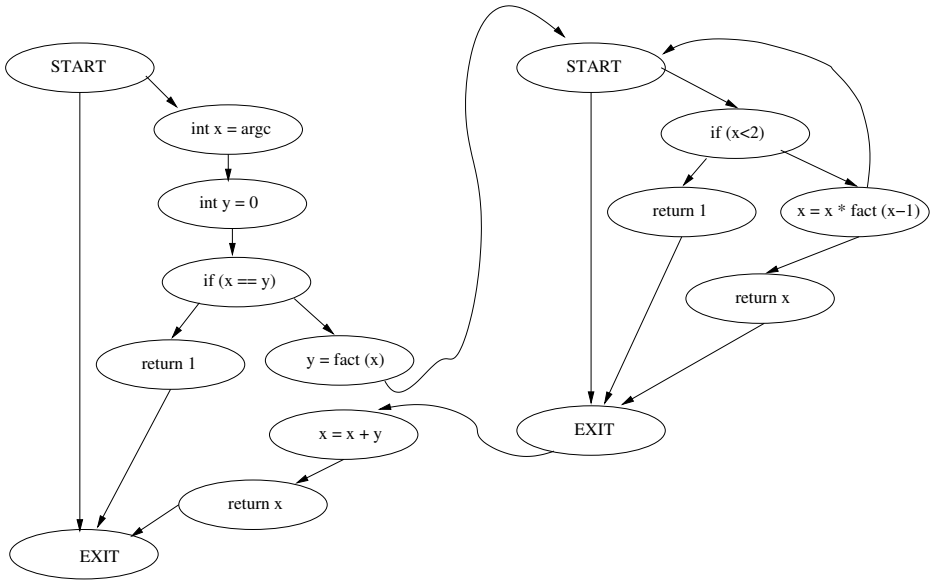

Figure 4.2: Interprocedural dataflow framework generated from our example program from Figure 2.3

- $E^1 = \bigcup_{p \in P^*} E^1_p$ is the collection of procedure call and procedure return edges.

  We also define the functions:

- $calledBy(p, G^*) \rightarrow N'$ where $N' \subset N^*$ is the set of call nodes that call procedure p

- $calcIntra(p)$ calculates the intraprocedural solution for procedure p as given by the framework of Section 4.3

The basic algorithm for calculating the interprocedural maximum allocation rate is as follows:

```
Interprocedural Data Flow
  Initialize
1   for each p ∈ P* do
2     calcIntra(p)

  Update
3   while there are changes in G* do
4     for each p ∈ P* do
5       N' ← calledBy(p, G*)
6       s_p.in ← ⋀_{n∈N'} n.in
7       calcIntra(p)

8       for all n ∈ N' do
9         n.out ← e_p
```

In the above algorithm, $n.in$ refers to the combined input to node n, $n.out$ refers to the output of node n, and s_p and e_p refer to the start and exit nodes of procedure p, as mentioned in Section 2.5. The most important steps of the algorithms are line 6 and 9. At line 6, all the calls made to procedure p are combined into one using the meet operator. The reason for doing this is twofold. First, it reduces the computational complexity because several procedure calls are merged, reducing the number of times $calcIntra(p)$ needs to be called. Also, if p makes any procedure calls, then for each data flow solution created by $calcIntra(p)$, each procedure called by p would have to be reevaluated. Second, it reduces space complexity. To see this, consider that fact that each data flow solution resulting from a call to $calcIntra(p)$ is contained in G_p, and thus G_p must be stored from iteration to iteration. By combining all procedure calls to p, we never have to keep more that one copy of G_p at any given time.

The price we pay for the decrease in computational and space complexity is that our interprocedural analysis will be more conservative than it otherwise would. However, our results in Section 4.5 confirm that we obtain reasonable solutions with this approximation.

4.3.2 Accounting for Allocation Size

We now revisit the issue of allocation size, focusing first on scalar objects and then on arrays. While most programs allocate objects of varying size, we have observed that

most allocations are small—on the order of 12 bytes. Because object size depends on object type in Java, most programs exhibit a *locality of size*, meaning that object sizes that have been frequently allocated in the past are likely to be allocated in the future [2]. However, we are obligated to compute a maximum allocation rate, and this cannot be based on average or expected behavior.

In most cases, determining the size of an allocated object statically is relatively straightforward. An object's storage can be computed as the sum of the sizes of all the fields in the object plus the object's header. Our results were obtained using Sun's JDK Java execution environment, in which objects have a header of 8 bytes and in which almost all fields are 4 bytes. The only exceptions are the types `double` and `long` which occupy 8 bytes. At an allocation, we compute each object's size using the Java reflection package.

Statically-Bounded Array Allocations

Computing the size of statically allocated arrays adds more complexity to our allocation size calculations. The size of these arrays can be bounded by a constant statically in Java, but such an analysis is slightly complicated, as demonstrated by the following example, where φ is some boolean condition that is not known statically:

```
Array Allocation
1   MyObj a[]
2   size ← 100
3   if φ then
4     size ← 10
5   a ← MyObj[size]
```

Static determination of the size of statically allocated arrays is in itself a data flow problem called range propagation [9]. While similar to constant propagation [23], the difference is that we do not propagate whether or not a variable is a constant; instead, we propagate bounds (a range) of the possible value of a variable. We have incorporated this range analysis, implemented by Morgan Deters [9], into our static analysis. When the number of elements of the array is known, determining its size is simply a matter of multiplying the number of elements with the size of the array type. In Java, arrays of objects are in fact arrays of reference type, so for object arrays we do not have to worry about the size of the constituent objects when computing the memory footprint of an array—each array element is of pointer size. For now, we

assume that the size of all arrays can be known statically. In the next section, we will relax this assumption.

As we are accounting for the size of what is being allocated, our meet lattice L and set of transfer functions F must be modified. Modification of the transfer function is straightforward: instead of shifting in a 1 for an allocation, we shift in the actual size of the object being allocated. The modification of L is shown below and an example is given in Figure 4.3.

- $A = \{0, 1, 2, \ldots, M\}$ where M is the maximum number of bytes that the allocator can allocate at one time
- $\top = \langle 0, 0, 0, \ldots, 0 \rangle$
- $\bot = \langle M, M, M, \ldots, M \rangle$
- $\preceq$ is defined in terms of the $\wedge$ operator such that $a \preceq b \Leftrightarrow a \wedge b = a$
- $\wedge$ is shown in Figure 4.3 and described below

Previously, we defined the meet function in terms of a left-to-right scan of the input vectors. When all allocations were equal we could simply align the most recent allocations in each vector, then the second most recent, and so on. This meet function can be generalized to work with allocations of varying size. Below we present the algorithm for computing the meet of two allocation vectors.

```
meet(v1[], v2[])
  Initialize
1    ret[v1.length]
2    diff1 ← 0
3    diff2 ← 0

  Compute Meet
4    for i ← 0 to result.length do
5      ret[i] ← max(v1[i] - diff1, v2 - diff2)
6      diff1 ← diff1 + (ret[i] - v1[i])
7      diff2 ← diff2 + (ret[i] - v2[i])

8    return ret
```

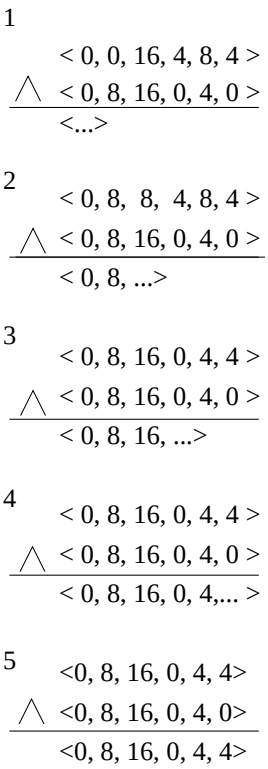

Figure 4.3: Computing meet when accounting for object allocation size.

Figure 4.3 shows how the above algorithm works on two specific input vectors. Step 1 shows the original input vectors. These are never modified—steps 2–5 work with copies of the original vectors.

At step 2, in Figure 4.3, 8 bytes are moved from the most recent allocation of the top vector to compensate for the fact that the bottom vector has an allocation of 8 bytes occurring more recent. The resulting vector of the meet can now be filled up to this point. At step 3, both vectors have an allocation at the same position, but now the allocation of the top vector is 8 bytes smaller. Consequently, we move bytes from earlier (further to the right) allocations to compensate, and we can update the resulting vector. At step 4, both allocations occur at the same position and they are equal in magnitude, the result vector is updated accordingly. Finally, at step 5, the top vector has an allocation but the bottom vector has no more allocations. From here on, had the top vector had more allocations left, the bottom vector can be ignored and the result vector is simply filled with the allocations in the top vector.

Arraylets

The fact that languages like Java allow dynamically allocated arrays complicates our analysis by forcing us to attempt to place a static bound on an allocation, the size of which cannot be statically known. To this end, we look at how existing work handles large allocations. Bacon et al. [2] suggest the use of arraylets to solve the problem that large objects cause for real-time garbage collectors. The idea is to represent large arrays as a sequence of arraylets where each arraylet, except for the last, is of a constant size, C. Siebert [21] uses a similar idea and represents large arrays as a tree structure of fixed sized blocks.

As mentioned in the beginning of this chapter, we have assumed that each virtual machine instruction is executed in one clock cycle. This is not the case for many instructions. In fact, instructions that allocate memory take time proportional to the size of the allocation. When any object in Java is allocated, first the amount of memory needed is reserved from the heap. Then all fields are initialized to zeroes (typically 4 bytes at a time on a 32-bit processor). This means that each allocation of size K bytes is followed by x number of assignments, where $x = \lceil \frac{K}{4} \rceil$. However, as aforementioned, the clock cycle assumption is valid because assuming that all instructions take one clock cycle to execute cannot lower the upper bound we are computing—in fact, it might raise it.

To maintain the generality of this implementation, we will not include the initialization instructions for allocations of objects, other than arrays, in our analysis. We will include the initialization instructions for array allocations in order for us to be able to compute an upper bound on the allocation rate resulting from these allocations. Using the idea of arraylets, we assume that the size of all array allocations of (statically) unknown size is some multiple of the arraylet size, C, reported by Bacon et al. [2] as $C = 2\text{KB}$. If we assume that our window size W is smaller then $\frac{2KB}{4B}$, then we can bound the allocation rate behavior of all array allocations of unknown size.

Figure 4.4 shows how allocations of dynamic arrays can be represented. Directly following the allocation of the array, we assume that one arraylet has been allocated. We can do this since we are assuming that each unknown-size array allocation is allocated as arraylets and that each arraylet is allocated and initialized before the next arraylet is allocated. As aforementioned, $W < \lceil \frac{2KB}{4B} \rceil$. This means that when the next arraylet is allocated, the allocation for the first one will have fallen out

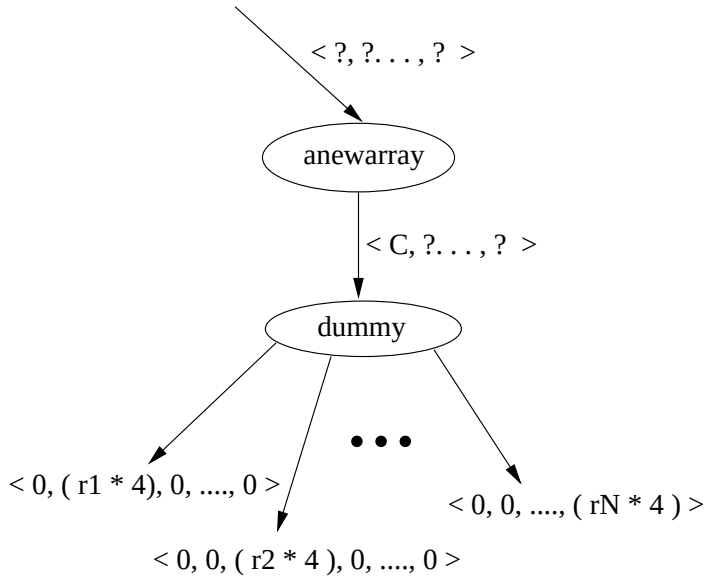

Figure 4.4: An array allocation, as represented in the control flow graph. ('?' represents any instruction.) $0 < r \leq N$, $N = W - 1$

of the window. The key point here is that the number of arraylets that are allocated will have no effect on the overall allocation rate.

Following the array allocation instruction we insert a dummy node. This node accounts for the fact that the last arraylet to be allocated may not be large enough for its initialization instructions to push that allocation out of the window. The range of r, the number of elements in the last arraylet, that we must account for is $0 < r \leq N$, where $N = W - 1$. Because we do not know the size of r, only its range, we must account for all possible values of r, with its subsequent initialization instructions. This is the output from the dummy node in Figure 4.4. Taking the meet of all the output vectors from the dummy node gives us the vector $\langle 0, 4, \ldots, 4 \rangle$.

We have placed an upper bound on the allocation rate that can result from the allocation of a statically-unbounded array allocation. This bound is based on the assumption that the allocator will allocate arrays as a sequence of fixed sized arraylets. Similarly, if the allocator allocates large objects as a sequence of smaller allocations, this technique can be used to estimate an allocation rate for those allocations, assuming that we are including the initialization instructions. In this case, and in the case of statically allocated arrays, r will be known and thus the output from the dummy node will be one of the output vectors in Figure 4.4 rather than the meet of all of them.

If the allocator does not handle statically-unbounded array allocations as arraylets, there is little we can do to compute a good upper bound on the allocation rate. We would be forced to assume that $C = M$ in Figure 4.4. Since the array actually allocated may not be large enough for the subsequent initialization instructions to "push" the allocation out to the window, we would still need to use the dummy node and meet all of its output vectors.

4.3.3 Properties of the Framework

First, we must show our meet operator satisfies the properties enumerated in Section 2.5 For any $a, b \in A$:

1. $a \preceq b \Leftrightarrow a \wedge b = a$
2. $a \wedge a = a$
3. $a \wedge b \preceq a$
4. $a \wedge b \preceq b$
5. $a \wedge \top = a$
6. $a \wedge \bot = \bot$

Clearly our meet satisfies $2, 5$ and 6. Properties $1, 3$ and 4 hold because our definition of $\preceq$ is based on meet.

Second, to guarantee that our data flow framework converges, we must show our framework is monotone, see Equation 2.3 in Section 2.5. In our framework, a node's transfer function shifts in the amount of memory allocated at each instruction (0 for a non-allocating instruction). The value that is shifted in cannot affect the rankings of the vectors because all vectors have the same value shifted in. The value that is shifted out is at the right-most position of the allocation-vector. The comparison ($\preceq$) is based on the leftmost values, which means that the right shift operation cannot affect the relationship, with respect to $\preceq$, of the vectors involved. Thus, no $f \in F$ can output a better solution given a worse input and therefore our framework is monotone.

As a result of the above, a data flow solution will converge such that the maximum allocation rate we compute at any point in a procedure is no lower than what could be seen on any path arriving at that point.

Thus far, we have a solution that is valid and that is guaranteed to converge, but at issue still is the quality of our solution relative to what could ideally be computed on each path separately through a procedure. As mentioned in Section 2.5, a distributive framework (see Equation 2.4) guarantees that the (intraprocedural) solution we compute is the best possible static solution of the framework.

Consider two generic vectors, a and b, containing n elements. The effect that the transfer function, f, has on a and b is that all values in the vectors are shifted one step to the right: a_2 takes on the value of a_1 and so on. a_n and b_n are shifted out of the window and a_1 and b_1 take on the value that is shifted in.

Let $f(a) = a', f(b) = b', a' \wedge b' = c'$ and $a \wedge b = c$. We want to show that $f(c) = c'$ to prove distributivity. For a given node, the semantics of f guarantee that $a'_1 = b'_1$ and since $a \wedge a = a$, it follows that $a'_1 = b'_1 = c'_1$. Thus c'_1 is the value shifted in by f, which by definition is $(f(c))_1$. Given any vector y, the values at $y_1 - y_{n-1}$ prior to applying $f(y)$ will still be in the vector after applying $f(y)$. $f(y)$ will shift all values one step to the right. Thus, for $1 < i \leq n$ $c'_i = c_{i-1}$. $f(c)$ moves c_{i-1} to c_i for all $1 < i \leq n$, and we already know that $c'_1 = (f(c))_1$. It follows that, $f(c) = c'$, so our framework is distributive and our solution is no worse than the meet-over-all-paths (MOP) solution.

Lastly, we will consider the speed of evaluation of our framework. Recall the definition of a rapid framework, Equation 2.5. It would be ideal if our framework were rapid, because we would be able to converge upon a solution more quickly. However, our framework is not rapid, since each trip around a loop can shift in another 1-bit and thus one iteration of the graph cannot collect all information needed for convergence.

4.4 Experimental Setup

We have implemented our static analysis for maximum allocation rate and array allocation bounds on top of *Clazzer* [12], a byte-code manipulation framework in which data flow problems can be explicitly defined and solved. Clazzer implements the *Role* software pattern [3], which allows the graph of instructions for a program to "play" different roles. To implement our framework, we defined the appropriate roles and plugged those into the Clazzer framework. This allowed us to leverage all the power of Clazzer, which decreased the amount of implementation we had to do.

For each procedure called by the mutator program, our framework needed to construct an instruction graph. In our current implementation, all instruction graphs

used are kept in memory, which means that a fairly large heap needs to be allocated for this analysis. We ran our experiments using a heap of size 1GB. If this memory requirement causes problems, instruction graphs can be written out to disk to save space. This, of course, incurs a penalty in efficiency. As the instructions of the mutator program are being examined to build the instruction graphs, the software pulls in class files as calls to procedures defined in them are encountered. In our first implementation, several of the class files were read from disk multiple times because the file handle was closed after the graph for the needed procedure was built. Later implementation utilizes a cash like structure to minimize the number of disc I/O operations.

4.5 Results

In this section, we report on the application of our analysis on some Java benchmarks. While those benchmarks are not real-time benchmarks, portions of what they do (audio decoding, expert shell problem resolution, image rendering, etc.) could arguably be included in a real-time application. When the real-time community accepts real-time garbage collection—we hope this thesis takes steps in that direction—then real-time Java programs and benchmarks should be more plentiful.

Figure 4.5 displays our static determination of maximum allocation rates of benchmarks in the jvm98 SPEC benchmark suite.[1] We used window sizes of 16, 32, 64, 128, 256, and 512 clock cycles. Figure 4.5 illustrates the problem associated with relatively small window sizes: When the window size is small each allocation has a dramatic effect on the overall maximum allocation rate. The plot also shows that as the window size increases, the maximum allocation rate decreases, asymptotically approaching a bound of the average allocation rate of the entire program. This is expected; in Chapter 3 we presented a dynamic analysis of a subset of the jvm98 SPEC benchmark suite demonstrating that the maximum allocation rate approaches the average rate as the window size increases.

We know that doubling the window size can never increase the allocation rate. Intuitively, we can show this by considering a window of size n with a maximum allocation rate of $\frac{x}{n}$ where x is the maximum number of bytes allocated in any window of size n in the program. If doubling the window size increases the maximum allocation

[1]The 'mtrt' benchmark is currently excluded because our approach has not yet been extended to support multithreaded target programs.

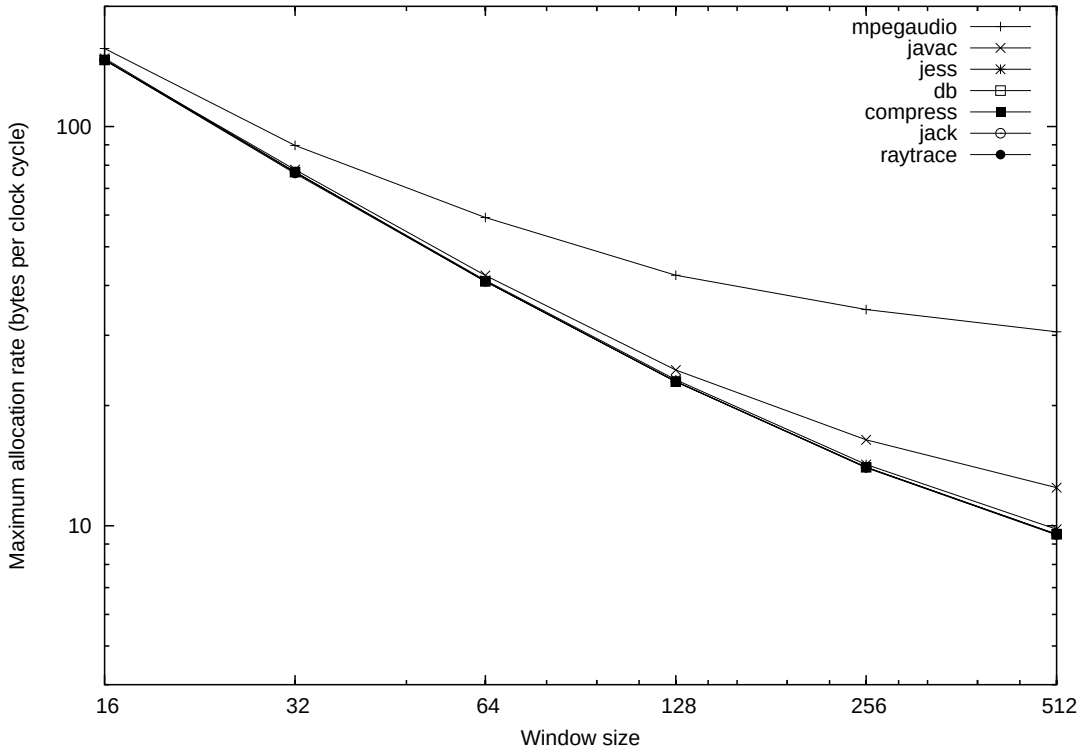

Figure 4.5: Maximum allocation rate *vs* window size (statically-determined bound).

rate of the program, then there exists an x' such that $\frac{x}{n} < \frac{x'}{2n}$. This implies that $x' > 2x$. It must also be the case that $x' \leq 2x$ because x is the maximum number of bytes allocated in any window of size n—doubling n cannot more than double x. We have a contradiction, so doubling the window size cannot increase the maximum allocation rate.

As a consequence, the static upper bound of the maximum allocation rate for a sufficiently large window can be used to approximate an upper bound for an arbitrarily large window. For example, the results in Figure 4.5 suggest that using a window size of 256 as an approximation is not overly conservative.

The dynamic analysis performed in Chapter 3 used the metric $\frac{bytes}{ms}$ to quantify the allocation rate of the mutator program. This is a temporal metric and therefore the results from Chapter 3 cannot be compared directly to our statically computed results. To enable a comparison between our statically computed maximum allocation rate and the observed maximum allocation rate of a given execution of the program, we modified our dynamic data-collection mechanism in the Java Virtual Machine (JVM) to record the allocation rate as $\frac{bytes}{clockcyle}$. For this, too, we limited

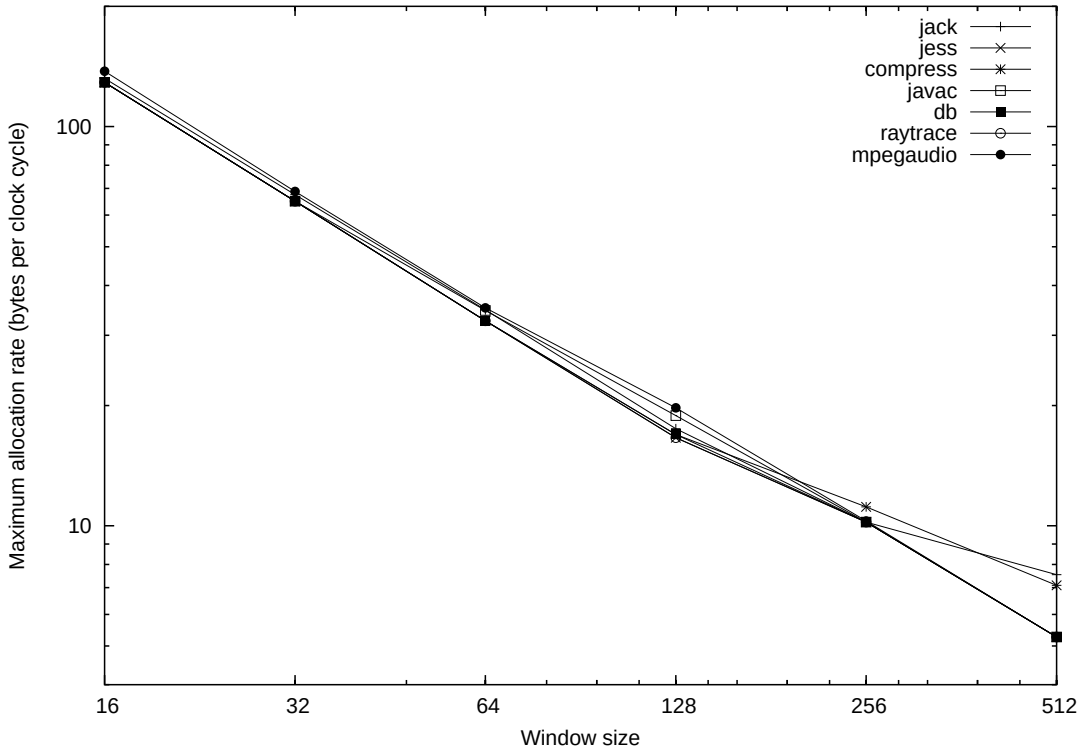

Figure 4.6: Maximum allocation rate *vs* window size (actual observation).

array allocations to a two-kilobyte arraylet size and inserted enough zero-allocation entries in the window to account for initialization of the array memory. Figure 4.6 shows the maximum allocation rate observed during a run of size 100 of each of these benchmarks.

We offer comparisons of our static bounds and dynamically-collected results in Figures 4.7 and 4.8—Figure 4.7 compares the static bound to the observed allocation rate in the jess benchmark, and Figure 4.8 makes the comparison over all benchmarks in the suite.

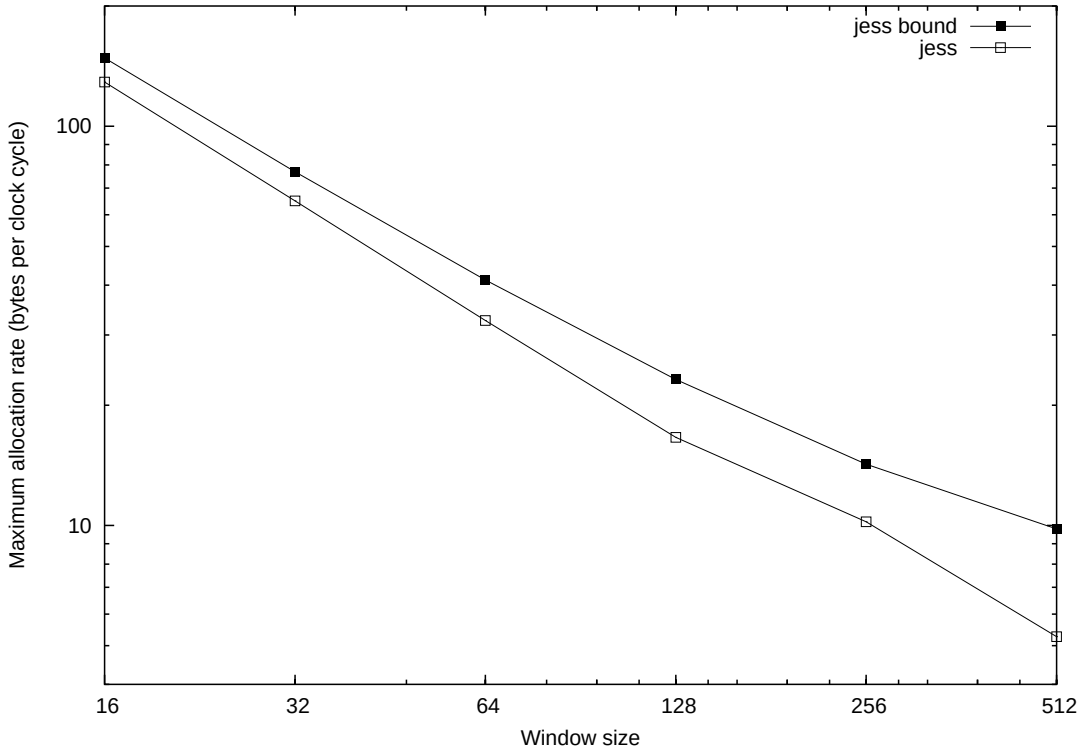

Figure 4.7: Comparison of bounded and actual maximum allocation rates for jess.

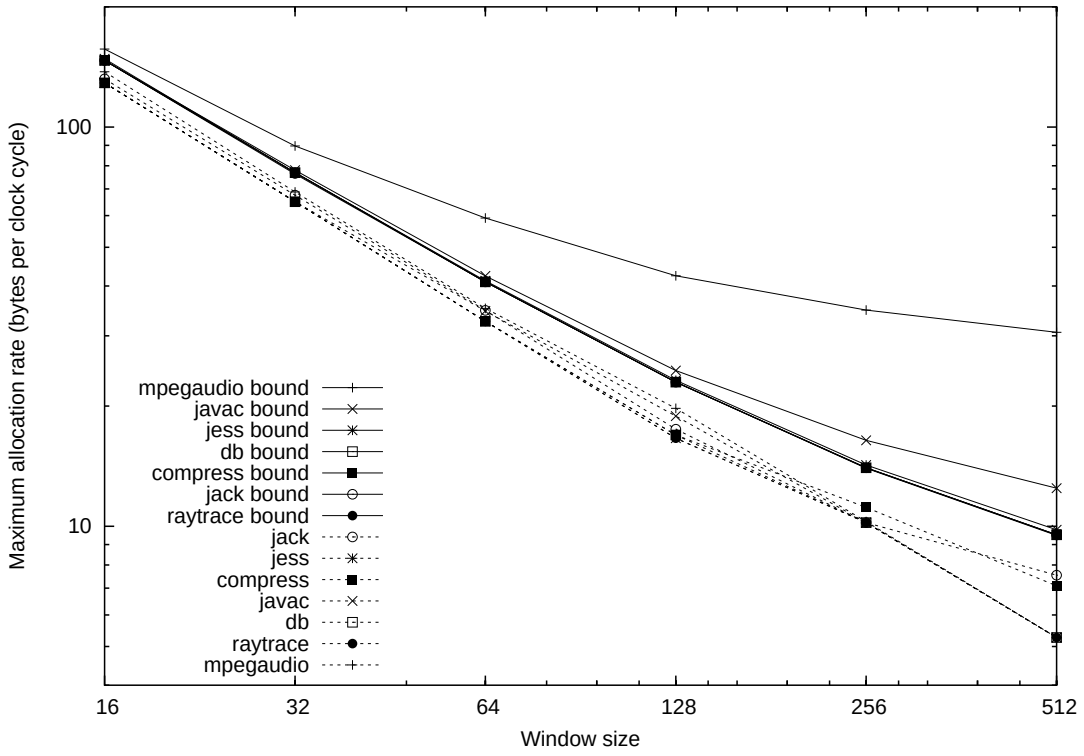

Figure 4.8: Comparison of bounded and actual maximum allocation rates for all benchmarks (both plotted together).

Chapter 5

Analysis of Findings

Previously in Chapters 3 and 4 we presented results from a dynamic and a static analysis of the allocation rates of programs in the jvm98 SPEC benchmark suite. The significance of these results is the topic of this chapter.

5.1 Dynamically Quantified Allocation Rate

The results from Chapter 3 are not directly applicable to the problem that we are addressing, but they do serve to highlight some properties of our problem domain. Figure 3.3 and Figure 3.1 show that for garbage collection cycle times, $\Delta\tau$, exhibited by a real-time collector the maximum allocation rate is a poor estimate of the average allocation rate. Using the maximum allocation rate to determine when to schedule the collector and the memory requirement of the program means that resources are wasted whenever the mutator does not exhibit the maximum rate.

However, because the garbage collector must budget for the worst-case scenario one could argue that, although the maximum allocation rate is an abnormality, budgeting for worst-case necessitates the use of this value. This is a valid argument, but by observing that the maximum allocation rate of the mutator is dependent on the $\Delta\tau$ used to measure the allocation rate, it becomes clear that the worst-case will vary with the time allotted for one complete collection cycle. The average allocation rate of the program is calculated using a window size that is the entire length of the program, $\gamma(0, T)$. This means that as $\Delta\tau$ increases we can expect the maximum allocation rate to approach the average allocation rate, as shown in Figure 3.3.

Furthermore, Figure 3.4 shows that as $\Delta\tau$ increase we can expect an increase in MMU. This follows from the fact that by increasing $\Delta\tau$ the work of the GC is

spread out over a longer period of time thus giving the mutator a larger portion of CPU time. The observation is obvious, but it serves to highlight an additional benefit to increasing $\Delta\tau$.

Ideally it appears that we would want to run the program with $\Delta\tau$ as large as possible. Yet, there is a problem with this argument. The largest possible $\Delta\tau$ is the entire length of execution, which is synonymous to executing the program without any garbage collection at all. In general, increasing $\Delta\tau$ will increase the memory requirements of the program. Nonetheless, because max does not approximate average, budgeting for the worst-case means that most of the time when the mutator does not allocate at the maximum rate, a significant part of the heap will be unutilized. This unutilized memory leaves room for improvement.

The collector must be scheduled so that the amount of memory available when collection starts makes it impossible for the mutator to deplete its memory resources during the collection cycle. If this decision is based on the assumption that the mutator allocates memory at the maximum allocation rate of the program then, on average, the collector will be scheduled prematurely. If the work of the collector is mostly dependent on amount of garbage being collected, then this would not be significant because the collector would perform less work during each cycle. However, as discussed in Chapter 2, the work of a mark-sweep collector is dependent on the amount of live memory that needs to be processed during the marking phase. This means that scheduling the collector more frequently will not have a considerable affect on the amount of work performed during each collection cycle. Consequently, additional collection cycles result in the collector needlessly occupying CPU resources.

There are two approaches to solving this problem, and both will require knowing more about the allocation rate of the mutator than its maximum allocation rate. Either the collector can adapt its scheduling policy as the allocation rate changes and execute less frequently during periods of low allocation rates, or it can adapt the $\Delta\tau$ used to the change in allocation rate. The mutator is provided with enough heap memory to handle its maximum allocation rate. Therefore, there is room for increasing $\Delta\tau$ when the mutator does not exhibit this maximum.

Out of these two approaches, we prefer the latter one because increasing $\Delta\tau$ also increases MMU during that part of the execution. Hence, for all periods of execution time where $\gamma(\tau, \Delta\tau) < \gamma^*(\Delta\tau)$ we can safely increase $\Delta\tau$ and thus achieve a higher MMU than what has been previously reported, [2], without increasing the resources allocated to the program.

To be able to harvest these advantages, we require that the garbage collector can adapt to the changes in the allocation rate of the mutator. This means that the collector must be able to predict future allocation rates of the mutator. This prediction is provided by statically analyze the mutator program, as described in Chapter 4.

5.2 Statically Quantified Allocation Rate

How well does our static analysis bind the observed maximum memory allocation rate? In Figure 4.5, we see that using a window size of 256 clock cycles, the bound for most of our tested benchmarks is close to 15–20 bytes allocated per clock cycle. These bounds are artificially high, because all array allocations not bounded statically are assumed to be large, as described in Section 4.3.2. This means that even a very small array allocation could have a large effect on the upper bound. Figure 5.1 shows that the maximum allocation rate computed for the SPEC benchmark `jess`, is not representative of most procedures executed by the benchmark; most procedures in `jess` allocate between 5 and 7 bytes per clock cycle, and many allocate 0 bytes. Array allocations without a static bound force us to make a highly conservative assumption about their size—we might expect that procedures allocating such arrays actually allocate between 5 and 7 bytes per clock cycle, but we cannot determine that statically. Figure 5.2 shows that indeed array allocations are the problem here; when we don't make pessimistic assumptions about array size, the allocation rates of all procedures in `jess` are bounded by 7.1 bytes per clock cycle.

Figure 5.1 and Figure 5.2 show the maximum allocation rates exhibited by individual procedures in `jess`. These graphs give the appearance that many procedures exhibit fairly high allocation rates. This is misleading because it does not mean that all procedures with a high maximum allocation rate actually are heavy allocators. The analysis we are performing is interprocedural and thus allocations that occur in procedure p_1 might affect the overall allocation rate of a procedure p_2, called by p_1. This "spill-over" effect is what creates the appearance that many procedures are heavy allocators. The contrast between Figures 5.1 and 5.3, and Figures 5.2 and 5.4 makes it clear that the large numbers of interprocedurally-analyzed procedures with a high maximum allocation rate is caused by heavy allocation in relatively few procedures. The implication of this is that the maximum allocation rate of the whole program could potentially be lowered by modifying a few procedures. In addition

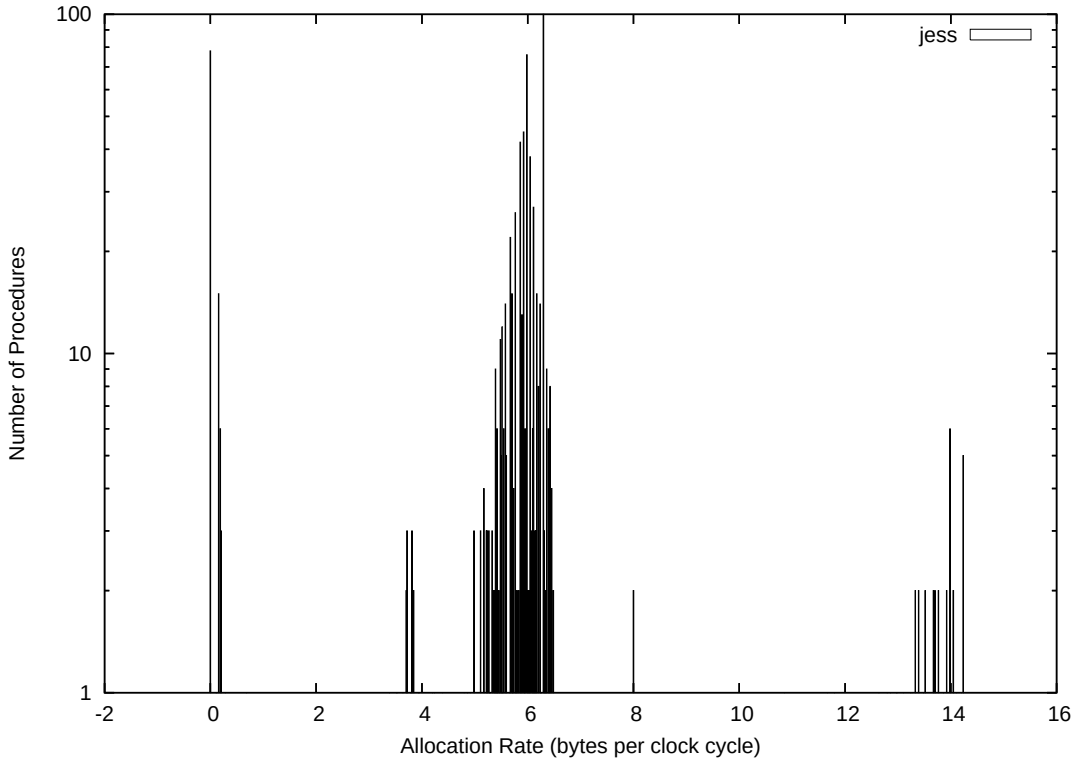

Figure 5.1: Number of procedures with a given upper bound for `jess` with a window size of 256, running interprocedural analysis using arraylets.

to binding the maximum allocation rate of the program, this framework can also be used to locate the procedures where rewriting the code carries the greatest potential gain.

As expected, the intraprocedural plots, Figures 5.3 and 5.4, also show that the maximum allocation rates of the heavily allocating procedures are caused by allocations of arraylets. Figure 5.3 has a spike at 8 bytes per clock cycle, which does not appear in Figure 5.4. $8 \times 256 = 2048 = 2\text{KB} =$ Arraylet size.

Figure 4.8 demonstrates that our static bounds indeed bound the maximum allocation rates, and that they were reasonable bounds for these benchmark runs. In particular, with the exception of the `mpegaudio` benchmark, our static bounds on allocation rate is within a factor of 2.5 of the actual, observed allocation rate over all tested window sizes. For `mpegaudio`, our static bound is 5.8 times the observed rate for a window size of 512. (The static bound on `mpegaudio` at smaller window sizes is considerably closer to the observed rate.) This is illustrated in Figure 5.5.

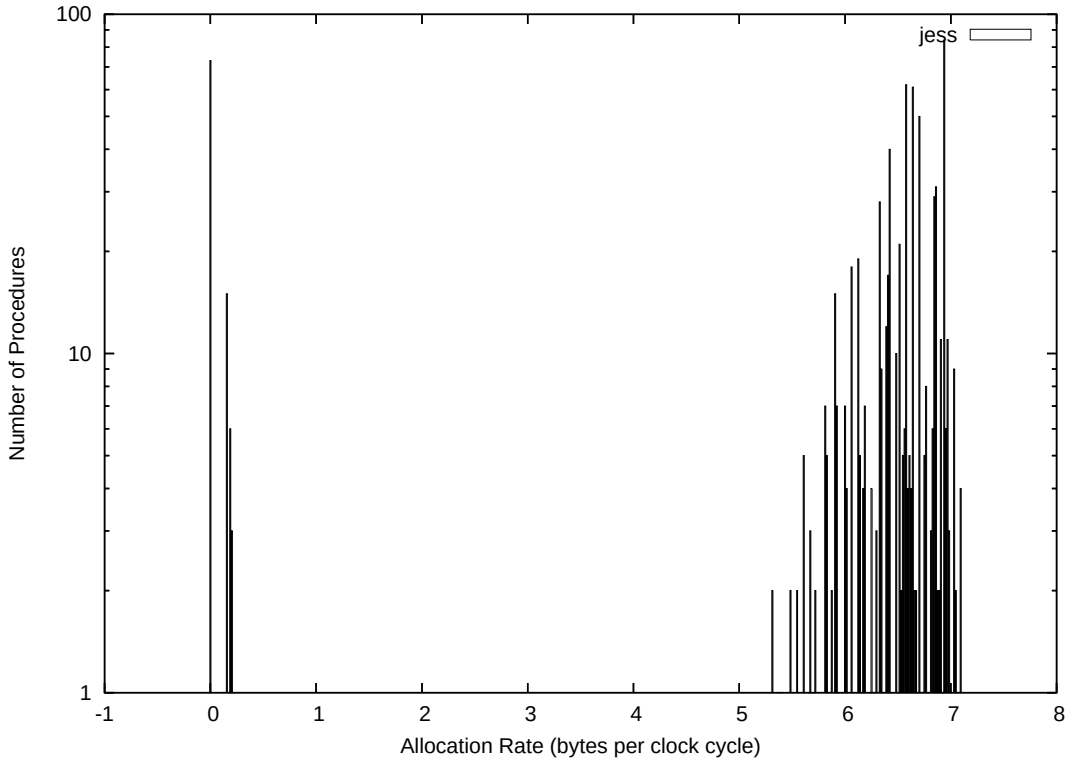

Figure 5.2: Number of procedures with a given upper bound for `jess` with a window size of 256, running interprocedural analysis assuming each array allocation is 16 bytes.

The static bound for `mpegaudio` deviates more from the observed rate than does the other benchmarks. The reason for this is that `mpegaudio` allocates one large array up-front and allocates very few objects during the rest of the run. Thus, the program experiences a "spike" of allocation, which we correctly bound, though by a factor of 5.8 off of its observed rate for that particular run. Static analysis must account for any path that could be taken in the code. In this case, such analysis determines that the allocation could happen in a loop (though it happens just once) and the steady-state worst-case allocation rate is 5.8 times higher than what was seen.

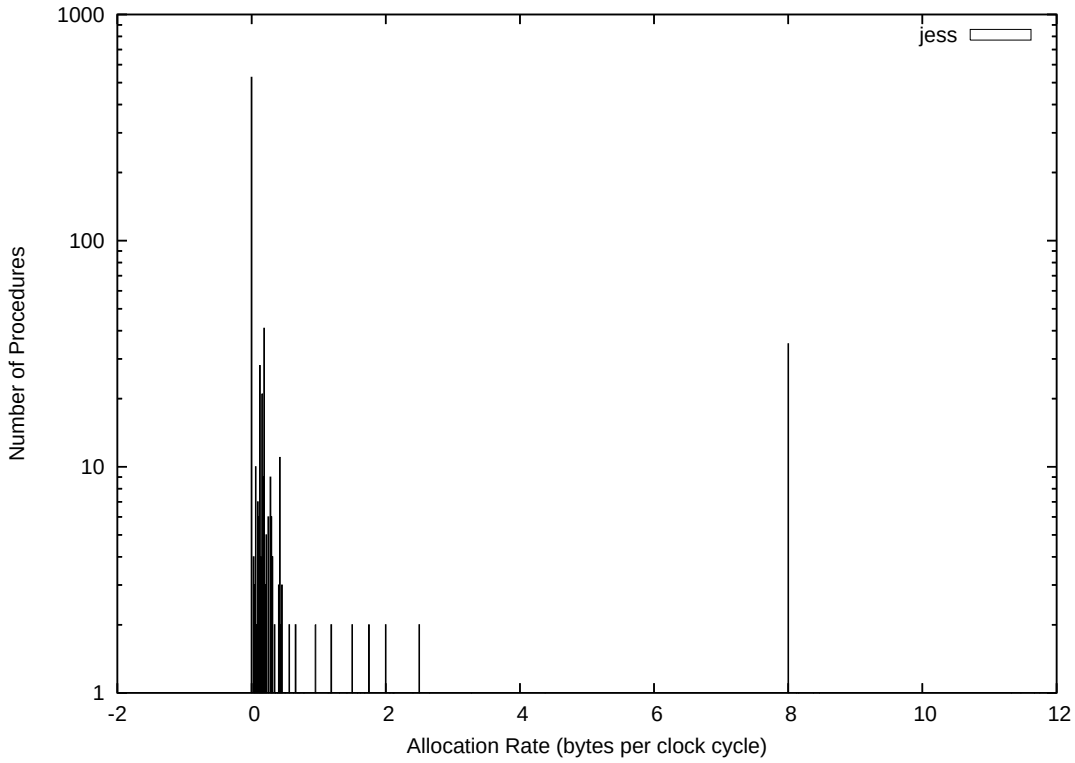

Figure 5.3: Number of procedures with a given upper bound for `jess` with a window size of 256, running intraprocedural analysis using arraylets.

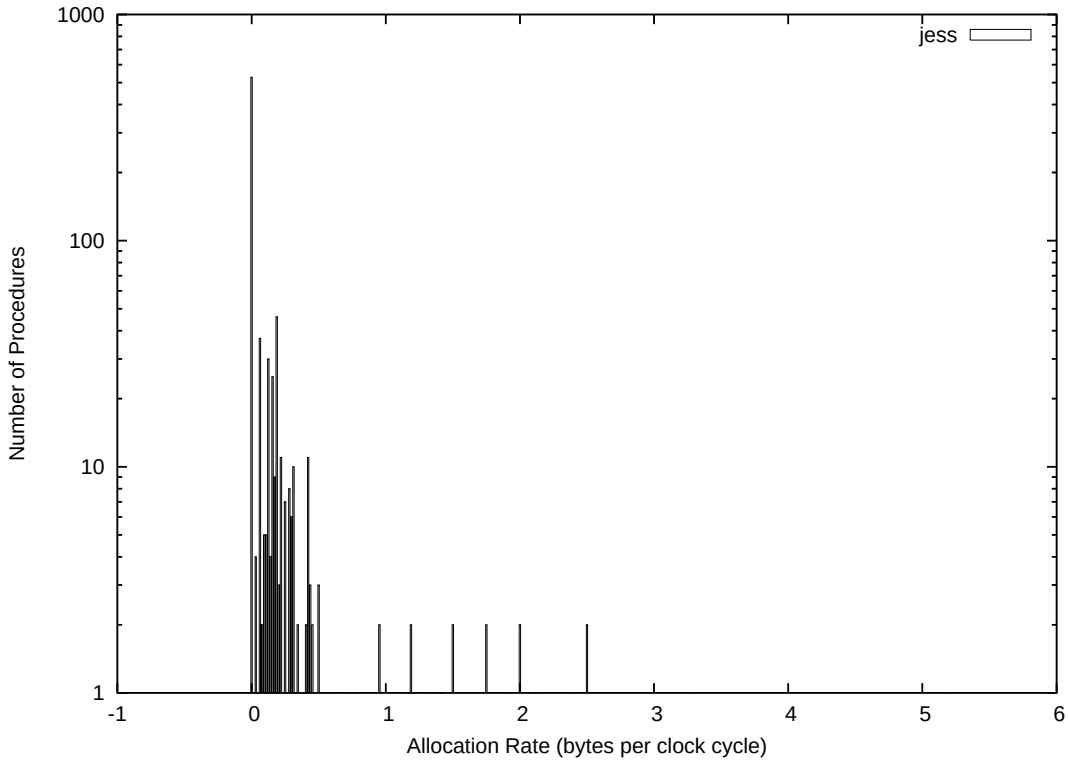

Figure 5.4: Number of procedures with a given upper bound for `jess` with a window size of 256, running intraprocedural analysis assuming each array allocation is 16 bytes.

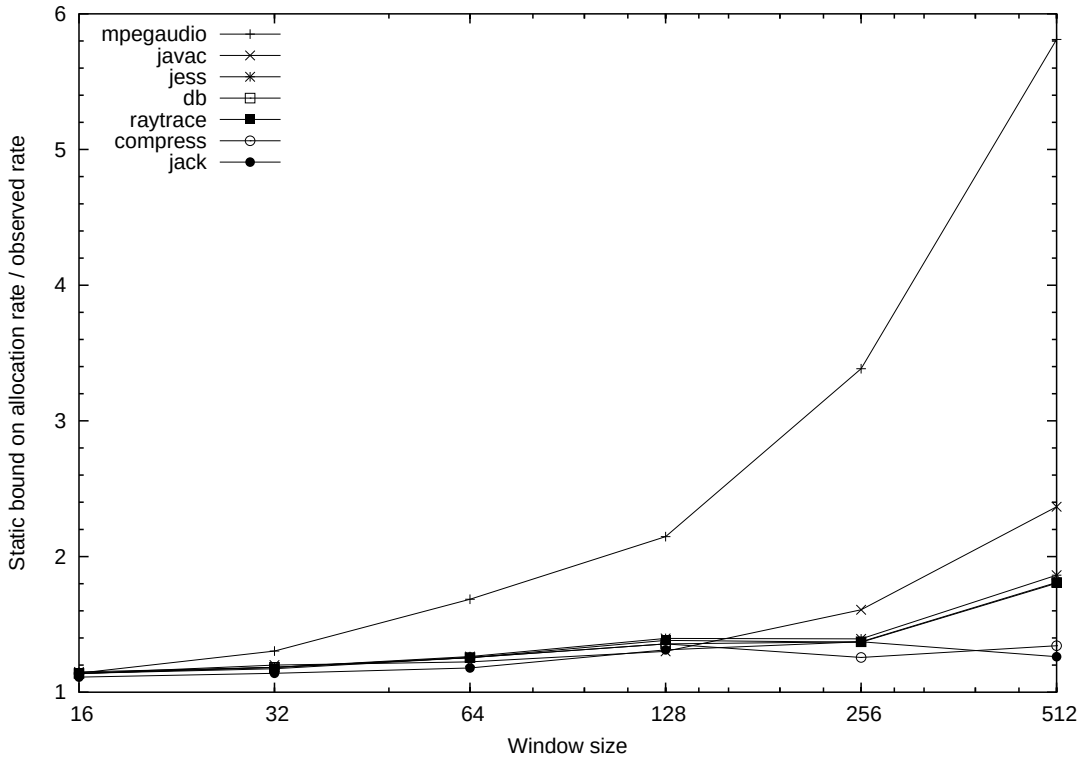

Figure 5.5: Comparison of bounded and actual maximum allocation rates for all benchmarks (static bound on rate / observed rate).

Chapter 6

Handling Multithreaded Programs

Until now we have purposely avoided the issue of multithreaded mutator programs, which will will be the topic of this chapter. Multithreading is an important issue because most programs operating under real-time constraints rely on this capability. However, threading present statical analysis with the problem of predicting when context switches will happen and how these switches will affect properties of the executing program.

6.1 Worst-Case Scenario

Below is an example of the instruction trace of three executing threads. For clarity, we have labeled non-allocation instructions as "instruction" and allocating instructions as "allocation". We also assume that in this particular example each allocation allocate 4 bytes of memory. If context switches can happen at any time during the execution of the mutator, assuming 0 overhead in terms of instructions executed for the switch, the statically analyzed allocation rate of the mutator must consider the following.

	Thread A	Thread B	Thread C
	...	...	...
1	instruction	instruction	instruction
2	instruction	allocation	instruction
3	allocation	instruction	allocation
4	instruction	instruction	instruction
	...	...	...

Analyzed in isolation, each of the example threads has one allocation in the window of four instructions displayed. Assuming that this mutator program has no other allocations, a static analysis using a window size of 4 instructions would report an upper bound on the mutator's allocation rate of 4 bytes per 4 instructions. However, there are numerous ways in which the threads can be scheduled to yield an actual allocation rate that is higher than the reported thread ignorant upper bound.

For example: Thread A executes instructions 1 and 2 and then yields to thread B. Thread B executes instruction 1 and then yields to thread C. Thread C executes instructions 1, 2, and 3 before yielding to thread A. Thread A executes instruction 3 and yields to thread B. Thread B executes instruction 2, and since there are no more allocations, the remainder of the execution pattern is of no concern. In this example, the actual allocation rate exhibited by the mutator program was 12 bytes per 4 instructions, a factor of 3 higher than the thread ignorant statically computed upper bound.

The problem is that if we cannot assume anything about how the threads are scheduled then any interleaving of instructions is possible. This means that a mutator with four threads, each of which performs 4 consecutive allocations at some point during their execution, could perform a total of 16 consecutive allocations. In general, if all mutator threads have the same number of consecutive allocations, the number of consecutive allocations that the mutator could perform increases by a factor of the number of threads. When considering allocations of varying size there are even more ways for the thread scheduler to demonstrate the incorrectness of the thread ignorant static analysis. For example, the largest allocation in each thread could occur one after the other.

6.2 Timing Context Switches

Fortunately, the above discussion is overly pessimistic. For one, we assumed that context switching between threads is free, meaning that its done with no effect on the allocation rate of the running program. In actuality, it takes some amount of time (clock cycles) for the operating system to perform a context switch. The context of the thread that just finished executing must be saved and placed in the appropriate queue. The next thread that is ready to execute [1], must be found and removed from its queue. Finally, the context of the new thread must be loaded into the CPU.

[1] According to some scheduling policy

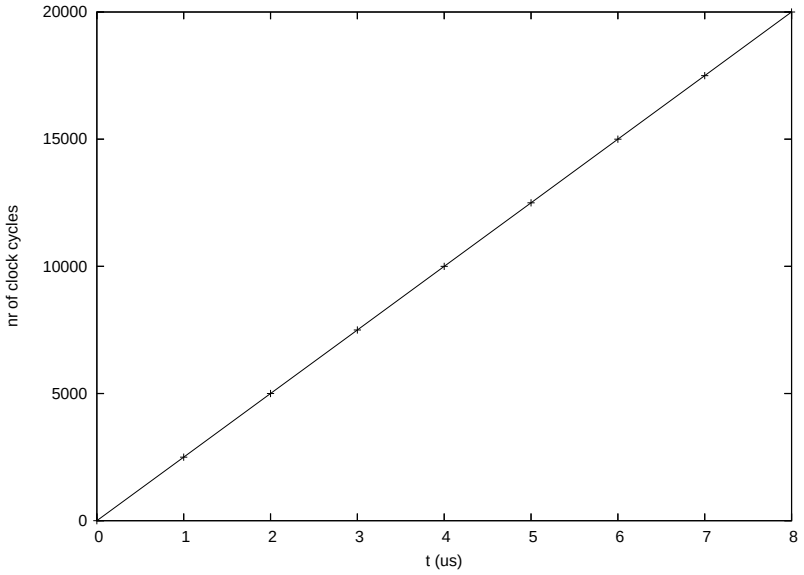

Figure 6.1: The number of clock cycles executed per μs on a 2.5G-Hz processor

In other words, some number of non-allocating instructions will be executed by the operating system between the last instruction of the thread being switched out, and the first instruction of the thread being switched in.

These instructions will not allocate any memory on the heap reserved for the mutator program. However, the CPU time needed to execute these instructions will be billed to the time quantum of the mutator. This has the implication that for each instruction executed by the OS during the context switch, a 0 (non-allocating instruction) is shifted into the allocation vector of our static solution. It follows that if the context switch execute more instructions than there are places in the allocation vector, by the time the new thread starts executing, all allocations from previous threads will have been shifted out of the window. The examples of Section 6.1 would not apply because whenever a new thread starts executing, its allocation window will be empty.

Figure 6.1 graphs the linear relationship between the cost, in number of clock cycles executed, of a context switch, and the time required to complete the switch. The worst-case is, of course, $t = 0$, as in the example shown in Section 6.1. As t increases, more and more non-allocation instructions are executed during the context switch.

To measure the time of a context switch, we modified a C program written by Dr. Edward G. Bradford, a senior programmer at IBM, presented in the online article

Runtime: Context Switching, Part I. [2] This program passes a one byte token back and forth between two threads using a UNIX pipe. One thread blocks on receiving the token, the other sends the token and then waits for it to be returned. This process is repeated a fixed number of times to measure the total number of context switches per second. The cost of passing the token back and forth was reported by Dr. Bradford as being negligible compared to the cost of context switching. The code, in the two threads that is being timed is shown below:

Thread A

```
gettimeofday(&start, 0);
for( i=0; i < count; i++){
  if(!send(pipeA, &token)
    break;
  if(!recv(pipeB, &token)
    break;
}
gettimeofday(&end, 0);
```

Thread B

```
gettimeofday(&start, 0);
for( i=0; i < count; i++){
  if(!recv(pipeA, &token)
    break;
  if(!send(pipeB, &token)
    break;
}
gettimeofday(&end, 0);
```

Using the this program we ran an experiment using two threads, context switching back and forth 1 million times. On average, 1 million context switches took 4.022 seconds to perform, meaning that each context switch needed, on average, 4.022 μs to complete. Figure 6.1 show that this roughly equivalent to 10000 clock cycles.

The remaining question is: How many Java byte code instructions can be executed during the context switch? A conversion factor between clock cycles and executed byte code is needed. This conversion must be conservative so that the fewest number of byte code instructions that can be executed during the time of the context switch is used. Consequently, we need to know how many clock cycles the most expensive byte code instruction needs to execute.

The literature on this topic agrees that the most expensive Java byte code instruction is the *invoke* instruction. However, the reports on how many clock cycles this instruction needs to execute varies from 15 [25] to 175 [18]. This large spread is due to the differences in the hardware used. In his thesis work, Martin Schoeberl [18] reported that invoke needs 175 clock cycles running on a Cyclone FPGA,

[2] `http://www-106.ibm.com/developerworks/linux/library/l-rt9/`

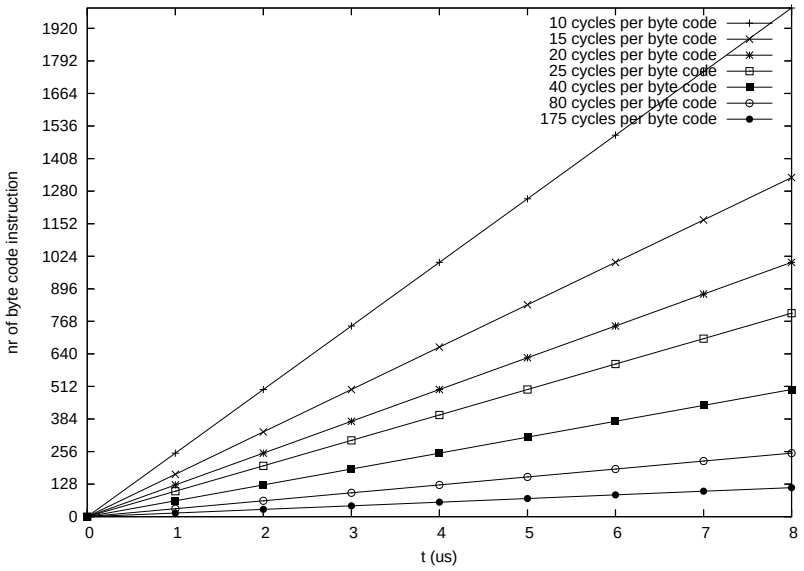

Figure 6.2: The fewest number of bytecode instructions executed per μs on a 2.5G-Hz processor. Each line represents a specific maximum number of clock cycles needed to execute any byte code instruction

while `NanoAmp Solutions` report 20 clock cycles using an ARM CPU [11]. Figure 6.2 plots the fewest number of byte code instructions that can execute during a specific time interval. Each line in the graph was generated using a specific maximum number of clock cycles per byte code instruction.

In Section 4.5, we showed that, for the tested jvm98 SPEC benchmarks, the maximum allocation rate computed using a window size larger than 256 clock cycles will not be significantly less conservative than using the rate computed for a window of size 256 to estimate larger windows. As previously mentioned, the experimentally determined context switching time was 4.022 μs. Figure 6.2 shows that if the most expensive byte code instructions need less than 40 clock cycles to execute, then any allocation in an instruction window of size 256 or less will be shifted out of the window by the instructions executed during the context switch.

This means that if the assumption that the byte code instruction invoke needs less than 40 clock cycles to execute is valid, then context switching between threads will not alter the thread ignorant computed static upper bound for the window sizes we are considering. At issue then, is whether or not it is reasonable to make this assumption. In the literature, we have not encountered any cases, other than the thesis work of Martin Schoeberl [18], where this assumption would not hold. However,

as we already pointed out, the data reported in his thesis was obtained using a Cyclone FPGA. Therefore, we feel that the previously mentioned ratio's of 15 - 20 clock cycles per byte code instruction [25, 11], are more in line with what we can expect. This puts the number of clock cycles executed by the invoke instruction well below 40, which is why we feel we can make this assumption.

Under the assumptions specified in this chapter, the allocation rate of multi-threaded mutator programs can be properly bounded using our framework. If these assumptions are too constraining for a particular application, then it may still be possible to bind the allocation rate using *safe points* (as in Jikes RVM) so that threads are interrupted only at predetermined points. This avenue has not been explored in this thesis. One could also imagine a thread scheduler that is memory allocation aware, which could enforce particular interleavings to optimize time vs. space tradeoff in the application.

Chapter 7

Conclusions and Future Work

We have studied the allocation rate of mutator programs dynamically, and have determined that there is great variation in the memory allocation rates they exhibit. Therefore, the choice of window size for the collector is an important issue. Longer windows tend to decrease the significance of allocation spikes, lowering the apparent maximum allocation rate. This in turn can reduce the storage needed in reserve during a collection cycle. However, longer windows also increase the storage requirements once the maximum allocation rate is fixed. The tension between longer and shorter windows is application-specific and merits further study. Moreover, the variance in maximum allocation rate throughout a program indicates the potential of an adaptive approach based on phases of the application's allocation behavior.

We have provided a framework for static determination of maximum allocation rates and have applied this framework to some Java benchmarks. We have demonstrated that for our benchmarks, our statically-determined allocation rate is within a constant factor of the observed allocation rate. Whether or not this constant factor constitutes a reasonable upper bound must be evaluated on a case by case basis. The size of this factor will have an affect on the memory footprint and the MMU [8] of the application. If an closer upper bound is needed a more careful interprocedural analysis could potentially decrease the magnitude of this factor. In either case, our statically computed upper bound offers an improvement over the current technique where, in the worst-case, the user can do little but guess a upper bound on the allocation rate. However, before using our system to deploy a garbage collector in a real-time environment, further study on the affect of converting from bytes per instruction to bytes per unit time, is needed.

Admittedly, our benchmarks are not real-time benchmarks, but one reason for a lack of real-time Java code is the effort required to use the Real-Time Specification for Java^{TM} (RTSJ). To date, the only substantial RTSJ code is under development at NASA and they are not releasing that code yet.

Our implementation can be improved in a number of ways. One idea is to investigate path-sensitive approaches, including a *meet-over-all-valid-paths* approach [17]. We would like to investigate static approaches to bounding pointer density for real-time programs. As many realistic programs do not maintain a constant rate of allocation at runtime [14], we plan to adapt our approach to handle variable allocation rates. This is especially important for programs in which not all methods are called by real-time threads. The maximum allocation rate within the execution of real-time threads is the relevant statistic for the real-time collector.

References

[1] David F. Bacon, Perry Cheng, and V. T. Rajan. Controlling fragmentation and space consumption in the Metronome, a real-time garbage collector for Java. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems*. ACM Press, 2003.

[2] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 285–298. ACM Press, 2003.

[3] Dirk Baumer, Dirk Riehle, Wolf Siberski, and Martina Wulf. Role Object. In Brian Foote, Neil Harrison, and Hans Rohnert, editors, *Pattern Languages of Program Design 4*. Addison-Wesley, Reading, Massachusetts, 2000.

[4] Greg Bollella, James Gosling, Ben Brosgol, Peter Dibble, Steve Furr, David Hardin, and Mark Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[5] Dante J. Cannarozzi, Michael P. Plezbert, and Ron K. Cytron. Contaminated garbage collection. *Programming Language Design and Implementation*, pages 264–273, 2000.

[6] Lisa Carnahan and Marcus Ruark. Requirements for real-time extensions for the Java platform (final draft). Technical report, NIST, 1999.

[7] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

[8] Perry Cheng and Guy Belloch. A parallel, real-time garbage collector. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 125–136, 2001.

[9] Morgan Deters. Applicability of Range Propegation to Memory Analyses. Technical report, Washington University in St. Louis, 2005. To Appear.

[10] Robert R. Fenichel and Jerome C. Yochelson. A LISP garbage collector for virtual-memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[11] NanoAmp Solutions Inc. MOCA-J Technical Brochure. www.nanoamp.com/MOCA-J_Tech_Brochure.pdf.

[12] Martin R. Linenweber. A study in Java bytecode engineering with PCESjava. Master's thesis, Washington University in St. Louis, 2003.

[13] C.L. Liu and J.W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *JACM*, 20(1):46–61, January 1973.

[14] Tobias Mann and Ron K. Cytron. Automatic Determination of Factors for Real-Time Garbage Collection. In *Washington University technical report #WUCS-04-45*, St. Louis, Missouri, 2004.

[15] Steven S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann Publishers Inc., 1997.

[16] Kelvin Nilsen. Issues in the design and implementation of real-time Java. *Java Developer's Journal*, 1(1):44, 1996.

[17] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, 1995.

[18] Martin. Schoeberl. *JOP Java Optimized Processor.* PhD thesis, Technical University of Vienna, Vienna, Austria, September 2000. Available through http://www.jopdesign.com.

[19] Edsger W. Dijkstra Leslie Lamport A.J. Martain C.S. Scholten and E.F.M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.

[20] Ran Shaham, Elliot Kolodner, and Mooly Sagiv. Heap profiling for space-efficient Java. *ACM SIGPLAN Notices*, 36(5):104–113, May 2001.

[21] Fridtjof Siebert. Eliminating external fragmentation in a non-moving garbage collector for Java. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 9–17, 2000.

[22] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, February 1997.

[23] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, 1991.

[24] Paul R. Wilson. Uniprocessor garbage collection techniques (Long Version). Submitted to ACM Computing Surveys, 1994.

[25] Mao Zhi-gang, Wang Tao, and Ye Yi-zheng. Designing JCVM in Hardware. www.ifip.or.at/con2000/icda2000/icda-2-5.pdf.

Vita

Tobias Mann

Date of Birth March 1, 1978

Place of Birth Stockholm, Sweden

Degrees B.Sc. Computer Science, 2003,
from Pacific Lutheran University, Tacoma WA.

Publications Tobias Mann and Ron K. Cytron. "Automatic Determination of Factors for Real-Time Garbage Collection" *Washington University Technical Report,* August 2004.

Tobias Mann, Morgan Deters, Rob LeGrand and Ron K. Cytron. "Static Determination of Allocation Rates to Support Real-Time Garbage Collection" *LCTES Conference* June 2005. To Appear

May 2005