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

Tobias Mann Morgan Deters Rob LeGrand Ron K. Cytron

Department of Computer Science and Engineering
Washington University in St. Louis
{tmann, mdeters, legrand, cytron}@cs.wustl.edu

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 for storage. 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 paper we present a static analysis to bound conservatively an application's *allocation rate*. The analysis is based on a data flow framework that requires interprocedural evaluation. We present the framework and results from analyzing some Java benchmarks. Because static analysis is necessarily conservative, we also present measurements of our benchmarks' actual allocation rates.

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.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Memory management (garbage collection)

General Terms Algorithms, Languages, Measurement

Keywords Real-time Garbage Collection, Static Analysis, Allocation Rate

* This report is an extended version of [22]. The work was sponsored by DARPA under contract F33615–00–C–1697 and by the AFRL under contract PC Z40779.

1. Introduction

There is considerable interest in Java as a software development vehicle for real-time and embedded applications. There are several reasons for this, some of which are listed by the **National Institute of Standards and Technology** (NIST) [7]:

- 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 like the **Real-Time Specification for Java** (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) automatic deallocation reasonably predictable. Included as a core requirement in the NIST specification for a real-time Java is the following [7]:

> 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 garbage collector suitable for real-time applications must be able to collect sufficient storage so that the application does not fail for lack of storage, and must do so without denying the application reasonable use of the CPU(s). Currently, there are two approaches to satisfying that requirement:

Avoid traditional garbage collection. Specialized storage-allocation structures can be introduced to obviate the need for traditional garbage collection. For example, the RTSJ introduces *scopes* in which objects can be allocated. Hard real-time threads are allowed

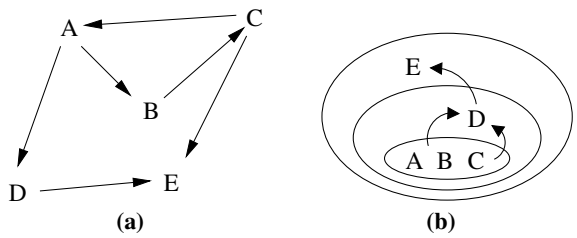

Figure 1. Use of scoped storage. **(a)** shows the references between objects at one point in time; **(b)** shows a legal scope assignment. If E tried to reference D, the program would fail given this scope assignment.

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. When the last thread exits a scope, all of the storage described by the scope is reclaimed *en masse*.

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 1: objects in one scope may not refer to objects in another scope that might be shorter-lived.[2]
- Scopes are a specialized form of *regions* [33], 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 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.
- Scoped memory areas make the construction of complex software highly unwieldy. Software designs utilizing scopes often involve *ad hoc* user-level mechanisms for reusing scoped storage (and the scopes themselves), use threads not for concurrent computation but to keep scopes from being reclaimed, or suffer from a high degree of complexity in thread intercommunication. A useful discussion of software development using the RTSJ's scoped memory mechanism can be found in [26].
- It is *undecidable* whether a program adheres to the various rules associated with scoped memory. This can be shown by reduction from the halting problem [14] as follows. Suppose that we simulate a regular Java program in a universal Turing machine M. Because a regular Java program does not use scopes, we can arrange M so that it executes an illegal scoped memory reference just after the simulated program halts. Thus, M suffers an illegal scope reference if and only if the simulated program halts. Thus, deciding if M executes an illegal scoped memory reference also decides whether the simulated program halts.

Traditional garbage collection can also be avoided by using techniques such as reference counting [36] and contaminated garbage collection [6], but those collectors are inexact and could thus suffer from the same leakage problems as scopes.

Use a real-time garbage collector. A real-time garbage collector, such as Metronome [3] or Perc [25], is assigned the responsibility of detecting and collecting dead storage. The application, often called the *mutator* in the literature, need not change, but the application's *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 [3, 2] that collectors such as Metronome operate correctly *if* the mutator's behavior is properly described. Fortunately, a mutator's relevant behavior can be distilled into a few statistics.

At issue is whether a programmer can reliably provide such 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 for storage. One could try to overspecify these statistics, but this is still an educated guess on the part of the developer. Also, overspecification of the 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 paper, we present a static method (data flow framework) for determining a bound on the **Maximum Mutator Allocation Rate** (MMAR) of a program. This kind of analysis is done at compile time and it is crucial to the correct operation of a real-time collector.

Our paper is organized as follows. Section 2 provides some background to our problem domain. In Section 3 we summarize the statistics required by collectors such as Metronome to guarantee correct operation and identify the particular property that is the target of this work. For real-time programs, correctness here implies never running out of storage and never starving the mutator of access to the CPU for an unreasonable or unbounded amount of time. Section 4 presents our solution. This is followed by a discussion of multi-threading in Section 5, and experiments determining the MMAR are presented in Section 6. Because static approaches are necessarily conservative, we also report on our benchmarks' MMAR from actual executions. Section 7 discusses related work. Finally, Section 8 concludes. An appendix provides additional details about a range propagation implementation that we have developed to support our MMAR analysis.

2. Background

Before going into details on what a real-time garbage collector needs to know about a mutator program we will provide some background information. 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. Readers familiar with these areas can skip or skim through this section.

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

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

[2] This constraint on object referencing behavior is intended to avoid the manufacture of dangling pointers. Together with other constraints that the RTSJ makes, it ensures that no external pointers exist to scoped objects when their scope is reclaimed.

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 Garbage Collection Techniques

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 [36].

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 1 this technique is inexact, and thus may suffer from memory leakage problems.

The problem of leaking memory makes a reference counted system unsuitable for deployment in a real-time environment. 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.

2.2.1 Mark-Sweep Collection

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 [36]:

Fragmentation: If the mutator allocates objects with varying sizes, the heap will likely become fragmented, with the adverse effect that the allocator may fail to allocate memory for an object even when enough space is available. 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: Objects retain their place in memory throughout their lifetime. This means that when a collection cycle finishes live objects will be interspersed in memory with the free space gathered from the collected objects. New objects are then allocated in these free spaces. 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.

2.2.2 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 region of memory. The remainder of the heap is then known to be garbage. The most common copy collector is the *semispace* collector [12] using the Cheney copying traversal algorithm [8]. 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 over mark-sweep is that it avoids fragmentation of the heap and locality of reference may be improved because it avoids the problem caused when live objects retain their place. However, the as with mark-sweep there is an inherent limit on efficiency because all live objects must be moved. Copying collectors also suffers from the following disadvantages.

Increased Memory Footprint: The mutator only has access to fromspace, reserving tospace for use by the garbage collector. This means that the amount of heap memory available to the mutator is cut in half, effectively doubling the memory footprint of the application.

Read Barrier: When implemented incrementally (see Section 2.3.1) a copying collector must employ a read-barrier (see Section 2.3.2, which adds additional cost.

2.2.3 Hybrid

Some collectors, such as the Metronome [3] and many generational garbage collecting systems [36], that utilize both the mark-sweep and copying techniques. In the case of the Metronome the bulk of the collection work is done using the mark-sweep technique and copying is only used to defragment the heap as needed. Generational collectors on the other hand can be seen as being mostly a copying collector where objects that are found to be live are copied, or tenured, to a different region of memory. Each region, holds a generation of objects, thus the name of the technique, where a generation is defined by how many collections an object has "survived". The region of memory holding the oldest generation is often collected using a mark-sweep technique. For more details on generational collection and other garbage collection techniques see Wilson's thorough survey paper [36].

2.3 Real-Time Garbage Collection

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. It is impossible for a real-time scheduling routine to determine whether or not the mutator program will be able to meet its temporal constraints if the garbage collector can take control over the CPU at anytime during execution and keep control for an unbounded period of time. A fine-grained incremental garbage collector, which will interleave the collectors work with that of the mutator, is needed. All the techniques discussed in Section 2.2 can be made incremental.

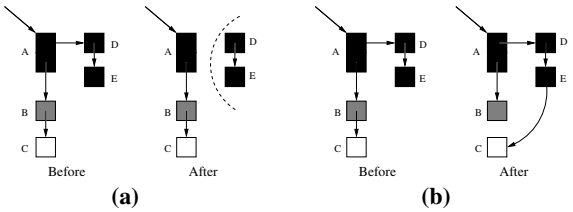

Figure 2. Mutating the Heap. **(a)** shows how the mutator may alter the heap to create floating garbage, objects D and E; **(b)** shows how the mutator may alter the heap so that the collector will reclaim an object that is still live, object C.

2.3.1 Incremental

Using an incremental technique for garbage collection is the only viable approach for a system that wants to claim real-time performance. However, it does add additional complexity to the collector. If the mutator is allowed to alter (mutate) the heap while the garbage collector 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 [11]. In this scheme, memory objects are classified using the three colors white, grey, and black. For the remainder of this section we focus on what happens during the collection cycle.

White: White means that the memory object has not been visited by the garbage collector. 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, make up the *floating garbage* mentioned in Section 1. 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 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 illustrates the problems here discussed.

Stated more formally, any correct incremental garbage collector 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 garbage collector ever completes its collection cycle. Next, we will discuss two techniques that address this problem.

2.3.2 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 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 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 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, as mentioned in Section 2.2.2 copying collectors, need to use a read barrier to ensure that the mutator only sees references to valid copies of objects.

2.4 How a Real-Time Garbage Collector Affects the Mutator

To quantify the effect that the real-time garbage collector has on the mutator programs ability to meet its temporal requirements we let $\Delta\tau$ symbolize the total amount of time given to the mutator during a collection cycle.

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. [3], 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 [9] 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 $\Delta\tau$. Equations 1 and 2 (taken from [3]) shows how MMU can be computed assuming a time based scheduling algorithm where Q_T and C_T are the mutator and garbage collector time quanta respectively. A time quantum is the smallest amount of non-preemptive execution time that is guaranteed.

$$\text{MMU}(\Delta\tau) = \frac{Q_T \cdot \left\lfloor \frac{\Delta\tau}{Q_T + C_T} \right\rfloor + x}{\Delta\tau} \tag{1}$$

$\Delta\tau$ is the time window for which MMU is calculated and x is the remaining partial mutator quantum, defined in Equation 2.

$$x = \max\left(0, \Delta\tau - (Q_T + C_T) \cdot \left\lfloor \frac{\Delta\tau}{Q_T + C_T} \right\rfloor - C_T\right) \quad (2)$$

If the mutator, prior to runtime, is guaranteed that it will always get a certain fraction of the CPU over a small window of time $(Q_T + C_T)$, then static scheduling analysis [19, 16] can determine whether or not the system is feasible.

3. Mutator Statistics for Real-Time Collection

Now we return to the discussion of what statisticall properties of the mutator program that a real-time collector must have access to. While in this paper we focus primarily on the Metronome real-time collector [3], all tracing, real-time collectors function similarly, in the sense that the following statistics are necessary:

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* [29, 6] 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. Maxlive is needed to put an upper bound on the amount of work that the collector may need to perform during a collection cycle. As mentioned in Section 2.2 the work done by the collector is proportional to the amount of live memory

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*, 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.

MMAR: A real-time collector cannot suspend a mutator indefinitely. Thus, the work of a traditional collection cycle is 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 thus continue to allocate objects. Some of those objects may become dead during the cycle. Once dead, such objects do not count toward *maxlive*, but the real-time collectors cannot collect them in the current cycle. Such objects are called "floating garbage" in the literature.

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. A bound on floating garbage is computed as the product of the mutator's execution time during an entire collection cycle and the maximum rate at which the mutator can allocate storage (MMAR). That product is influenced by the mutator in terms of its MMAR, but the fraction of time given to the mutator is the key parameter used by the collector to guarantee pacing with the mutator.

Underestimating the MMAR could cause the program to fail because of insufficient storage budgeted for the collection cycle—a situation unacceptable for real-time applications. Overestimating the rate 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.[3]

Thus, MMAR is the most influential statistic but also the one most difficult for a developer to estimate. In this paper we present static analysis that accurately bounds the MMAR. This analysis assumes that the whole program is available. While it's true that Java dynamically loads classes, real-time allocators need a whole-program conservative estimate of the MMAR. Short of a guess, the whole program must be available to a human or to our analysis to make the estimate possible. We further assume that only classes that are known to our system can be instantiated using reflection.

4. Static Determination of Allocation Rates

A common technique for analyzing a static property of a program is to formulate the problem as a data flow framework [23]. To this end a control flow graph representing the program is constructed. Below we show an example C program, and in Figure 3 we show the control flow graphs for the two methods.

```
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: the set of elements, A, forming the domain of the problem; top, $\top$,

[3] in the sense that rate-monotonic analysis cannot guarantee that all deadlines are met.

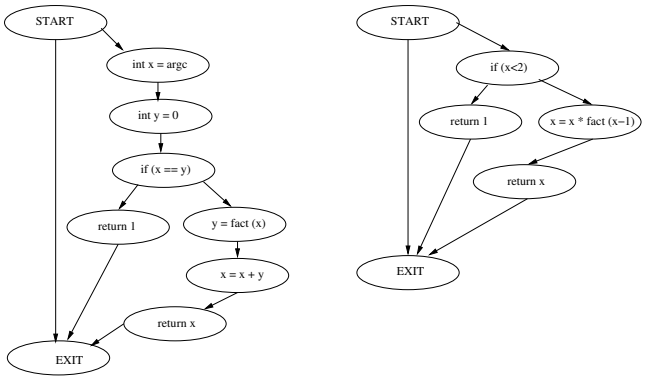

Figure 3. Intraprocedural data flow framework generated from our example program.

and bottom, $\perp$, representing the best and worst possible solutions to the problem; a reflexive partial order operator, $\preceq$, which is used to compare different solutions to each other; and the meet operator, $\wedge$, which combines solutions.

The last element of the DF triple is the set of transfer functions, F. A transfer function f maps the combined input to a node, $n.in$, to its output $n.out$.

4.1 Solutions using Data Flow Frameworks

As described above, we aim to compute the maximum rate at which a mutator consumes memory. Our framework uses a window that essentially slides over a program's instructions, and we compute the MMAR seen in that window. The window could be expressed using units of time, but for our purposes it was more convenient to size the window with respect to a program's Java bytecode 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.

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. At this point in our paper, we assume that all allocations are of unit size. We take into account actual object size in Section 4.3.

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 (leftmost) bit represents the most recent instruction.

4.1.1 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, where W is the size of the window:

- $A = \{0, 1\}^W$
- $\top = \langle \underbrace{0, 0, 0, \ldots, 0}_{W} \rangle$
- $\perp = \langle \underbrace{1, 1, 1, \ldots, 1}_{W} \rangle$

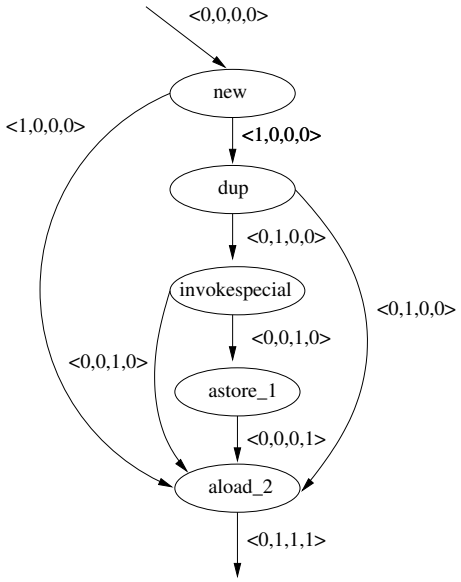

Figure 4. The control flow graph representations of Java's `try...catch` blocks (and `try...catch...finally` blocks) contain many edges to exception-handling code (and code in `finally` blocks, respectively). This can fill a window unnecessarily with allocations when using the naïve framework of Section 4.1.1; in this example, the final node reports three consecutive allocations in its window, when only one allocation can really occur along any execution path.

- $\wedge$ 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; $\perp$ 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 ago, respectively. Using the above meet, $\langle 0, 0, 1, 0 \rangle \wedge \langle 0, 1, 0, 0 \rangle = \langle 0, 1, 1, 0 \rangle$. This resulting vector assumes that allocations have occurred both three *and* two instructions ago. Clearly all information has been retained and thus the result can never be better than the input vectors. However, as we shall see, 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 the node encompasses the instruction represented by the node. 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 and lost.

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.

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 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, 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 of its input vectors.

When all incoming vectors only contain one 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 have more than one allocation? Clearly, the result should contain the same number of allocations as the vector with the most allocations, but where should 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 can be seen as better than logical bitwise *or* because it does not increase the number of allocations.

Nonetheless, this meet function produces a solution that reflects that the last instruction it encountered was an allocation when none of its input vectors reflected that fact. Also, it violates the important meet property of idempotency $a \wedge a = a$ (when a has at least one allocation but none in the most recent instruction). A better idea is to have the meet place allocations in the positions of the most significant set bits in its input vectors: $\langle 0,1,0,0,1\rangle \wedge \langle 0,0,1,1,0\rangle = \langle 0,1,1,0,0\rangle$. In other words, note the number of allocations n in the input vector with the most allocations, use the bitwise-or meet, then keep only the n leftmost ones in the result. This *limited-bit-or* 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. Unfortunately, it leads to a sub-optimal reflexive partial order, as seen in the next section.

4.1.2 Better Framework

Thus far, we have been constructing the framework's meet lattice by defining the meet function directly and defining the reflexive partial order in terms of it: $a \preceq b$ if and only if $a \wedge b = a$. Each meet function we proposed gave results that were no "better" than the inputs in an obvious way, but it was never clear that we had found the least conservative reflexive partial order that works for our application. Now we construct a better framework by motivating a reflexive partial order directly from our application and using it to find a better meet function.

Essentially, since we want to find the tightest possible bound on allocation rate in every given window, we want $a \preceq b$ to be true when a has at least as many allocations as b and will no matter what sequence of transfer functions is applied to both a and b; put another way, b will give at least as good a result as a regardless of the future path through the program graph. (Remember that in our bit-vector framework there are only two transfer functions: f_0, which shifts in a 0 at a non-allocating node, and f_1, which shifts in a 1 at an allocating node.) For example, using the limited-bit-or meet above,

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

and so $\langle 1,0,1\rangle \not\preceq \langle 0,1,0\rangle$. But we want $\langle 1,0,1\rangle \preceq \langle 0,1,0\rangle$ since $\langle 1,0,1\rangle$ has more allocations and will never have fewer no matter what sequence of transfer functions is applied to both. Applying f_1 and then f_0 to both, for example, will result in $\langle 0,1,1\rangle$ and $\langle 0,1,0\rangle$. To see that this holds for any sequence of transfer functions, notice that every prefix of $\langle 1,0,1\rangle$ has at least as many allocations as the equal-length prefix of $\langle 0,1,0\rangle$. So our application motivates the following definition, where a_i denotes the ith bit of a:

DEFINITION 1. *$a \preceq b$ if and only if for all $1 \leq i \leq W$, the i-bit prefix of a has at least as many ones as the i-bit prefix of b, i.e., $\sum_{j=1}^{i} a_j \geq \sum_{j=1}^{i} b_j$.*

This reflexive partial order leads to the following meet function, which calculates $c = a \wedge b$ from inputs a and b:

for $i = 1$ **to** W **do**
  if $\max\left(\sum_{j=1}^{i} a_j, \sum_{j=1}^{i} b_j\right) > \sum_{j=1}^{i-1} c_j$ **then**
    $c_i = 1$
  else
    $c_i = 0$

The **if** statement assigns a 1 to c_i only when it is necessary to prevent $c \not\preceq a$ or $c \not\preceq b$ according to definition 1. Induction can be used to prove that this meet function gives the best possible result while satisfying $a \preceq b \iff a \wedge b = a$ for all a and b; see Section 4.3.2 for a more general proof. So the overall problem motivates the transfer functions in an obvious way, those transfer functions motivate a reflexive partial order, and a meet function is constructed to impose that reflexive partial order.

Perhaps a more intuitive way to compute this meet function 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 c by taking the bitwise *or* of a_i and b_i. If the result $c_i = 1$, 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.

4.2 Framework Evaluation

Recall that in the introduction to this section we presented the intraprocedural data flow graph for an example C program (Figure 3). Forming an interprocedural solution from this graph is conceptually trivial. As show in Figure 5 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 graph involves connecting the call node in A with the start node in B, and the exit node in B with the successors of 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 also have to be reevaluated, and so forth. In our detailed description of the algorithm 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. [27], to specify our interprocedural data flow framework formally:

- $G^* = (N^*, E^*)$
- $P^* =$ the set of all procedures p represented in G^*

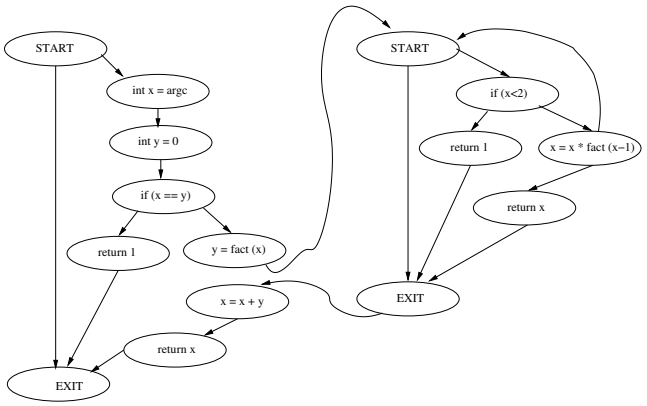

Figure 5. Interprocedural data flow framework generated from our example program from Figure 3.

- $N^* = \bigcup_{p \in P^*} N_p$
- $E^* = E^0 \bigcup E^1$
- $E^0 = \bigcup_{p \in P^*} E_p^0$ is the set of intraprocedural control-flow edges
- $E^1 = \bigcup_{p \in P^*} E_p^1$ is the set of procedure call and procedure return edges.

We also define the functions:

- $calledBy(p, G^*) = N'$ 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 4.1.2.

The basic algorithm for calculating the interprocedural MMAR 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 4. The most important steps of the algorithm are lines 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 evaluated. 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 6 confirm that we obtain reasonable solutions with this approximation.

4.3 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 [3]. However, we are obligated to compute the MMAR, 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 fields of type `double` or `long` which occupy 8 bytes. We recognize that individual Java implementations may vary, however, we feel that these are reasonable assumptions because all Java implementations do something very similar to this. At an allocation, we compute each object's size using the Java reflection package.

4.3.1 Statically-Bounded Array Allocations

The size of some arrays can be statically bounded by a constant in Java, but such an analysis is slightly complicated. Consider the following example, where φ is some boolean condition that is not known statically:

```
Array Allocation
1    int size = 10;
2    if (φ)
3       size = 100;
4    MyObj[ ] a = new MyObj[size];
```

Static determination of the size of statically-allocated arrays is itself a data flow problem. Typically, constant propagation is used for this purpose. However, for our purposes we require a bound on array size but are not interested in the actual array size; accordingly, we implemented an intraprocedural range propagation analysis instead, which tracks value ranges for variables even when they are nonconstant [13]. 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.

We have implemented such an analysis for bounding the size of array allocations when possible, and we have incorporated this analysis into our static analysis for the MMAR. Additional details of our range propagation implementation can be found in Appendix A. In Section 4.3.3, we discuss array allocations that we cannot statically bound. For now, we assume we have a static bound on each allocation, whether of object or array type.

4.3.2 A Data Flow Framework with Allocation Size

As we must account 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 6:

- $A = \{0, 1, 2, \ldots M\}^W$ where M is the maximum number of bytes that the allocator can allocate at one time
- $\top = \langle \underbrace{0, 0, 0, \ldots, 0}_{W} \rangle$
- $\bot = \langle \underbrace{M, M, M, \ldots, M}_{W} \rangle$
- $\wedge$ is illustrated in Figure 6 and formally defined below
- $a \preceq b$ holds if and only if $a \wedge b = a$

In Section 4.1.2, 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. Figure 6 shows how the meet function works when all allocations are not equal. 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 6, 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 earlier. 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.

This "borrowing" meet algorithm is intuitive, but using a more formal algorithm makes it easier to show that it is the best possible meet. First we define the reflexive partial order for the allocation-size framework similarly to the one in the bit-vector framework from Section 4.1.2, but instead of looking at number of allocations in a given window, we look at total allocation size. So we want $a \preceq b$ to be true when a has at least as large a total allocation size as b and will no matter what sequence of transfer functions is applied to both a and b, so that b will give at least as good a result as a regardless of the future path through the program graph. Thus the following definition, where a_i denotes the ith entry of a:

DEFINITION 2. *$a \preceq b$ if and only if for all $1 \leq i \leq W$, the i-entry prefix of a has at least as great an allocation total as the i-entry prefix of b, i.e., $\sum_{j=1}^{i} a_j \geq \sum_{j=1}^{i} b_j$.*

So, for example, $\langle 4, 4, 16, 4, 0, 4, 8 \rangle \preceq \langle 0, 8, 4, 8, 4, 8, 4 \rangle$. This reflexive partial order leads to the following meet function, which calculates $c = a \wedge b$ from inputs a and b:

for $i = 1$ **to** W **do**
$c_i = \max\left(\sum_{j=1}^{i} a_j, \sum_{j=1}^{i} b_j\right) - \sum_{j=1}^{i-1} c_j$

At step i the assignment statement assigns to c_i the smallest integer that is necessary to prevent $c \not\preceq a$ or $c \not\preceq b$ according to definition 2. Induction can be used to prove that this meet function

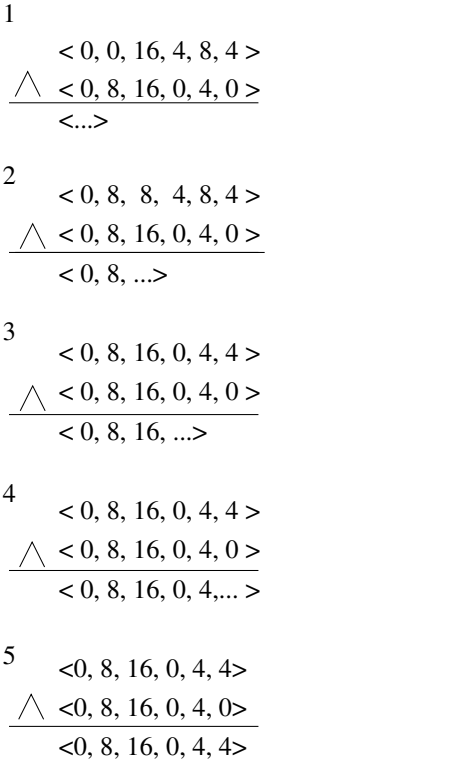

Figure 6. Computing meet when accounting for object allocation size.

is the only one possible that, for all a and b, satisfies $a \wedge b \preceq a$ and $a \wedge b \preceq b$ and for which no d exists such that $d \preceq a$, $d \preceq b$ and $d \not\preceq a \wedge b$:

First, say that the meet algorithm is on the last, Wth, step; c_1 through c_{W-1} have been calculated in some way. If c_W were assigned a value less than $\max\left(\sum_{j=1}^{W} a_j, \sum_{j=1}^{W} b_j\right) - \sum_{j=1}^{W-1} c_j$, then it would be true that

$$\max\left(\sum_{j=1}^{W} a_j, \sum_{j=1}^{W} b_j\right) > \sum_{j=1}^{W} c_j$$

and so either $c \not\preceq a$ or $c \not\preceq b$ by the above definition of $\preceq$. On the other hand, say c_W were assigned a value greater than $\max\left(\sum_{j=1}^{W} a_j, \sum_{j=1}^{W} b_j\right) - \sum_{j=1}^{W-1} c_j$. Then letting $d_j = c_j$ for $1 \leq j \leq W-1$ and $d_W = \max\left(\sum_{j=1}^{W} a_j, \sum_{j=1}^{W} b_j\right) - \sum_{j=1}^{W-1} c_j$ will give a d such that $d \preceq a$, $d \preceq b$ and $d \not\preceq c$, this last because $d_W < c_W$ and so $\sum_{j=1}^{W} d_j \not\geq \sum_{j=1}^{W} c_j$. Therefore $\max\left(\sum_{j=1}^{W} a_j, \sum_{j=1}^{W} b_j\right) - \sum_{j=1}^{W-1} c_j$ is the correct value to assign to c_W no matter what c_1 through c_{W-1} are.

Now assume that the meet algorithm is on the ith step; c_1 through c_{i-1} have been calculated in some way and c_{i+1} through c_W will be calculated according to the above meet algorithm. If c_i were assigned a value less than $\max\left(\sum_{j=1}^{i} a_j, \sum_{j=1}^{i} b_j\right) -$

$\sum_{j=1}^{i-1} c_j$, then it would be true that

$$\max\left(\sum_{j=1}^{i} a_j, \sum_{j=1}^{i} b_j\right) > \sum_{j=1}^{i} c_j$$

and so either $c \not\preceq a$ or $c \not\preceq b$ by the above definition of $\preceq$. On the other hand, say c_i were assigned a value greater than $\max\left(\sum_{j=1}^{i} a_j, \sum_{j=1}^{i} b_j\right) - \sum_{j=1}^{i-1} c_j$. Then letting $d_j = c_j$ for $1 \leq j \leq i-1$ and $d_i = \max\left(\sum_{j=1}^{i} a_j, \sum_{j=1}^{i} b_j\right) - \sum_{j=1}^{i-1} c_j$ and assigning values to d_{i+1} through d_W according to the algorithm above will give a d such that $d \preceq a$, $d \preceq b$ and $d \not\preceq c$, this last because $d_i < c_i$ and so $\sum_{j=1}^{i} d_j \not\succeq \sum_{j=1}^{i} c_j$. (In fact, it will be the case that $c \prec d$.) Therefore $\max\left(\sum_{j=1}^{i} a_j, \sum_{j=1}^{i} b_j\right) - \sum_{j=1}^{i-1} c_j$ is the correct value to assign to c_i no matter what c_1 through c_{i-1} are. It follows by induction on i that the above meet algorithm gives the best possible result that satisfies $a \wedge b \preceq a$ and $a \wedge b \preceq b$.

4.3.3 Unbounded Arrays and Arraylets

Bacon et al. [3] 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 [30] uses a similar idea and represents large arrays as a tree structure of fixed-size blocks.

As mentioned in Section 4.1, thus far 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 instruction is followed by x number of assignments, where x is the number of bytes being allocated divided by 4. However, 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 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. [3] as $C = 2$KB. If we assume that our window size, W, is smaller than $\frac{2KB}{4B}$ then we can bound the allocation rate behavior of all array allocations of unknown size.

Figure 7 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, $W < \frac{2KB}{4B}$. This means that when the next arraylet is allocated, the allocation for the first one will have fallen out 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 values of r, with its subsequent initialization instructions. This is the output from the dummy node in Figure 7. Taking the meet of all the output vectors from the dummy node gives us the vector $\langle 0, 4, \ldots, 4 \rangle$.

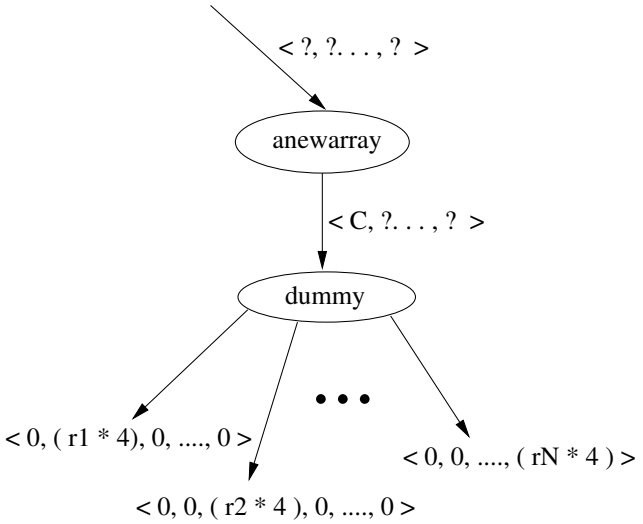

Figure 7. An array allocation, as represented in the control flow graph. ('?' represents any instruction.)

We have placed an upper bound on the MMAR 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-size arraylets. Similarly, if the allocator allocates large objects as a sequence of smaller allocations, this technique can be used to estimate the 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 7 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 MMAR. We would be forced to assume that $C = M$ in Figure 7. Since the array actually allocated may be small, we would still need to use the dummy node and meet all of its output vectors.

4.4 Analysis of the Framework

To guarantee that our data flow framework converges we must show our framework is monotone:

$$(\forall f \in F)(\forall x, y) \quad x \preceq y \longrightarrow f(x) \preceq f(y)$$

A node's transfer function shifts in the amount of memory allocated at each instruction (0 for a non-allocating instruction). The shifts occur at the right hand side of a bit-vector, while the comparison ($\preceq$) is based on the leftmost bits scanning to the right. Thus, no $f \in F$ can output a better solution given a worse input.

We must show our meet operator satisfies the following rules for all $a, b \in A$:

1. $a \wedge a = a$
2. $a \wedge b \preceq a$
3. $a \wedge b \preceq b$
4. $a \wedge \top = a$

5. $a \wedge \perp = \perp$

It is easy to see from our meet algorithm that the meet satisfies 1, 4 and 5 (note that $a_i \geq \top_i$ and $a_i \leq \perp_i$ for all a and i). Also, our meet algorithm was carefully designed to satisfy properties 2 and 3 with our definition of $\preceq$; see Section 4.3.2.

As a result of the above, a data flow solution will converge such that the MMAR 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. If we have a *distributive* framework, then the (intraprocedural) solution we compute is the best possible static solution to our problem. In a distributive framework,

$$(\forall f \in F)(\forall x, y) \quad f(x \wedge y) = f(x) \wedge f(y)$$

Consider two generic vectors a and b, containing n elements. The effect of f 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 $f(y)$. All $f(y)$ does is a simple right shift, thus $c'_i = c_{i-1}$ for $1 < i \leq n$. $f(c)$ moves c_{i-1} to c_i for all $1 < i \leq n$, and we already know that $c'_1 = (f(c))_1$. Thus $f(c) = c'$, so our framework is distributive and our solution is no worse than the **meet-over-all-paths** (MOP) solution.

A framework is *rapid* if and only if

$$(\forall a \in A)(\forall f \in F) \quad a \wedge f(\top) \preceq f(a)$$

It would be ideal if our framework were rapid, because we would be guaranteed to converge upon a solution more quickly. However, our framework is not rapid, since each trip around a loop can shift in another allocation. More precisely, a vector a and transfer function f can be found that violate the above definition: If $W = 2$, $a = \langle M, 0\rangle$ and f is the transfer function that shifts in an M (indicating an allocation of maximum size), then

$$a \wedge f(\top) = \langle M, 0\rangle \wedge f(\langle 0, 0\rangle) = \langle M, 0\rangle \wedge \langle M, 0\rangle = \langle M, 0\rangle$$

but

$$f(a) = f(\langle M, 0\rangle) = \langle M, M\rangle$$

and $\langle M, 0\rangle \not\preceq \langle M, M\rangle$, so our framework cannot be rapid.

Similarly, it is not *fast*. A fast framework is one for which

$$(\forall a \in A)(\forall f \in F) \quad a \wedge f(a) \preceq f(f(a))$$

But if $a = \langle 0, M\rangle$ and f is the transfer function that shifts in an M, then

$$a \wedge f(a) = \langle 0, M\rangle \wedge f(\langle 0, M\rangle) = \langle 0, M\rangle \wedge \langle M, 0\rangle = \langle M, 0\rangle$$

but

$$f(f(a)) = f(f(\langle 0, M\rangle)) = f(\langle M, 0\rangle) = \langle M, M\rangle$$

and $\langle M, 0\rangle \not\preceq \langle M, M\rangle$, so our framework cannot be fast.

5. Multithreaded Environments

Until now we have purposely avoided the issue of multithreaded mutator programs, which will will be the topic of this section.[4] Multithreading is an important issue because most programs operating under real-time constraints rely on this capability. However, threading presents static analysis with the problem of predicting how context switches can affect properties of the executing program.

5.1 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 allocates 4 bytes of memory. If context switches can happen at any time during the execution of the mutator, assuming zero overhead in terms of instructions executed for the switch, our process for statically determining a bound on MMAR 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, our analysis using a window size of 4 instructions would report an MMAR of 4 bytes per 4 instructions. However, there are numerous ways in which the threads can be scheduled to yield an actual MMAR 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 MMAR observed is 12 bytes per 4 instructions, three times that of the thread-ignorant statically-computed bound.

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 four 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.

5.2 Timing Context Switches

Fortunately, the above discussion is overly pessimistic. For one, we assumed that context switching between threads is free, meaning that it has no effect on the MMAR. 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 is being preempted must be saved and placed in the appropriate queue. The next thread that is ready to execute (according to some scheduling policy) must be found and removed from its queue. Finally, the context of the new thread must be loaded into the CPU. In other words, some number of non-allocating instructions will be executed by the operating system between the last instruction of the thread being

[4] Masters Thesis [20]

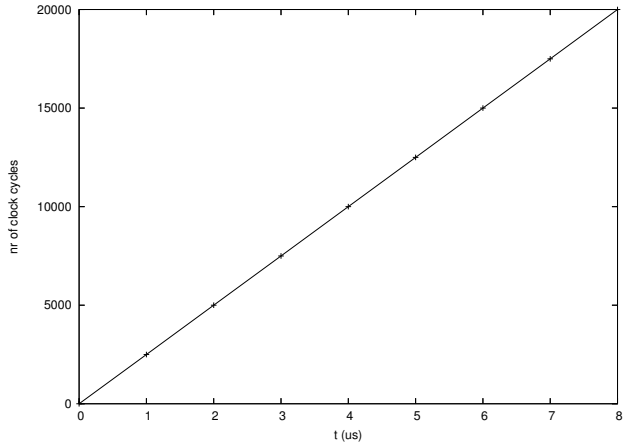

Figure 8. The number of clock cycles executed per μs on a 2.5-GHz processor

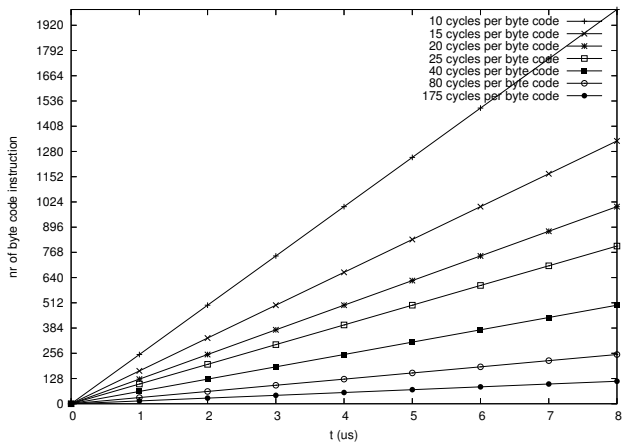

Figure 9. The fewest number of bytecode instructions executed per μs on a 2.5-GHz processor. Each line represents a specific maximum number of clock cycles needed to execute any byte code instruction

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 5.1 would not apply because whenever a new thread starts executing, its allocation window will be empty.

Figure 8 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 5.1. As t increases, more and more non-allocation instructions are executed during the context switch.

To measure the time of a context switch, we adapted the work of Bradford [5]. In particular, we use a C program (quite similar to Bradford's) that 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 Bradford as being negligible compared to the cost of context switching. The code is shown below:

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

Using this program we ran an experiment using two threads, context switching back and forth one million times. On average, each context switch took 4.022μs. Figure 8 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? Here we cannot assume that each instruction takes only one clock cycle, because the worst case is that each instructions executed during the context switch take as long as possible minimizing the number of instructions executed during the switch. A conversion factor between clock cycles and executed byte code is needed. As aforementioned, 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 two pieces of information. 1) Which is the most expensive Java byte code instruction and 2) How many clock cycles does this instruction need to execute.

The literature on this topic agree that the most expensive Java byte code instructions are the `invoke` instructions. However, the reports on how many clock cycles the invoke instruction needs to execute varies from 15 [37] to 175 [28]. This large spread is due to the differences in the hardware used. In his thesis work, Schoeberl [28] reported that invoke needs 175 clock cycles running on a Cyclone FPGA, while NanoAmp Solutions report 20 clock cycles using an ARM CPU [24]. Figure 9 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.

As previously mentioned, the experimentally determined context switching time was 4.022μs. Figure 9 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. In Section 6, we will show that, for the tested SPEC jvm98 benchmarks [31], the MMAR 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.

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 work of Schoeberl [28], 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 ratios of 15–20 clock cycles per byte code instruction [37, 24], 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 section, the MMAR of multithreaded 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 MMAR using *safe points* (as in Jikes RVM [15]) so that threads are interrupted only at predetermined points. One could also construction an allocation-aware thread scheduler which would enforce particular interleavings to optimize the time *vs.* space tradeoff in the application. These avenues have not been explored in this work.

5.3 Sporadic Real-Time Tasks

Many real-time applications require the notion of *sporadic scheduling*, or another rate-limiting feature for certain tasks. Commonly used for event-handling tasks, the idea of sporadic scheduling is to budget only for (and, generally, to permit only) a certain number of executions over some period. For example, a human operator may push a button on a real-time system many times a second (or hold the button down), but the system may choose only to respond to that button press once per second. Sporadic scheduling can make it possible to deploy provably feasible task sets that interact with the statically-unknowable physical world.

In many cases, these tasks can be statically shown never to execute in parallel (for example, calls to an event handler are serialized in the RTSJ), and, further, values of sporadic scheduling parameters can sometimes be determined statically. This additional information about the target program can reduce the set of possible thread interleavings and the frequency with which certain blocks of code can execute, improving our bound for multithreaded programs.

6. Experiments

In this section we report on the application of our analysis on some Java benchmarks. While those benchmarks are admittedly 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 work takes steps in that direction—then real-time Java programs and benchmarks should be more plentiful.

We have implemented our static analysis for the MMAR and array allocation bounds on top of *Clazzer* [18], a byte-code manipulation framework in which data flow problems can be explicitly defined and solved. Figure 10 displays our static determination of the MMAR of benchmarks in the SPEC jvm98 benchmark suite [31]. We used window sizes of 16, 32, 64, 128, 256, and 512 clock cycles. Figure 10 illustrates the problem associated with relatively small window sizes: When the window size is small each allocation has a dramatic effect on the overall MMAR. The plot also shows that as the window size increases, the MMAR decreases, asymptotically approaching a bound of the average allocation rate of the entire program. This is expected; in previous work [21] we presented a dynamic analysis of a subset of the SPEC jvm98 benchmark suite demonstrating that the MMAR rapidly approaches the average rate as the window size increases.

We know that doubling the window size can never increase the MMAR. Intuitively, we can show this by considering a window of size n with an MMAR 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 MMAR 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 MMAR.

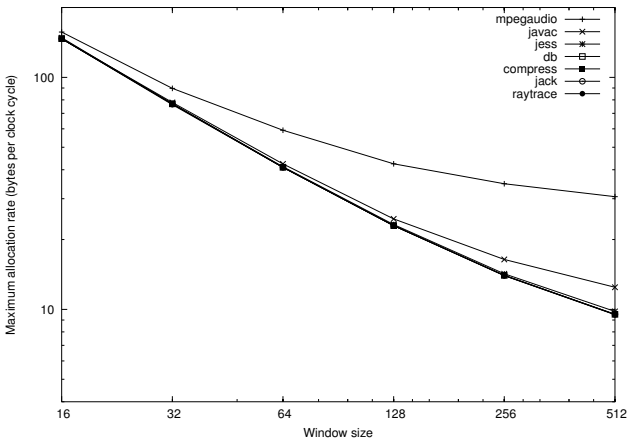

Figure 10. MMAR *vs.* window size (statically-determined bound).

As a consequence, the static upper bound of the MMAR for a sufficiently large window can be used to approximate an upper bound for an arbitrarily large window. For example, the results in Figure 10 suggest that using a window size of 256 as an approximation is not overly conservative.

Using a window size of 256 clock cycles, we find bounds for most of our tested benchmarks 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 4.3.3. This means that even a very small array allocation could have a large effect on the upper bound. Figure 11 shows that the MMAR computed for the 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 12 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 11 and Figure 12 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 MMAR 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 11 and 13 and Figures 12 and 14 makes it clear that the large numbers of interprocedurally-analyzed procedures with a high MMAR is caused by heavy allocation in relatively few procedures.

As expected, the intraprocedural plots (Figures 13 and 14) also show that the MMAR of the heavily allocating procedures are

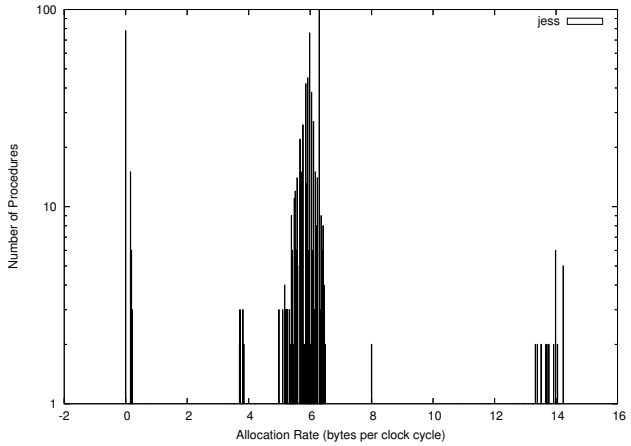

Figure 11. Number of procedures with a given upper bound for jess with a window size of 256, running interprocedural analysis using arraylets.

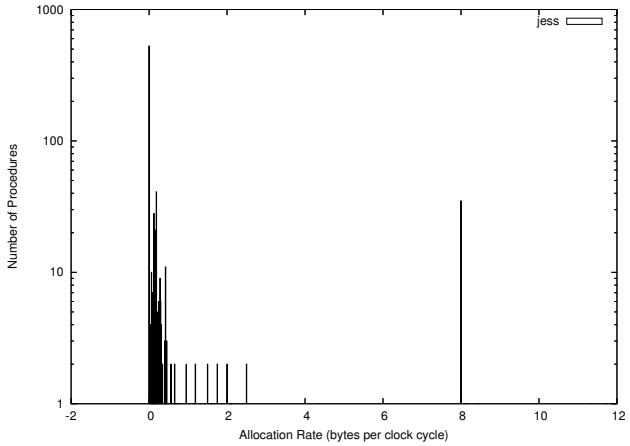

Figure 13. Number of procedures with a given upper bound for jess with a window size of 256, running intraprocedural analysis using arraylets.

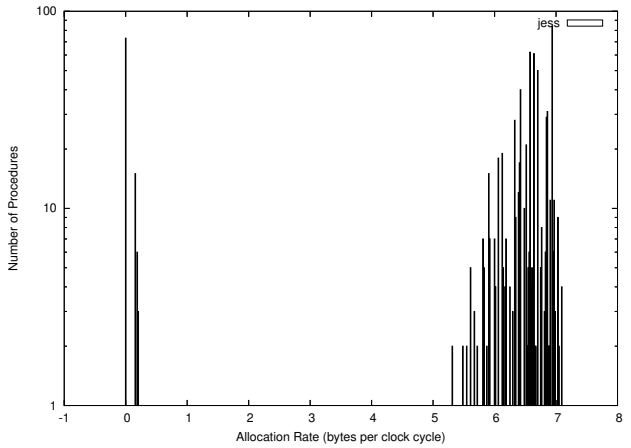

Figure 12. Number of procedures with a given upper bound for jess with a window size of 256, running interprocedural analysis assuming each array allocation is 16 bytes.

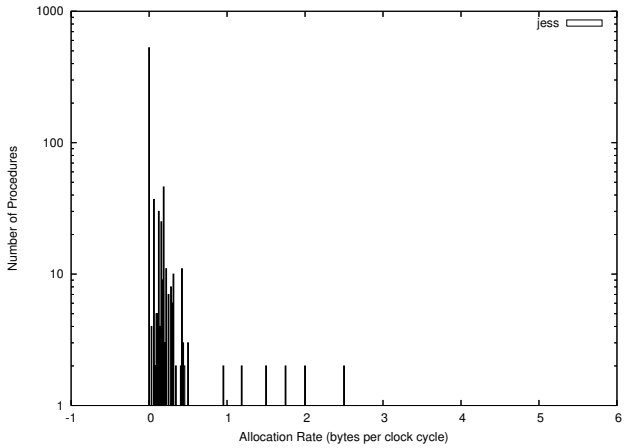

Figure 14. Number of procedures with a given upper bound for jess with a window size of 256, running intraprocedural analysis assuming each array allocation is 16 bytes.

caused by allocations of arraylets. Figure 13 has a spike at 8 bytes per clock cycle, which does not appear in Figure 14. $8 \times 256 = 2048 = 2\text{KB} =$ Arraylet size.

We also implemented a dynamic data-collection mechanism in a **Java Virtual Machine** (JVM) to capture the *actual* MMAR of our benchmarks in various window sizes. For this, too, we limited 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 15 shows the MMAR 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 16 and 17—Figure 16 compares the static bound to the observed MMAR in the jess benchmark, and Figure 17 makes the comparison over all benchmarks in the suite. As the figures demonstrate, we found that our static bounds did indeed bound the MMAR, and that they were reasonable bounds for these benchmark runs.

In particular, with the exception of the mpegaudio and javac benchmarks, our static bounds on MMAR is within a factor of less then 2 of the actual, observed MMAR over all tested window sizes. We bind javac with a factor less then 2.5 for all tested window sizes and 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.)

The static bound for mpegaudio deviates more from the observed rate than does the other benchmarks because 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 thinks the allocation could happen in a loop (though it happens just once) and the steady-state MMAR is 5.8 times higher than what was seen.

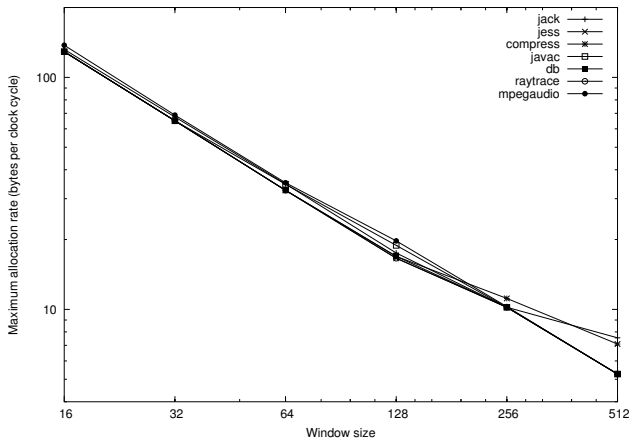

Figure 15. MMAR *vs* window size (actual observation).

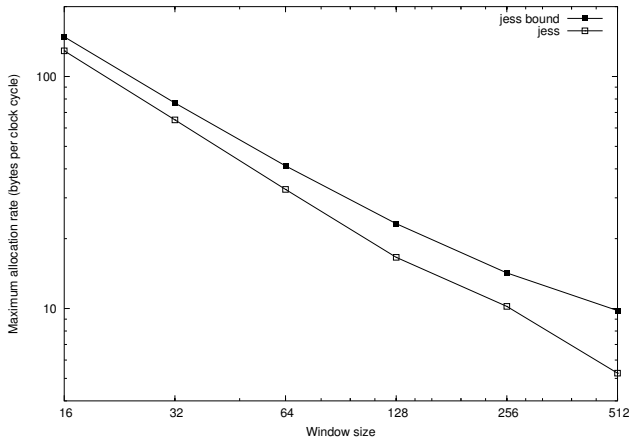

Figure 16. Comparison of bounded and actual MMAR for jess.

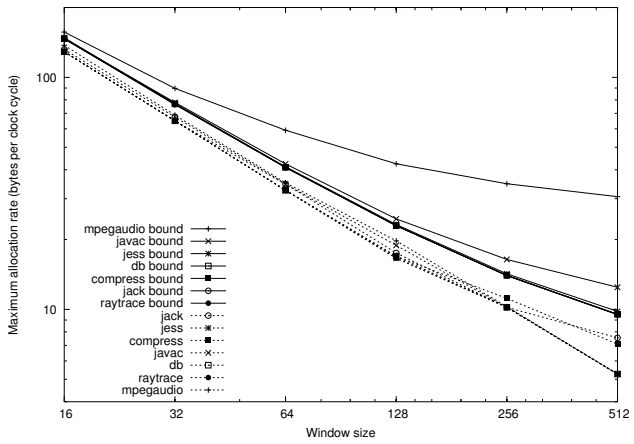

Figure 17. Comparison of bounded and actual MMAR for all benchmarks (both plotted together).

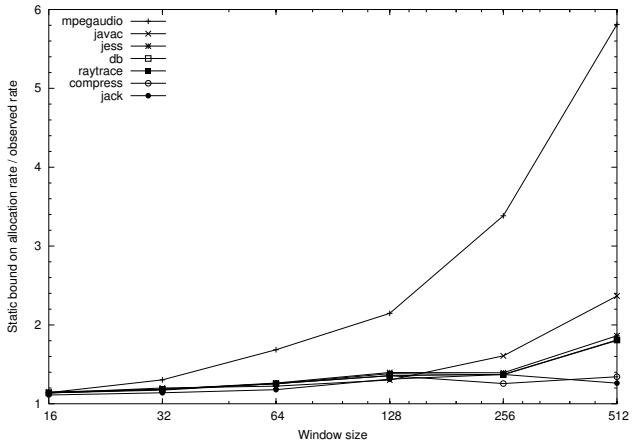

Figure 18. Comparison of bounded and actual MMAR for all benchmarks (static bound on rate / observed rate).

Figure 18 shows this comparison over all benchmarks and window sizes.

7. Related Work

Write me!

Memory studies of benchmarks, footprint bounds (dynamic & static), etc?

8. Conclusion

We have provided a framework for determining MMAR and have applied this framework to some Java benchmarks. We have demonstrated that for our benchmarks, our statically-determined MMAR is within a constant factor of the observed MMAR. Whether or not this constant factor constitutes a reasonable upper bound is a subjective issue. The size of this factor will have an effect on the memory footprint and the MMU [9] of the application. If a 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 MMAR. However, before using our system to deploy a garbage collector in a real-time environment, further study on the effect of converting from bytes per instruction to bytes per unit time is needed.

Admittedly, our set of benchmarks are not real-time benchmarks, but one reason for a lack of real-time Java code is the effort required to use the 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 [27]. 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 [21], 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. MMAR within execution of real-time threads is the relevant statistic for the real-time collector.

We also expect to be able to get a tighter bound on MMAR by relaxing the assumption that all instructions execute in one clock cycle. If the number of needed clock cycles for each instruction is

known, modifying the transfer function to account for this would not be difficult.

Acknowledgements

We thank Martin Linenweber for his efforts in developing the PCESjava bytecode-engineering and analysis framework [18].

We thank Richard Souvenir for his careful reading of this paper. We thank Joe Cross for his suggestion concerning the usefulness of per-method allocation rates as a guide to help developers rewrite code to improve CPU utilization. We also thank the LCTES 2005 reviewers for their insightful suggestions on an earlier version of this paper that appeared there [22].

References

[1] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley, Boston, third edition, 2000.

[2] 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 (LCTES 2003)*. ACM Press, 2003.

[3] 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 (POPL 2003)*, pages 285–298. ACM Press, 2003.

[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] Edward G. Bradford. Runtime: Context switching, part 1. `www-106.ibm.com/developerworks/linux/library/l-rt9/`, July 2002.

[6] Dante J. Cannarozzi, Michael P. Plezbert, and Ron K. Cytron. Contaminated garbage collection. In *Proceedings of the ACM SIGPLAN '00 conference on Programming Language Design and Implementation (PLDI 2000)*, pages 264–273, 2000.

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

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

[9] Perry Cheng and Guy Belloch. A parallel, real-time garbage collector. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2001)*, pages 125–136, 2001.

[10] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. *ACM SIGPLAN Notices*, 34(10):1–19, 1999.

[11] 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.

[12] 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.

[13] William H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Transactions on Software Engineering*, SE-13(3), May 1977.

[14] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[15] IBM developerWorks. Jikes RVM Home Page. `jikesrvm.sourceforge.net`, 2005.

[16] J. Lehoczky, L. Sha, and Y. Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In *Proceedings of the 10th IEEE Real-Time Systems Symposium (RTSS 1989)*, pages 166–171. IEEE Computer Society Press, 1989.

[17] Tom Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, Massachusetts, second edition, 1999.

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

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

[20] Tobias Mann. Static determination of allocation rates to support real-time garbage collection. Technical Report WUCSE–05–24, Washington University in St. Louis, Department of Computer Science and Engineering, May 2005. M.S. Thesis.

[21] Tobias Mann and Ron K. Cytron. Automatic Determination of Factors for Real-Time Garbage Collection. Technical Report WUCSE–04–45, Washington University, St. Louis, Missouri, 2004.

[22] Tobias Mann, Morgan Deters, Rob LeGrand, and Ron K. Cytron. Static determination of allocation rates to support real-time garbage collection. In *Proceedings of the ACM Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2005)*, pages 193–202. ACM Press, 2005.

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

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

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

[26] F. Pizlo, J.M. Fox, D. Holmes, and J. Vitek. Real-time Java scoped memory: Design patterns and semantics. In *Proceedings of the Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2004)*, Vienna, Austria, May 2004. IEEE.

[27] 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.

[28] Martin Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, Vienna, Austria, 2005.

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

[30] 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.

[31] SPEC Corporation. Java SPEC benchmarks. Technical report, SPEC, 1999. Available by purchase from SPEC.

[32] Alexandru Sălcianu and Martin C. Rinard. Pointer and escape analysis for multithreaded programs. In *Principles Practice of Parallel Programming*, pages 12–23, 2001.

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

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

[35] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. *ACM SIGPLAN Notices*, 34(10):187–206, 1999.

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

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

A. Range Propagation

As described in Section 4.3.1, we employ range propagation [13] to bound statically the size of allocated arrays. This appendix pro-

```
1   int n = 10;
2   if(needBigArray) {
3       n *= 10;
4   }
5   Object[] objArray = new Object[x];
```

Figure 19. A Java code snippet.

vides additional details about our implementation and presents experiments demonstrating its utility.

A.1 Background

Constant propagation [34, 23] is an optimization that aims to reduce the number of data accesses required by a program. First, locations of the target program resulting in statically-known, constant state (such as an assignment of x to a constant value) are identified. These static constants are pushed forward through the control flow graph of the program, transitively resulting in additional statically-known state (*e.g.*, if y is assigned to x while x is constant and statically-known). If state is tainted with statically-unknown data (such as user input), or if conflicting state is detected, classic constant propagation analysis gives up and declares the data nonconstant ($\perp$).

Sometimes, however, program state, while nonconstant, is drawn from a statically-known, constrained range (*i.e.*, it is statically *bounded*). This defeats the analysis underlying classic constant propagation but is a common behavior of programs (see Section A.4). A more general analysis can establish and maintain statically-known *bounds* on program variables and arithmetic expressions. This allows other static program analyses requiring bounds on particular values (as, for example, an analysis that requires a static bound on the size of a dynamic allocation request) to provide better results than if they were to use the results of traditional constant propagation.

Applications. Range propagation analysis gives useful results even when those results provide nonconstant ranges of variable values. Some static analyses, including bounding the size of array allocation requests as discussed in Section 4.3.1, depend not on knowing the specific value of a program variable, but a *bound* on the value. In the context of this paper, range propagation analysis can statically bound the size of some array allocation sites.

As a simple example, consider the Java code snippet in Figure 19 and its associated control-flow graph in Figure 20. In a classic constant propagation analysis, `n` is constant at the start of execution of lines 2 and 3 (where $\texttt{n} = 10$), and after the execution of line 3 (where $\texttt{n} = 100$). However, at lines 4 and 5, when the two program paths merge, `n` is nonconstant (being either 10 or 100), so classic constant propagation gives up on `n` at this point.

Range analysis, however, observes the merging program state at line 4 and calculates the bounds on `n` to be $[10, 100]$. This is also conservative, of course, as range analysis doesn't indicate that 10 and 100 are the *only* values permissible (which is statically knowable); instead, range analysis simply indicates that `n` must be valued between 10 and 100, inclusive. For an analysis needing to know only a bound on the size of the allocated object array, this is a sufficient result; nothing more would be gained by knowing the specific values that `n` may take.[5]

[5] The astute reader will note that this statement holds true for *this* example, but may not in others; Section A.2 discusses some cases where range analysis can fail to establish bounds that would be statically determined by an analysis that tracked *sets* of values, rather than *bounds* on values, that program variables may take.

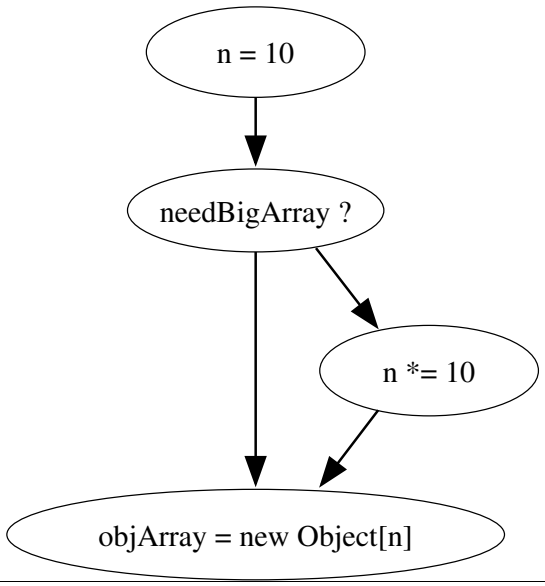

Figure 20. The control-flow graph of the Java code snippet of Figure 19.

A.2 Approach

We formulate the problem as a dataflow framework over integer variables with fixed-width binary representations.[6] For our presentation here, we assume a single integer class, so that all integers are of the same size. We assume `MININT` is the smallest value that may be held by the data type, and that `MAXINT` is the largest value that may be held by the data type. Note that $-(\texttt{MININT} + 1) = \texttt{MAXINT}$.

This representation suits our needs for bounding array allocation sizes. Range propagation can be extended to other data types, but that is outside the scope of this paper.

First, we some preliminary definitions. If L is a lower bound on a variable and U is an upper bound on the same variable, inclusive, we write a value range for the variable $[L, U]$. When $x > y$, a range $[x, y]$ contains no elements. A range is *smaller* than another range if it admits fewer elements of $\mathbb{Z}$. The set $\mathcal{V}$ is the set of program variables.[7]

With these preliminaries, we now formally describe our framework:

- $A = \mathcal{V} \longmapsto \{[L, U] \mid -\texttt{MAXINT} \leq L, U \leq \texttt{MAXINT}\}$
- $\top = [\texttt{MAXINT}, \texttt{MININT}]$
- $\perp = [\texttt{MININT}, \texttt{MAXINT}]$
- $\wedge$ is a widening operator that computes the smallest range that includes the two input ranges
- $a \preceq b$ holds if and only if $a \wedge b = a$

Thus A maps variables to ranges, $\perp$ provides no information (the variable's value range includes all values available to the data type), and $\wedge$ expands a range to include two ranges. $\top$ contains no elements, and it yields immediately to any other range under

[6] In Java, all instructions (except `iushr` and friends, right-shifts without sign extension) interpret integer operands as signed values under two's complement.

[7] In Java, the "program variables" include stack cells and virtual machine registers (the latter of which are referred to as "local variables" in the documentation [17]. Since we consider stack cells as program variables, we don't need to evaluate complex arithmetic expressions or treat temporary values specially.

meet; this curious definition is analogous to constant propagation's traditional definition of $\top$.

The transfer function for range propagation is more complicated. Our approach evaluates arithmetic statically, when possible, to bound the result. In the following sections, we define the transfer function for different types of control flow nodes.

Assignment and negation. Given a simple assignment of `x` to `y`:

```
x := y
```

the procedure is obvious—`x` inherits `y`'s upper and lower bounds.

Similarly, we would expect that

```
x := -y
```

would flip the bounds; that is, `x`'s upper-bound would be the additive inverse of `y`'s lower-bound, and `x`'s lower-bound would be the additive inverse of `y`'s upper-bound. However, this is not necessarily so.

[discussion of finite fields?]

The problem is that `y` might be the most negative value available for its storage class (with its most-significant bit set and al others unset). In this case, `y == -y`. Because of this, if `y`'s range includes this most negative possible value for its storage class, `x`'s lower-bound must also be this minimum value. If `y` is nonconstant, then `x`'s upper-bound must be the maximum possible value for the storage class. Because this calculated range for `x` includes all possible values that a member of its storage class can take, effectively *nothing is known statically* about its value range.[8]

It is interesting to note that the Java programming language forbids code that might cause overflow in this way [1]. However, Java bytecode that exhibits such behavior *is* legal, so a sequence like '`sipush -32768; ineg; i2s`' results in -32768. Other languages, like C, do not forbid such source code and can also exhibit the behavior that $-x = x$ for some nonzero value of x.

Addition. Consider two values x and y, their lower bounds L_x and L_y, and their upper bounds U_x and U_y.

$$\begin{array}{rcccl} L_x & \leq & x & \leq & U_x \\ L_y & \leq & y & \leq & U_y \end{array}$$

When these values are added, we get their sum $x + y$. Upper and lower bounds on this sum are denoted U_{x+y} and L_{x+y}:

$$L_{x+y} \quad \leq \quad x + y \quad \leq \quad U_{x+y}$$

Our goal is to find suitable values for U_{x+y} and L_{x+y} from the bounds we do have on x and y. So, we would expect:

$$\begin{array}{rcl} L_{x+y} & = & L_x + L_y \\ U_{x+y} & = & U_x + U_y \end{array}$$

However, in a program analysis context we must carefully consider overflow conditions, as x and y are in fact fixed-width fields.

$$\begin{array}{rcll} L_{x+y} & = & \begin{cases} L_x + L_y & \text{if no overflow} \\ \texttt{MININT} & \text{if overflow} \end{cases} \\ U_{x+y} & = & \begin{cases} U_x + U_y & \text{if no overflow} \\ \texttt{MAXINT} & \text{if overflow} \end{cases} \end{array}$$

Any overflow condition affects both bounds; if $L_x + L_y$ doesn't overflow but $U_x + U_y$ does, then $L_{x+y} = \texttt{MININT}$ and $U_{x+y} = \texttt{MAXINT}$. Since in our problem $\perp = [MININT, MAXINT]$ for integers, we have effectively given up on bounding the integer at this point.

[8] A more sophisticated analysis could track more than simple lower-bound–upper-bound ranges as we are doing here, to provide a statically-known range of possible values in this case.

Subtraction. Once again, we have:

$$\begin{array}{rcccl} L_x & \leq & x & \leq & U_x \\ L_y & \leq & y & \leq & U_y \\ L_{x-y} & \leq & x - y & \leq & U_{x-y} \end{array}$$

And, once again, we would expect:

$$\begin{array}{rcl} L_{x-y} & = & L_x - U_y \\ U_{x-y} & = & U_x - L_y \end{array}$$

but we get similar overflow conditions resulting in $\perp$.

Multiplication. Multiplication introduces additional complications:

$$\begin{array}{rcll} L_{x*y} & = & \begin{cases} L_x \cdot L_y & \text{if } x, y \geq 0 \\ U_x \cdot U_y & \text{if } x, y \leq 0 \end{cases} \\ U_{x*y} & = & \begin{cases} U_x \cdot U_y & \text{if } x, y \geq 0 \\ L_x \cdot L_y & \text{if } x, y \leq 0 \end{cases} \end{array}$$

With negative x and nonnegative y (or nonnegative x and negative y):

$$\begin{array}{rcl} L_{x*y} & = & \min\{U_x \cdot L_y, L_x \cdot U_y\} \\ U_{x*y} & = & \max\{U_x \cdot L_y, L_x \cdot U_y\} \end{array}$$

However, we cannot guarantee that x (resp. y) is nonpositive unless $U_x \leq 0$ (resp. $U_y \leq 0$), and we cannot guarantee that x (resp. y) is nonnegative unless $L_x \geq 0$ (resp. $L_y \geq 0$). So the general form for multiplication is:

$$\begin{array}{rcll} L_{x*y} & = & \begin{cases} L_x \cdot L_y & \text{if } x, y \geq 0 \\ U_x \cdot U_y & \text{if } x, y \leq 0 \end{cases} \\ U_{x*y} & = & \begin{cases} U_x \cdot U_y & \text{if } x, y \geq 0 \\ L_x \cdot L_y & \text{if } x, y \leq 0 \end{cases} \end{array}$$

With negative x and nonnegative y (or nonnegative x and negative y):

$$\begin{array}{rcl} L_{x*y} & = & \min\{U_x \cdot L_y, L_x \cdot U_y\} \\ U_{x*y} & = & \max\{U_x \cdot L_y, L_x \cdot U_y\} \end{array}$$

However, we cannot guarantee that x (resp. y) is nonpositive unless $U_x \leq 0$ (resp. $U_y \leq 0$), and we cannot guarantee that x (resp. y) is nonnegative unless $L_x \geq 0$ (resp. $L_y \geq 0$). So the general form for multiplication is:

$$\begin{array}{rcl} L_{x*y} & = & \min\{L_x \cdot L_y, U_x \cdot U_y, L_x \cdot U_y, U_x \cdot L_y\} \\ U_{x*y} & = & \max\{L_x \cdot L_y, U_x \cdot U_y, L_x \cdot U_y, U_x \cdot L_y\} \end{array}$$

with the familiar overflow conditions resulting in $\perp$.

Division. Similar to multiplication. The general form is similar, with guards against division by zero (see Figure 21). In our target programming languages, if the program *does* divide by zero, the undefined result is not stored into the program variable on the left-hand side of the assignment. For that reason, we are justified in excluding zero from the divisor's range.

In fact, if the divisor is zero, a branch to an exception handler (or exiting the method) will be taken. Along this exceptional branch path, the divisor is known to have been zero. On the non-exceptional path, the divisor cannot have been zero. This knowledge can be used to further constrain the ranges calculated by the analysis along these distinct paths. As an example, consider Figure 22. At a division instruction, `x` and `y` are each known to have bounds $[0, 10]$. The division $\texttt{x} \div \texttt{y}$ is performed. Along the exceptional path, `x`'s bounds are unchanged but `y` is known to be 0. Along the non-exceptional path, `x`'s bounds are also unchanged[9] and `y`'s bounds are unchanged *except* for the fact that it is known to be nonzero.

[9] To see why this is so, consider cases where $\texttt{x} = 0$ before the division (it will also be zero after) and where $\texttt{x} = 10$ and $\texttt{y} =$

$$L_{x \div y} = \begin{cases} \text{undefined} & \text{if } U_x = L_x = U_y = L_y = 0 \\ \min\{L_x \div \downarrow U_y, U_x \div \uparrow L_y, U_x \div \downarrow U_y, L_x \div \uparrow L_y\} & \text{otherwise} \end{cases}$$

$$U_{x \div y} = \begin{cases} \text{undefined} & \text{if } U_x = L_x = U_y = L_y = 0 \\ \max\{L_x \div \downarrow U_y, U_x \div \uparrow L_y, U_x \div \downarrow U_y, L_x \div \uparrow L_y\} & \text{otherwise} \end{cases}$$

where

$$\uparrow x = \begin{cases} 1 & \text{if } x = 0 \\ x & \text{otherwise} \end{cases}$$

$$\downarrow x = \begin{cases} \text{-1} & \text{if } x = 0 \\ x & \text{otherwise} \end{cases}$$

Figure 21. The range propagation transfer function for integer division control flow nodes.

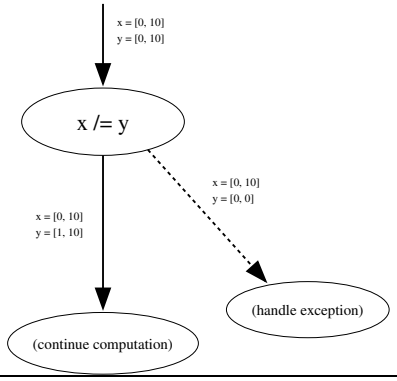

Figure 22. A division, possibly by zero. Edges are tagged with statically-known bounds on program variables `x` and `y`.

Comparisons. can conclude $x > y$ if $L_x > U_y$ etc.

Conditionals. on a conditional branch, can relate new knowledge (result of comparison etc.) to previous knowledge (e.g., DUPs)

Summary. The range propagation data flow result at each node is a function mapping program variables to value ranges. Our transfer function statically evaluates arithmetic and other operations, and it updates this mapping to take advantage of the new information provided.

A.3 Implementation

We have implemented an intraprocedural range propagation analysis for Java bytecode using the PCESjava framework [18]. Our implementation simulates the Java stack and registers found in Java virtual machines, except that our stack slots and registers contain ranges instead of values. These stacks and registers are pushed through each function's control flow graph so that each program point (node in the control flow graph) is tagged with bounds on program state.

Limitations. There are several limitations of the implementation that are not necessarily limitations of the general approach to range propagation. First, as noted, our implementation is intraprocedural, so while bounds might be discovered by this approach for program variables assigned to the result of a procedure call, or bound to a procedure's formal arguments, our implementation does not discover such bounds.

Also, our implementation doesn't track bounds on class and object fields. This may seem a trivial extension to an implementation that tracks other program state. However, many Java programs are multithreaded, and we would have to take into account the possible actions of other threads, and in order to obtain an effective solution, we would perhaps be required to include further analyses to determine which object fields are inaccessible by other threads at certain points in the program's execution[10].

We don't do conditional stuff. (like tracking dups and inferring bounds later)

Discussion of overflow conditions?

no tracking of array size after creation e.g. array.length could be statically determined but isn't.

A.4 Experimentation

We performed a simple experiment to verify a claim we make in Section A.1 and demonstrate that range propagation can be useful as an extension to classic constant propagation.

Show vs. classic constant propagation: how many program variables are statically-known constants; how many array allocation points can be statically-bound

Also show: distribution of bounds sizes

how often overflow actually occurs

A.5 Conclusions

We have described and demonstrated our implementation of range propagation [13], an extension to constant propagation [34, 23] that statically provides upper and lower bounds on program variables in cases where they are nonconstant. These upper and lower bounds may result in future variables being statically-known constants (where constant propagation would fail), and provides more accurate results to other analyses, such as allocation-size analysis.

1 (`x` is still 10 after the division). In terms of the general formula, $L_{x \div y} = \min\{0 \div 10, 10 \div 0, 10 \div 10, 0 \div 0\}$ and $U_{x \div y} = \max\{0 \div 10, 10 \div 0, 10 \div 10, 0 \div 0\}$.

[10] For example, an object that has been newly instantiated but a reference to which has not yet *escaped* [32, 35, 10] would be inaccessible to other currently-executing thread contexts