Appearing in Proceedings of the 2005 ACM Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2005)

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. Semi-automatic 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 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. Standards such as the **Real-Time Specification for Java** (RTSJ) [3] have emerged that offer facilities for the specification, scheduling, and management of real-time structures, such as periodic threads, asynchronous events, and high resolution timers. There is general agreement that the efficient and predictable execution of such structures is necessary for the acceptance of the RTSJ or any other Java implementation that claims real-time performance.

However, when it comes to storage management, there is not (yet) universal agreement as to *how* to make object allocation and (in particular) deallocation and garbage collection reasonably predictable. Included as a core requirement in the NIST specification for a real-time Java is the following [5]:

> 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 run out, 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 to access only these objects allocated in scopes.[1] The rules for scope creation are established so that a reference count on the entire scope suffices to determine liveness of all objects in the scope. The reference count is affected by threads entering and exiting the scope.

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

- The application is constrained as to how objects in scopes can reference each other, as depicted in Figure 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* [14], and programs can leak an unbounded amount of dead storage in a region.

[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.

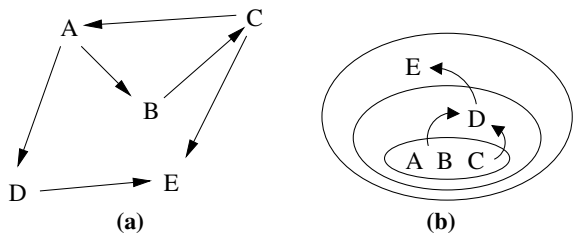

Figure 1. Use of scoped storage. **(a)** shows the references between objects at one point in time; **(b)** shows a legal scope assignment. If E tried to reference D, the program would fail given this scope assignment.

> 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.

Traditional garbage collection can also be avoided by using techniques such as reference counting [16] and contaminated garbage collection [4], 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 [2] or Perc [10], 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 [2, 1] 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 the 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 program's maximum allocation rate. This kind of analysis is done at compile time and it is crucial to the correct operation of a real-time collector. Once properly bounded, a program's allocation rate determines the necessary fraction of execution time that must be devoted to real-time garbage collection.

Our paper is organized as follows. In Section 2 we summarize the statistics required by collectors such as Metronome to guarantee correct operation. 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 3 presents our data flow framework, and experiments determining allocation rates are presented in Section 4. Because static approaches are necessarily conservative, we also report on our benchmarks' allocation rates from actual executions. Finally, Section 5 concludes.

2. Mutator Statistics for Real-Time Collection

While in this paper we focus primarily on the Metronome real-time collector [2], 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* [12, 4] is computationally intensive, as the garbage collector must be run when any stack or heap cell is modified.

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

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

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

Allocation rate: A real-time collector cannot 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. That product is influenced by the mutator in terms of its allocation rate, but the fraction of time given to the mutator is the key parameter used by the collector to guarantee pacing with the mutator.

Underestimating a program's allocation rate 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, allocation rate 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 a program's allocation rate. 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 program's allocation rate. 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.

3. 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 [9]. To this end a control flow graph representing the program is constructed. Below we show an example C program, and in Figure 2 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.

- $G_p = (N_p, E_p, s_p, e_p)$
- $L = (A, \top, \bot, \preceq, \wedge)$
- $F \subseteq \{f : L \rightarrow L\}$

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, $\top$, and bottom, $\bot$, 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 $n.out$.

3.1 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

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

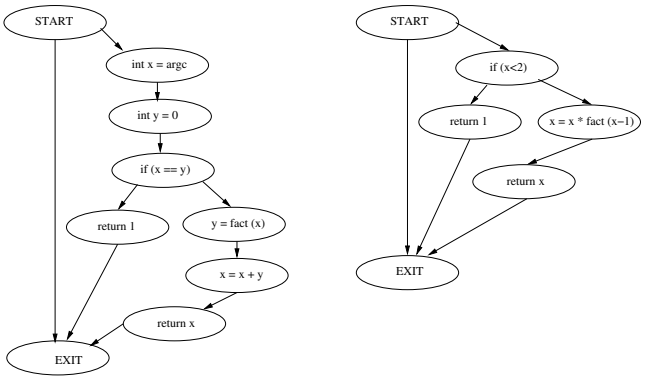

Figure 2. Intraprocedural dataflow framework generated from our example program.

essentially slides over a program's instructions, and we compute the maximum allocation rate 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 3.3.

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 bit represents the most recent instruction.

3.1.1 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:

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

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

For example, the bit-vectors $\langle 0, 0, 1, 0 \rangle$ and $\langle 0, 1, 0, 0 \rangle$ inform us that on their respective paths through G_p an allocation has occurred three and two instructions ago, respectively. Using the above meet, $\langle 0, 0, 1, 0 \rangle \wedge \langle 0, 1, 0, 0 \rangle = \langle 0, 1, 1, 0 \rangle$. Clearly all information has been retained and thus the result can never be better

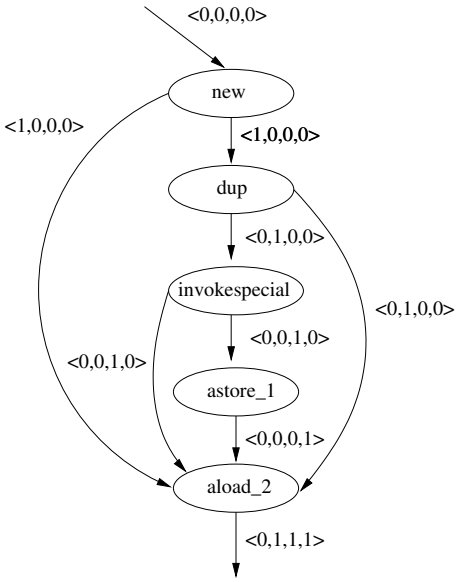

Figure 3. The control flow of a `try...catch` block fills a window unnecessarily with allocations in the naïve framework.

than the input vectors. However, as we shall see in the next section, this meet function is overly conservative.

Each transfer function $f \in F$ must update the solution at a given node, n, so that the output of 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.

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 3.

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 3 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 3, 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 will contain the same number of allocations as the vector with the most allocations, but where will they be placed? For example, say we need $\langle 0,1,0,0,1\rangle \wedge \langle 0,0,1,1,0\rangle$. One idea is to set the most significant bits in the result: $\langle 0,1,0,0,1\rangle \wedge \langle 0,0,1,1,0\rangle = \langle 1,1,0,0,0\rangle$. This is better than logical bitwise *or* because it does not increase the number of allocations.

Nonetheless, this meet produces a solution that reflects that the last instruction it encountered was an allocation when none of its input vectors reflected that fact. A better idea is to let the meet place allocations in the positions of the most significant set bits in its input vectors: $\langle 0,1,0,0,1\rangle \wedge \langle 0,0,1,1,0\rangle = \langle 0,1,1,0,0\rangle$. This meet will never increase the number of allocations and it will never place an allocation at a position that all of its input vectors regard as a non-allocation.

3.1.2 Better Framework

A better way to compute *meet* in light of the example shown in Figure 3 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 3.3.

3.2 Framework Evaluation

Recall that in the introduction to this section we presented the intraprocedural data flow graph for an example C program (Figure 2). Forming an interprocedural solution from this graph is conceptually trivial. As show in Figure 4 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. [11], to specify our interprocedural dataflow framework formally:

- $G^* = (N^*, E^*)$
- P^* = the set of all procedures p represented in G^*
- $N^* = \bigcup_{p \in P^*} N_p$
- $E^* = E^0 \bigcup E^1$
- $E^0 = \bigcup_{p \in P^*} E^0_p$ is the set of intraprocedural control-flow edges
- $E^1 = \bigcup_{p \in P^*} E^1_p$ 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 3.1.2.

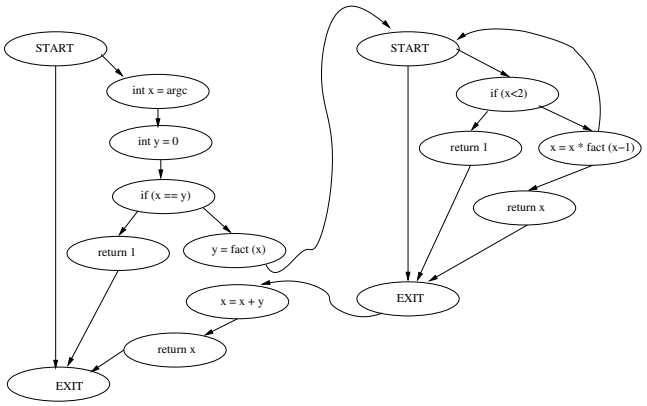

Figure 4. Interprocedural dataflow framework generated from our example program from Figure 2.

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

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

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

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

In the above algorithm $n.in$ refers to the combined input to node n, $n.out$ refers to the output of node n, and s_p and e_p refer to the start and exit nodes of procedure p, as mentioned in Section 3. 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 4 confirm that we obtain reasonable solutions with this approximation.

3.3 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 [2]. However, we are obligated to compute maximum allocation rate, and this cannot be based on average or expected behavior.

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

3.3.1 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, as demonstrated by the following example, where φ is some boolean condition that is not known statically:

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

Static determination of the size of statically-allocated arrays is in itself a data flow problem. While similar to constant propagation [15], the difference here is that we do not propagate whether or not a variable is a constant; instead, we propagate bounds on the possible value of a variable. 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 maximum allocation rate. In Section 3.3.2, 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.

As we are accounting for the size of what is being allocated, our meet lattice L and set of transfer functions F must be modified. Modification of the transfer function is straightforward: instead of shifting in a 1 for an allocation, we shift in the actual size of the object being allocated. The modification of L is shown below, and an example is given in Figure 5:

- $A = \{0, 1, 2, \ldots, M\}$ where M is the maximum number of bytes that the allocator can allocate at one time
- $\top = \langle 0, 0, 0, \ldots, 0 \rangle$
- $\bot = \langle M, M, M, \ldots, M \rangle$
- $\preceq$ is defined in terms of the $\wedge$ operator such that
$$a \preceq b \iff a \wedge b = a$$
- $\wedge$ is shown in Figure 5 and described below.

In Section 3.1.2, 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 5 shows how the meet function works when all allocations are not equal. Step 1 shows the

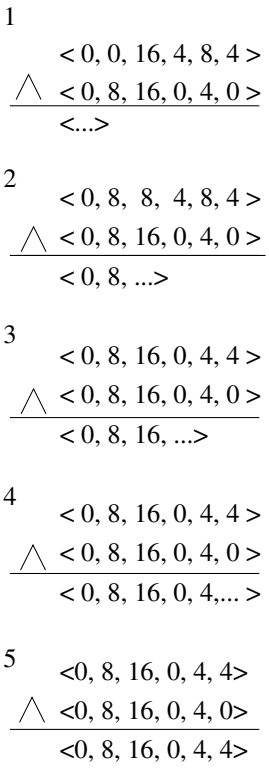

Figure 5. Computing meet when accounting for object allocation size.

original input vectors. These are never modified—steps 2–5 work with copies of the original vectors.

At step 2, in Figure 5, 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.

3.3.2 Arraylets

Bacon et al. [2] suggest the use of arraylets to solve the problem that large objects cause for real-time garbage collectors. The idea is to represent large arrays as a sequence of arraylets where each arraylet, except for the last, is of a constant size, C. Siebert [13] uses a similar idea and represents large arrays as a tree structure of fixed-size blocks.

As mentioned in Section 3.1, 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

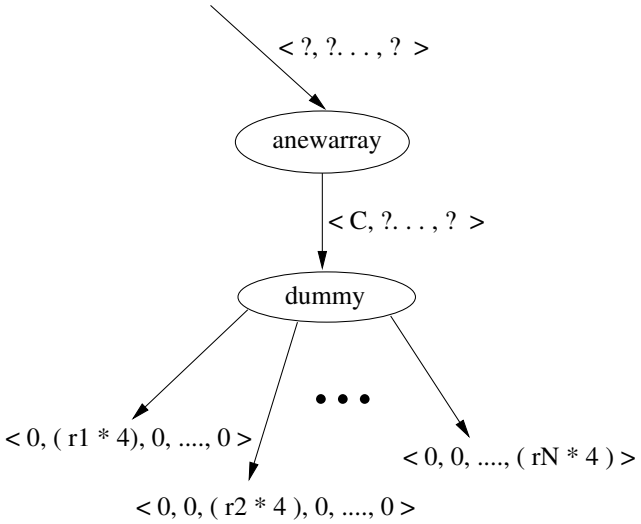

Figure 6. An array allocation, as represented in the control flow graph. ('?' represents any instruction.)

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. [2] 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 6 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 6. Taking the meet of all the output vectors from the dummy node gives us the vector $\langle 0, 4, \ldots, 4 \rangle$.

We have placed an upper bound on the allocation rate that can result from the allocation of a statically-unbounded array allocation. This bound is based on the assumption that the allocator will allocate arrays as a sequence of fixed-size arraylets. Similarly, if the allocator allocates large objects as a sequence of smaller allo-

cations, this technique can be used to estimate 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 6 rather than the meet of all of them.

If the allocator does not handle statically-unbounded array allocations as arraylets, there is little we can do to compute a good upper bound on the allocation rate. We would be forced to assume that $C = M$ in Figure 6. Since the array actually allocated may be small, we would still need to use the dummy node and meet all of its output vectors.

3.4 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 rules of a monotone data flow framework 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. $a \wedge \bot = \bot$

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

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

Thus far, we have a solution that is valid and that is guaranteed to converge, but at issue still is the quality of our solution relative to what could ideally be computed on each path separately through a procedure. 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* iff

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

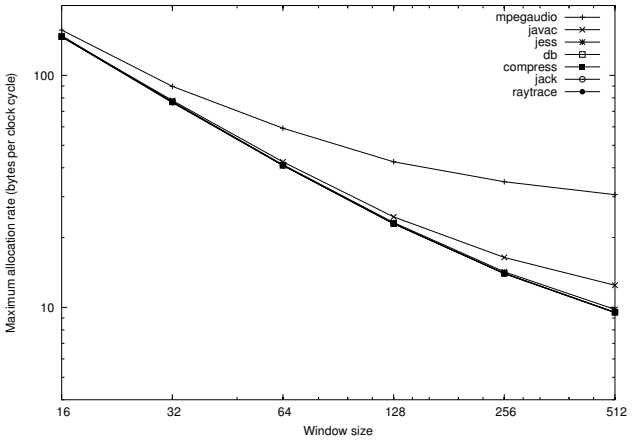

Figure 7. Maximum allocation rate *vs* window size (statically-determined bound).

It would be ideal if our framework were rapid, because we would be able to converge upon a solution more quickly. However, our framework is not rapid, since each trip around a loop can shift in another 1-bit.

4. Experiments

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

We have implemented our static analysis for maximum allocation rate and array allocation bounds on top of *Clazzer* [7], a bytecode manipulation framework in which data flow problems can be explicitly defined and solved. Figure 7 displays our static determination of maximum allocation rates of benchmarks in the jvm98 SPEC benchmark suite.[4] We used window sizes of 16, 32, 64, 128, 256, and 512 clock cycles. Figure 7 illustrates the problem associated with relatively small window sizes: When the window size is small each allocation has a dramatic effect on the overall maximum allocation rate. The plot also shows that as the window size increases, the maximum allocation rate decreases, asymptotically approaching a bound of the average allocation rate of the entire program. This is expected; in previous work [8] we presented a dynamic analysis of a subset of the jvm98 SPEC benchmark suite demonstrating that the maximum allocation rate approaches the average rate as the window size increases.

We know that doubling the window size can never increase the allocation rate. Intuitively, we can show this by considering a window of size n with a maximum allocation rate of $\frac{x}{n}$ where x is the maximum number of bytes allocated in any window of size n in the program. If doubling the window size increases the maximum allocation rate of the program then there exists an x' such that $\frac{x}{n} < \frac{x'}{2n}$. This implies that $x' > 2x$. It must also be the case that $x' \leq 2x$ because x is the maximum number of bytes allocated in any window of size n—doubling n cannot more than

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

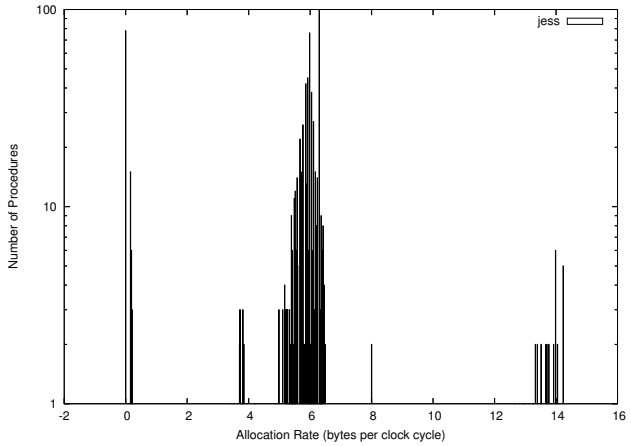

Figure 8. Number of procedures with a given upper bound for jess with a window size of 256, running interprocedural analysis using arraylets.

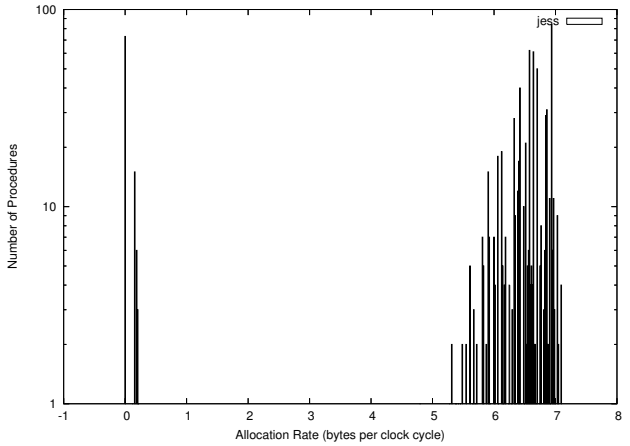

Figure 9. Number of procedures with a given upper bound for jess with a window size of 256, running interprocedural analysis assuming each array allocation is 16 bytes.

double x. We have a contradiction, so doubling the window size cannot increase the maximum allocation rate.

As a consequence, the static upper bound of the maximum allocation rate for a sufficiently large window can be used to approximate an upper bound for an arbitrarily large window. For example, the results in Figure 7 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 3.3.2. This means that even a very small array allocation could have a large effect on the upper bound. Figure 8 shows that the maximum allocation rate computed for the SPEC benchmark jess, is not representative of most procedures executed by the benchmark; most procedures in jess allocate between 5 and 7 bytes per clock cycle, and many allocate 0 bytes. Array allocations without a static bound force us to make a highly conservative assumption about their size—we might expect that procedures allocating such arrays actually allocate between 5 and 7 bytes per clock cycle, but we cannot determine that statically. Figure 9 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 8 and Figure 9 give the appearance that many procedures exhibit fairly high allocation rates. This is misleading because it does not mean that all procedures with a high maximum allocation rate actually are heavy allocators. The analysis we are performing is interprocedural and thus allocations that occur in procedure p_1 might affect the overall allocation rate of a procedure p_2, called by p_1. This "spill-over" effect is what creates the appearance that many procedures are heavy allocators. The contrast between Figures 8 and 10 and Figures 9 and 11 makes it clear that the large numbers of interprocedurally-analyzed procedures with a high maximum allocation rate is caused by heavy allocation in relatively few procedures.

As expected, the intraprocedural plots (Figures 10 and 11) also show that the maximum allocation rates of the heavily allocating procedures are caused by allocations of arraylets. Figure 10 has a spike at 8 bytes per clock cycle, which does not appear in Figure 11. $8 \times 256 = 2048 = 2\text{KB} =$ Arraylet size.

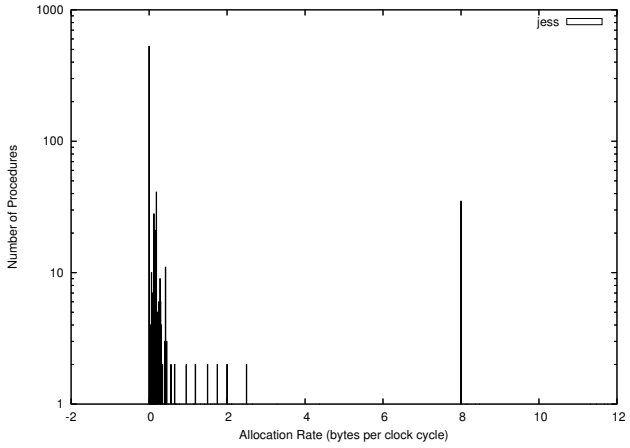

Figure 10. Number of procedures with a given upper bound for jess with a window size of 256, running intraprocedural analysis using arraylets.

We also implemented a dynamic data-collection mechanism in a **Java Virtual Machine** (JVM) to capture the *actual* maximum allocation rate 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 12 shows the maximum allocation rate observed during a run of size 100 of each of these benchmarks.

We offer comparisons of our static bounds and dynamically-collected results in Figures 13 and 14—Figure 13 compares the static bound to the observed allocation rate in the jess benchmark, and Figure 14 makes the comparison over all benchmarks in the suite. As the figures demonstrate, we found that our static bounds did indeed bound the maximum allocation rates, and that they were reasonable bounds for these benchmark runs.

In particular, with the exception of the 'mpegaudio' benchmark, our static bounds on allocation rate is within a factor of 2.5 of the actual, observed allocation rate over all tested window sizes. For

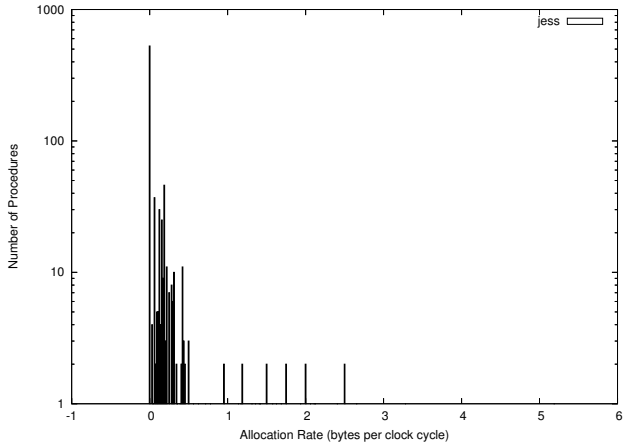

Figure 11. Number of procedures with a given upper bound for jess with a window size of 256, running intraprocedural analysis assuming each array allocation is 16 bytes.

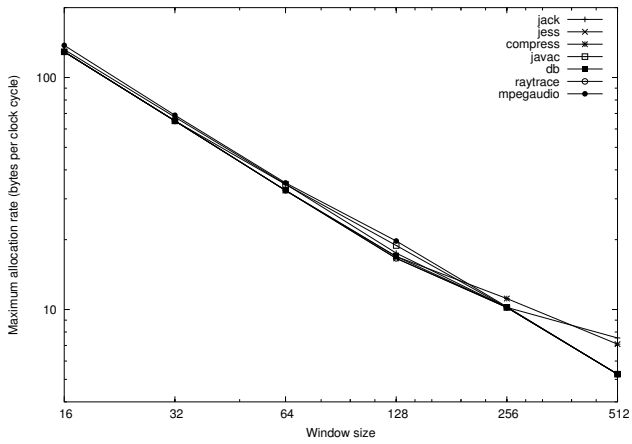

Figure 12. Maximum allocation rate *vs* window size (actual observation).

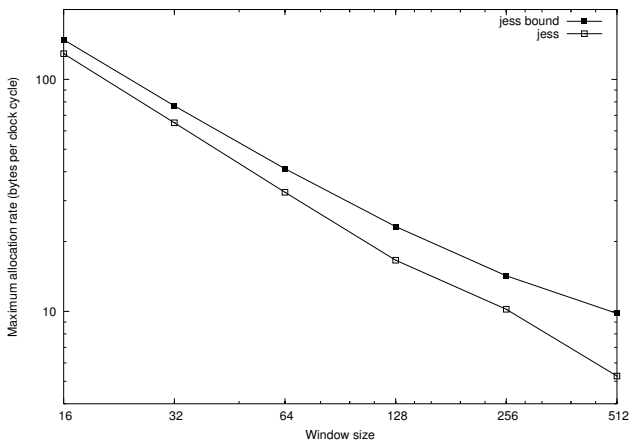

Figure 13. Comparison of bounded and actual maximum allocation rates for jess.

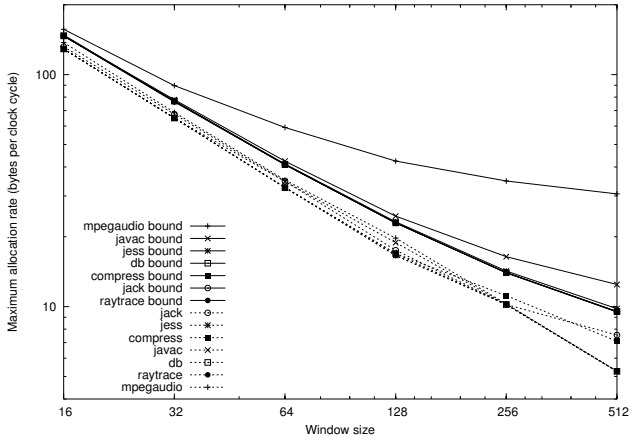

Figure 14. Comparison of bounded and actual maximum allocation rates for all benchmarks (both plotted together).

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 worst-case allocation rate is 5.8 times higher than what was seen.

Figure 15 shows this comparison over all benchmarks and window sizes.

5. Conclusion

We have provided a framework for determining maximum allocation rates and have applied this framework to some Java benchmarks. We have demonstrated that for our benchmarks, our statically-determined allocation rate is within a constant factor of the observed allocation rate. Whether or not this constant factor constitutes a reasonable upper bound is a subjective issue. The size of this factor will have an effect on the memory footprint and the **Minimum Mutator Utilization** (MMU) [6] 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 the allocation rate. 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.

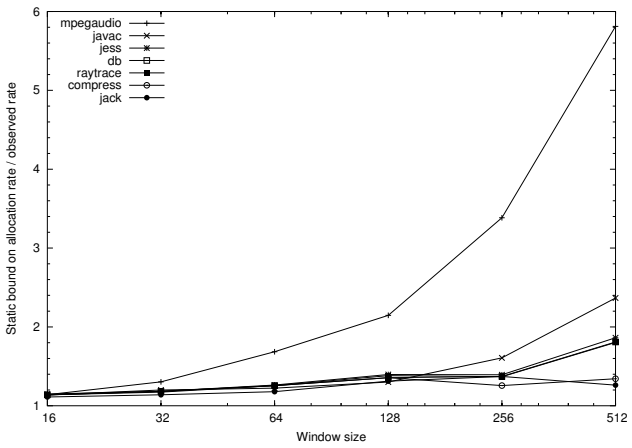

Figure 15. Comparison of bounded and actual maximum allocation rates for all benchmarks (static bound on rate / observed rate).

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 [11]. 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 [8], we plan to adapt our approach to handle variable allocation rates. This is especially important for programs in which not all methods are called by real-time threads. The maximum allocation rate within execution of real-time threads is the relevant statistic for the real-time collector.

We have not addressed threads in this paper. If threads can be interrupted at any point, then the windows we compute could overlap in any manner, which would make the allocation rate look unnecessarily high. We plan to make use of *safe points* (as in Jikes RVM) so that threads are interrupted only at predetermined points.

Acknowledgements

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 reviewers for their insightful suggestions.

References

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

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

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

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

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

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

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

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

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

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

[11] 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.

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

[13] 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.

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

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

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