Report Number: WUCSE-2004-45

2004-04-30

Automatic Determination of Factors for Real-Time Garbage Collection

Tobias Mann and Ron K. Cytron

Several approaches to hard, real-time garbage collection have been recently proposed. All of these approaches require knowing certain statistical properties about a program's execution, such as the maximum extent of live storage, the rate of storage allocation, and the number of non-null object references. While these new approaches offer the possibility of guaranteed, reasonably bounded behavior for garbage collection, the determination of the required information may not be straight forward for the application programmer. In this paper we present evidence suggesting that the necessary factors can vary widely over the program's execution, indicating that an automatic, phased approach may... **Read complete abstract on page 2.**

Automatic Determination of Factors for Real-Time Garbage Collection

Tobias Mann and Ron K. Cytron

Complete Abstract:

Several approaches to hard, real-time garbage collection have been recently proposed. All of these approaches require knowing certain statistical properties about a program's execution, such as the maximum extent of live storage, the rate of storage allocation, and the number of non-null object references. While these new approaches offer the possibility of guaranteed, reasonably bounded behavior for garbage collection, the determination of the required information may not be straight forward for the application programmer. In this paper we present evidence suggesting that the necessary factors can vary widely over the program's execution, indicating that an automatic, phased approach may be wise for a real-time collector. We present a static framework for determining some of the factors and present run-time statistics on Java benchmarks.

2004-45

Automatic Determination of Factors for Real-Time Garbage Collection

Authors: Mann, Tobias; Cytron, Ron K

April 30, 2004

Abstract: Several approaches to hard, real-time garbage collection have been recently proposed. All of these approaches require knowing certain statistical properties about a program's execution, such as the maximum extent of live storage, the rate of storage allocation, and the number of non-null object references. While these new approaches offer the possibility of guaranteed, reasonably bounded behavior for garbage collection, the determination of the required information may not be straightforward for the application programmer.

In this paper we present evidence suggesting that the necessary factors can vary widely over a program's execution, indicating that an automatic, phased approach may be wise for a real-time collector. We present a static framework for determining some of the factors and present run-time statistics on Java benchmarks.

Automatic Determination of Factors for Real-Time Garbage Collection[*]

Tobias Mann and Ron K. Cytron

Washington University in St. Louis, Department of Computer Science and Engineering, April 30, 2004

Abstract

Several approaches to hard, real-time garbage collection have been recently proposed. All of these approaches require knowing certain statistical properties about a program's execution, such as the maximum extent of live storage, the rate of storage allocation, and the number of non-null object references. While these new approaches offer the possibility of guaranteed, reasonably bounded behavior for garbage collection, the determination of the required information may not be straightforward for the application programmer.

In this paper we present evidence suggesting that the necessary factors can vary widely over a program's execution, indicating that an automatic, phased approach may be wise for a real-time collector. We present a static framework for determining some of the factors and present run-time statistics on Java benchmarks.

1 Introduction

Real-Time Specification for Java^{TM} (RTSJ) [3] has emerged as a viable standard for developing real-time systems in Java. Unfortunately, to obtain real-time guarantees, a program must not touch the garbage-collected heap and must instead use *scoped memories*. Use of such memories is difficult [5, 6] and may involve an unbounded amount of leaked (dead) storage [10]. While automatic techniques for using scoped memories have been developed [7, 11, 2], the best situation would result if the ordinary, garbage-collected heap could be used for real-time Java programs.

To realize that goal, garbage-collection must occur in real-time, with absolute guarantees on its performance and its effect on the application program (sometimes called the *mutator*). Several approaches to real-time garbage collection have been proposed, including Metronome [1], NewMonics PERC[1], and the Jamaica VM[2]. The guarantees associated with those collectors are dependent on certain program properties. Statistical bounds on those

[*]This work is supported by DARPA under contract F33615-00-C-1697

[1]`http:/www.newmonics.com/`

[2]`http://www.aicas.com/`

properties, currently provided by the application developer, must be specified in order for the collector to deliver its promised performance.

In fact, all methods of storage reclamation that can claim to be precise currently depend on information provided by the programmer:

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

Scoped memories: Because RTSJ places strict rules on the allowable references between scoped memories, and because violation of those rules causes an application to terminate (by exception), the programmer must understand scoping and lifetime issues of scope-allocated storage to make certain that the application contains only valid references.

Automatic collection: Approaches here are based on traditional garbage-collection methods [12]. As we describe in greater detail below, sophisticated information about a program's allocation, pointer-setting, and referencing behavior is required to obtain reasonable bounds on the collector's behavior.

2 Automatic Real-Time Garbage Collection

Of significant relevance to this paper is the specific information required to obtain real-time performance, as follows; the notation is reprised and adopted from papers on the Metronome collector [1]:

m is the maximum live storage (in bytes) of the application. In other words, the program requires at least m bytes to run with a perfect, continuously operating garbage collector. Determining m statically is undecidable. Even a dynamic approach to determining m [9, 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 programmers and those who execute Java applications know m for a given application. This follows from the fact that the size of the run-time heap must be specified when a Java program is executed. Failure to specify the heap size causes the program to execute with a heap of some default size, typically 16 Mbytes.

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 determined to be live. The cost of the marking phase, and so its length and the length of the entire garbage-collection phase, are thus dependent on the number of non-null references that can be discovered while marking live objects.

Thus, some real-time garbage collectors must know the number of non-null references to bind the time required for the mark phase.

Allocation rate: While the collector is operating, the program may continue to allocate objects. The number of such objects must be known, so that the extent of "floating garbage" can be determined, if such objects are allocated beyond the reach of the currently executing collector. If such objects can be collected by the current cycle of garbage collection, the number of such objects must still be known as it can affect the runtime of the collector.

For this reason, the following information is required concerning a program's allocation behavior:

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

Mutator and Collector time quanta: We assume a single-CPU system in which the application program (the *mutator*) and the garbage collector must alternate execution on the CPU. The following are therefore of interest:

$$\begin{aligned}
Q_T &= \text{mutator time quantum} \\
C_T &= \text{collector time quantum}
\end{aligned}$$

These refer to the smallest amount of non-preemptive execution time that the mutator and collector respectively are guaranteed.

P: is the *work-rate* of the collector (Storage/time unit)

Equations 1 and 2 show how this information is used by the Metronome collector to determine when to schedule a garbage collection cycle and to place an upper bound on the space requirement of the whole application.

The maximum amount of memory allocated during one collection cycle, e, is defined in Equation 1.

$$e = \alpha^*(\frac{m}{P} \cdot \frac{Q_T}{C_T}) \tag{1}$$

The expression $\frac{m}{P} \cdot \frac{Q_T}{C_T}$ is the amount of time that the mutator is allowed to execute during one collection cycle. Throughout this paper we will refer to this period of time as Δt. The garbage collector must be scheduled so that the amount of available memory never falls below e.

The total space requirement of the program:

$$s = m + ke \tag{2}$$

The k term in equation 2 refers to the number of garbage collection cycles that are needed to guarantee that the floating garbage is collected. In summary, while real-time collectors are an exciting development, they cannot operate without certain statistical bounds on some aspects of a program's behavior.

In this paper, we present analysis of Java programs—dynamic analysis as well as static analysis—to determine properties of interest to a real-time garbage collector. We focus in particular on the allocation rate of a program. Although the number of bytes allocated by a program is undecidable, the *maximum* allocation rate can be bound statically if the size of each allocation is known and if the time of each instruction is known. Section 4 presents our method for this computation.

3 Dynamic Approach

The allocation characteristics of applications have not been widely studied. To gain a deeper understanding of our problem domain and to get a sense of the results that we may expect from a static prediction we use dynamic techniques to study the allocation rate characteristics of benchmark programs for the following reasons:

1. Static analysis is conservative, and we are interested in knowing actually observed lower bounds on a program's maximum allocation rate.

2. Assumptions made in exant work speculate that a program's average allocation rate is similar to its allocation rate observed over relatively small windows of the program's execution. We show that for the benchmarks we studied, this is not the case. An implication of this discovery is the new merit that could be attributed to a garbage-collection scheme that operates in phases, changing its scheduling and resource consumption throughout a program's execution.

3. For some applications, dynamic measurements can suffice in providing bounds on the program's behavior, if the program is expected to behave similarly over many runs of the program.

Our approach measures the allocation rate of the program for a set of different window sizes (Δt) to analyze what effect Δt has on allocation rate. Specifically we are interested in seeing how the variability of the allocation rate changes with different Δt. In addition we want to see what effect the window has on the maximum allocation rate of the program.

3.1 Experiments

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

The information gathered by the instrumented JVM was processed offline to compute $\forall_t \ \gamma(t, \Delta t)$ for a range of different values of Δt. The data gathered provide us with the maximum allocation rate, as well as the allocation rate for any given part of the executing program.

3.2 Implementation

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

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

```
Queue Update
  $t_c \leftarrow$ the current time
  $|W| \leftarrow$ the size of the window for this queue
  $A.time \leftarrow$ the time of allocation A
  $queue.size \leftarrow$ the size of the queue in bytes
  $queue.front \leftarrow$ gets the front of the queue
  $queue.dequeue \leftarrow$ removes the front element

  $timeLimit \leftarrow t_c - |W|$
  $A_{old} \leftarrow queue.front$

  while $A_{old}.time < timeLimit$
    $queue.dequeue$
    $A_{old} \leftarrow queue.front$

  record $t_c$
  record $\frac{queue.size}{|W|}$
```

3.2.1 Implementation Analysis

The cost of each update to a queue is $O(k)$, where k is the number of dequeue operations. Each queue is updated $n + m$ times where n is the number of allocations of the program

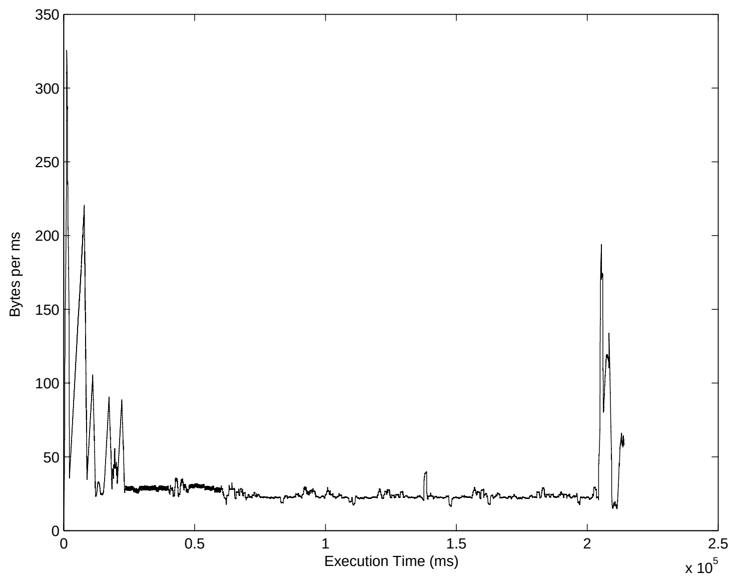

Figure 1: Allocation rate of jvm98 benchmark jack, as a function of execution time, $\Delta t = 1049ms$.

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

3.3 Results

The primary result of these experiments are depicted as graphs of the mutator's allocation rate for a given time window as a function of time. What is remarkable about these results is the high variability of the allocation rate for each of the benchmarks we tested. In particular all of the benchmarks exhibited shorter periods of time were the allocation rate was unusually high. An example of this is shown in Figure 1.

This is significant because, as previously mentioned, the only allocation rate information used by current real-time collectors is the maximum allocation rate. As we have seen, the maximum allocation rate of a mutator, for the window size considered, is not representative of the allocation rate throughout the execution of the program.

However, because the garbage collector must budget for the worst-case scenario one could argue that although the maximum allocation rate is an abnormality, budgeting for worst case necessitates the use of this value. This is a valid argument, but by observing that the maximum allocation rate of the mutator is dependent on the Δt used to measure the allocation rate, it becomes clear that the worst case will vary with the time alotted for one

Benchmark	Max Live	Coll. Rate	Δt	MMU
javac	34	39.4	707	0.446
jess	21	53.2	325	0.441
jack	30	57.4	429	0.441
mtrt	28	45.1	509	0.446
db	30	36.7	670	0.441

Figure 2: Time Windows for the Metronome Collector $Q_T = 10$, $C_T = 12.2$

complete collection cycle.

Using the maximum allocation rate to determine when to schedule the collector and the memory requirement of the program means that resources are wasted whenever the mutator does not exhibit the maximum rate. The average allocation rate of the program is calculated using a window size that is the entire length of the program, $\gamma(0,T)$. This means that as Δt increases we can expect the maximum allocation rate to approach the average allocation rate.

Ideally, the maximum rate should be equal to the average rate or, stated more clearly, we want the worst case to be as close to the average case as possible. To enable us to measure how close the worst case is to average we define the following metric, $\frac{\gamma(0,T)}{\gamma^*(\Delta t)}$.Where $\gamma^*(\Delta t)$ is the maximum allocation rate for a value of Δt. When this value is close to 1, the maximum rate provides a good approximation of the average. As this value gets further from 1 this assumption becomes more and more costly in terms of resources wasted.

The graph depicted in Figure 3 shows $\frac{\gamma(0,T)}{\gamma^*(\Delta t)}$ as a function of Δt. As expected, for all benchmarks tested this value is approaching 1 as Δt increases. As a frame of reference the Δt used in the experiments presented by Bacon et. al [1] are shown in Figure 2. Clearly for the window sizes normally used the maximum allocation rate is a poor estimate of the average with the aforementioned implication that resources are wasted.

In addition, Figure 4 shows an indirect benefit of an increased window size. Here **Minimum Mutator Utilization** (MMU) is graphed as a function of Δt. MMU is formally defined in equation 3. Informally it can be thought of as the minimum fraction of CPU time that the mutator is guaranteed during one collection cycle.

The MMU has been calculated using the following formula presented by Bacon et. al. [1]

$$MMU(\Delta\tau) = \frac{Q_T * \lfloor \frac{\Delta\tau}{Q_T + C_T} \rfloor + x}{\Delta\tau} \tag{3}$$

Where $\Delta\tau$ is the time window for which MMU is calculated, and Q_T and C_T are the mutator and collector time quanta respectively. $\Delta\tau$ is here $Q_T + C_T$. x is the remaining partial mutator quantum and is define in Equation 4.

$$x = max(0, \Delta\tau - (Q_T + C_T) * \lfloor \frac{\Delta\tau}{Q_T + C_T} \rfloor - C_T) \tag{4}$$

For large values of $\Delta\tau$, this expression reduces to

$$\lim_{\Delta\tau\to\infty} MMU(\Delta\tau) = \frac{Q_T}{Q_T + C_T} \tag{5}$$

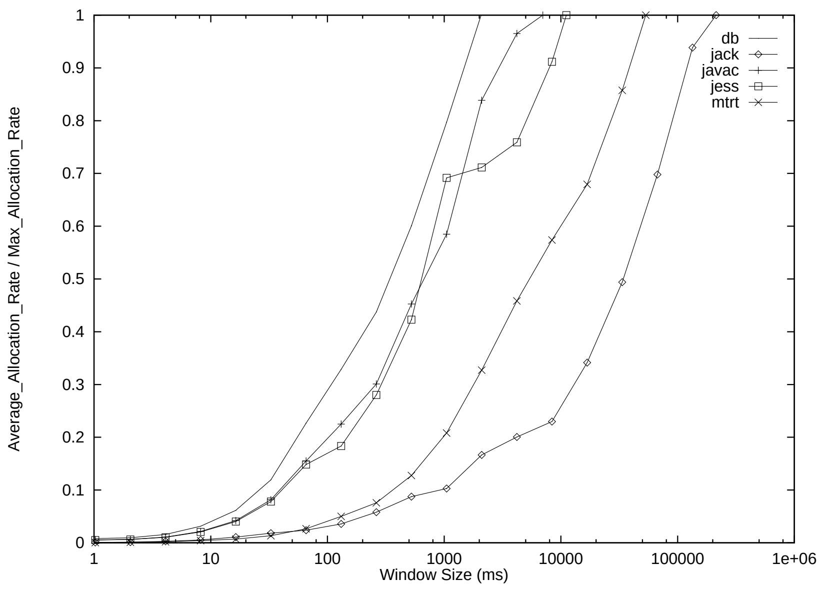

Figure 3: $\frac{\gamma(0,T)}{\gamma^*(\Delta t)}$ vs. Δt. This graph shows that as Δt increases, assuming that the allocation rate is $\gamma^*(\Delta t)$ becomes less costly for any given point in the programs execution. The reason for this is that $\gamma^*(\Delta)$ approaches $\gamma(0,T)$

Hence, we expect that as Q_T increases, thus increasing Δt and $\Delta\tau$, MMU will approach 1. This is an obvious observation since making the mutator's execution time per collection cycle equal to the entire execution time of the program is synonymous to running the program with out any garbage collection. Although the observation is obvious it does highlight an additional benefit to increasing Δt.

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

The collector must be scheduled so that the amount of memory available when collection starts makes it impossible for the mutator to deplete the amount of available memory during the collection cycle. If this decision is based on the assumption that the mutator allocates memory at the maximum allocation rate of the program then, on average, the collector will be scheduled prematurely. If the work of the collector is mostly dependent on amount of garbage being collected then this would not be significant because the collector would perform less work during each cycle.

However, the work of a mark-sweep collector is dependent on the amount of live memory that needs to be processed during the marking phase. This means that scheduling the collector more frequently will not have a considerable affect on the amount of work performed during each collection cycle. Consequently, additional collection cycles results in the collector

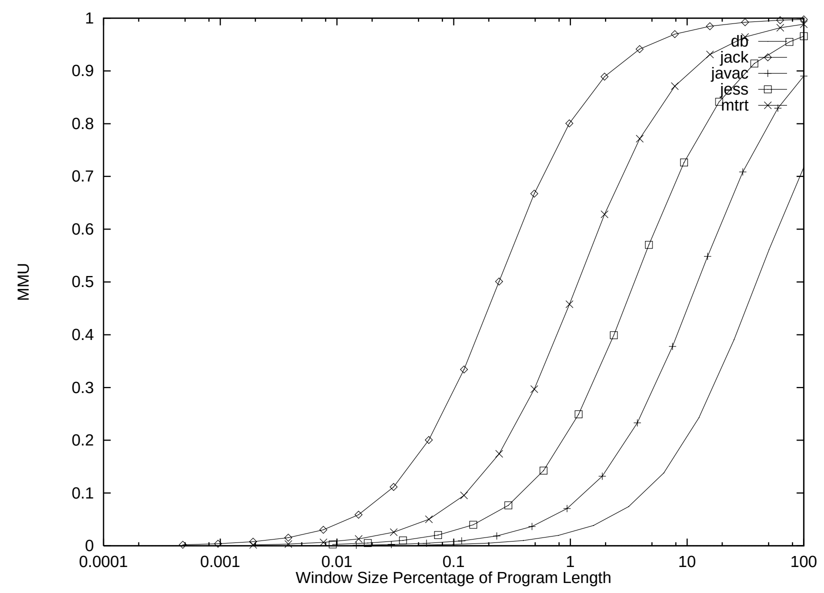

Figure 4: MMU vs. $\frac{\Delta t}{T} * 100$ Here we see experimental data depicting the relation ship between MMU and Δt described in equation 5

needlessly occupying CPU resources.

There are two approaches to solving this problem, and both will require knowing more about the allocation rate of the mutator than simply its maximum allocation rate. Either the collector can adapt its scheduling policy as the allocation rate changes and execute less frequently during periods of low allocation rates, or it can adapt the Δt used to the change in allocation rate. As mentioned previously increasing Δt will increase the amount of memory needed by the program. However, because the program is allocated enough heap memory to handle the programs maximum allocation rate there is room for increasing Δt when the program does not exhibit maximum allocation rate.

Out of these two approaches we prefer the latter one because of the fact that increasing Δt also increases MMU during that part of the execution. As shown in Figure 3, $\Delta t < \Delta t_1$ implies $\gamma^*(\Delta t) < \gamma^*(\Delta t_1)$. This means that for all periods of execution time where $\gamma(t, \Delta t) < \gamma^*(\Delta t)$ we can safely increase Δt and thus achieve a higher MMU than what has been previously reported, [1], without increasing the resources allocated to the program.

To be able to harvest these advantages we require that the garbage collector can adapt to the changes in the allocation rate of the mutator. This means that the collector must be able to predict future allocation rates of the mutator. In the next section we present a static framework for allocation rate determination that can be used to give the collector this prediction ability.

4 Static Approach

The problem with a dynamic approach to determine the allocation rate of a program is that it cannot provide any guarantees. The maximum allocation rate that is found dynamically during one run of a program may not be the maximum possible allocation rate that the program can exhibit. Basing parameters for the collector on any number of actual runs could underestimate the maximum allocation rate of the program. A static approach to allocation rate determination is needed.

The problem of statically predicting a mutator's allocation rate can be formulated as a dataflow problem [8]. The dataflow graph consists of nodes and edges and is denoted $G_{df}(N_{df}, E_{df})$ where N_{df} and E_{df} are the nodes and the edges in the graph. Each node in the graph represents an instruction in the program that the graph depicts. Each edge leaving a node points to an instruction that may execute just after this instruction. For our purposes each instruction is either an allocation or a non-allocation. The most trivial way to represent this is using one bit. When the bit is set the instruction is an allocation, when it is not then the instruction is a non-allocation. Thus, the solution to this dataflow problem will be represented as a bit-vector where the length of the bit-vector is the window size (Δt) considered. Here Δt is measured in number of instructions rather than execution time.

In addition to G_{df} we must provide further formal definitions of this dataflow problem to be able to reason about it. First, G_{df} is augmented as is customary with a *stop* and *start* node and an edge $(start, stop)$ connecting the start node with the stop node in the implied direction. Second, the meet lattice must be defined. The meet lattice is used to combine information from several paths that are converging at a node. It is defined as follows:

$$L = (A, T, \perp, \leq, \bigwedge)$$

- A is the domain of the dataflow problem, the set of bit-vectors

- T and $\perp$ is commonly called top and bottom. Top is a bit-vector consisting of only 0's while bottom is a bit-vector of only 1's

- $\leq$ is called reflective partial ordering. This operator is interpreted as if $a \leq b$ the a is no better than b.

- $\bigwedge$ is the meet operator. Here this operator is the bitwise or of its bit-vector operands.

$\forall a, b \in A$ the following must hold.
$a \leq b$ implies $a \bigwedge b = a$
$a \bigwedge a = a$
$a \bigwedge b \leq a$
$a \bigwedge b \leq b$
$a \bigwedge T = a$
$a \bigwedge \perp = \perp$

We also need to define a transfer function, f. Each node $n \in N_{df}$ uses f to transform its input to output. $f(IN) = OUT$ where IN is the input to the node. The input is generated by taking the meet of all inputs into n. OUT is the output of n generated by f. To guarantee

that our G_{df} converges we require f to be monotone. Monotoneicity is defined as follows:
$(\forall x, y)x \leq y \rightarrow f(x) \leq f(y)$
The proof that dataflow frameworks with monotone transfer functions must converge is beyond the scope of this paper, see [8] for details.

For our purposes we want the transfer function at node n to update the input bit-vector so that the output includes the instruction represented by n. This is achieved by simply shifting in the bit representing allocation (1) or non-allocation (0) into the bit-vector. This transfer function satisfies all constraints that we have discussed.

So far we have shown a solution that is guaranteed to converge but we have not show how to evaluate the framework. Ideally we would like our data flow framework to compute the **meet-over-all-paths** (MOP) solution. The MOP solution would calculate the allocation rate for possible paths of execution of the program. Obviously this is not feasible. Instead we use a forward iterative solution. This iterative solution can be show to produce a result that is no worse than the MOP solution, see [8], provided that the dataflow framework is distributive. A dataflow framework is distributive if:
$(\forall a, b \in A)f(a \bigwedge b) = f(a) \bigwedge f(b)$

The meet property of our framework gives us:
$a \bigwedge b \leq a$
$a \bigwedge b \leq b$

Monotonicity gives us:
$f(a \bigwedge b) \leq f(a)$
$f(a \bigwedge b) \leq f(a)$

Combining these two properties gives us:
$f(a \bigwedge b) \bigwedge f(a \bigwedge b) \leq f(a) \bigwedge f(b)$
$f(a \bigwedge b) \leq f(a) \bigwedge f(b)$

Thus, the dataflow framework we have devised gives us an iterative solution that is no worse than the MOP solution.

However, as with most static analysis this analysis will err on the side of caution. It is overly conservative. This is an artifact of how the meet operator combines information from several paths of execution. To show this consider a node n. Let n represent a non-allocation instruction. There are two inputs edges a and b leading into n. Further assume that at the values of a and b are:
$a = 0100$ meaning that out of the last four instructions the third most recent one was an allocation
$b = 0001$ meaning that out of the last four instructions the most recent one was an allocation

The meet operator will produce the following result:
$a \bigwedge b = 0100$ OR $0001 = 0101$

This will be transformed by the transfer function:
$f(0101) = 1010$

Thus the output of node n would indicate that out of the past four instructions, two where allocations. However, since none of the inputs into n contained more than one allocation and n itself is a non-allocation, this estimate is obviously overly conservative.

Thus far the described static approach for determining allocation rates have not yet been fully implemented and tested. However our dynamic analysis has given us an idea of the results that we may expect from performing this static analysis.

5 Conclusion

We have studied the allocation rate of programs dynamically, and have determined that there is great variation in the allocation rate. As such, the choice of window size for the collector is an important issue. Longer windows tend to decrease the significance of allocation spikes, lowering the apparent maximum allocation rate. This in turn can reduce the storage needed in reserve during a collection cycle. However, longer windows also increase the storage requirements once the maximum allocation rate is fixed. The tension between longer and shorter windows is application-specific and merits further study.

Moreover, the variance in maximum allocation rate throughout a program indicates the potential of an adaptive approach based on phases of the application's allocation behavior.

Finally, we propose a static method for measuring the maximum allocation rate of a program given a window size. This is useful, in that most problems of this form are undecidable. One cannot determine the number of bytes allocated nor can the running time of a program be computed statically. However, the number of bytes allocated per fixed window size can be bounded from above by making conservative estimates of time and space where necessary. The actual accuracy of this upper bound must be explored, and that is a subject of future work for us.

References

[1] 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.

[2] William S. Beebee, Jr. and Martin Rinard. An implementation of scoped memory for real-time java. In *EMSOFT*, pages 289–305, 2001.

[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] Angelo Corsaro and Ron K. Cytron. Efficient Memory-reference Checks for Real-Time Java. In *Proceedings of the 2003 ACM SIGPLAN conference on Language, Compiler, and Tools for Embedded Systems*, pages 51–58. ACM Press, 2003.

[6] Angelo Corsaro and Ron K. Cytron. Implementing and optimizing real-time java. In *Proceedings of The 11th International Workshop on Parallel and Distributed Real-Time Systems*. IEEE, 2003.

[7] Morgan Deters, Nicholas Leidenfrost, and Ron K. Cytron. Translation of Java to Real-Time Java using aspects. In *Proceedings of the International Workshop on Aspect-Oriented Programming and Separation of Concerns*, pages 25–30, Lancaster, United Kingdom, August 2001. Proceedings published as Tech. Rep. CSEG/03/01 by the Computing Department, Lancaster University.

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

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

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

[11] F. Vivien and M. Rinard. Incrementalized pointer and escape analysis. In *Proceedings of the SIGPLAN '01 Conference on Program Language Design and Implementation*, Snowbird, Utah, June 2001.

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