Automated Discovery of Scoped Memory Regions for Real-Time Java *

Morgan Deters
mdeters@cs.wustl.edu

Ron K. Cytron
cytron@cs.wustl.edu

Department of Computer Science
Washington University
St. Louis, MO 63130

ABSTRACT

Advances in operating systems and languages have brought the ideal of reasonably-bounded execution time closer to developers who need such assurances for real-time and embedded systems applications. Recently, extensions to the Java libraries and virtual machine have been proposed in an emerging standard, which provides for specification of release times, execution costs, and deadlines for a restricted class of threads. To use such features, the code executing in the thread must never reference storage that could be subject to garbage collection. The new standard provides for region-like, stack-allocated areas (scopes) of storage that are ignored by garbage collection and deallocated *en masse*. It now falls to the developer to adapt ordinary Java code to use the real-time Java scoped memory regions.

Unfortunately, it is difficult to determine manually how to map object instantiations to scopes. Moreover, if ordinary Java code is modified to effect instantiations in scopes, the resulting code is difficult to read, maintain, and reuse. Static analysis can yield scopes that are correct across all program executions, but such analysis is necessarily conservative in nature. If too many objects appear to live forever under such analysis, then developers cannot rely on static analysis alone to form reasonable scopes.

In this paper we present an approach for automatically determining appropriate storage scopes for Java objects, based on *dynamic analysis*—observed object lifetimes and object referencing behavior. While such analysis is perhaps unsafe across all program executions, our analysis can be coupled with static analysis to *bracket* object lifetimes, with the truth lying somewhere in between. We provide experimental results that show the memory regions discovered by our technique.

Categories and Subject Descriptors

D.3.4 [**Software**]: Processors—*optimization*

*Sponsored by DARPA under contract F33615-00-C-1697

ISMM'02, June 20-21, 2002, Berlin, Germany.

General Terms

Languages, Algorithms, Experimentation, Performance

Keywords

Memory management, real-time Java, regions, garbage collection, trace-based analysis

1. INTRODUCTION

The Java[1] programming language [1] has enjoyed wide acceptance, from applications deployed on large servers to those deployed on small, hand-held devices. However, the real-time community has largely ignored Java until recently, when an experts group issued the Real-Time Specification for Java [5] (RTSJ) document.

The RTSJ provides extensions to Java in support of real-time programming. These extensions are not accomplished by changes to the base language; instead, new class libraries and Java Virtual Machine (JVM) extensions are introduced, so that compilers and other program-development tools are largely unaffected by the mechanisms that pertain to real-time programming.

The RTSJ touches many areas of the Java programming language, including threads, priorities, synchronization, and exceptions. In this paper, we focus on that area of the RTSJ concerned with storage management. We present an algorithm and experimental results for automatically assigning objects to RTSJ scopes for ordinary Java programs, using the RTSJ notion of *scoped memory* as explained in detail in Section 3.

The main advantage of using scoped memory in an RTSJ-compliant JVM is the reduced reliance on the garbage collector; objects allocated from scoped memory regions are not subject to garbage collection and are instead collected when the region is closed.

This paper's main contributions are to describe the automated harvesting of potential RTSJ scoped memory areas in ordinary, single-threaded Java programs with the flat Java memory model and to provide some early evaluation of the objects that can take advantage of these memory areas instead of relying upon the standard garbage collected heap.

While this paper and the resarch it describes are concerned primarily with Java programs, we believe our technique is applicable to the more general problem of assigning objects to stack frame-associated memory regions. Further, our assumption of a single-threaded target program is at present a limitation of our technique; this assumption can, however, be lifted, and a discussion of this extension is included.

[1]Java is a registered trademark of Sun Microsystems, Inc.

```
class Example {
  void zero() {
    RefObject E = new RefObject();     *
    one(E);
  }

  void one(RefObject E) {
    Object D = two(E);
  }

  Object two(RefObject E) {
    RefObject D = new RefObject();     *
    Object F = new Object();           *

    RefObject A = new RefObject(D),    *
              B = new RefObject(E),    *
              C = new RefObject(F);    *

    D.ref = E;

    refTemporarily(A, B);
    refTemporarily(B, C);
    refTemporarily(C, A);

    Object G = new Object();           *
    refTemporarily(E, G);

    return D;
  }

  void refTemporarily(RefObject source,
                      Object target) {
    source.ref = target;
    source.ref = null;
  }
}

class RefObject {
  Object ref;
  RefObject() { }
  RefObject(Object o) { ref = o; }
}
```

(a)

```
import javax.realtime.*;

class RTExample {
  void zero() {
    final RefObject E = new RefObject();
    // memory area of 1 kilobyte
    new LTMemory(1024,1024).enter(
      new Runnable() {
        public void run() {
          one(E);
        }
      });
  }

  void one(final RefObject E) {
    final ScopeReturnValue returnObj =
      new ScopeReturnValue();
    // memory area of 1 kilobyte
    new LTMemory(1024,1024).enter(
      new Runnable() {
        public void run() {
          returnObj.set(two(E));
        }
      });
    Object D = returnObj.get();
  }

  Object two(RefObject E) {
    RealtimeThread thisThread =
      RealtimeThread.getRealtimeThread();
    ScopedMemory thisScope =
      (ScopedMemory)thisThread.getMemoryArea();
    MemoryArea
      outerScope = thisScope.getOuterScope(),
      outerOuterScope =
        ((ScopedMemory)outerScope).getOuterScope();
    RefObject D = (RefObject)
      outerScope.newInstance(RefObject.class);

    Object F = new Object();

    RefObject A = new RefObject(D),
              B = new RefObject(E),
              C = new RefObject(F);

    D.ref = E;

    refTemporarily(A, B);
    refTemporarily(B, C);
    refTemporarily(C, A);

    Object G =
      outerOuterScope.newInstance(Object.class);
    refTemporarily(E, G);

    return D;
  }

  void refTemporarily(RefObject source,
                      Object target) {
    source.ref = target;
    source.ref = null;
  }
}

class ScopeReturnValue {
  Object obj;
  void set(Object o) { obj = o; }
  Object get() { return obj; }
}

// RefObject definition remains the same
```

(b)

Figure 1: (a) A Java program and (b) its RTSJ version. Allocation sites are marked in (a).

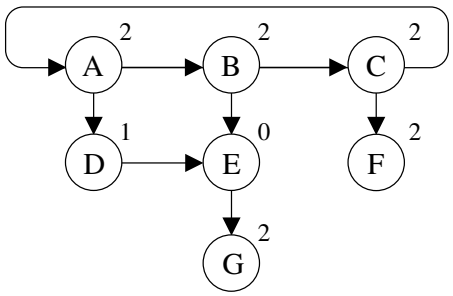

Figure 2: The *doesReference graph* for the program of Figure 1(a), vertices augmented with object liveness information.

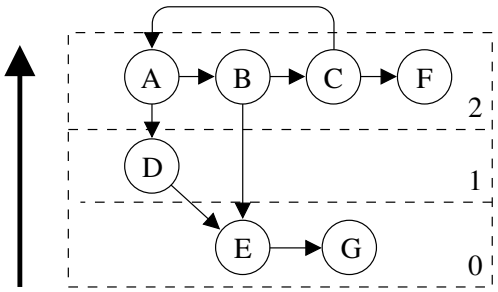

Figure 3: Optimal RTSJ scope assignment for the program of Figure 1(a) with references superimposed.

The paper is organized as follows. Section 2 introduces a motivating example used throughout. Section 3 introduces the RTSJ and describes the setting for our problem in detail. Section 4 describes our approach, presenting an algorithm for scope assignment. Section 5 presents some experimental results concerning the assignment of objects to scopes in standard benchmarks, Section 6 identifies some related work, and Sections 7 and 8 offer concluding remarks and a discussion of planned future work.

2. MOTIVATION

We motivate the problem we study using the example shown in Figure 1(a), which defines methods `zero`, `one`, and `two`—methods that are invoked at depth 0, 1, and 2 in the call-stack, respectively. The example also contains 7 *allocation sites* (`new` statements). The problem at hand is to map each object instance o to a call-stack frame $F \in \{0, 1, 2\}$, subject to the following criteria:

1. Object o is known to be dead after F is popped;

2. Any object that could refer to object o is mapped to a frame popped no sooner than F.

With regard to criterion 1, our problem is analogous to escape analysis [28, 8, 4] or to the "regions" problem [26, 25]. As explained in Section 3, criterion 2 is introduced to ensure safe pointer-accesses in accordance with the RTSJ.

The ultimate outcome of our analysis is a program conforming to the RTSJ;[2] Figure 1(b) shows this intended result. Each instantiation site is modified to allocate its object from a storage area, in a manner consistent with the RTSJ.

To respect criterion 1, we first run the program in Figure 1(a) and observe the following object behavior:

Object	Instantiated in frame	Becomes collectible
`A`	2	when frame 2 pops
`B`	2	2
`C`	2	2
`D`	2	1
`E`	0	0
`F`	2	2
`G`	2	2

[2]This paper focuses primarily on the detection of scoped memory areas for real-time Java, not the instrumentation of the original Java code to use them. A discussion of our approach is provided in Section 4.

For the purposes of scope-creation for RTSJ, it suffices to assume that an object cannot die in a frame above its birth-frame. Analysis has been considered [23] to move allocation sites like these up the call-stack to decrease the lifetime of the instantiated object.

Next, to ensure legal references (criterion 2 above), we maintain a graph of inter-object references observed during runs of the program; in this *doesReference graph*, shown in Figure 2, each object is represented as a vertex, and an edge is placed between v and w if object v stores a reference to object w at some point when the program is run. We augment the graph with the collectibility information from the above table, as shown in the figure.

The following observations are key to solving our problem.

- Objects `A`, `B`, and `C` are involved in a *cycle* in the *doesReference* graph. Based on criterion 2 above, these objects must be mapped to the same frame.

- Object `A` references `D`, implying by criterion 2 that `D` must be mapped to a frame at or below the frame associated with `A`. The same can be said for objects `B` $\rightarrow$ `E`, `C` $\rightarrow$ `F`, and also `D` $\rightarrow$ `E` and `E` $\rightarrow$ `G`.

- Object `D` is instantiated in method `two` (at frame 2) but does not actually become dead until frame 1 pops, because it is returned to the calling method.

- Object `E` is allocated in frame 0, and must therefore live forever.

One possible solution to the above problem is to allocate *all* objects in frame 0's memory region—this is the last frame to be popped. This clearly satisfies the above criteria, because all objects will live for the duration of the entire program. However, we seek a solution with the objective of minimizing the lifetime of each object, so as to reduce the program's storage footprint.

The best RTSJ solution for this problem is depicted in Figure 3, which corresponds to the scopes used in the code of Figure 1(b). Inter-object references have been superimposed in the figure; when visualized in this manner, object references are required by criterion 2 to always point down (toward the base of) the stack.

Object `E` is around for the entire lifetime of the program and thus is not dead until frame 0 is popped. Object `D` cannot die until frame 1 is popped. While objects `A`, `B`, `C`, and `F` could be allocated in frame 1's memory region, that would unnecessarily increase their lifetimes. Instead, they can be allocated in the region at frame 2. Object `G` is instantiated during the execution of frame 2, but must be allocated in frame 0 because of the reference from object `E`, even though the reference was temporary.

3. REAL-TIME JAVA

The work presented in this paper is driven mainly by the RTSJ specification, which itself was motivated by the desire to make Java attractive to developers of real-time applications. While the RTSJ could be subject to criticism, for the purposes of this paper we assume its views concerning how Java can be used while still providing real-time guarantees for critical tasks. In this section we discuss the relevant portions of the RTSJ for our work.

3.1 The RTSJ meets garbage collection

The RTSJ[3] [5] covers many issues related to real-time programming, but for the purposes of this paper we concentrate on storage, threads, and their relationship to the garbage collector.

Although research has addressed the viability of real-time garbage collection to some extent [21, 7], it is still questionable [22] as to whether a collector and allocator can *always* provide storage for an allocation request (*i.e.*, a `new` command) in time proportional to the size of the request. Consider the case where the heap is full except for a few dead objects. All exact collectors require some form of marking phase to discover objects that are still live. Generational collection [29] can limit the extent of marking but at the expense of ignoring potentially dead objects in older generations. The result is that the time taken to perform a `new` command cannot be reasonably bounded if garbage collection is required to liberate storage to satisfy the request.

Such bounds are essential if the thread requesting the storage must adhere to an execution-cost budget. In the RTSJ, tasks' budgets are submitted to the scheduler so that the *feasibility* of scheduling the tasks can be determined. Currently, such decisions are based on Rate-Monotonic Analysis (RMA) [18]. If the garbage collector causes a thread to experience unbudgeted delay, then that task as well as others could miss their deadlines.

As a result, the RTSJ insists that threads for which scheduling criteria are taken seriously—the so-called `NoHeapRealTimeThreads` (NHRTTs)—cannot allocate storage in the garbage-collected heap. In fact, such threads cannot even *reference* an object in the garbage-collected heap, since it is possible that heap objects could be locked during a garbage-collection cycle, causing the referencing thread to experience unbudgeted delay.

3.2 Uncollected storage areas

The RTSJ defines two kinds of storage areas that can used for allocation and object-references from NHRTTs:[4]

`ImmortalMemory` is a singleton memory area all of whose component objects' lifetimes are that of the entire program.

`ScopedMemory` is an area of storage whose constituent objects' lifetimes are collectively tied to a particular execution scope.

Both of the above storage areas are completely ignored by garbage collection in an RTSJ-compliant JVM; objects allocated in the singleton immortal memory area are uncollectable, and objects allocated in a `ScopedMemory` area are collected *en masse* when the threads that could access the memory area exit the associated scope.

To associate a `ScopedMemory` area to a particular execution scope and be permitted to allocate from it, an RTSJ program calls the `enter` method on a `ScopedMemory` instance, as in the code of Figure 1(b). The `enter` method takes a single parameter—a `Runnable` object whose `run` method becomes the execution scope for the scoped memory area. A reference count of threads currently accessing a given scoped memory area is kept by the JVM, and when all of these threads exit their scopes' top-level `run` methods, the objects within the memory area are known to be collectible—no live objects in the system can reference them because of particular reference constraints established by the RTSJ and shown here in Table 1.

	Reference to Heap	Reference to Immortal	Reference to Scoped
Heap	Yes	Yes	No
Immortal	Yes	Yes	No
Scoped	Yes	Yes	Yes*

* If to an object in the same or outer scope

Table 1: References between storage areas [5].

In accordance with Table 1, code executing in an RTSJ thread may not access any storage that could be reclaimed while the thread is still accessing it. It is safe for any storage area to access immortal memory. Also, a thread can access any scope that it has *entered*—any scope whose reference count was bumped on behalf of that particular thread. To guard against potential "dangling references" to objects that have been reclaimed, inter-scope references are regulated.

Scoped storage can contain references to the garbage-collected heap, but if an NHRTT tries to access any cells in the heap, an exception is thrown.[5]

3.3 Nested storage scopes and instantiations

While thread acquisition is the mechanism for bumping the reference count of a `ScopedMemory` area, it is important to understand the intent of `ScopedMemory` in the RTSJ. Garbage collection is avoided in such areas, so it is essentially left to the programmer to determine the effective lifetime of all instantiated objects and to allocate such objects in the appropriate scope (respecting criteria 1 and 2 discussed in Section 1). If `ScopedMemory` creation and deletion were free, then ideally a program would allocate them to hold objects with coinciding lifetimes: such objects are then freed collectively when the scope is exited and its memory area is deleted.

This bias towards "many scopes" may encounter performance difficulties if `ScopedMemory` creation or deletion is expensive. In this paper, we take the view that more, precise scopes are better than fewer, conservative scopes. We draw from research on *regions* [26, 25, 13] and tie a scoped area's life to the pushing and popping of stack frames in response to method invocations and exits. Some regions work is more general, as the beginning and ending of a frame could correspond to any reasonable pair of stack frames. For our present work, we tie the life of a scope to that of a single stack frame. Thus, when any method is entered, we can open a scope, and that scope exists for the lifetime of that method's frame. When the method returns, and its associated frame is popped, the storage scope is deleted.

At an object allocation site (a `new` command), we rewrite the instantiation to cause the instantiated object to be allocated n frames

[3]The RTSJ specification is presently undergoing substantial revision, but the mechanisms described in this section will exist in some form in subsequent releases of the specification. The description given here is according to the published RTSJ 1.0 [5].

[4]We do not discuss the `PhysicalMemory` classes of the RTSJ here; such classes permit access to specific locations of memory and can also be scoped or immortal, but they are not used in our work.

[5]The RTSJ does offer a weaker form of a real-time thread, `RealtimeThread`, that can access such cells, but for which no scheduling guarantees can be made because of the issues discussed in Section 3.1.

back in the call stack, where the value of n is determined by the analysis we detail in Section 4. Returning to the example in Section 2, variables `A`, `B`, `C`, and `F` would be allocated in their own allocating method's associated `ScopedMemory` area. Since object `D` is returned to its allocating method's caller, however, the instantiation of `D` must be moved back one frame, so its allocation site can be rewritten as described in Section 4 to allocate `D` in method `one`'s stack frame. Similarly, object `G` must be moved back two frames due to the reference of `G` by `E`.

The `new` command still executes where it was originally specified, but code is inserted to cause the storage for that allocation to be obtained from the outer method's scope. Object `E` is allocated in frame 0, as specified in the original program.

3.4 Issues in using scoped storage

Considering the rules of Table 1, we observe that:

- Marking up the source code for a class with explicit scope-allocation gestures (as in Figure 1(b)) will hamper readability and reuse of that class.
- Library classes provided by different vendors often coexist in a system in which they must interoperate; to promote separation of concerns, portability, and reusability in object-oriented systems, these separate products should not be required to know of and react to each other's individual memory requirements.

Furthermore, it is often difficult to determine the best scopes for object instantiation for the following reasons:

- The answer depends on how objects can reference each other and the effective lifetime of objects. As shown in Figure 1(a), object references need not persist to affect object allocation. The references from `A`, `B`, and `C` are all temporary, but the programmer who uses the RTSJ must ensure that each such access is safe according to the rules given in Table 1.
- Similarly-lived objects often unrelated in design *accidentally* become collectible at the same time due to their placement in other, aggregate structures or the behavior of a common thread that uses them.
- Java code is generally not written with object deallocation in mind (as code in a non-garbage collected language like C++ may be), and it is therefore not always reasonable to pick out places in the code where finalization and deallocation can occur.
- Allocation and deallocation events may not be clearly or inherently scoped; two objects of a given type or instantiated in a similar way may be live a very different amount of time. Standard library utilities (*e.g.*, Java's `Collection` classes) may be used in different ways with different associated liveness.

Based on these observations, we next present our solutions for determining scopes for objects in Java programs and ideas for transforming them into scoped memory-aware RTSJ programs.

4. APPROACH

Our approach for translating a Java program P into a RTSJ-compliant, scoped memory-aware program P', is based on two kinds of analysis. In particular, for a given object instance x, we seek to answer these two questions, analogous to the two criteria introduced in Section 2:

1. What is the lifetime of x?

   Based on this information, we can map x's lifetime to the stack frames F_P of program P. In this way, we ensure that x's allocation in P' occurs prior to its use and that x is collected (together with other objects in the same scoped memory area) only after it is dead.

2. What other objects must have access to x?

   With knowledge of those objects' lifetimes, we can ensure that x is allocated in a scope that does not violate the scope-referencing rules of the RTSJ, as explained in Section 2.

The results of the above analyses are represented in a *doesReference graph*, similar to the one shown in Figure 2. Once that graph has been determined, the algorithm presented later in this section can be applied to obtain the optimal scope assignments of Figure 3. Before discussing our scope-assignment algorithm, we explore possible choices for the structure and contents of a *doesReference* graph.

4.1 Analysis techniques

Analysis of object lifetime and object references can be carried out using one of the following techniques:

Static: Compile- or load-time analysis could examine the instructions of a program to determine lifetime [28] and potential storage-referencing behavior [17, 15, 12, 27].

The information obtained in this manner would hold over *any* execution of the program. Unfortunately, such information is necessarily conservative, as the problems defined above are undecidable in general.

Dynamic tracing: Traces from one or more runs of the program could be examined to determine the lifetime of objects and their storage-referencing behavior.

While this information is precise for a given set of runs, it may not represent the program's behavior in general. Thus, such information is necessarily imprecise; if programs are adapted to RTSJ based only on dynamic trace information, then the criteria listed above and those introduced in Section 2 could potentially be violated.

As there is no "escape" mechanism for objects once they have been allocated from a region, an adaptive, runtime approach to scoped region assignment would result in the collection of objects with outstanding references or in objects unnecessarily existing for the entire execution of the program.

The *truth* concerning a program's behavior lies somewhere between the static answer (which holds over *all* executions) and the dynamic tracing answer (which holds only over an observed set of executions). For this reason it is potentially useful to consider an approach combining the two into a single, unified approach; in our current research—and therefore in this paper as well—we are pursuing answers based on *dynamic, trace-based* analysis rather than static analysis. Our reasons are as follows.

- Static analysis may turn out to be overly conservative, to the point of not providing useful results for the real-time applications developer. For example, if all objects appear to be immortal, then the resulting RTSJ program will essentially preallocate all objects in one global scope.
- Although the answers don't hold for all executions, the results based on dynamic analysis can inform static analysis

on the distance between its solution and some solutions that hold for particular executions. If that distance is small, then static analysis has worked well.

- If static analysis turns out to be overly conservative, then the developer will inevitably be involved in the ultimate decision about the program's scopes [3]. Our work can serve to validate or invalidate such a manual scope assignment based on actual executions of a program.

The unifying principle between static and dynamic analysis for the problem we consider is the *doesReference graph*, an example of which is shown in Figure 2. For dynamic analysis, this graph shows reference behavior observed in an actual run of a program. Static analysis could at best produce a "may reference" graph, which would indicate potential referencing behavior, but would suffer imprecision because of the uncertainty of aliasing and other phenomena that must be estimated in a static analysis approach to the problem. Further, with delayed class loading and reflective calls, it can be difficult to make strict static guarantees for production Java code, since the conservative estimate reached is often far too conservative: a static analysis tool may not be able to determine the set of methods called by a reflective invocation, for example, and thus must either assume that any method in the universe may be called or relax its guarantees of accuracy.

For these reasons we use dynamic analysis in determining the interconnections between elements of the *doesReference* graph. If each vertex of the graph is a set of objects (or a characteristic that defines a set of objects), as we will discuss below, then we admit an edge $[u, v]$ to the *doesReference* graph whenever an object belonging to u stores a reference to an object belonging to v.

4.2 Vertex granularity of *doesReference*

When fully built, the *doesReference* graph will influence object instantiation so that the safe memory accesses required by the RTSJ (and shown here in Table 1) are respected. Before building this graph, though, we must define the granularity of the vertices of the *doesReference* graph, which could be one of the following:

- A vertex could represent all objects instantiated directly by a given method. This would simplify expressing the results of our analysis, since each method could enter the appropriate scope upon invocation and exit the scope prior to returning. The resulting graph will merge the behavior of objects of different types and instances. For the purposes of this paper, we would like to obtain a more precise result.

- A vertex could represent all objects of a given type (class) in the program. If so, then the edges of the *doesReference* graph would indicate referencing behavior over all instances of objects of that type. While such a graph might contain relatively few vertices, the resulting graph may be unnecessarily conservative. For example, if any `String` object must live forever, then so would all instances of `String` according to such a graph.

- A vertex could represent a dynamic instance of an object; this was the approach used to construct the *doesReference* graph of Figure 2. This way, each `String` object instantiated at runtime would be represented by its own vertex. The resulting graph would have many vertices, but the behavior of each instantiation is nicely distinguished.

  Unfortunately, there is then the problem of recognizing such dynamic objects across multiple analysis runs of the program and eventually on instrumentation of the program to place objects into scopes.

- As a compromise of these three ideas, each vertex could represent all instances of objects created at a particular *allocation site* (`new` command) of the program.

Because the ultimate result of our analysis is the instrumentation of allocation sites, we choose each vertex to represent a unique allocation site. If each object instance is represented separately, we would require context-sensitive information as to how the appropriate instantiation statement was reached.

To summarize, for the work we present in this paper, the vertices of the *doesReference* graph correspond to object-instantiation sites of the program and the edges are determined dynamically by instruction traces. The edges of the *doesReference* graph reflect inter-object references; each vertex represents all objects instantiated at a given site.[6]

In addition to the referencing information, we require per-vertex knowledge concerning when the objects associated with that vertex become collectible. Such information can be recorded as a frame number (or frame offset, as discussed below) like that shown in Figure 2. For the remainder of this section, we assume that the *doesReference* graph has been faithfully constructed. Section 5 explains the mechanics of how the relevant information can be collected.

4.3 Scope determination

In the observations listed in Section 2, and in Figure 3, we stored frame information based on the *absolute depth* of the call stack. We diverge from that approach here to introduce the more precise concept of *relative depth* that our analysis actually produces.

Relative depth is measured as a call-stack frame offset between the instantiation of an object and the point at which it is first determined to be dead (collectible). If an object is instantiated at frame 10 and becomes collectible when frame 4 pops, its lifetime is said to be 6 frames (it is live until the seventh frame pop). A relative liveness metric is much more suitable for our purposes than an absolute liveness metric; any method that acts as a factory (*e.g.*, the commonly-used `iterator()` method of Java's `Collection` classes) likely generates objects with very different liveness characteristics, since the constructed object's liveness is heavily based upon the caller, for whom the object is being instantiated. Relative frame information alleviates this problem somewhat by reducing the penalty for a call to a factory method deep in the call stack.[7]

We now present our algorithm for determining scope assignments for a Java program P. We seek a mapping

$$Scope : V_P \rightarrow \{0, 1, 2, \ldots, maxstack(P) - 1\}$$

from a vertex $v \in V_P$ in the *doesReference* graph for P to an integer indicating the lifetime of objects aggregated by that vertex.

At the point of instantiation of object o, a nonnegative frame number o_i represents the method whose stack frame occurs at absolute depth o_i in the current call stack. We also define o_c to be the absolute frame number which, when popped, causes o to become collectible; if o is not collectible until the program exits, $o_c = 0$. For our purposes an object cannot be collected until at least the call-stack frame from which it was instantiated pops; thus $o_c \leq o_i$.

[6] With this definition of the *doesReference* graph, Figure 2 remains the same, as there is a one-to-one mapping from the 7 instantiated objects to the 7 instantiation sites.

[7] Early experiments with absolute frame liveness information showed little promise, as expected, compared to the relative frame liveness results presented in Section 5.2 of this paper, for the purposes of precise scope detection.

The output of scope determination, then, is the $Scope$ mapping such that $Scope(v)$ is the expected lifetime $o_i - o_c$ of objects described by vertex v, expressed as an integer.

Our algorithm is based on the constraints imposed by a *doesReference* graph for a given program as follows:

1. Lifetime information associated with a vertex v indicates the frame by which all objects associated with v have become dead. This is easily computed by retaining the maximum relative lifetime observed of any object associated with v. We then have constraints of the following form:

   $$Scope(v) \geq \max_{o \in Objects(v)} o_i - o_c$$

   where $Objects(v)$ is the set of all objects instantiated at the site represented in the *doesReference* graph by v.

   For the program shown in Figure 1(a), liveness information imposes the following constraints:

Vertex	$Scope(v)$
A, B, C, F, and G	≥ 2
D	≥ 1
E	≥ 0

2. Referencing information constrains the relative scope allocation for the related references. According to the RTSJ, object x can reference object y only if y's associated scope fully encloses x's scope in a nested-scope relationship for the currently executing thread. In terms of the *doesReference* graph, reference information is mapped to the vertices of that graph.

   Thus, a reference from an object x to an object y is mapped to an edge from X to Y in the graph, where X and Y are the vertices corresponding to the allocation sites of x and y, respectively. Each reference contributes such an edge to the *doesReference* graph if one is not already present to represent the reference. Each edge then imposes the following constraint between all pairs (x, y) where $x \in X, y \in Y$:

   $$y_c \leq x_c$$

   Letting $Vertex$ be a mapping from objects to *doesReference* vertices, we can obtain

   $$Scope(Vertex(y)) \geq y_i - x_c$$

   and since $x_i - x_c = Scope(Vertex(x))$ in the instrumented program, we have

   $$Scope(Vertex(y)) \geq Scope(Vertex(x)) + y_i - x_i$$

   We can then use the observed values for y_i and x_i during analysis.

In summary, once the *doesReference* graph is constructed, assignment of scopes can be reduced to solving a relatively simple constraint system. Each vertex is constrained by liveness information to be allocated at or below a certain stack depth; referencing behavior places edges between vertices to ensure that the relative stack-depth allocation of objects respects the rules of the RTSJ.

The constraint system that arises in this fashion has at least the following solution

$$\forall\, v\ Scope(v) = \max_{x \in Objects(v)} x_i$$

which effectively assigns all objects to the primordial scoped memory area. At present, ignoring the cost of scope creation and destruction, the solution we seek *maximizes* the solution for each scope, so as to minimize object lifetime. This solution minimizes storage footprint at the expense of creating (perhaps too) many scopes.

Benchmark	vertices	edges
`jess`	1,072	2,068
`raytrace`	879	1,721
`javac`	1,212	3,592
`mpegaudio`	818	1,605

Table 2: Constraint system size for benchmarks (size 10) using *allocation site* granularity to construct the *doesReference* graph.

The solution that minimizes footprint but respects all constraints can be obtained via a worklist approach:

```
generate doesReference graph G = (V, E)
fill in Scope(v) ∀v ∈ V from collected liveness information
initialize worklist ← V
while worklist ≠ ∅ do
  get and remove u from worklist
  foreach v ∈ Successors(u) do
    apply reference constraint (u ref v)
    if Changed(v) then
      worklist ← worklist ∪ {v}
    end if
  end foreach
end while
```

When calculating relative frame depth scopes, the application of a reference constraint (u ref v) is performed by assigning:

$$Scope(v) \leftarrow \max_{w \in \{u,v\}} Scope(w)$$

The result is a scope index $Scope(v)$ for all vertices v of the *doesReference* graph (and thus for all instantiation sites of program P).

4.4 Target program instrumentation

With the above analysis performed, we can complete our goal of instrumenting the original program P to take advantage of discovered memory scopes. We instrument each method and constructor invocation to allocate and enter a new `ScopedMemory` area and each allocation site (that is, each vertex v of the *doesReference* graph) to allocate its object in the appropriate stack frame. Thus, an instantiation site with an assigned *relative depth* of 4 would allocate its object in the `ScopedMemory` area associated with the call-stack frame that is buried 4 frames under the presently active one.

Future work will target *code splitting* and other techniques that can be performed at this step to reduce the overhead and number of `ScopedMemory` areas in the instrumented program.

4.5 Multithreaded target programs

Thus far in this paper we have considered only single-threaded programs. It is possible, however, to extend this approach to operate with similar liveness and reference traces collected from multiple threads executing concurrently.

In a multithreaded environment, objects that are not shared between threads can be handled in a similar way as in the single-threaded case; such objects receive a scope assignment with respect to their respective owning threads.

Benchmark	# objects	# allocation sites
jess	106,727	1,098
raytrace	559,287	925
javac	211,080	1,239
mpegaudio	9,166	827

Table 3: Benchmark sizes: the number of objects and distinct allocation sites encountered during execution.

Objects shared between two or more threads[8] must have a scope assignment in each of those threads. When all threads sharing an object have popped the stack frame associated with the object's lifetime, the object is dead.

With those observations, an extension to our approach to allow for multiple threads may be constructed. A particular scoped memory area corresponds not to a single stack frame but to a particular cut across all participating threads' call stacks. Each thread is instrumented to enter this shared scope at its own appropriate call-stack frame.

5. EXPERIMENTATION

In this section we present results that quantify the scopes formed by our analysis and compare the resulting scoped object lifetimes to observed object lifetimes. As stated earlier, we assume the cost of scope creation and deletion is negligible; thus, we are interested in creating the most refined scopes possible—as near the top of stack as possible—so as to minimize the time an object spends from its logical death until the point of collection (by having its associated scope deleted).

5.1 Experimental method

We instrumented Sun's JVM version 1.1.8 to collect the information required by our algorithm: birth and death frames of objects and object inter-reference information. We also tagged each instantiated object with its instantiation site—that is, the `new` instruction that created that object. This information is required to relate referencing behavior to the vertices of the *doesReference* graph.

We then built the *doesReference* graph and computed a mapping from vertices to object liveness for the objects using the approach presented in Section 4. We report results using the following benchmarks (size 10) from the Java SPEC suite [9]: `raytrace` renders an image; `javac` is the Java compiler from Sun's JDK 1.0.2; `mpegaudio` is a computational benchmark that performs compression on sound files; `jess` is an expert-system shell application that solves a puzzle in logic. The size of these benchmarks is shown in Table 3.

Object references were traced by instrumenting the JVM to emit messages in response to `putfield` and similar instructions that cause one object to reference another. Object lifetimes were collected by using a simple reference-counting garbage collector implementation that finds dead objects at each frame pop. This particular reference-counting implementation counts only inter-object references, and handles local references (from local variables and Java's operand stack) by associating the object to the lowest stack frame that exhibits such behavior. When a frame pops, its object list is traversed and any object with a reference count of zero is then collected. While the reference-counting collector is approximate, it is close enough for these benchmarks [6]. Use of an exact collector would cause our experiments to take too much time [23].

Some inexact methods of garbage collection are not suitable for this analysis as they closely couple inter-object references and object lifetime. For example, contaminated garbage collection [6] groups objects that reference each other into *equilive sets*, expecting that they will become collectible at the same time. If based on a contaminated collector, our scope analysis would not output a properly nested scoping structure; the scopes would be unnecessarily conservative due to the collector's bidirectional association between an object making a reference and the target of the reference.

We are working on the instrumentation of Java code as described in Section 4.4 to use the scopes harvested by our analysis; we cannot at this time offer measurements of improved real-time determinism due to the use of these scoped memory regions.

5.2 Results

The results presented here show some relevant information about the benchmarks and demonstrate scope formation using our technique. We compute an assignment of each object to a non-garbage collected scoped memory area corresponding to its instantiation site. The scope assignment satisfies both the observed object lifetimes and the reference constraints of the RTSJ introduced in Section 3.

Figure 4 shows the number of objects we assign into each scope for the `mpegaudio` benchmark. Column 0 accounts for objects that are collected in their allocating frame, using the scoped memory areas we've discovered. Column 1 accounts for objects that are collected when the stack frame of their allocating method's caller pops.

Figure 5 shows the corresponding comparison for `raytrace`; Figure 6 shows the `javac` benchmark, and Figure 7 shows the results for `jess`. Of the four benchmarks, three allocate a plurality of their objects such that they become dead when their allocating frame pops; `mpegaudio`, a primarily computational benchmark that allocates fewer objects than any of the other three (see Table 3), allocates more objects that ultimately live longer when placed in scoped memory.

In the `jess` benchmark, a substantial number of objects are paired off in circularly-referencing relationships; due to this referencing behavior, our reference-counting garbage collector does not collect such objects until program termination; since this information serves as input to our analysis, our analysis assumes that such objects are longer-lived than they actually would be under an exact garbage collector.

For practical reasons, inter-procedural escape analysis is typically performed with limits on the maximum call-stack depth;[9] thus escape analysis [28] could perform rather well for a benchmark like `raytrace` in which most of the objects become collectible after their allocating stack frame is popped. Our results indicate that for programs that allocate longer-lived objects (like `mpegaudio` in Figure 4), the added expense of deeper static analysis is justified.

Of course, due to our RTSJ-imposed reference constraints and the fact that we aggregate object behavior by *allocation site*, we cannot always collect (by closing a scoped memory) all objects as soon as they become collectible. To measure this, we consider the lifetime (in frames) of each object according to the reference-counting collector and subtract from this the lifetime of the object when placed in its assigned scope according to its allocation site's assigned scope index. The result is its "extra longevity" under the scope assignment.

[8] An object x can be shared either *directly* between threads (multiple threads access the object directly), or *indirectly* through a reference from a shared object. In either case, the combined behavior of all participating threads affects the scope assignment of x, so it constitutes a shared object for our purposes.

[9] Personal communication, Martin Rinard.

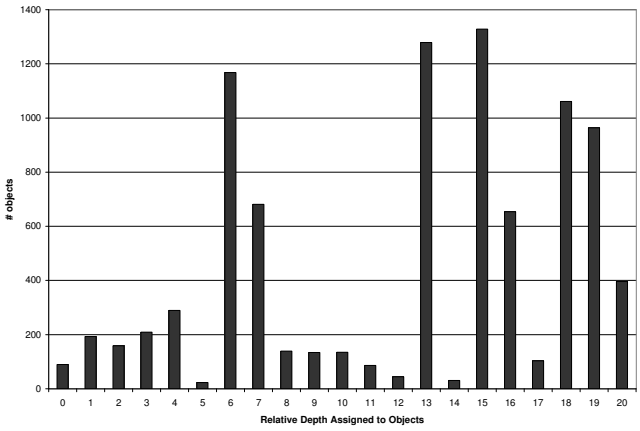

Figure 4: Scoping assignments for objects in `mpegaudio`.

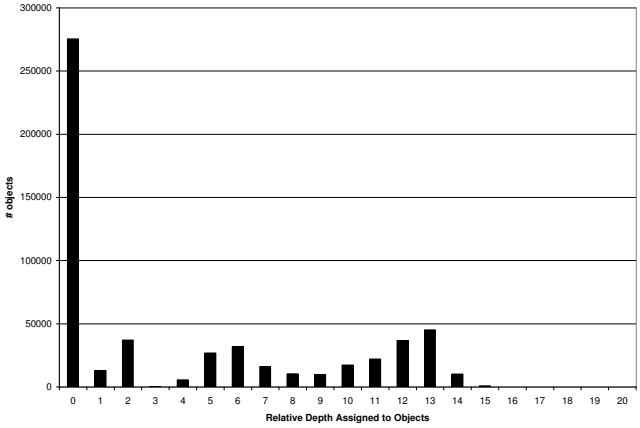

Figure 5: Scoping assignments for objects in `raytrace`.

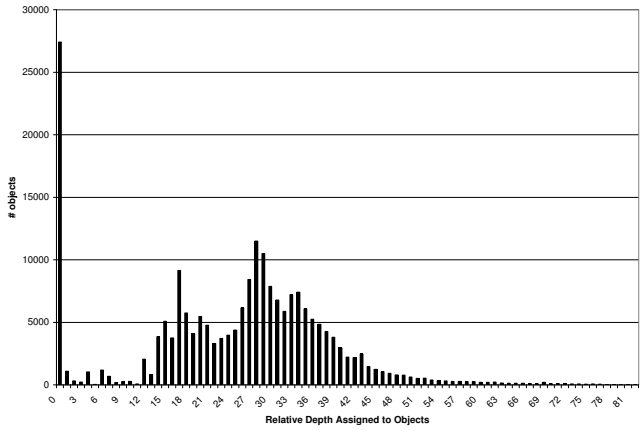

Figure 6: Scoping assignments for objects in `javac`.

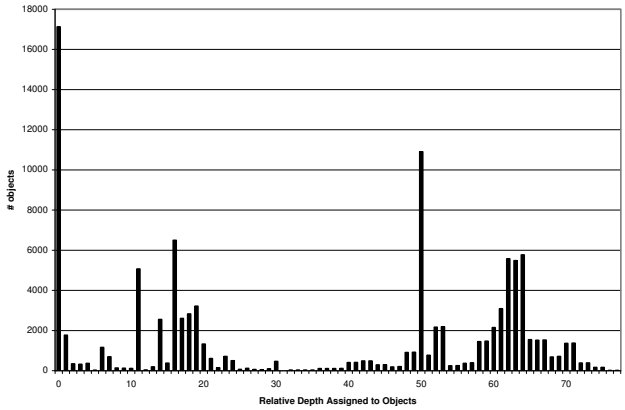

Figure 7: Scoping assignments for objects in `jess`.

Figure 8 shows the "extra longevity" afforded the objects of the `raytrace` benchmark. The plot indicates the number of objects that ultimately live longer than necessary when placed in our scoped memory areas and the additional number of frames for which they live. For example, when placed in scoped memory, there are 27,588 objects in `raytrace` that live one frame longer than is necessary.

Figure 9 shows a similar plot for the `mpegaudio` benchmark. Only 700 objects of this benchmark live one frame longer than necessary. In fact, we can accurately scope approximately 2,300 of the 9,000 objects in `mpegaudio` such that they live no longer than they would under the reference-counting collector.

6. RELATED WORK

Gay and Aiken [13] provide a regions library with such operations as `newregion` to create a new region, a corresponding `deleteregion`, and `ralloc` to allocate within a region; programmers target a region-annotated dialect of C, called RC, that permits them to indicate restricted pointers that may only point to objects in their own region or a limited (possibly empty) set of other regions. Their RC compiler performs static checks and injects code for runtime checks that ensure that the behavior of these restricted pointers fits their annotation. A reference count of external pointers into a region is kept, and a region may only be deleted when no outstanding external references exist.

A stack of regions is suggested by Tofte and Talpin [26, 25]. A programmer can bind the evaluation of an expression to a particular memory region, similar to the use of scoped memory regions in RTSJ. This view of region-based memory management has been realized in the ML Kit [20], a Standard ML [19] compiler. Recently the ML Kit has been extended to allow for the garbage collection of these regions; an account of this extension is found in [14].

Static pointer and escape analysis [27, 28] seeks to collect information about the referencing and lifetime behavior of objects; such analysis is often targeted to a variety of optimizations, including the allocation of objects on the stack and synchronization elimination. The analysis of Sălcianu and Rinard [24] introduces the *parallel interaction graph*, provides safety checking of supplied region gestures, and can be used to remove unnecessary runtime instructions for region verification, all for a multithreaded target program. These analyses are performed statically and may need to make overly conservative assumptions about the behavior of the program, as mentioned in Section 4 of this paper. Manual modifications may be required to the discovered regions to achieve desired results [3].

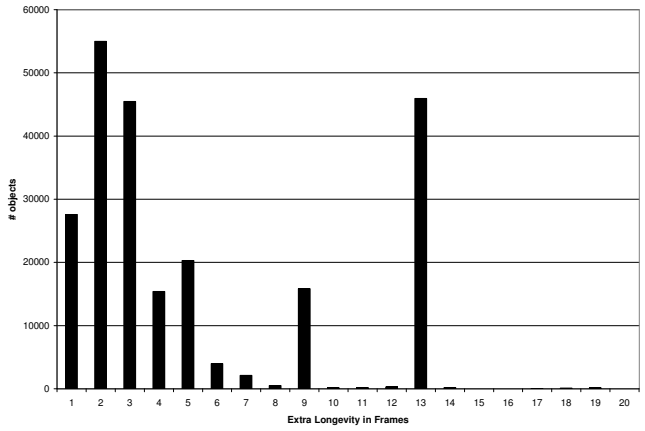

Figure 8: Extra longevity of objects in `raytrace` using scoped memory.

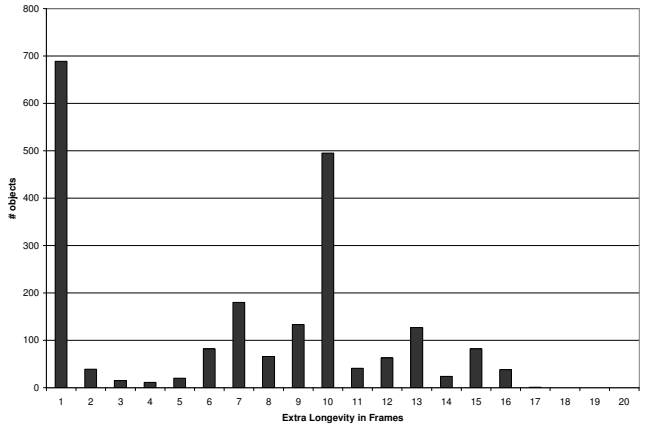

Figure 9: Extra longevity of objects in `mpegaudio` using scoped memory.

7. CONCLUSIONS

Automatic memory management is a useful technique that substantially releases the programmer from concerns about the lifetime of objects in a software system. There is a cost to using automatic memory management, however, and because the state of a system's memory and garbage collector are often highly unpredictable, a real-time system can miss important deadlines if it seizes or slows progress for an unbounded (or unreasonably bounded) period of time to internally manage memory.

The Real-Time Specification for Java addresses this problem by offering regions of memory that are not garbage collected but instead are reclaimed after a particular scope of execution exits. In this paper, we have presented an algorithm to discover how and where to instrument Java programs to take advantage of these non-garbage collected, scoped memory regions defined by the RTSJ. We have presented experimental results to verify that much of the memory used by standard benchmarks is locally allocated (or is live for a short time on the stack) but that a significant amount can live longer; these long-lived allocations can be difficult for static analysis to handle without resorting to overly conservative assumptions.

8. FUTURE WORK

We are presently investigating numerous improvements to our scope-detection tool:

Aspects. Aspect-Oriented Programming (AOP) [16] can be used to implement the reference and lifetime dynamic analyses and also to instrument the program to take advantage of scoped memory regions. In this approach, the analysis is designed, implemented, and then *weaved* into the program we wish to trace. Similarly, after the analysis is performed a behavioral description of the output of the analysis is weaved into the program. We have already implemented such profiling aspects [11, 10], but are unable to use these implementations to analyze Java library code or objects created internally by a JVM due to limitations in the current version of the AspectJ compiler [2]. The primary advantages to using AOP for this purpose are to avoid instrumenting and re-targetting Java interpreters individually to log appropriate data for our analysis and to cleanly separate the Java and RTSJ code of the target program.

Multiple threads. Further evaluation of our threading-aware analysis techniques introduced in Section 4.5 is necessary. We have a prototype threading-aware AspectJ aspect to perform this multi-threaded analysis and plan to continue its development as a part of this work.

Alternative liveness measures. Types of object death measurement other than the *relative death* metric described in this paper are possible. One example is the time of death of an object relative to the entry of a method in a particular library or class. This type of measure could be quite useful in analyzing several modular components separately, then composing those analyses' data into a full, coherent runtime system that makes use of all of those components.

Code splitting. The granularity of a vertex in the *doesReference* graph can be made more precise by "code splitting" at an instantiation site, with the resulting `new` commands tailored by some calling context. This could help objects created by factory methods (and other methods that generate objects on behalf of the caller), as we could then consider not only the allocation site of the object but also the caller of the allocating method.

Combination of analyses. Our dynamic analysis could be combined with static escape analysis [28] to provide static analysis with areas where precision could be improved and to inform our analysis of areas where we should be more conservative.

9. ACKNOWLEDGEMENTS

We would like to thank Guy Steele, Jr. and Sun Microsystems for access to their JVM 1.1.8. We would like to thank Dante Cannarozzi, Conrad Warmbold, and Nick Leidenfrost of Washington University for their help with instrumenting Sun's JVM 1.1.8 to collect the data for this paper.

10. REFERENCES

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

[2] The AspectJ Organization. The AspectJ Programming Guide: Implementation Limitations. `www.aspectj.org/doc/dist/progguide/apc.html`, 2001.

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

[4] B. Blanchet. Escape analysis for object-oriented languages. *ACM SIGPLAN Notices*, 34(10):20–34, 1999.

[5] Bollella, Gosling, Brosgol, Dibble, Furr, Hardin, and Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[6] Dante J. Cannarozzi, Michael P. Plezbert, and Ron K. Cytron. Contaminated garbage collection. *Proceedings of the ACM SIGPLAN '00 conference on Programming language design and implementation*, pages 264–273, 2000.

[7] Perry Cheng and Guy Belloch. A parallel, real-time garbage collector. *ACM SIGPLAN Notices*, 36(5):125–136, May 2001.

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

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

[10] Morgan Deters and Ron K. Cytron. Introduction of program instrumentation using aspects. In *Proceedings of the OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Tampa Bay, FL, October 2001. ACM. `http://www.cs.ubc.ca/~kdvolder/Workshops/OOPSLA2001/ASoC.html`.

[11] 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.

[12] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. *ACM SIGPLAN Notices*, 33(5):106–117, 1998.

[13] David Gay and Alex Aiken. Language support for regions. In *Proceedings of ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI)*, pages 70–80, Snowbird, Utah, May 2001. ACM.

[14] Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining region inference and garbage collection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*. ACM Press, June 2002. Berlin, Germany.

[15] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21(4):848–894, 1999.

[16] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, June 1997.

[17] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 235–248, New York, NY, 1992. ACM Press.

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

[19] Robin Milner, Mads Tofte, Robert Harper, and David McQueen. *The Definition of Standard ML (Revised)*. MIT Press, May 1997.

[20] The ML Kit. `www.it-c.dk/research/mlkit/`, 2002.

[21] Scott Nettles and James O'Toole. Real-time replication garbage collection. *ACM SIGPLAN Notices*, 28(6):217–226, 1993.

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

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

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

[25] Mads Tofte. A brief introduction to regions. *Proceedings of the International Symposium on Memory Management (ISMM)*, pages 186–195, 1998.

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

[27] 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.

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

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