Free-Me: A Static Analysis for Automatic Individual Object Reclamation

Samuel Z. Guyer
Tufts University
sguyer@cs.tufts.edu

Kathryn S. McKinley *
The University of Texas at Austin
mckinley@cs.utexas.edu

Daniel Frampton
Australian National University
daniel.frampton@anu.edu.au

Abstract

Garbage collection has proven benefits, including fewer memory-related errors and reduced programmer effort. Garbage collection, however, trades space for time. It reclaims memory only when it is invoked: invoking it more frequently reclaims memory quickly, but incurs a significant cost; invoking it less frequently fills memory with dead objects. In contrast, explicit memory management provides prompt low cost reclamation, but at the expense of programmer effort.

This work comes closer to the best of both worlds by adding novel compiler and runtime support for compiler inserted frees to a garbage-collected system. The compiler's *free-me* analysis identifies when objects become unreachable and inserts calls to free. It combines a lightweight pointer analysis with liveness information that detects when short-lived objects die. Our approach differs from stack and region allocation in two crucial ways. First, it frees objects incrementally exactly when they become unreachable, instead of based on program scope. Second, our system does not require allocation-site lifetime homogeneity, and thus frees objects on some paths and not on others. It also handles common patterns: it can free objects in loops and objects created by factory methods.

We evaluate free() variations for *free-list* and *bump-pointer* allocators. Explicit freeing improves performance by promptly reclaiming objects and reducing collection load. Compared to mark-sweep alone, free-me cuts total time by 22% on average, collector time by 50% to 70%, and allows programs to run in 17% less memory. This combination retains the software engineering benefits of garbage collection while increasing space efficiency and improving performance, and thus is especially appealing for real-time and space constrained systems.

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

General Terms Languages, Performance, Experimentation, Algorithms

Keywords Adaptive, Generational, Compiler-Assisted, Liveness, Pointer Analysis, Locality, Mark-Sweep, Copying

* This work is supported by NSF ITR CCR-0085792, NSF CCR-0311829, NSF CCR-0311829, NSF EIA-0303609, DARPA F33615-03-C-4106, IBM, and Intel. Any opinions, findings and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

PLDI'06 June 10–16, 2006, Ottawa, Ontario, Canada

1. Introduction

This work seeks to combine the software engineering benefits of automatic memory management (garbage collection) with the low-cost incremental reclamation of explicit memory management [6, 17, 29]. If programmers free objects immediately after their last use, they can minimize the program's memory footprint. Garbage collection instead minimizes programmer effort at the expense of increased memory requirements [23, 25, 33]. Garbage collectors require extra memory, beyond what the application is using, because they only periodically reclaim memory, allowing dead objects to accumulate. Collecting more frequently is expensive due to the overhead of identifying dead objects. For example, a copying collector traverses the roots (stacks, statics, registers), copying all the reachable objects, which can be costly. Invoking the collector less frequently amortizes this cost, and gives objects more time to die, making each collection more productive. Garbage collection thus makes a space-time tradeoff that increases the memory footprint to reduce collection costs [7, 23, 25, 26, 33, 38].

We present new runtime and compiler support to obtain the best of both approaches. We add an explicit *free(object)* operation to the garbage collector and a *free-me* compiler analysis that automatically inserts a call to free() at the point an object dies. Free-me analysis combines simple, flow-insensitive pointer analysis with flow-sensitive liveness information. An allocation site can produce some objects that escape the method or loop, and free-me can still free some of them. Since its scope is mostly local, it typically finds very short-lived objects. Free-me includes an interprocedural component that summarizes connectivity and identifies *factory* methods.[1] A factory method returns one newly allocated object and has no other side effects.

The underlying collection discipline dictates the implementation of free. We implement free for mark-sweep free-lists and for copying contiguous *bump-pointer* allocation. Our free-list implementation segregates by size [7, 8]. Free returns memory to the front of the appropriate free-list, as does explicit memory management. For bump-pointer allocation, we explore a version of free that reclaims the object only when it is the last allocation. This version is fairly restrictive because the compiler must perfectly order (last-in-first-out) frees. We also explore a more powerful version of free that tracks one unreclaimed region closest to the bump pointer. If a

[1] Freeing long-lived objects would require a substantially more precise interprocedural pointer and liveness analysis.

subsequent free brings it to the top, free reclaims it as well. Finally, we show a version of free that simply reduces the required copy reserve size by the number of free bytes.

Compared with region [12, 24, 32, 37] and stack [10, 13, 20, 39] allocation, our approach differs in two key ways. First, region and stack allocation require lifetimes to coincide with a particular program scope, whereas our approach frees objects exactly when they become unreachable. Second, region and stack allocation require specialized allocation sites, forcing them to decide the fate of each object at allocation time. In many systems, this limitation requires each allocation site to produce objects with the same lifetime characteristics. Even if some objects become unreachable, these systems must wait until all objects become unreachable. Although stack and region allocation reduce collector load and have the potential to reduce the memory footprint, neither has delivered consistent improvements over generational collectors.

We implement these techniques in Jikes RVM and MMTk, a high performance Java-in-Java virtual machine and memory management toolkit [1, 2, 7, 8]. For mark-sweep, our results on SPECjvm98, SPECjbb2000, and DaCapo [9, 35, 36] Java benchmarks show that free reclaims on average 32% of all objects: on three benchmarks, it reclaims less than 10%, but it reclaims 50% or more of the objects on four benchmarks. These frees translate directly into total and garbage collection performance improvements in small and moderate sized heaps (up to a factor of 2), and do no harm in large heaps.

Our technique was unable to improve over a state-of-the art generational collector. Although free reclaims similar numbers of objects as free in mark-sweep and these frees translate into many fewer nursery collections, the survival rate of each collection increases. As the collection cost of the nursery is proportional to the size of live objects, we see little reduction in overall collection cost. We had hoped that by eliminating some short-lived objects from the nursery we would increase the virtual size of the nursery giving other objects more time to die, but this effect was not visible in our results. Furthermore, mutator time degraded in some cases due to the overhead of incrementally *unbumping* objects. These results combined with similar ones for stack allocation [10, 13, 20, 39] indicate that collecting short-lived objects seems best left to a copying nursery.

Although generational collectors typically perform much better than whole-heap mark-sweep collectors [7], non-moving collectors remain critical for certain applications. For example, embedded systems use mark-sweep for space efficiency. Real-time collectors are especially sensitive to garbage collection load [4]. For these systems, we recommend compiler-inserted frees which yield substantial improvements in memory efficiency and performance.

2. Motivating Example

Figure 1 shows an example that motivates the use of free-me analysis instead of escape analysis. The code in Figure 1(a) is inspired by a method in javac. In this method, stream.readToken() is a factory method that produces a single new object and has no other side effects. The variable `idName` points to this newly allocated object. The code only adds idName to the symbol table if it is not already there. Since programs tend to use symbols repeatedly, many more die than live. Since the surviving objects escape the method, it is not safe to stack allocate them. Free-me compiler analysis detects that stream.readToken() is a factory method and that the resulting object is only stored in the symbol table on the true branch of the conditional statement. It determines that the object can be freed on the false branch, and it inserts a call to free(idName), as shown in Figure 1(b). We use this running example throughout the paper.

```
1  public void parse(InputStream stream) {
2   while (...) {
3     String idName = stream.readToken();
4     Identifier id = symbolTable.lookup(idName)
5     if (id == null) {
6       id = new Identifier(idName);
7       symbolTable.add(idName, id);
8     }
9     computeOn(id);
10  }
11 }
```

(a) Only adds idName to the symbolTable once.

```
1  public void parse(InputStream stream) {
2   while (...) {
3     String idName = stream.readToken();
4     Identifier id = symbolTable.lookup(idName)
5     if (id == null) {
6       id = new Identifier(idName);
7       symbolTable.add(idName, id);
8     }
9     else            // idName no longer used:
10      free(idName); //  -> free it immediately
11    computeOn(id);
12  }
13 }
```

(b) Compiler inserts free on else branch.

Figure 1. Example of conditional free from javac.

3. Free-me Compiler Analysis

This section describes our *free-me* compiler phase which automatically inserts calls to `free()`. For each allocation site, it computes the set of program points where a newly allocated object exists, but is no longer reachable. It consists of four parts.

1. A flow-insensitive pointer analysis builds a connectivity graph for the objects in each method. Each node represents a local variable, a global variable, or a new object (one for each allocation site). Edges represent may points-to relationships.
2. A flow-sensitive liveness analysis sharpens the connectivity graph. Used alone, the flow-insensitive pointer analysis can only determine *if* objects are reachable – for example, which objects escape and which do not. We add liveness information about variables and points-to edges so that we can determine exactly *when* objects are reachable.
3. A free-me instrumentation pass selects program points and inserts calls to free(). Since the liveness analysis may identify numerous potential places to free() an object, our heuristic chooses the subset that (a) frees an object as soon as possible, and (b) inserts as few calls to free() as possible.
4. An offline interprocedural pass that summarizes the may points-to effects of each method and identifies *factory* methods: methods that only return newly allocated objects. We perform this pass first.

We apply free-me compilation in two passes. The first pass performs only the pointer analysis to compute *method summaries* offline. We perform this first pass bottom-up on the call graph to ensure that method summaries are available at all call sites. The call graph is constructed on-the-fly as method calls are encountered. The second pass adds the liveness information and instruments methods with calls to free(). We perform the second pass only on *hot* methods (Section 5).

The analysis uses the Jikes RVM compiler internal representation (IR). The IR is a control-flow graph of basic blocks, where each basic block contains a list of instructions. The instructions correspond to Java operations, such as *getfield*, *putfield*, array operations, and assignments. We perform our analysis when the IR is in

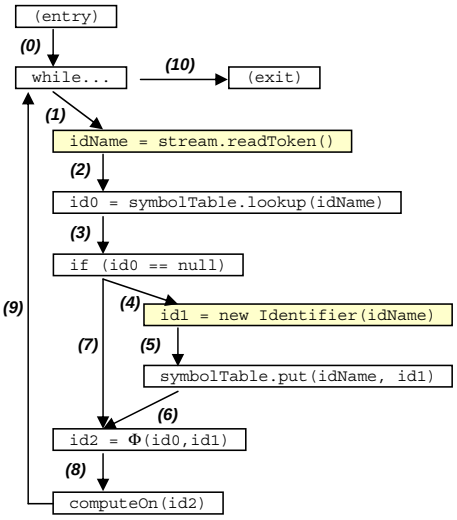


Figure 2. Control-flow graph for the example.

SSA form [15], which provides flow sensitivity for local variables. Figure 2 shows the IR for our example (Figure 1). Note that this control-flow graph includes edges between each instruction, which allows our algorithm to reason about liveness at a fine granularity.

3.1 Pointer Connectivity

Our pointer analysis is flow insensitive, field insensitive, and inclusion based, similar to Andersen's pointer analysis [3]. Our algorithm is primarily intraprocedural: it only analyzes one method at a time, and it only represents local objects in its heap model. It does, however, use method summaries to provide interprocedural information. At each call site the analysis consults the summary for the callee, which describes: (1) its effects on the connectivity of the objects passed into it, and (2) if it is a factory method. The analysis treats a call to a factory method like an allocation site, since it returns a new object that the caller is allowed to free.

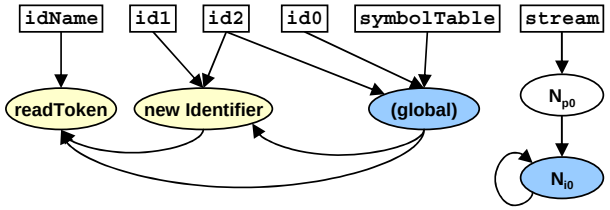


Figure 3. Connectivity graph for the running example: each node represents a variable or object, and each edge represents a points-to relationship.

The analysis builds a connectivity graph with nodes for each allocation site, input parameters, and one node for all globals. It also keeps track of the targets of all local variables. Figure 3 shows the connectivity graph for the running example. Variables are represented by boxes at the top, and objects in the heap are represented by ovals. The algorithm identifies `readToken()` as a factory and creates an allocation node for it.

Table 1 presents the analysis data structures. Note that each parameter has an associated pair of nodes that represent the incoming objects. The parameter nodes N_P represent the immediate targets of the parameters, while inner nodes N_I represent any other objects reachable from the parameter nodes. We assume no aliasing between parameters, which is safe for free-me analysis because reachability from *any* parameter will prevent the analysis from freeing an object. The points-to analysis starts by initializing the points-to functions of parameters to reflect this structure:

S	Set of statements
V	Set of variables
v_i	Local variable i
$p_i \in V$	Formal parameter i
N	Nodes in connectivity graph
$N_P \subset N$	Nodes for targets of parameters
$N_I \subset N$	Parameter "inner" nodes
$N_A \subset N$	Allocation nodes – one for each `new()`
$N_G \in N$	Node for all globals (statics)
$PtsTo : (V \cup N) \rightarrow 2^N$	Points-to function
$PtsTo* : (V \cup N) \rightarrow 2^N$	Transitive closure of points-to

Table 1. Data Structures for Free-Me Analysis

$\forall i, PtsTo(p_i) = \{N_{Pi}\}$	Initialize parameter variables
$\forall i, PtsTo(N_{Pi}) = \{N_{Ii}\}$	Parameter nodes point to inner nodes
$\forall i, PtsTo(N_{Ii}) = \{N_{Ii}\}$	Inner nodes have a self-loop

The algorithm iterates over the instructions in the method and adds edges to the graph until no new edges are added. The following table describes how we update the points-to function at each kind of instruction. We treat array operations as field operations: *astore* uses the same rule as *putfield*; *aload* uses the same rule as *getfield*.

assignment	`v1 = v2;`	$PtsTo(v_1) \cup= PtsTo(v_2)$
getstatic	`v = Cls.f;`	$PtsTo(v) \cup= \{N_G\}$
putstatic	`Cls.f = v;`	$PtsTo(N_G) \cup= PtsTo(v)$
putfield	`v1.f = v2;`	$\forall n \in PtsTo(v_1)$
		$PtsTo(n) \cup= PtsTo(v_2)$
getfield	`v1 = v2.f;`	$PtsTo(v_1) \cup= PtsTo*(v_2)$

The rules for *assignment*, *getstatic*, *putstatic*, and *putfield* are straightforward: they transfer points-to edges from the right-hand side to the left-hand side, as appropriate. The only unusual rule is *getfield*: it adds all nodes *reachable* from the right-hand side to the left-hand side. This conservative rule allows us to use very simple interprocedural summaries (see Section 3.2), yet has little effect on accuracy because it does not drastically change the overall reachability of objects.

3.2 Procedure Summaries

Our analysis algorithm benefits significantly from information about the callees of a method. During the first analysis pass we compute simple method summaries that capture (1) whether or not the method qualifies as a factory method, and (2) how the method connects objects passed to it. Each summary consist of a set of pairs of parameter numbers. The pair (p_i, p_j) indicates that the method connects these two parameters, by some sequence of pointers, so that argument p_j becomes reachable from argument p_i.

For example, line 7 in the code in Figure 1 calls the method `symbolTable.add(idName, id)`. Internally, this method manages a complex data structure (a hash table). Our analysis, however, only needs to know that this method attaches both the `idName` and `id` objects to the `symbolTable`. We summarize this behavior using two pairs, (0,1) and (0,2), indicating that arguments 1 and 2 become reachable from argument 0 (the receiver argument). In our heap model, the algorithm adds direct pointers from `symbolTable` to the two other objects. Figure 3 shows these edges from the global node to the `readToken` node and to the new identifier node.

The following table shows how we compute each entry and includes special entries for global pointers and return objects. The notation $*p_j$ means apply *getfield* to the argument first to obtain the inner node. The last type of entry identifies factory methods. Our analysis requires that a method return *only* new objects in order to qualify as a factory.

$N_{pj} \in PtsTo*(p_i)$	$\Rightarrow$	record entry (p_i, p_j)
$N_{Ij} \in PtsTo*(p_i)$	$\Rightarrow$	record entry $(p_i, *p_j)$
$N_{pj} \in PtsTo*(N_G)$	$\Rightarrow$	record entry $(global, p_j)$
$N_{pj} \in PtsTo*(return)$	$\Rightarrow$	record entry $(return, p_j)$
$PtsTo(return) \subset N_A$	$\Rightarrow$	record method is a factory

When the analysis encounters a method call, it looks up the possible targets (virtual calls may have more than one) and applies each method summary to the actual arguments. It applies entries of the form (p_i, p_j) using the putfield operation, and entries of the form $(return, p_j)$ by assigning the points-to set of p_j to the left-hand side of the call site. For each factory method call site, it introduces an allocation node.

This summary scheme has two significant implications. First, it necessitates the getfield rule used during pointer analysis because a single pointer link in the summary may represent many pointer links in the callee, and therefore may represent multiple pointers in the concrete heap . For example in Figure 3, the edge from the global node to the `new Identifier()` elides the internal structure of the symbol table. Our conservative getfield rule ensures that none of the possible targets are missed. Second, since the summaries refer only to the parameter positions, they effectively make the pointer analysis context sensitive (i.e., it does not introduce unrealizable paths.) The results of this pointer connectivity analysis, however, are only suitable for testing overall reachability.

3.3 Object reachability and liveness

Liveness analysis is a crucial part of our system because it identifies *when* objects become unreachable, not just whether or not they escape. In Figure 3, for example, both of the newly created objects appear to escape because both are reachable from the global node. Sharpening the flow-insensitive connectivity graph with flow-sensitive liveness information reveals that these pointers do not exist on all paths through the method, allowing us to free the objects on some paths and not others.

Our algorithm for computing object reachability is based on the observation that an object is reachable only when the pointers to it are live. If several variables point to an object, for example, then the object will be reachable at any of the program points where those variables are live. We apply this observation to our connectivity graph as follows. We start by computing live sets for each variable using traditional liveness analysis. We then propagate these live sets from the variables to their targets: the reachability of a node is the union of the live sets of the variables that point to it. We can then use the reachability of a node to infer the liveness of its outgoing pointers, and propagate those live sets to their respective targets. We continue this process until we have computed reachability for all nodes.

We compute liveness and reachability as sets of edges in the control-flow graph. The control-flow graph in Figure 2 motivates this design: we want the analysis to determine that `idName` is freeable on edge 7 (the else branch). Traditional liveness analysis, which uses sets of statements, cannot distinguish this edge. In order to support fine-grained freeing, we use a representation in which each instruction is a separate node in the CFG.

This formulation alone, however, is not sufficient to detect the free() opportunity in the example (Figure 2) because the global variable `symbolTable` is live (conceptually) throughout the method, implying that `idName` is always reachable and can never be freed. We can determine that it is dead, however, by considering two extra facts. First, `readToken()` returns a new object every time through the loop. Therefore `idName` is the only pointer that can refer to that object immediately following the call. Second, the object returned from `readToken()` only becomes reachable from the global variable after the call to `symbolTable.add()`. We can conclude, therefore, that this pointer first exists on CFG edge #6, continues through CFG edge #8 and the back-edge #9, and ends at CFG edge #1 (right before the call to `readToken()`). The object is also reachable from the local variable `idName` on CFG edges #2, #3, #4, and #5. Therefore it is unreachable on CFG edge #7, which is exactly where we want to place the call to free(). Figure 4 shows the liveness and reachability computation for this object.

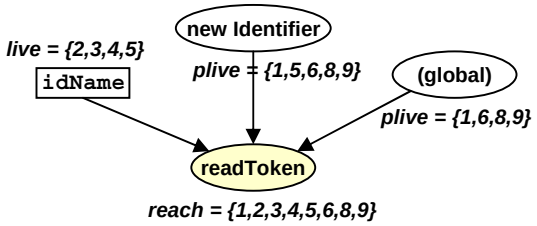


Figure 4. Liveness: each set indicates the CFG edges for which the variables and points-to edges are live. Objects represented by the readToken node are reachable at the union of the CFG edges of the incoming pointers.

To incorporate these refinements into our analysis we introduce a notion of *potential liveness* for points-to edges, which restricts the liveness of pointers stored in global variables and in the heap. We define potential liveness as the set of program points where a pointer *might* exist, according to the following rules: (1) such a pointer can only exist after it is stored in the source object (putfield or putstatic), and (2) such a pointer cannot exist before the target object is allocated. We compute potential liveness for each points-to edge starting at the pointer store that creates it and visiting all CFG edges reachable in the method without going past the allocation site of the target object. Notice that rule (2) is only important in loops – in code without loops the pointer store can never precede the allocation of the target. The data structures for the analysis are initialized as follows:

- $plive(N_i \rightarrow N_j)$: the potential liveness of a points-to edge consists of all CFG edges reachable from the initial assignment, but not extending past the allocation site of the target object N_j.
- $live(v)$: liveness of variables is computed by traditional liveness analysis.
- $reach(N_i), N_i \notin N_A$: parameter nodes and the global node are reachable on all CFG edges in the method.
- $reach(N_i), N_i \in N_A$: reachability of the allocation nodes – initially empty.

The algorithm computes liveness and reachability iteratively for nodes and edges in the graph using two principles: (a) the reachability of a node is the union of the liveness of all incoming points-to edges, and (b) the liveness of a points-to edge is the intersection of the potential liveness of the edge with the reachability of the source node. The second rule allows us to free data structures that consist of multiple objects. It prevents the liveness of a points-to edge from extending past the point that the source object becomes dead. Once the source object is dead, the outgoing points-to edge is no longer live, allowing the target object to become dead.

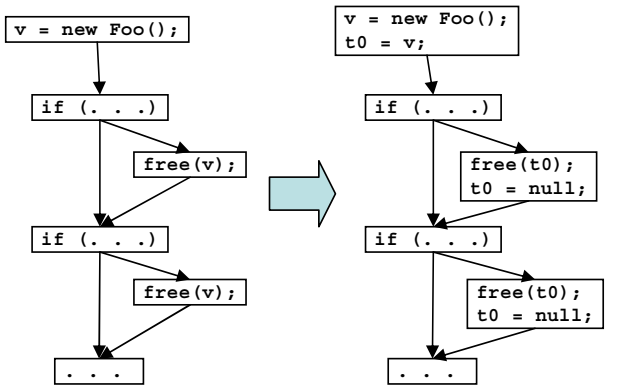


Figure 5. Free-me instrumentation uses temporary variables to allow free() on multiple paths through the same code.

Formally, the algorithm applies the following rules iteratively over the connectivity graph until a fixpoint is reached:

$$\begin{array}{ll} reach(N_i) = & \bigcup live(v)\ \forall v \in V, N_j \in PtsTo(v) \\ & \bigcup live(N_j \rightarrow N_i)\ \forall N_j, N_i \in PtsTo(N_j) \\ live(N_j \rightarrow N_i) = & plive(N_j \rightarrow N_i) \cap reach(N_j) \end{array}$$

Figure 4 shows how this computation proceeds for the object returned by `readToken`, which has three incoming pointers: (1) The pointer from the global node, created in the call to `symbolTable.add()`, which is live on CFG edges $\{1,6,8,9\}$, (2) the pointer from `new Identifier`, created by the constructor, which is live on CFG edges $\{1,5,6,8,9\}$, and (3) the variable `idName`, which is live on CFG edges $\{2,3,4,5\}$. The union of these live sets shows that the `readToken()` object is reachable on CFG edges $\{1,2,3,4,5,6,8,9\}$, and therefore, the object is not reachable on CFG edge 7, the else branch, and we can place a call to free() there.

3.4 Free Placement

Finally, we compute the possible places to insert a call to free for each allocation site. We start with all control-flow edges on which a new object from the site exists (all edges after the `new`), and subtract all edges on which the object becomes reachable. The result is often a *set* of possible program points because our liveness analysis is so fine-grained. To avoid excessive calls to free() we first select the earliest available program point, then iteratively eliminate any other program points dominated by it.

As with manual memory management, we do not want to call free() more than once on a single object. To simplify our analysis, we sidestep this error. We instrument each allocation site to save a copy of the newly allocated object in a temporary. Calls to free() take this temporary as an argument and immediately set it to null. Subsequent calls to free() recognize the null value as a special case and simply return. Figure 5 shows an example of this instrumentation. One significant benefit of this approach is that it allows our system to handle multiple paths through the same set of free() calls.

In addition, the use of temporaries prevents the system from accidentally freeing the wrong object when a pointer value changes between the allocation site and the free site. The following example shows how the use of temporaries avoids this error:

```
v = new Object();          v  = new Object()
v = x.f;                   t0 = v;
free(v);  // Wrong!        v  = x.f;
                           free(t0);  // OK
```

3.5 Analysis design tradeoffs

Our free-me analysis represents one point in the design space. This section discusses some of our design tradeoffs and alternatives. In particular, a central limitation of free-me is that it can only free objects for which there is an explicit pointer in the code. This choice, however, allows us to use a local model of the heap in our analysis, and it allows us to use a simple implementation of deallocation.

In principle, explicit deallocation can completely replace garbage collection. With this ideal in mind we inspected our benchmarks to determine which objects free-me is missing and how we might extend the system to improve its coverage. We find that pushing free-me further would require a significantly more complex and expensive analysis, and a more costly run-time system.

Large data structures. Free-me is not effective on large, long-lived data structures constructed by multiple methods. Extending the analysis to find these opportunities would require several costly changes. First, we need a global model of the heap in order to capture the connectivity of these structures. Second, we need to extend our notion of liveness to handle multiple methods. This analysis would need to be both interprocedural and flow sensitive, and be able to operate over much larger connectivity graphs. Finally, we might need to make the heap model context sensitive, since commonly used methods (such as class libraries) produce objects which are dead in some contexts and not in others.

Freeing large data structures also requires significant changes to the run-time system. In particular, the free() implementation would need to trace through the data structure to free each of its components. This capability not only adds complexity, it adds run-time overhead, which we find is already a serious concern even in our current implementation of free().

Container internals. Free-me also misses internal components of container classes. For example, when the `Vector` class resizes its internal array, the old array is immediately garbage. As with large data structures, this case requires more powerful analysis. Unless the array never escapes the `Vector` class, we need to perform interprocedural, whole-program pointer and liveness analysis to free it. Even then, the analysis might be insufficient. In xalan, for example, the arrays that are dead are themselves stored in arrays, so determining that no aliases exist between them is extremely difficult.

Factory variability. Free-me also misses objects returned by methods that *mostly* behave like factories, but not always. The String method `substring()` is an example. In most cases, `substring()` returns a new String object, but it includes an optimization that simply returns the input string if it equals the search string. In this special case, which is only determined at run-time, the return value is not a new string and might not be dead in the caller. This pattern occurs frequently in the standard libraries. For example, one implementation of `Hashmap.get()` usually returns a new iterator, but will return a global NULL object in certain circumstances. It is not clear how any analysis could automatically determine the conditions under which the output of `substring()` or `get()` can be freed.

4. Runtime Support for Free-Me

This section describes the free() implementations for current allocation disciplines: free-list and bump-pointer allocation. Figure 6 shows four implementations of free(*obj*). The free() interface takes two arguments: a reference to the object to delete, and the size of the object in bytes. The size information is precomputed by the compiler and provided as a constant, when possible. There are cases, such as arrays, where size information is not known statically. In these cases, the system computes the size of the object at

```
1 final int free(ObjectReference obj)
2         throws InlinePragma {
3   int sizeClass = getSizeClass(obj);
4   Memory.zero(obj, bytesInSizeClass(sizeClass));
5   // -- Push object on front of free list
6   setNext(obj, freeList[sizeClass]);
7   freeList[sizeClass] = obj;
8 }
```

(a) Free-List Free

```
1 final int free(ObjectReference obj, int size)
2         throws InlinePragma  {
3   Address  end = obj + size;
4   Memory.zero(start, size);
5   // -- Is object at the end of bump pointer?
6   if (end >= bumpPointer.cursor)
7     // -- Unbump the bump pointer
8     bumpPointer.cursor = obj;
9 }
```

(b) Unbump: Bump-Pointer Free of Top

```
1 final int free(ObjectReference obj, int size)
2         throws InlinePragma  {
3   Address  end = obj + size;
4   Memory.zero(start, size);
5   if (end >= bumpPointer.cursor) {
6     bumpPointer.cursor = obj;
7     // -- Did we expose region?
8     if (unbumpEnd >= bumpPointer.cursor) {
9       // -- Unbump again
10      bumpPointer.cursor = unbumpStart;
11      unbumpStart = 0;
12      unbumpEnd = 0;
13    }
14  } else { // -- Or, remember this object
15    unbumpStart = obj;
16    unbumpEnd = end;
17  }
18 }
```

(c) Unbump Region: Bump-Pointer Free with Top Region

```
1 final int free(ObjectReference obj, int size)
2         throws InlinePragma  {
3   copyReserve = copyReserve - size;
4 }
```

(d) Unreserve: reduce the copy reserve

Figure 6. Selected based on the collector at build-time.

run-time by querying its type information. We found that dynamically querying object size can be a significant overhead.

We simplify these code listings as follows. First, we elide the differences between objects and addresses, and adjustments pointers to account for object headers and alignment. Second, the implementation includes null pointer tests that prevent double frees (see Section 3.4). Third, the implementation includes a test that ensures the input object is in the proper allocation space. We ignore calls to free on objects that are in the immortal space or the boot image space. The implementations also include an inline pragma notifying the optimizing compiler to inline these short sequences.

4.1 Free for a Lazy Free-List

This section explains free() for the MMTk lazy free-list for its mark-sweep collector [7, 8]. This implementation is suitable for mark-sweep-compact, reference counting and any other collector that uses a free-list allocator, with or without size-class blocks. In addition, it is suitable for a more aggressive compiler analysis that can free long and short-lived objects [34], and for systems that cannot move some objects, e.g., with C# and pinning.

MMTk organizes memory into *k* size-segregated *free-lists* using blocks of contiguous memory for same-size objects. Each free-list is unique to a size class. The free-list *collector* traces and marks live objects using bit maps associated with each block. Tracing is thus proportional to the number of live objects. It then places all the partially free blocks on a list. The free-list *allocator* puts a new object into the first free cell of the smallest size class that accommodates the object. If the size-class free-list is exhausted, the allocator creates a new free-list from one of the partially filled blocks or an empty block. Reclamation is thus incremental and proportional to allocation. Although MMTk creates free-lists a block at a time, it does not depend on that feature.

Free() simply links objects to the front of the appropriate size class free-list as shown in Figure 6(a) lines 6 and 7. A subsequent allocation of the same size object will thus reuse it.

4.2 Free for a Bump-Pointer Allocator

A bump-pointer allocator is typically coupled with a copying or compacting collector. We assume a copying collector and an allocator that uses a contiguous block of memory, allocating objects in program order by bumping a pointer until it exhausts the block. We add to this discipline two versions of free(): *unbump* and *unbump region*. Unbump can only free the most recently allocated object, whereas unbump region can free deeper into the recently allocated objects. Any subsequent allocation can reuse this memory, not just same size objects. We also present a variation called *unreserve* that simply reduces the copy reserve by the size of the object, rather than immediately reusing the memory.

Unbump. Figure 6(b) shows pseudocode for the simplest implementation of free(*obj*) in a bump-pointer allocator. If the given object was the last allocation by the bump pointer, free moves the bump pointer back to the start of the object. A subsequent allocation will reuse this memory and move the bump pointer forward again. Notice that there is no way to reuse memory further back behind the bump pointer. In these cases, free simply returns.

Unbump Region. The above implementation forces the compiler to issue the frees in last-in-first-out order. To simplify the compiler analysis, we also investigate a free() that keeps track of a free, but unreclaimed contiguous region closest to the cursor. This free() can always reclaim the top three objects in any order, and may reclaim more. The implementation delimits an unreclaimed region with two pointers, `unbumpStart` and `unbumpEnd`, as shown in Figure 6(c). If *obj* is the top object free() retreats *cursor* to the start of *obj*, the most common case. If the new top object is also free, free() retreats the *cursor* further and returns. Figure 7 shows an example of this case, where unreclaimed memory is shaded.

Our implementation is slightly more sophisticated than the Figure 6(c). It remembers the unreclaimed region closest to the current bump pointer (rather than the most recently freed object), which is the most likely to be reclaimed later. It also coalesces free objects that are adjacent to the current region. Figure 8 shows this case.

Figures 7 and 8 show a limitation of this implementation. Some older free memory goes unreclaimed even though it may eventually reach the top. We investigated weaving a free-list through all free regions, but it did not reclaim significantly more memory, and it is expensive. A free on short-lived objects matches the best behavior of the bump pointer. This structure is a high performance design point because it forms the underpinnings for generational collectors in use in the current best performing systems with garbage collection (e.g., IBM JDK version 1.4.1, and Sun's HotSpot 1.4.2).

Unreserve. As we show in Section 6, free() with a bump pointer does not deliver a performance improvement because, in part, the overhead of manipulating the bump pointer and free regions outweighs space efficiency. We therefore investigated an even simpler version of free() that instead reduces the size of the copy reserve for the copying collector, as shown in Figure 6(d). Since potentially all objects could survive a collection, every copying collector must keep in reserve, memory equal to the size collection region.

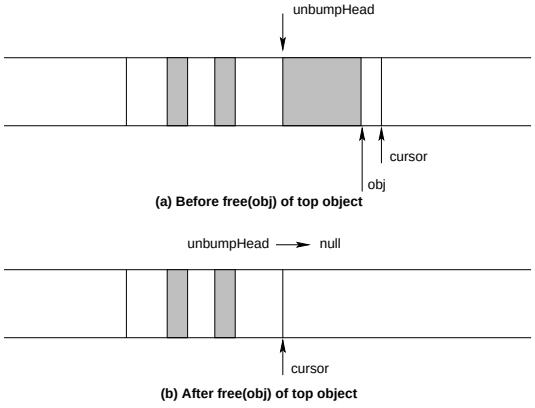


Figure 7. Unbump Freeing of Top Object

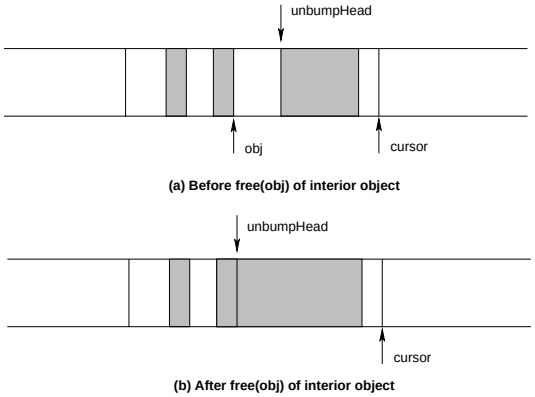


Figure 8. Unbump Region Freeing an Interior Object

Instead of retreating the bump pointer, unreserve simply reduces the reserve space, postponing garbage collection.

5. Methodology

We add free-me compiler analysis and free() runtime support to version 2.4.3 of Jikes RVM and MMTk. Jikes RVM is a high performance VM written in Java with an aggressive adaptive just-in-time optimizing compiler [1, 2, 27]. We use configurations that pre-compile libraries and the optimizing compiler itself (the *Full* build-time configuration), and turn off assertion checking. To measure applications in a deterministic setting, we use *replay methodology* with pre-compilation. Replay builds and uses an advice file that selects the hot methods and their optimization level. It eliminates non-determinism due to the adaptive optimizing compiler and focuses on the application itself. (Eeckhout et al. show that including the optimizing compiler in timing runs on short running programs obscures the application behavior [16].)

We use replay together with pre-compilation in three steps. (1) During construction of the Jikes RVM boot image, we analyze and instrument large portions of the Java standard class libraries. For some benchmarks calls to free() in the class libraries contribute significantly to performance improvements. (2) Offline, we pre-compute method summaries for all methods in the benchmark, store them in a file, and retrieve them during free-me compilation. The summaries mark factories and compute pointer connectivity, as described in Section 3.2. This analysis pass does not modify the benchmark code. (3) Immediately before running the benchmark, we pre-compile the hot methods using full free-me compilation. This compilation pass instruments the methods with calls to free(), as described in Section 3.

The cost of free-me compilation is significant – it almost doubles the time spent in the optimizing compiler. Several factors mitigate this expense. First, free-me compilation is only applied to hot methods, which are a very small fraction of all methods. Second, this cost would be quickly recouped in long-running applications. Third, we have not spent much time optimizing the implementation of the analysis.

The use of precomputed method summaries raises two issues with respect to Java and the Java execution model. First, dynamic class loading may invalidate precomputed summaries, possibly rendering some free-me decisions incorrect. Second, while the cost of computing summaries is not extreme (on average, just a few seconds per benchmark), it is too high to perform in a just-in-time setting for these benchmarks. For long-running server applications, high compilation costs can be amortized over long execution times (days or weeks). Embedded and real-time applications may not use dynamic class loading and may already use ahead-of-time compilation, and are thus likely to benefit from the space efficiency of free-me.

The runtime system is implemented using MMTk – a composable Java memory management toolkit that implements a variety of high performance collectors that reuse shared components [8]. MMTk manages large objects (8K or bigger) separately in a non-copy space, and puts the compiler and a few other system pieces in the boot image, an immortal space. We experiment with MMTk's mark-sweep full heap collector, and a generational collector with an unbounded copying nursery and a mark-sweep older space. Previous work [7] shows these collectors perform well.

We report results on SPECjvm98 [35], pseudojbb, a fixed workload version of SPECjbb2000 [36] and the DaCapo [9] benchmarks. We measure results on a 2.0 GHz Intel Pentium M (755) with a 32 KB L1 data cache, a 32 KB L1 instruction cache, a 2 MB L2 cache, 533 MHz front-side bus, and 2 GB of main memory, running Linux 2.6.12.

6. Experimental Results

This section first presents statistics about the effectiveness of free-me compiler analysis, and then presents the total time, garbage collection time, and mutator time improvements obtained with compile-time inserted frees.

6.1 Effectiveness of Free-Me Analysis

Table 2 presents allocation and free statistics for our free-me compiler analysis. We gather statistics in special instrumented (non-timed) runs with the mark-sweep collector. On average, the free-me analysis frees 32% of all objects and up to 80% in our benchmarks (the *Free* columns). The last two columns (labeled *Stack-like*) show a version of our analysis modified to detect only those cases that could be stack allocated, i.e., if we restrict our analysis to inserting frees for allocations in the same method, and restrict the free instrumentation to the end of the method. This eliminates the benefit of our factory method detection, and conditional freeing. These restrictions reduce the average effectiveness from 32% to 21%, with several benchmarks showing dramatic reductions.

Comparing the *Free* columns with the unconditional (*Uncond*) columns shows the influence of free acting on some paths and not others. For *Uncond*, we modify the free-me analysis to prove objects are dead on all paths – as required for stack allocation. On average, this restriction finds 7% less than if we allow free on some paths and not others. Conditional freeing makes quite a difference to several of the more complex benchmarks: bytes freed is reduced by half or more for javac, jack, antlr, bloat, and pmd.

	alloc	Free		Uncond.		Stack-like	
	MB	MB	%	MB	%	MB	%
SPEC							
compress	105	0	0%	0	0.0%	0	0.0%
jess	263	16	6%	16	6%	16	6%
raytrace	91	73	81%	72	80%	72	80%
db	74	45	61%	45	61%	45	61%
javac	183	24	13%	15	9%	15	9%
mtrt	98	73	75%	73	75%	73	74%
jack	271	163	60%	127	47%	103	38%
pseudojbb	180	34	19%	16	9%	6	3%
DaCapo							
antlr	1544	673	44%	335	22%	146	10%
bloat	716	222	31%	46	7%	35	5%
fop	103	30	30%	24	24%	21	20%
hsqldb	515	57	11%	34	7%	28	6%
jython	348	75	22%	67	20%	3	1%
pmd	822	278	34%	140	17%	56	7%
ps	523	22	4%	18	4%	14	3%
xalan	8195	1607	20%	1584	20%	1566	19%
Average			32%		25%		21%
Potential							
javac-inl	188	51	27%	25	14%	25	14%
xalan-mod	8195	7290	89%	7267	89%	7249	88%
db-mod	74	65	88%	65	88%	65	87%

Table 2. Compile-time Free Decisions: *alloc*: Total allocation, *Free*: Free Amount, *Uncond.*: Unconditional free amount if frees must correspond to allocations, and *Stack-like*: Free amount without factory methods or conditional frees

The last three rows in the table show further potential of our approach on three benchmarks. Unfortunately, the Jikes RVM inliner does not inline *symbolTable.lookup* in the javac benchmark, which is why we only free 13%. If we force the compiler to inline this method, free-me finds 27% (javac-inl). An enhanced analysis could automatically detect this case.

For the two modified benchmarks, db `-mod` and xalan `-mod`, we manually added three frees in key routines that grow array-based containers. For example, the `ArrayList` container increases the size of its array to accommodate new elements. Its `add()` method allocates a new, larger array and copies the elements from the old array. The old array is immediately garbage. We believe a more powerful compiler analysis could detect and exploit such opportunities. Note that even with more powerful analysis, stack and region allocation are unlikely to ever handle these cases. Container expansion is an unpredictable event that does not coincide with any particular program scope, precluding stack allocation. Furthermore, at least one of the arrays is always live, making region allocation extremely inefficient or impossible. Note that we do not include these three versions in any further experiments.

6.2 Free-me in a Mark-Sweep Collector

This section presents the effect of free on GC time, mutator time, and overall execution time in a pure mark-sweep collector, where explicit free helps reduce GC costs significantly. Space limitations prohibit including results for all 16 programs, and thus we present the geometric mean and results for select substantial benchmarks with representative (but not the best!) improvements. We show the geometric mean over all benchmarks with free-me in Figure 9, and bloat, xalan, and javac in Figures 10, 11 and 12, respectively. These figures plot time on the y-axis relative to the best time on the left and time in seconds on the right. The x-axis plots relative heap sizes that vary from the smallest in which the collectors execute to three times that minimum on the bottom, and MB on the top.

Figure 9 shows that on average, free improves total performance by an average of 50% in small heaps, 10% in moderate heaps, and 5% in large heaps. In addition, by examining the smallest heap size for each collector, we see free-me reduces by 25% the smallest heap size in which the benchmarks can execute on average. We see these benefits for almost all benchmarks, with the improvements roughly proportional to the amount of memory explicitly freed.

Specifically, free-me improves raytrace, db, mtrt, jack, javac, pseudojbb, antlr, bloat, fop, hsqldb, jython, pmd, and xalan as expected from examining the data from Table 2. Free-me attains these improvements for the most part by reducing garbage collection time, as illustrated in part (b) of each figure. Free-me also provides improved mutator time, as shown in part (c) of each figure, despite the overhead of calling free(). With free-me the allocator reuses dead objects right away, rather than waiting for the garbage collector to reclaim them. This features helps performance in two ways. First, it improves temporal locality by immediately reusing recently freed memory. Second, it populates the free-list which reduces allocator work. With free-me, the allocator allocates fewer new size-class blocks, and creates fewer free-lists during lazy sweeping to satisfy allocation requests.

Figure 10, and Figure 11 show bloat and xalan, and are typical of programs for which free-me works well. Free-me explicitly deallocates 31% of the memory allocated in bloat and 20% of the memory in xalan. In larger heaps, these numbers translate into modest improvements in GC time, since collection is less frequent. When collection is more frequent in small heaps, free-me yields more significant improvements by reducing memory pressure and delaying collection. Mutator time also improves significantly for both benchmarks. For bloat, the improvement is probably due to reduced allocator work: free-me improves more in smaller heaps, which puts more pressure on the lazy sweeping mechanism. For xalan, improvements are probably due to improved temporal locality, since it improves consistently across heap sizes.

Figure 12 shows javac, a program for which free-me provides only modest improvements. Free-me cannot deallocate a significant amount of memory in javac, limiting the improvement in GC time. However, free-me still improves mutator time by rapidly recycling a commonly used object size.

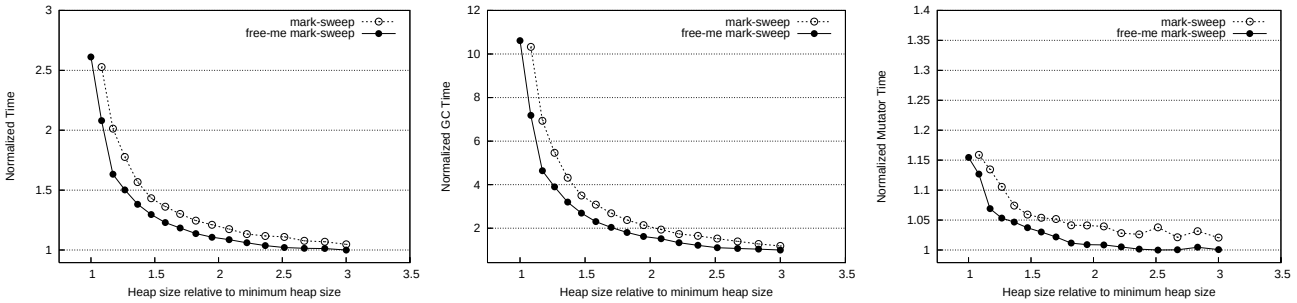
These results demonstrate that free-me improves performance and reduces the memory requirements over a wide variety of benchmarks in a mark-sweep collector. For some benchmarks the improvements are dramatic, while in others they are more modest. However, in no case does free-me degrade performance by any noticeable amount.

6.3 Free-Me in a Generational Collector

In a generational copying nursery and a mark-sweep older space, we find that the nursery reclaims dead objects cheaply and quickly enough that explicit deallocation provides a benefit only for those programs where a large fraction of objects can be explicitly freed. We believe that this effect brings into question any technique that targets short-lived objects, such as stack allocation.

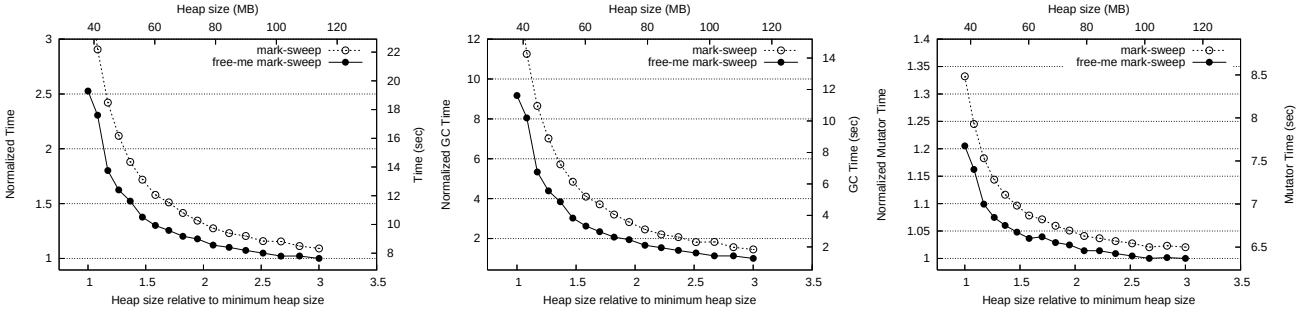
Figure 13 shows the geometric mean of overall time, collection time, and mutator time for two variations of free(): "unbump top" which frees the last object allocated, and "unreserve" which reduces the size of the copy reserve. We do not show the results for unbump region which performs strictly worse than unbump top.

Figures 14 and 15 show representative performance graphs. For javac, free-me has practically no effect on GC time. It does, however, produce an improvement in mutator time, probably for the same reason it does in the mark-sweep collector. For bloat, on the other hand, free-me does result in a modest improvement



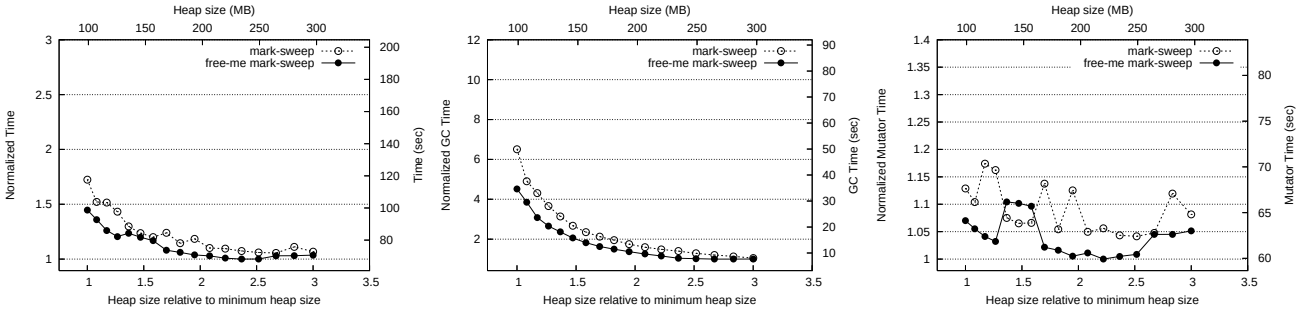
(a) Total Time (b) GC Time (c) Mutator Time

Figure 9. Geometric means over all benchmarks with and without free-me in a mark-sweep collector



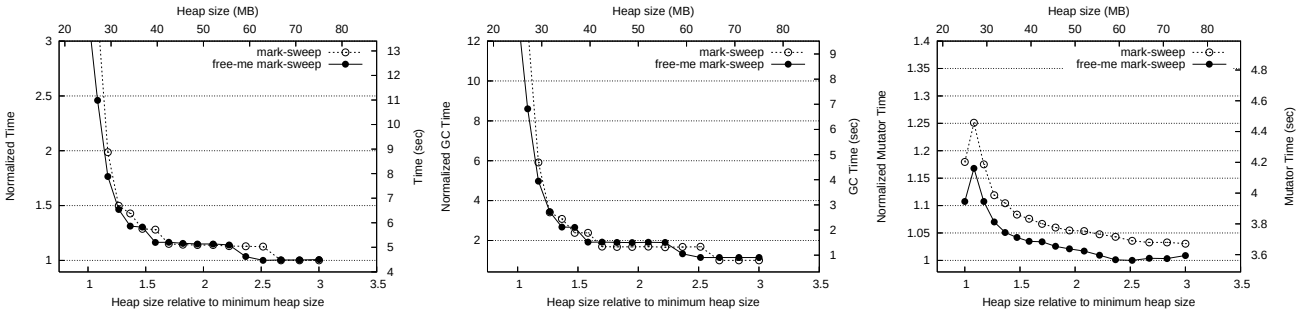
(a) bloat Total Time (b) bloat GC Time (c) bloat Mutator Time

Figure 10. bloat with and without free-me in a mark-sweep collector



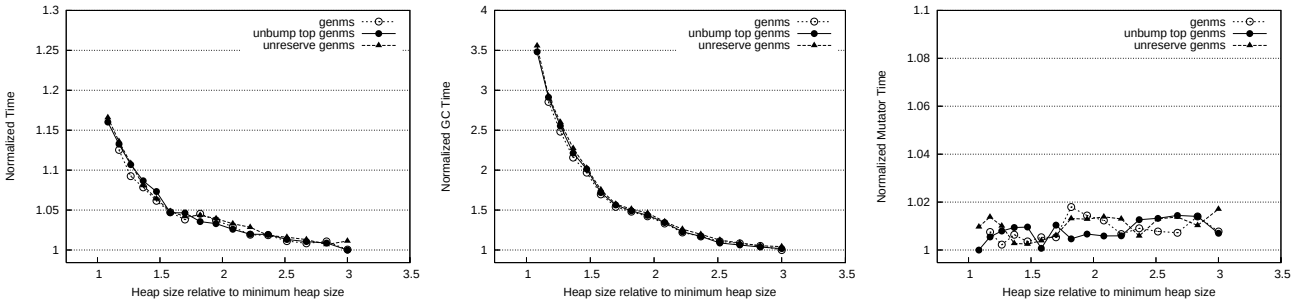
(a) xalan Total Time (b) xalan GC Time (c) xalan Mutator Time

Figure 11. xalan with and without free-me in a mark-sweep collector



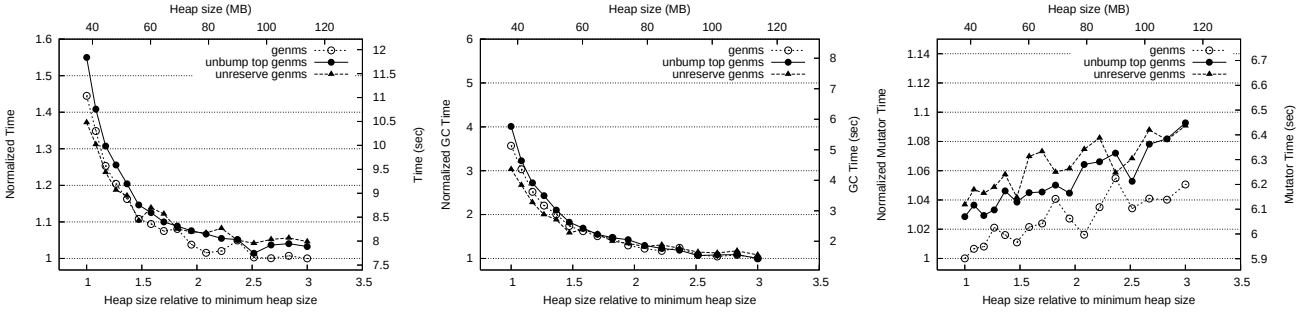
(a) javac Total Time (b) javac GC Time (c) javac Mutator Time

Figure 12. javac with and without free-me in a mark-sweep collector



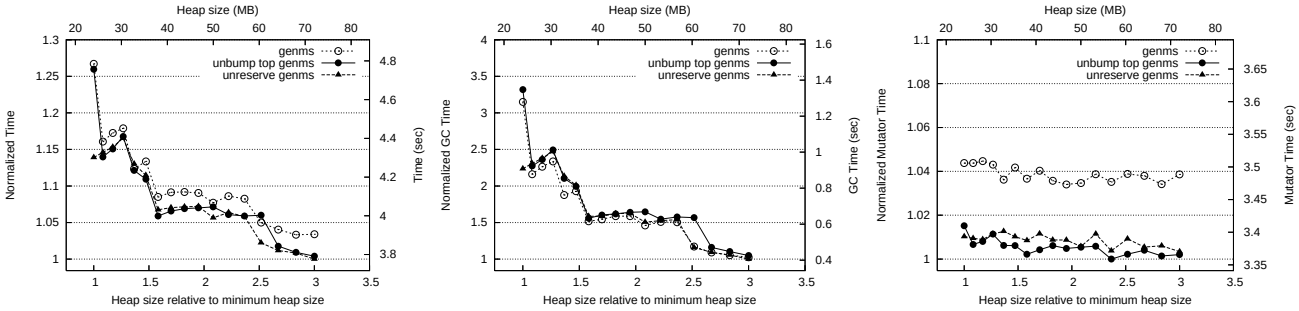
(a) Total Time (b) GC Time (c) Mutator Time

Figure 13. Geometric means over all benchmarks with and without free-me in a generational collector



(a) bloat Total Time (b) bloat GC Time (c) bloat Mutator Time

Figure 14. bloat with and without free-me in a generational collector



(a) javac Total Time (b) javac GC Time (c) javac Mutator Time

Figure 15. javac with and without free-me in a generational collector

in GC time in smaller heaps. This benefit is offset by a loss in performance for the mutator. This loss is due to the overhead of free(); in particular, the cost of querying objects and computing their sizes. This result suggests that any system for individual object reclamation needs to be careful to keep the cost of free() low. Even if we discount the mutator overhead, as stack allocation might achieve, the gains are modest.

For almost all benchmarks free-me has the intended effect of significantly reducing the number of nursery collections. For example, it reduces nursery collections from 16 to 4 on db in the smallest heap size. Unfortunately, this reduction has practically no effect on collection time because it does not significantly reduce the number of objects that survive nursery collections. Since the bulk of a copying collector's work is proportional to nursery *survivors*, free-me hardly affects collection time at all.

While free-me is not an improvement in this setting, it does allow the mark-sweep collector to perform more competitively with the generation collector. Comparing the graphs for bloat in Figures 10 and 14, the two collectors perform similarly in large heaps. In tight heaps, however, mark-sweep suffers considerably. With free-me, though, mark-sweep moves closer to the performance of the generational collector.

7. Related Work

This section overviews the related work on compile-time object reuse (also known as object scalar replacement) and lifetime analysis, and compiler analysis for stack and region allocation.

7.1 Compile-Time Free and Reuse Analysis

Shaham et al. [34] is closest to our work. They identify the last *use* of an object and free it to eliminate the need for *any* garbage

collection. They also null any pointers to it (since the object may still be reachable). Their analysis is very precise and expensive since it seeks to prove liveness for heap variables across the entire program, and thus they demonstrate it only on toy programs. Our approach is simpler and cheaper since it limits its scope.

Other work automates object merging (hash consing, object reuse, and object scalar replacement) [5, 21, 27, 28, 30]. These approaches attain reuse only for same size objects with lifetime homogeneity. Lee and Yi's analysis inserts frees only for immediate reuse, i.e., before an allocation of the same size [30]. Gheorghioiu et al. [21] find allocation sites for which only one object instance is ever live, finding many fewer dynamic objects than our approach. Our free implementations do not require call-site lifetime homogeneity. Our bump-pointer free is not restricted to same size objects and thus reuses the same memory for different sized objects.

Marinov and O'Callahan profile to find *object equivalence* in which their contents are the same, but their lifetimes are disjoint [31]. For SPECjvm98 and two Java server programs, they report memory savings of 2% to 50% if all equivalent objects could be merged. Their results provide motivation for our work, but we use the compiler to realize these savings and are not restricted to equivalent content or sized objects.

Inoue et al. [26] explore the limits of lifetime predictability for allocation sites. They find that many objects have zero lifetimes, and our free-me analysis finds a similar number of objects to free. Our technique differs from lifetime analysis because it detects exactly which update kills an object, rather than its lifetime.

7.2 Stack Allocation

Prior work explores using pointer *escape analysis* to detect allocation sites that produce objects whose lifetimes correspond to method scope and allocate them on the stack [10, 13, 20, 39]. Dynamic stack allocation changes the allocator [13, 14, 39], and static stack allocation adjusts the stack frame [20]. The static approach cannot stack allocate allocations in loops. The dynamic approach can grow the stack without bound. Both implementations assume that stack frame lifetimes are relatively small, and thus the system will normally reclaim this memory faster than the collector. Our free scheme guarantees prompt reclamation since it need not wait for the method return and can free objects in loops and from allocation sites where some objects escape.

Whaley and Rinard [39] provide the most precise escape analysis [10, 13, 20], but no implementation of stack allocation. They measure the amount of memory classified as stack allocatable and report a higher percentages compared to Choi et al. or Blanchet on similar programs [10, 13], e.g., 25% for javac. Choi et al. describe a flow-sensitive and insensitive escape analysis that allocates from 2% to 65% on the stack. However, they state: "Performance gains come mainly from synchronization elimination rather than from stack allocation." Choi et al. point out the potentially unconstrained stack growth did not occur in practice. Blanchet's system dynamically stack allocates between 13 and 95% of memory, 13% for javac. Blanchet reports a mark-sweep free-list collector on one heap size. He finds excellent collection time reductions and mutator locality benefits from contiguous stack allocation. We find more substantial improvements in small to moderate heap sizes.

Gay and Steensgaard [20] and Blanchet [10] provide faster less precise analysis than other escape analyses [13, 39]. Their static stack allocation mechanism increases the stack frame size by up to 24KB, but usually by 1KB or less. They speedup a copying nursery generational collector on one heap size by 11% on jack, but on average, performance benefits are limited [18, 20]. Stack allocation has less overhead than our free with a copying nursery, but does not deliver consistent improvements. Copying nurseries reclaim short-lived dead objects very efficiently.

Our compiler analysis is simpler and less precise than prior work. It should thus be more amenable to use in a just-in-time compiler, although we have not yet performance tuned it. Prior evaluations of stack allocation have only ever used one collector and one heap size. We evaluate compile-time inserted free in several garbage collectors with a variety of heap sizes which exposes the space time tradeoffs inherent in garbage collection.

7.3 Region Allocation

Region allocation either manages all of memory based on allocation-site lifetime scoping [12, 19, 32, 37] or adds regions as a special purpose component management [6, 22, 24]. Regions provide programmability benefits for real-time systems and offer safety features such as thread isolation in server applications, but these features come without the software engineering advantages of garbage collection. Potential advantages include improved memory efficiency, but prior work has not consistently demonstrated this improvement. For example, Hicks et al. [24] show space efficiency improvements in Cyclone over garbage collection alone, but Cherem and Rugina [11] actually increase the memory footprint in Java programs by up to 101%. These mixed results have their roots in requiring a program point when all objects from a specific allocation site are dead, rather than our approach that decouples object allocation from its free.

8. Conclusions

This paper presents a new analysis for identifying short-lived objects and inserting explicit memory deallocation at the points where the objects die. Our analysis properly identifies a large fraction of short-lived objects for our Java programs, which results in rapid, incremental reclamation of memory. For mark-sweep collectors, explicitly freeing objects yields substantial performance improvements from 50% to 200%. However, our experiments show that generational collectors are extremely effective at reclaiming short-lived objects. We believe it is unlikely that any technique can beat the performance of copying generational collection on short-lived objects. However for real-time systems and memory constrained embedded systems, free-me offers a way to combine the software engineering benefits of garbage collection with the memory and performance benefits of incremental collection.

Acknowledgments

We would like to thank Steve Blackburn for getting us started, the entire Jikes RVM research team for making their system publicly available, and Mike Bond and Gene Novark for their suggestions.

References

[1] B. Alpern et al. Implementing Jalapeño in Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 314–324, Denver, CO, Nov. 1999.

[2] B. Alpern et al. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, February 2000.

[3] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[4] D. F. Bacon, P. Cheng, D. Grove, and M. T. Vechev. Syncopation: Generational real-time garbage collector in the metronome. In *ACM Languages, Compilers, and Tools for Embedded Systems*, pages 183–192, Chicago, IL, June 2005.

[5] J. M. Barth. Shifting garbage collection overhead to compile time. *Communications of the ACM*, 20(7):513–518, July 1977.

[6] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. In *ACM Conference on Object-Oriented*

Programming Systems, Languages, and Applications, pages 1–12, Seattle, WA, Nov. 2002.

[7] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: The performance impact of garbage collection. In *ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*, pages 25–36, NY, NY, June 2004.

[8] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and water? High performance garbage collection in Java with JMTk. In *International Conference on Software Engineering*, pages 137–146, Scotland, UK, May 2004.

[9] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, S. Z. Guyer, A. Hosking, M. Jump, J. E. B. Moss, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java benchmarking development and analysis. Technical Report TR-CS-06-01, Deptartment of Computer Science, Austrailian National University, Mar. 2006. http://ali-www.cs.umass.edu/DaCapo/-Benchmarks.

[10] B. Blanchet. Escape analysis for Java: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 25(6):713–775, Nov. 2003.

[11] S. Cherem and R. Rugina. Region analysis and transformation for Java programs. In *ACM International Symposium on Memory Management*, pages 85–96, Vancouver, BC, 2004.

[12] W. Chin, F. Craciun, S. Qin, and M. Rinard. Region inference for object-oriented language. In *ACM Conference on Programming Languages Design and Implementation*, pages 243–354, Washington, DC, June 2004.

[13] J. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Transactions on Programming Languages and Systems*, 25(6):876–910, Nov. 2003.

[14] C. Click. Stack allocation, Jan. 2005. Personal Communication.

[15] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.

[16] L. Eeckhout, A. Georges, and K. D. Bosschere. How Java programs interact with virtual machines at the microarchitectural level. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 244–358, Anaheim, CA, Oct. 2003.

[17] Y. Feng and E. D. Berger. A locality-improving dynamic memory allocator. In *ACM Conference on Memory System Performance*, pages 1–12, Chicago, IL, June 2005.

[18] R. P. Fitzgerald, T. B. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi. Marmot: An optimizing compiler for Java. *Software—Practice and Experience*, 30(3):199–232, 2000.

[19] D. Gay and A. Aiken. Language support for regions. In *ACM Conference on Programming Languages Design and Implementation*, pages 70–80, Snowbird, UT, 2001.

[20] D. Gay and B. Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *International Conference on Compiler Construction*, pages 82–93, Berlin, Germany, 2000.

[21] O. Gheorghioiu, A. Salcianu, and M. Rinard. Interprocedural compatibility analysis for static object preallocation. In *ACM Symposium on the Principles of Programming Languages*, pages 273–284, New Orleans, LA, Jan. 2003.

[22] N. Hallenberg, M. Elsman, and M. Tofte. Combining region inference and garbage collection. In *ACM Conference on Programming Languages Design and Implementation*, pages 141–152, Berlin, Germany, June 2002.

[23] M. Hertz and E. Berger. Quantifying the performance of garbage collection vs. explicit memory mananagement. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, San Diego, CA, Oct. 2005.

[24] M. Hicks, G. Morrisett, D. Grossman, and T. Jim. Experience with safe manual memory-management in Cyclone. In *ACM International Symposium on Memory Management*, pages 73–84, Vancouver, BC, 2004.

[25] M. Hirzel, A. Diwan, and J. Henkel. On the usefulness of type and liveness accuracy for garbage collection and leak detection. *ACM Transactions on Programming Languages and Systems*, 24(6):593–624, Nov. 2002.

[26] H. Inoue, D. Stefanović, and S. Forrest. Object lifetime prediction in Java. Technical Report TR-CS-2003-28, University of New Mexico, May 2003.

[27] Jikes RVM. IBM, 2005. http://jikesrvm.sourceforge.net.

[28] S. B. Jones and D. Le Métayer. Compile-time garbage collection by sharing analysis. In *ACM Inteornational Conference on Functional Programming Languages and Computer Architecture*, pages 54–74, Nov. 1989.

[29] D. Lea. A memory allocator. http://gee.cs.oswego.edu/dl/html/malloc.html, 1997.

[30] O. Lee and K. Yi. Experiments on the effectiveness of an automatic insertion of memory reuses into XSML-like programs. In *ACM International Symposium on Memory Management*, pages 97–108, Vancouver, BC, 2004.

[31] D. Marinov and R. O'Callahan. Object equality profiling. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 313–325, Anahiem, CA, Oct. 2003.

[32] F. Qian and L. Hendren. An adaptive, region-based allocator for Java. In *ACM International Symposium on Memory Management*, Berlin, Germany, June 2002.

[33] R. Shaham, E. K. Kolodner, and M. Sagiv. Heap profiling for space-efficient Java. In *ACM Conference on Programming Languages Design and Implementation*, pages 104–133, Snowbird, UT, 2001.

[34] R. Shaham, E. Yahav, E. K. Kolodner, and M. Sagiv. Establishing local temporal heap safety properties with application to compile-time memory management. In *Static Analysis Symposium*, pages 483–503, San Diego, CA, June 2003.

[35] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, release 1.03 edition, March 1999.

[36] Standard Performance Evaluation Corporation. *SPECjbb2000 (Java Business Benchmark) Documentation*, release 1.01 edition, 2001.

[37] M. Tofte and J. Talpin. Region-based memory management. *Information and Computation*, 1997.

[38] D. M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *ACM Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, April 1984.

[39] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 187–206, Nov. 1999.