A Static Analysis for Automatic Individual Object Reclamation in Java

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

Automatic garbage collection has proven software engineering benefits: fewer memory-related errors and less programmer effort. However, to attain performance competitive with explicit memory management, garbage collection needs much more memory. This key tradeoff exists because the collector only reclaims memory when it is invoked: invoking it more frequently reclaims memory quickly, but incurs a significant cost, while invoking it less frequently fills memory with dead objects.

This work comes closer to the best of both worlds by adding novel runtime and compiler support for compiler-inserted frees to a garbage-collected system. The compiler analysis automatically identifies when objects become unreachable and inserts calls to free. The analysis combines a simple interprocedural pointer analysis with liveness information. The free() implementation depends on the allocator, and we demonstrate variations for *free-list* and *bump-pointer* allocators. Explicitly freeing objects reduces memory requirements by reclaiming memory quickly, reduces garbage collection load, and improves performance, often substantially for mark-sweep collectors. Compared to the baseline, free-me cuts total time by 22% on average, collector time by 50% to 70% on average, and allows programs to run in 17% less memory on average.

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 can free objects on some program paths and not on others. Our system also handles common design patterns, such as freeing inside loops and objects created by factory methods.

Categories and Subject Descriptors

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

General Terms

Languages, Performance, Experimentation, Algorithms

Submitted to PLDI'06, June, 2006, Ottawa, Canada.

Keywords

adaptive, generational, compiler-assisted, locality

1. INTRODUCTION

This work seeks to combine some of the performance benefits of explicit memory management with the software engineering benefits of automatic memory management (garbage collection). Explicit memory managers typically implement free-lists optimized for low-cost allocation and freeing [5, 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, 34]. Garbage collectors require extra memory, beyond what the application is using, because they only periodically reclaim memory, allowing dead objects to acculumate. Collecting more frequently is prohibitively 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, which increases the memory footprint to reduce collection costs [23, 25, 26, 34, 39].

Our goal is to obtain the best of both of these approaches by combining the software engineering benefits of garbage collection with the space efficiency of incremental object reclamation. We present new runtime and compiler support for automating prompt and cheap object reclamation in a garbage collection system. 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. Free-me analysis can thus identify and free an object that dies on one path, but not another. 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. However, it includes an interprocedural component that identifies *factory* methods: those methods whose only side effect is to return one newly allocated object.

The underlying garbage collection discipline dictates the implementation of free. We implement several variants a mark-sweep collector that uses free-lists, and for a copying collector. Our free-list implementation [6, 7] provides multiple free-lists segregated by size. Free simply returns the memory cell to the front of the appropriate free-list, as would explicit memory management. Since we leave unchanged the bump-pointer allocator in a copying collector, we explore a version of free that reclaims the object only when it is the last allocation. We find this version is too restrictive because

it requires the compiler to perfectly order (last-in-first-out) frees. We then explore a more powerful version of free that tracks one unreclaimed region closest to the top. If a subsequent free brings it to the top, free reclaims it as well. Finally, we show a version of free that simply reduces the required size of the copy reserve by the number of bytes freed.

Compared with region [11, 24, 33, 38] and stack [9, 12, 20, 40] 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 become unreachable. Although stack and region allocation reduces collector load and has 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, 6, 7]. For mark-sweep, our results on SPECjvm98, SPECjbb2000, and DaCapo [8, 36, 37] Java benchmarks show that free reclaims on average 28% of all objects: on 4 benchmarks it reclaims less than 10%, but for 9 other benchmarks it reclaims 19% or more, including 5 benchmarks where it reclaims at least 50% of the objects. 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 copying generational collector results are more surprising. Although free reclaims almost the same objects as mark-sweep free and these frees translate into many fewer nursery collections, the survival rate of each collection goes up. Eliminating short-lived objects from the nursery did not have the expected effect of yielding fewer survivors by giving objects more time to die. In fact, short-lived objects do not contribute much to collection cost since the copying collector never touches them. Furthermore, mutator time degraded due to the overhead of incrementally *unbumping* objects. These results combined with similar ones for stack allocation [9, 12, 20, 40] indicate that collecting short-lived objects seems to be best left to a copying nursery.

Although generational collectors typically perform much better than whole heap mark-sweep collectors [6], non-moving collectors remain critical in certain applications. For example, embedded systems use mark-sweep for space efficiency, and C# uses mark-sweep to support pinning an object in memory. For these systems, we recommend adding compiler-inserted frees, which yields substantial improvements in memory efficiency and performance benefits.

2. 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.

2.1 Compile-Time Free and Reuse Analysis

Shaham et al. [35] is closest to our work. Their analysis identifies the last *use* of an object and frees it to eliminate the need for garbage collection. They also null any pointers to it (since the object may still be reachable) to communicate object death the garbage collector. 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 simpler and cheaper approach limits its scope to a single method.

Other prior work automates object merging (hash consing, object reuse, and object scalar, replacement) [4, 21, 27, 28, 30]. These approaches require the same size object to attain reuse and call site lifetime homogeneity. For example, Lee and Yi's analysis inserts frees only for immediate reuse, i.e., before an allocation of a new object of the same size [30]. The analysis of Gheorghioiu et al. [21] finds allocation sites for which only one object instance is ever live. In practice, their analysis finds many fewer dynamic objects than ours. Our free implementations do not require call-site lifetime homogeneity, and can free an object at any point it becomes dead. 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 [32]. Two objects are equivalent if their contents are the same and their lifetimes are disjoint. For SPECjvm98 and two Java server programs, they report a 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] explores 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.

2.2 Stack Allocation

Prior work explores using pointer *escape* analysis to detect allocation sites that produce objects whose lifetimes correspond to the current method [9, 12, 20, 40]. This work then allocates these objects on the stack. It seeks to improve garbage collection by reclaiming memory sooner than the collector would. Implementations of stack allocation dynamically add objects by changing the allocator [12, 13, 40], or size the stack frame statically [20]. The later static approach cannot stack allocate allocations in loops. The former can grow the stack without bound. Both implementations assume that the lifetime of a stack frame is 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 [40] provide the most precise escape analysis [9, 12, 20]. They do not provide an implementation of stack allocation, but measure the amount of memory they classify as stack allocatable. Their precision pays off; they report a higher percent of stack allocatable objects than Choi et al. or Blanchet on similar programs [9, 12]. For example on javac from SPECjvm98 [36], they classify 25% of all allocation as stack allocatable. Choi et al. describe a flow-sensitive and insensitive analysis and allocate from 2% to 65% on the stack. However, they state: "The performance gains come mainly from synchronization elimination rather than from stack allocation." Choi et al. point out that since they use dynamic stack allocation, they are subject to unconstrained stack growth, but did not find it in practice. Blanchet's system dynamically stack allocates between 13 and 95% of memory, 13% for javac and 21% for jess from SPECjvm98. Blanchet's 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 [9] provide faster less precise analysis than other escape analyses [12, 40]. They capture between 1% and 62% of all memory. Their static stack allocation mechanism increases the stack frame size by up to 24KB, but usu-

ally by 1KB or less. They report speed ups in a copying nursery generational collector on one heap size of up to 11% on jack from SPECjvm98, but on average, performance benefits are limited [18, 20]. As we mentioned in the introduction, copying nurseries reclaim short-lived dead objects very efficiently. Their stack allocation mechanism has less overhead than our free with a copying nursery, but does not deliver consistent improvements.

Our compiler analysis is simpler and less precise than prior work for stack allocation. It should thus be more amenable to use in a just-in-time compiler, although we have not yet performance tuned it. Prior work on stack allocation has only 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.

2.3 Region Allocation

Region allocation is a general memory management approach that either manages all of memory based on allocation-site lifetime scoping [11, 19, 33, 38] or adds regions as a special purpose component in explicit or automatic memory management [5, 22, 24]. Regions can 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 [10] actually increase the memory footprint in Java programs by 10% on average on the SPECjvm98 benchmarks, and 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 more incremental approach that decouples object allocation from its free.

3. MOTIVATING EXAMPLE

Figure 1 shows an example that motivates the use of free-me analysis instead of escape analysis. Figure 1(a) is taken from javac. In this method, *stream.readToken* is a factory method that produces a single allocation and has no other side effects. The variable `idName` points to this newly allocated object. The loop only adds the symbol (`idName`) to the symbol table if it is not already in it. Since programs tend use symbols repeatedly, more die than persist. Since those objects that persist escape the method, it is not safe to stack allocate them. Free-me compiler analysis detects that `stream.readToken` is a factory method that produces a single object, stored in `idName`. (Free-me analysis requires inlining to prove `symbolTable.lookup` does not store `idName`.) It then proves `idName` is only live by virtue of being stored in the global variable `symbolTable` on the `if` branch and is dead on the `else`. It then inserts `free(idName)`, as shown in Figure 1(b). We use this running example through out 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 }}
```

(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 free(idName);
10    computeOn(id);
11 }}
```

(b) Compiler inserts free on else branch.

Figure 1: Example of conditional free from javac.

4. FREE-ME COMPILER ANALYSIS

This section describes our *free-me* compiler analysis, which consists of three parts: (1) a simple pointer analysis that discovers object connectivity, (2) a liveness analysis that computes liveness for both variables and pointers in the connectivity graph, and (3) a free placement analysis that determines when objects are dead and selects program points to insert calls to free(). The analysis processes one method at a time and computes simple method summaries, which provide some interprocedural information.

The analysis works on the Jikes RVM optimizing compiler internal representation. This IR consists of 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. In addition, we perform our analysis when the IR is in SSA form [14], which provides flow sensitivity for the local variables.

4.1 Pointer Connectivity

The pointer analysis is flow insensitive and inclusion based, similar to Andersen's pointer analysis [3]. It builds a connectivity graph with nodes for each allocation site, each parameter, and one node for all globals. It also keeps track of the targets of all local variables. The following tables presents the analysis data structures.

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

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 the inner nodes N_I represent any other objects reachable from the parameter nodes. Note we assume no aliasing between parameters, which is correct here 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:

$\forall i, PtsTo(p_i) = \{N_{Pi}\}$	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 adding 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. The reason this rule is so conservative is so we can produce very simple interprocedural summaries (see Section 4.2). It has very little effect on accuracy because it does not drastically change the overall reachability of objects.

4.2 Procedure Summaries

Like many analysis algorithms, ours benefits significantly from information about the callees of a method. During analysis we compute simple summaries that capture how each method connects objects passed to it. The summaries consist of a set of pairs (p_i, p_j), where p_i and p_j refer to parameters. The pair (p_i, p_j) indicates that the method connects these two parameters, by some sequence of pointers, so that p_j is reachable from p_i. The following table shows how we compute each entry and includes special entries for global pointers and objects that are returned. The notation $*p_j$ means apply *getfield* to the argument first to obtain the inner node.

$N_{pj} \in PtsTo*(p_i)$	$\Rightarrow$	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

The last type of entry identifies "factory" methods: methods that return new objects. Our analysis requires that a method return *only* new objects in order to qualify as a factory. In the caller, we treat calls to such a method as an allocation, and apply the same free-me analysis as if it were an allocations site. For example, Figure 1 contains a factory method `Stream.readToken()`. The symbol table `add()` method stores each of the input parameters in the receiver object. We represent this information with two summary entries: (p_0, p_1) and (p_0, p_2).

When the analysis encounters a method call, it looks up the possible targets (there can be more than one for virtual calls) and applies the method summary to the actual arguments. Entries of the form (p_i, p_j) are implemented as putfield operations. Entries of the form $(return, p_j)$ are implemented by passing the points-to set of p_j to the left-hand side of the call site. Finally, methods marked as factories introduce a new allocation node associated with the call site.

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 actual callee. 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 analysis fully context sensitive. The results of the analysis, however, can only be used to test overall reachability. This algorithm is unsuitable for other problems, such as aliasing.

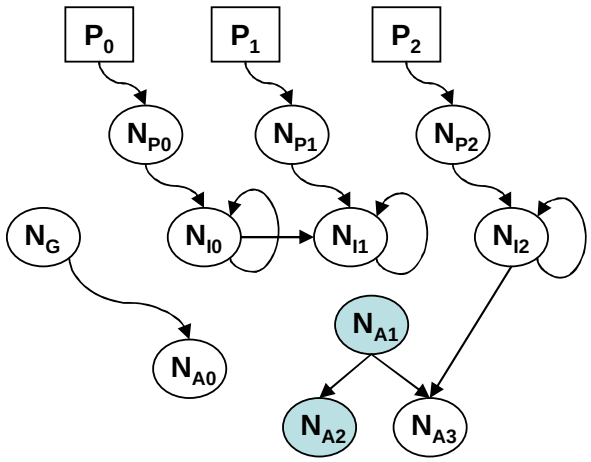


Figure 2: Example connectivity graph: each p_i is a parameter variable, each N_P is a parameter node, each N_A is an allocation node.

4.3 Variable Reachability and Liveness

Figure 2 shows an example connectivity graph for an unspecified program. It includes four allocation nodes, N_{A0} to N_{A3}. Two nodes, N_{A1} and N_{A2} do not escape the method, and therefore can be freed. The graph also suggests that N_{A0} and N_{A3} cannot be freed, since they are reachable from a parameter and a global, respectively. As Figure 1 shows, however, this conclusion depends on *when* the program creates these pointers.

To solve this problem, we sharpen the flow-insensitive connectivity graph with flow-sensitive liveness information. This analysis starts by computing the liveness of each local variable, and then propagates this information through the connectivity graph to determine the specific program points at which objects are reachable. A key insight we apply is that the program must have both allocated the object and instantiated any pointers to it, for the node to be live.

The liveness of a node is determined by the union of the liveness of all its incoming pointers. In the case of pointers from variables, liveness has a well-defined meaning: the live range of the variable. For our points-to graph, however, we define liveness as all program points reachable *after* the program creates the pointer. This definition is essential to our analysis because it distinguishes between program paths on which an object is live and program paths on which it does not.

Our analysis computes liveness as sets of edges in the control-flow graph. In order to support fine-grained freeing, we use a representation in which each instruction is a separate node in the CFG. The algorithm computes the liveness of each node iteratively as follows:

$$live(N_i) = \bigcup live(v)\ \forall v \in V, N_j \in PtsTo(v) \cup \bigcup live(N_j \rightarrow N_i)\ \forall N_j, N_i \in PtsTo(N_j)$$

$$live(N_j \rightarrow N_i) = \{\text{all edges reachable from the statement that creates the pointer}\}$$

In the example in Figure 1, statement 7 creates the pointer from the symbol table to the string `idName`. Therefore, the liveness of this pointer extends from statement 7 to 9, and on. It does not include the edge from statement 5 to 9, which is the false branch. Note that the loop could cause all edges to be marked reachable from statement 7. To handle this situation, we use the following observation: a pointer can only exist *after* the object is created. There is no way for the symbol table to point to `idName` between statements 3 and 7.

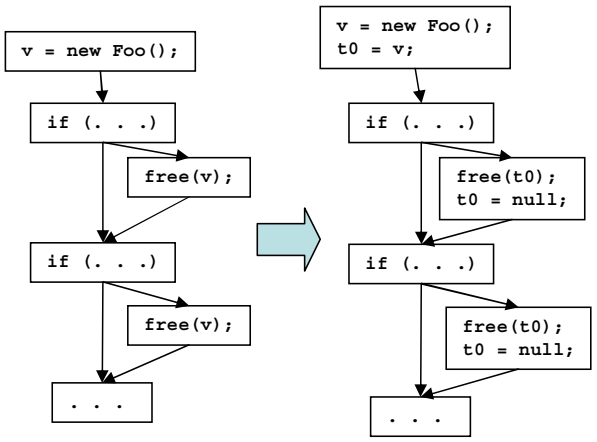


Figure 3: Free-me instrumentation uses temporary variables to allow free() on multiple paths through the same code.

4.4 Free Placement

Finally, we compute the possible places to insert a call to free for each object. We start with all control-flow edges on which the object exists (all edges after the `new`), and subtract all edges on which the object is live. This difference identifies the program points where the object becomes unreachable and therefore can be freed. It 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 must be careful not to call free() more than once on a single object. To avoid 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 the temporary to null. Subsequent calls to free() recognize the null value as a special case and simply return. Figure 3 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.

5. RUNTIME SUPPORT FOR FREE

This section describes the free implementations for current allocation disciplines: free-list and bump-pointer allocation. Figure 4 shows implementations of free(*obj*) which first zero the object, following the Java memory model. They include an inline pragma notifying the optimizing compiler to inline these short sequences. For simplicity, we omit the null check.

5.1 Free for a Lazy Free-List

A free-list allocator can be coupled with mark-sweep, mark-sweep-compact, reference counting, or other collectors. First we explain how MMTk implements a lazy free-list for its mark-sweep collector [6, 7], and then describe the implementation of free in this setting.

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. It lazily weaves a linked-list of free cells (a free-list) for a block during allocation, making reclamation incremental and proportional to allocation.

The free-list *allocator* puts a new object into the first free cell on

```
1  public final int free(ObjectReference obj)
2          throws InlinePragma {
3      Address block = getBlock(obj);
4      int sizeClass = getSizeClass(block);
5      int      size = getCellSize(sizeClass);
6      Address  cell = obj.address - HEADER_SIZE;
7
8      Memory.zero(cell, size);
9      cell.store(freeList[sizeClass]);
10     freeList[sizeClass] = cell;
11 }
```

(a) Free-List Free

```
1  public final int free(ObjectReference obj)
2          throws InlinePragma  {
3      Address start = obj.address - HEADER_SIZE;
4      Extent   size = size(obj);
5
6      Memory.zero(start, size);
7      if (!cursor.Space(obj)) return;
8      if (size.GTE(cursor))
9          cursor = start;
10 }
```

(b) Unbump: Bump-Pointer Free of Top

```
1  public final int free(ObjectReference obj)
2          throws InlinePragma  {
3      Address start = obj.address - HEADER_SIZE;
4      Extent   size = size(obj);
5
6      Memory.zero(start, size);
7      if (!cursor.Space(obj)) return;
8      if (size.GTE(cursor)) {
9          cursor = start;
10         if (unbumpReach.GTE(cursor)) {
11             cursor = unbumpHead;
12             unbumpHead  = Address.zero;
13             unbumpReach = Address.zero;
14     } }
15     else FreeTopRegion(obj, size);
16 }
```

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

Figure 4: Selected based on the collector at build-time.

the list 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 from an empty block. Although, MMTk creates free-lists from a single block, it does not require that free cells on the list come from the same block.

Free simply links objects to the front of the appropriate size class free-list as shown in Figure 4(a) line 5. A subsequent allocation of the same size object will thus reuse it. This implementation is suitable for any segregated-fits garbage collector or explicitly managed free-list 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 [35]. It is also appropriate for systems that prevents the collector from moving some objects, e.g., with C# and pinning.

5.2 Free for a Bump-Pointer Allocator

A bump-pointer allocator demands 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 that target the most recently allocated (*top*) object(s) by *unbumping* the pointer, *unbump* and *unbump region*. Any subsequent allocation can reuse this memory, not just one of the same size.

Unbump. Figure 4(b) shows pseudocode for the simplest implementation of free(*obj*) in a bump-pointer allocator. We omit alignment and let *cursor* be the allocation point. Free returns after zeroing the memory if an intervening collection moved it (zeroing the

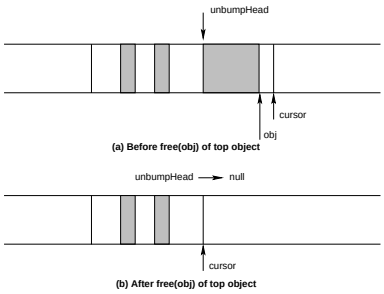


Figure 5: Unbump Freeing of Top Object

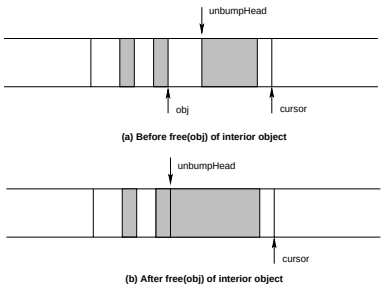


Figure 6: Unbump Region Freeing an Interior Object

object has the additional benefit of eliminating unnecessary retention the object may cause). If *obj* is the top object, free retreats *cursor* by (*size(obj)*). Otherwise, it does nothing.

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 also 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 can reclaim more in some cases. We optimize free of the top most object(s), the most common case, as shown in Figure 4(c). If *obj* is the top object free retreats *cursor* to the start of *obj*. If the new top object is also free, free retreats the *cursor* further and returns. Figure 5 shows an example of this case, where free but unreclaimed memory is shaded.

Otherwise, *obj* is an interior object and free calls *FreeTopRegion()*. This method tracks the top most free region, *unbumpHead*. It takes action in the following mutually exclusive cases. (1) If *unbumpHead* is empty, it sets *unbumpHead* to *obj*. (2) If *obj* is adjacent to *unbumpHead*, it coalesces the two by extending *unbumpReach*. Figure 6 shows an example of this case. (3) If *obj* is closer to cursor than *unbumpHead*, it sets *unbumpHead* to *obj*.

Figures 5 and 6 also 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, Sun's HotSpot 1.4.2, and Microsoft's .NET framework). Note that reclaiming short-lived objects is also the forte of a copying nursery since it only touches and copies live objects, and then reclaims all memory en masse. Since the cost of short-lived objects is low in this case, improving it has proven challenging for stack allocation and, as we show here, for free.

Unreserve. As the results will show, free in a bump-pointer does not deliver a performance improvement. Therefore, we investigated an even simpler version of free that simply reduces the size of the copy reserve for the copying collector. Since potentially all objects could survive a collection, every copying collector must keep in reserve memory equal to the size collection region. Instead of retreating the bump-pointer, unreserve simply reduces the reserve space which will postpone garbage collection. A limitation of our unbump implementation is that it dynamically looks up the object size, with unreserve, we simply and conservatively subtract the smallest object size.

6. METHODOLOGY

We add free-me compiler analysis and free runtime support to version 2.3.6 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 a replay precompile methodology. This methodology uses an advice file to select and optimize the hot methods before execution of the benchmark. This methodology 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].)

Our methodology also includes a compile-ahead component: we precompute method summaries for all methods in the benchmark before performing the free-me compilation on the hot methods. The summaries are stored in a file and retrieved as needed during compilation. This approach 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. One simple solution to this problem is to turn off free whenever classes are dynamically loaded. A more sophisticated solution could adopt the scheme often used for inlining: keep track of dependences, then recompile invalidated code. Note that embedded applications, which will be most likely to benefit from free-me, are unlikely to use features such as dynamic class loading.

The second issue is determining when, in the program lifecycle, to compute method summaries. While the analysis cost is not extreme, for many benchmarks it is still too high to perform in a just-in-time setting. For long running server applications, the one-time cost might be justified. For embedded applications, compiling the entire application ahead-of-time is more likely to be acceptable, since their limited resources preclude sophisticated optimizing compilers.

MMTk is a composable Java memory management toolkit that implements a variety of high performance collectors that reuse shared components [7]. 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 [6] shows these collectors perform well.

We report results on SPECjvm98 [36], pseudojbb, a fixed workload version of SPECjbb2000 [37] and the DaCapo [8] benchmarks. We measure results on a 3.2 GHz Intel Pentium 4 with hyper-threading enabled, an 8KB 4-way set associative L1 data cache, a 12Kμops L1 instruction trace cache, a 512KB unified 8-way set associative L2 on-chip cache, and 1GB of main memory, running Linux 2.6.0.

7. EXPERIMENTAL RESULTS

	alloc	Free		Uncond.		Stack-like	
	MB	MB	%	MB	%	MB	%
SPEC							
compress	105	0	0	0	0	0	0
jess	263	16	6%	16	6%	16	6%
raytrace	91	73	81%	72	80%	72	80%
db	75	45	61%	45	61%	45	61%
javac	184	35	19%	26	14%	26	14%
mtrt	98	73	74%	73	74%	73	74%
jack	276	164	60%	136	49%	113	41%
pseudojbb	209	38	19%	27	13%	12	6%
DaCapo							
antlr	254	96	38%	77	31%	64	25%
bloat	717	357	50%	258	36%	40	6%
fop	70	5	7%	4	5%	2	3%
hsqldb	516	57	11%	39	9%	40	7%
jython	387	83	22%	82	21%	21	6%
pmd	242	7	3%	7	3%	7	3%
ps	522	20	4%	17	3%	13	3%
xalan	5862	0.4	0%	0.4	0%	0.4	0%
Average			28		25		21
Potential							
javac-inl	188	51	27	25	14	25	14
xalan-mod	5862	5682	97	5682	97	5682	97
db-mod	74	65	88	65	88	65	87

Table 1: 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

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.

7.1 Effectiveness of Free-Me Analysis

Table 1 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 28% of all objects and up to 81% in our benchmarks (the *Free* columns). For javac it frees 19% of all objects, 50% for bloat, 38% for antlr, and 22% for jython. The last two columns (*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. We show that change reduces the average effectiveness from 28% to 21%, with jack, pseudojbb, antlr, bloat, and jython showing the most differences.

Comparing the *Free* columns with the *Uncond* columns shows the influence of free acting on some paths and not others. On average, it finds 3% more than if it required frees to correspond with their allocation site. Conditional freeing makes quite a difference to three of the more complex benchmarks: javac, jack, and bloat.

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 19%. If we force the compiler to inline this method, free-me finds 27% (javac-inl).

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 accomodate new elements. In this situation the 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 doesn't coincide with any particular program scope, precluding stack allocation. Furthermore, at all times at least one of the arrays is live, making region allocation extremely inefficient or impossible. These results are not included in any further discussion.

In general, the compiler analysis finds many opportunities to free objects and finds 7% more than is possible with a stack allocation analysis, since our analysis adds conditional frees or factory methods. In addition, free-me is prompt, since it does not have to wait for methods to complete.

It is also worth noting that while many calls to free() contribute to these results, it is often the case that just a few specific calls in each benchmark account for the majority of freed objects. This suggests that expending more analysis effort in these parts of the program could yield even better results.

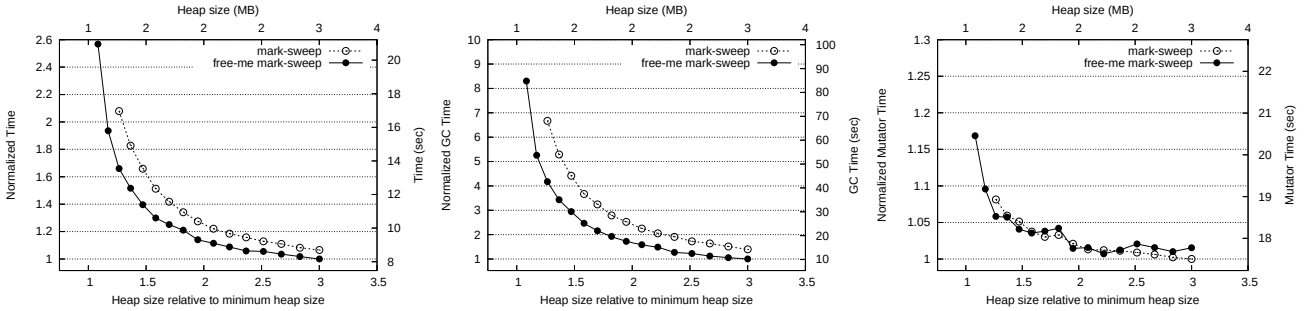
7.2 Free-me in a Mark-Sweep Collector

This sections 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 7, and javac and bloat in Figures 8 and 9, respectively. These figures plot time on the y-axis relative to the best time, and on the x-axis, heap sizes that vary from the smallest in which the collectors execute to three times that minumum.

Figure 7 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 size in which the benchmarks can execute on average. Thus free-me improves performance and reduces the memory requirements in a mark-sweep collector.

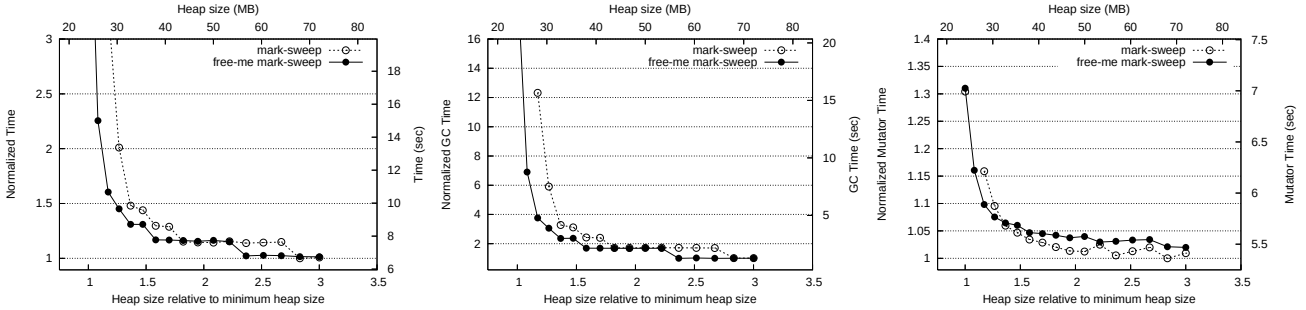
Specifically, free-me improves raytrace, db, mtrt, jack, javac, pseudojbb, antlr, bloat, hsqldb, and jython, as expected from examining the data from Table 1. Free-me attains these improvements for the most part, by reducing garbage collection time, as illustrated in part (b) of each figure. For javac, almost all its improvement is due to reduced collection time. However a few benchmarks also improve their mutator time, part (c) in each figure. We measured the effect of free-me on how often lazy sweeping needs to build free lists during allocation, and found it often substantially reduced this work. For bloat, free eliminates about half of the lazy sweeping, but this improvement does not vary with heap size. Thus, it does not explain the mutator improvements in Figure 9(c). We believe that in small heap sizes each collection disturbs age-order locality, and because free reduces the number of collections, it attains better mutator locality in smaller heaps. (We will verify this hypothesis with l1 and L2 performance counters in the final paper.)

These results demonstrate that free-me improves performance and reduces the memory requirements over a wide variety of bench-



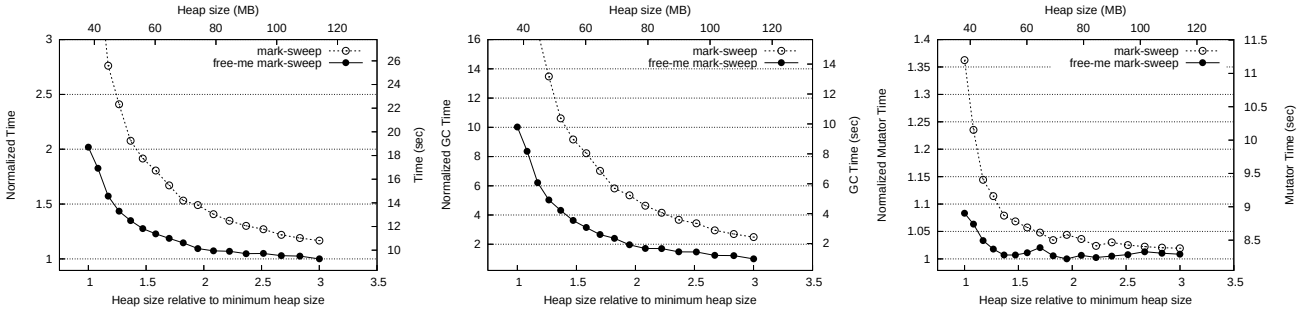
(a) Total Time (b) GC Time (c) Mutator Time

Figure 7: Geometric means over all benchmarks with and without free-me in a mark-sweep collector



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

Figure 8: javac with and without free-me in a mark-sweep collector



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

Figure 9: bloat with and without free-me in a mark-sweep collector

marks 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: when free() is not called, there is no cost.

7.3 Free-Me in a Generational Collector

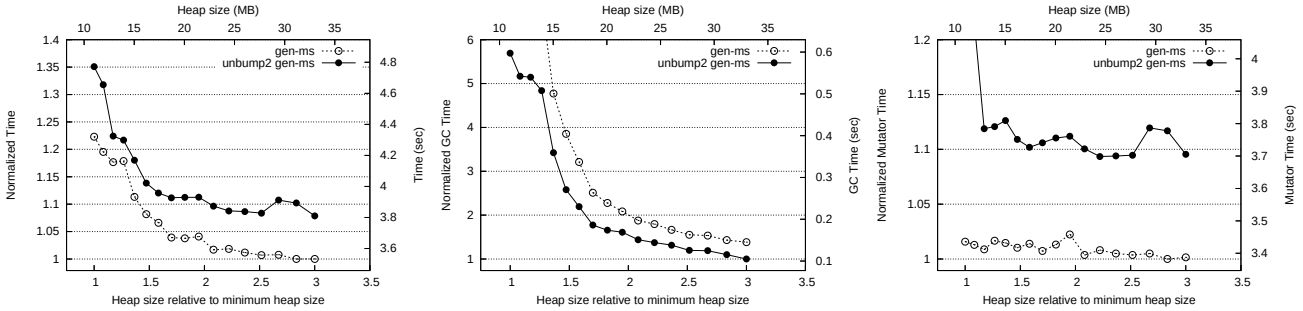
In a generational collector, with a copying nursery and a mark-sweep older space, we find that the nursery reclaims dead objects cheaply and quickly enough that explicit deallocation provides a benefit only for those programs where a large fraction of objects can be explicitly freed. Unfortunately, we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

Figures 10 and 11 shows representative performance graphs for free() implemented as an unbump operation (see Section 5). We show results for one of the two variations of unbump: the one that can free multiple objects by keeping track of frees that occur near the last object. Results for the other variant are similar. For jack we find a reasonable reduction in collection time, offset by an increase in mutator time, which is due to the extra work required to implement the unbump. Even if we discount the mutator overhead, as would be the situation with stack allocation, the gains are modest.

In the case of db, Figure 11, free has the intended effect of significantly reducing the number of nursery collections: it reduces the number from 16 collections to 4 collections in the smallest heap size. Other benchmarks show similar gains. Unfortunately, this has practically no effect on collection time because it does not significantly reduce the number of objects that survive the nursery. Since a copying collector does work proportional to nursery *survivors*, free-me hardly affects collection time at all.

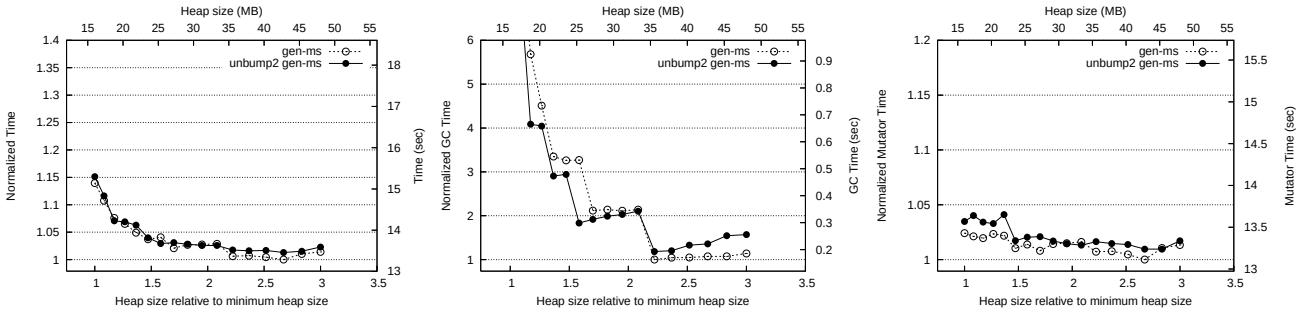
Figure 12 shows the geometric mean of overall time, collection time, and mutator time for free() implemented as a reduction in the size of the copy reserve (see Section 5). This implementation underestimates the sizes of objects: rather than compute object sizes, which can be expensive, it reduces the copy reserve by the minimum objects size; in our case, 24 bytes. This approach effectively eliminates the mutator overhead of free(), but cuts the gains as well.

8. CONCLUSIONS



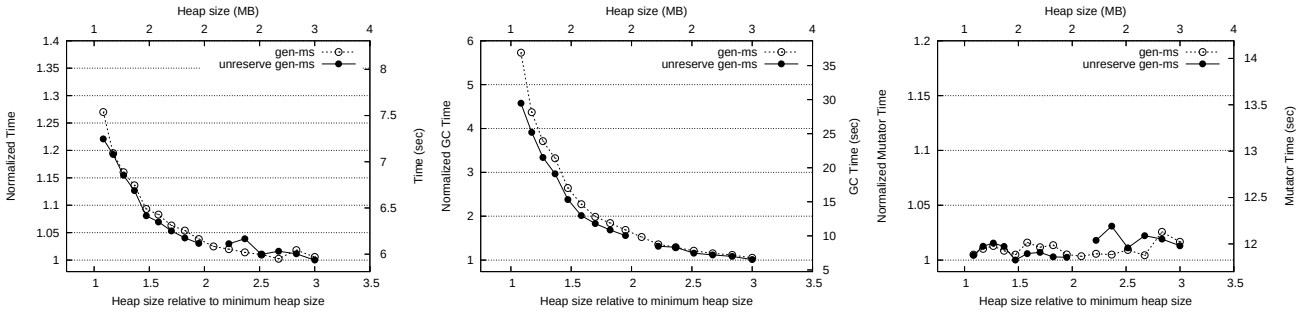
(a) Total Time (b) GC Time (c) Mutator Time

Figure 10: Performance of jack using unbump in a generational collector.



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

Figure 11: Performance of db using unbump in a generational collector.



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

Figure 12: Geometric means over all benchmarks with and without free-me using unreserve in a generational collector

In this paper we present a new static analysis for identifying short-lived objects in Java programs and inserting explicit memory deallocation at the points where the objects die. Our analysis properly identifies a large fraction of short-lived objects for the benchmark programs, which results in rapid reclamation of memory. For mark-sweep collectors operating in a memory-constrained environment, explicitly free objects yields substantial performance improvements from 50% to 200%. However, our experiments show that generational collectors are extremely effective at reclaiming short-lived objects. Therefore, it is unlikely that our technique, or any similar technique that targets short-lived objects can beat the performance of a generational garbage collector.

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

[5] 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.

[6] 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.

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

[8] S. M. Blackburn, K. S. McKinley, J. E. B. Moss, S. Augart, E. D. Berger, P. Cheng, A. Diwan, S. Guyer, M. Hirzel, C. Hoffman, A. Hosking, X. Huang, A. Khan, P. McGachey, D. Stefanović, and B. Wiedermann. The dacapo benchmarks. Technical report, 2004. http://ali-www.cs.umass.edu/DaCapo/Benchmarks.

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

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] L. P. Deutsch and D. G. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.

[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, Department of Computer Science, May 2003. http://www.cs.unm.edu/ darko/papers/objlife.pdf.

[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 International 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] H. Lieberman and C. E. Hewitt. A real time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983.

[32] 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.

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

[34] 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.

[35] R. Shaham, E. Yahav, E. K. Kolodner, and M. Sagiv. Establishing local temporal heap safety properties with application to compile-time memory management. *Science of Computer Programming Journal*, 2005. To appear.

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

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

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

[39] 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.

[40] 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.