

Object Equality Profiling

Darko Marinov
MIT Lab for Computer Science
marinov@lcs.mit.edu

Robert O'Callahan
IBM T. J. Watson Research Center
roca@us.ibm.com

ABSTRACT

We present *Object Equality Profiling* (OEP), a new technique for helping programmers discover optimization opportunities in programs. OEP discovers opportunities for replacing a set of equivalent object instances with a single representative object. Such a set represents an opportunity for automatically or manually applying optimizations such as hash consing, heap compression, lazy allocation, object caching, invariant hoisting, and more. To evaluate OEP, we implemented a tool to help programmers reduce the memory usage of Java programs. Our tool performs a dynamic analysis that records all the objects created during a particular program run. The tool partitions the objects into equivalence classes, and uses collected timing information to determine when elements of an equivalence class could have been safely collapsed into a single representative object without affecting the behavior of that program run. We report the results of applying this tool to benchmarks, including two widely used Web application servers. Many benchmarks exhibit significant amounts of object equivalence, and in most benchmarks our profiler identifies optimization opportunities clustered around a small number of allocation sites. We present a case study of using our profiler to find simple manual optimizations that reduce the average space used by live objects in two SpecJVM benchmarks by 47% and 38% respectively.

Categories and Subject Descriptors

D.1.5 [Programming Techniques]: Object-oriented Programming;
D.2.8 [Software Engineering]: Metrics—*Performance measures*;
D.3.4 [Programming Languages]: Processors—*Memory management, Optimization*; G.2.2 [Discrete Mathematics]: Graph Theory—*Graph Algorithms*

General Terms

Measurement, Performance

Keywords

Object equality, object mergeability, profiling, profile-guided optimization, space savings, Java language

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

OOPSLA'03, October 26–30, 2003, Anaheim, California, USA.
Copyright 2003 ACM 1-58113-712-5/03/0010 ...\$5.00.

1. INTRODUCTION

Object-oriented programs typically create and destroy large numbers of objects. Creating, initializing and destroying an object consumes execution time and also requires space for the object while it is alive. If the object to be created is identical to another object that the program has already created, it might be possible to simply use the existing object and avoid the costs of creating a new object. Even if a new object is created, if it later becomes identical to an existing object, one might be able to save space by replacing all references to the new “redundant” object with references to the existing object and then destroying the redundant object.

Anecdotal evidence suggests that these improvements are frequently possible and useful. For example, consider the white paper “WebSphere Application Server Development Best Practices for Performance and Scalability” [39]. Four of the eighteen “best practices” are instructions to avoid repeated creation of identical objects (in particular, “Use JDBC connection pooling”, “Reuse data-sources for JDBC connections”, “Reuse EJB homes”, and “EJBs and Servlets - same JVM - ‘No local copies’”). The frequent use of pooling and caching in applications in general suggests that avoiding object allocation and reusing existing objects is an important concern. Avoiding object allocation and initialization reduces memory usage, improves memory locality, reduces GC overhead, and reduces the runtime costs of allocating and initializing objects.

Figure 1 shows a situation where OEP helps determine whether an optimization opportunity exists. Two classes implement an `evaluate` method. The original `Eval` might be improved by switching to `CachedEval` that caches all previously created `Value` objects and reuses one of them if it has already been allocated for a given value of `i`. This is an attempt to save space as well as allocation and initialization time. Determining whether this optimization is actually valuable requires the information gathered by OEP. The optimization will only save time if a significant number of identical `Value` objects are created by `Eval`. Furthermore the optimization will only save space if the lifetimes of identical objects overlap significantly, otherwise it may increase space usage.

Such optimization techniques, including hash consing [5,22] and memoization [11, 12, 26, 28], burden the programmer and can impose overhead at run time. We have developed a tool that identifies and quantifies opportunities for eliminating redundant objects. Our tool automatically computes, over all the objects in a program, counts of identical objects and measurements of lifetime overlap. Our tool helps focus developer effort on program data that deserve optimization, just as a regular profiling tool helps detecting which parts of a program’s code deserve optimization. Our tool reveals some significant optimization opportunities in real Java applications and libraries. Furthermore, many of these optimizations require only relatively simple code restructuring, such as loop invari-

ant hoisting or lazy initialization. Hash consing or caching techniques are often not needed. Automatic optimizations that take advantage of the opportunities seem feasible, but are beyond the scope of this paper; currently, the tool provides information that is targeted at programmers.

Our basic approach is to detect when one object could be *merged* into another object. Merging an object o into object o' means replacing all references to o with references to o' . When o can be merged into o' , then o is redundant and it might be possible for an optimization to remove it. If o can be merged into o' early enough in the life cycle of o , then the creation of o can be optimized away. The concept of mergeability is complicated in a language such as Java which allows object mutation, reference equality testing, and synchronization on objects. We also need to identify particular points in time at which objects become mergeable; if the object lifetimes do not overlap, then optimization will not save space. We define a mergeability relation precisely in Section 3.

Our profiling tool performs post-mortem dynamic analysis. We instrument Java bytecode programs to monitor all relevant heap activity during a program run. We then perform post-mortem analysis to determine which objects could have been merged without affecting the execution of the program, and when they could have been merged. Our check for the mergeability of two objects requires, in general, checking two (potentially cyclic) labelled graphs for isomorphism. Thus determining all-pairs mergeability in reasonable space and time is not trivial, because typical heaps contain many cyclic structures and millions of objects. (We consider every object that the program ever created, which can be many more objects than are actually live at one time.) In Section 4 we give an efficient algorithm for mergeability checking, based on graph partitioning [4, 15] with modifications that make it work on more objects than can be stored in main memory.

We applied our profiling tool to the SpecJVM98 benchmarks and two production Web application servers. Section 5 shows that our tool detects a significant amount of mergeability in many benchmarks. To study its utility, we used the tool to extract detailed information for the SpecJVM98 benchmarks with the most mergeability, `db` and `markt`, and used the information to perform some manual optimization of those benchmarks. Section 6 shows how we were led directly to some simple program changes which greatly reduced the space used by objects in those benchmarks.

This paper makes the following contributions:

- We present the idea of Object Equality Profiling (OEP) as an aid to program optimization.
- We make OEP concrete by precisely defining the *mergeability* problem and showing how mergeability results lead to optimization opportunities.
- We describe how to efficiently compute mergeability by extending a graph partitioning algorithm to work well on data sets which do not fit in main memory.
- We evaluate our OEP implementation against a set of benchmarks which include two widely used Web application servers. We show that significant mergeability does exist.
- We demonstrate the utility of OEP by showing how we used our profiler to facilitate optimization of two of our benchmarks, resulting in major space savings.

2. BACKGROUND

Previous work in reusing existing objects has focused on specific optimization techniques. Two major groups of techniques are hash

```
class Eval {
    public Value evaluate(Integer i) {
        return new Value(i);
    }
}

class CachedEval extends Eval {
    private Hashtable cache = new Hashtable();
    public Value evaluate(Integer i) {
        Value v = (Value)cache.get(i);
        if (v == null) {
            v = new Value(i);
            cache.put(i, v);
        }
        return v;
    }
}
```

Figure 1: Example of Potentially Redundant Objects

consing and memoization. Other common optimizations can be viewed as special cases of these techniques.

A hash consing system [5, 22] compares an object o , at some point in its lifetime, against a set of existing objects. If the system finds an identical object o' , it replaces some or all references to o by references to o' . In automatic schemes, this can happen conveniently at allocation time or during garbage collection [8]. Hash consing is frequently applied as a manual optimization; for example, a programmer can manually hash-cons a Java String object `s` by writing `s = s.intern()`. Hash consing focuses on saving space, but it must be used carefully so that the overhead of hashing does not outweigh the benefits of sharing.

Memoization, caching, and pooling [11, 12, 26, 28] are techniques that reduce the need to construct new objects, and thus can save time (but may or may not save space). Suppose that we add memoization to a method m that would normally create and return a new object. The memoized m first checks to see if the result of a previous call with the same parameters is already available in a cache. If so, memoized m returns the previously computed result instead of building another, equivalent result object. A variety of policies are available for managing the cache.

Loop invariant hoisting of object allocation can be viewed as a special case of caching, which exploits knowledge of program semantics to make the cache lookup trivial. Lazy allocation is another example of a technique for avoiding object allocation. Instead of reusing an existing object, lazy allocation uses a special value (e.g., `null` or a flag setting) to indicate that the object is trivial.

Our work does not present a new optimization technique. Instead, we provide a tool to help determine where these known techniques may be useful. Our tool helps in avoiding the well-known problem of “premature optimization” and focusing programmer effort where it is most needed.

We divide the techniques into two groups: those that do not extend object lifetimes (e.g., most hash consing) and those that can extend object lifetimes (e.g., caching). Our profiler can focus on either group by making different assumptions about lifetimes as it checks which identical objects have overlapping lifetimes. If the profiler uses each object’s actual lifetime, it will detect hash consing opportunities but it may not detect caching opportunities. To find caching opportunities, the profiler extends the apparent lifetimes of objects, as if they were being stored in a cache. In this paper we only describe the tool operating in “hash consing” mode with no lifetime extension. Thus we focus on space savings, although some of the optimizations that we describe can save time as well.

3. DEFINING MERGEABILITY

3.1 Overview

Intuitively, two objects are mergeable at some step in a program run if, at that step, all references to one of the objects can be replaced with references to the other object without changing the future behavior of the program for that run. This definition of mergeability is rather general. Our tool for OEP uses a stronger condition that implies the general condition. Informally, it states that two objects are mergeable at a given point in time if:

- the objects are of the same class,
- each pair of corresponding field values in the objects is either a pair of identical values or a pair of references to objects which are themselves mergeable,
- neither object is mutated in the future, and
- neither object is subjected to a future operation that depends on the object's identity (e.g., passing the object to `System.identityHashCode` in Java).

This definition allows computing all pairs of mergeable objects with time complexity less than quadratic in the number of objects. If we chose a stronger definition, e.g., requiring that objects in cycles are mergeable iff they contain identical references, it would allow faster computing mergeability, but it would give more conservative results. If we chose a weaker definition, e.g., non-symmetric mergeability where one of the two objects is mutated in the future but the other object is not read, it would give better results, but it is not clear how an automatic optimization technique could use them. Also, our definition does not use any richer notion of equality such as that defined by the objects' `equals()` methods, because even correctly implemented `equals()` method does not imply that two equal objects will respond identically to all operations.

3.2 Formal definitions

Let O be the set of all objects that a Java program allocates in some run. Each object $o \in O$ has its dynamic class, $\text{class}(o)$, and the corresponding set of fully qualified (instance) field names, $\text{fields}(o)$. We treat indexes of array objects as fields. Let F be the set of all fully qualified field names in a program, and let P be the set of all (primitive) constant values: `null`, `true`, `false`, the set of `int` values, etc. A heap $h : O \times F \mapsto (O \cup P)$ is a partial function that gives the values to the fields; we consider only type-correct heaps that map each object and each of its fields to a value of the appropriate type.

Each Java program operates on a *state* that consists of a heap and additional elements, including a stack (for each thread) with local variables (for each activation record), a set of static fields (for each class) etc. We abstract the additional state into a *root* $r : L \mapsto O \cup P$, i.e., a function from some set of locations L to $O \cup P$. The whole state s is then a pair $\langle h, r \rangle$. We use $s[o.f]$ to denote the value of the field f for the object o in the (heap of) state s .

We next introduce parts of Java relevant for our definitions. We view a Java program as an infinite-state machine that makes transitions from one state to another during a program run. These transitions correspond to the executions of program statements and the evaluations of program expressions. Formally, a program run, R , is an alternating sequence of program states and transitions: $s_0, a_1, s_1, a_2, s_2, \dots$, where s_0 is the starting state with the empty heap. We use a_t^R and s_t^R , to denote the transition a_t and the state s_t , respectively, at the step t of the run R .

We consider the following predicates that specify when a transition a modifies an object or performs an action that depend on the object's identity (taking the identity-based hash code, locking on the object, or comparing the object to another object by identity):

- $\text{write}(a, o)$ holds iff a writes a field of the object o (e.g., with `o.f = v` or `o[i] = v`, as we treat array indexes as fields);
- $\text{idHash}(a, o)$ holds iff a evaluates identity-based hash code of o (e.g., with `System.identityHashCode(o)` or `o.hashCode()` if $\text{class}(o)$ does not override `hashCode` from `Object`);
- $\text{lockOp}(a, o)$ holds iff a performs a locking operation on o (i.e., `monitorenter(o)` or `monitorexit(o)`);
- $\text{idComp}(a, o)$ holds iff a performs an identity-based comparison of object o with another non-null object (i.e., `o == o'`).

We can now define when a run R , after a step t , does not modify an object o and has no actions that depend on the identity of o :

DEFINITION 1. An object o is a candidate for merging at a step t of a run R , in notation $C_t^R(o)$, iff

$$\neg \exists t' > t. \text{write}(a_{t'}^R, o) \vee \text{idHash}(a_{t'}^R, o) \vee \text{lockOp}(a_{t'}^R, o) \vee \text{idComp}(a_{t'}^R, o).$$

It is easy to show that if $C_t^R(o)$ then $C_{t'}^R(o)$ for all $t' \geq t$.

We next define the mergeability that OEP measures. The definition is recursive and to make it well-founded for objects that are in cycles, we take the *greatest fixed point*.

DEFINITION 2. Two objects o and o' are mergeable at a step t of a run R , in notation $M_t^R(o, o')$, iff $o = o'$ or

$$\begin{aligned} &\text{class}(o) = \text{class}(o') \wedge \\ &(\forall f \in \text{fields}(o). M_t^R(s_t^R[o.f], s_t^R[o'.f])) \wedge \\ &C_t^R(o) \wedge C_t^R(o'). \end{aligned}$$

For all objects o and o' , if $M_t^R(o, o')$ then at the step t of the run R all references to o can be replaced with references to o' , without changing the semantics of the run. This is easy to show by induction on the length of R , with a case analysis for transitions.

Mergeability also has the following property: if $M_t^R(o, o')$ then $M_{t'}^R(o, o')$ for all $t' \geq t$. Intuitively, if o and o' are mergeable at t , they have mergeable values of the corresponding fields, and these values do not change for any $t' > t$, so they remain mergeable. We leverage this property to determine the mergeability of all objects at once, in the *final state* of the run. This state contains all the objects that the run ever creates (not only the objects live at the end of the run) and the *last* edges between the objects (not all the edges that the run creates).

We finally define a notion of observable equivalence between objects in a given state. This definition is also recursive and we take the *greatest fixed point*.

DEFINITION 3. Two objects o and o' are observably equivalent in a state s , in notation $E_s(o, o')$, iff $o = o'$ or

$$\begin{aligned} &\text{class}(o) = \text{class}(o') \wedge \\ &(\forall f \in \text{fields}(o). E_s(s[o.f], s[o'.f])). \end{aligned}$$

This relation is reflexive, symmetric and transitive and is therefore a true equivalence relation.

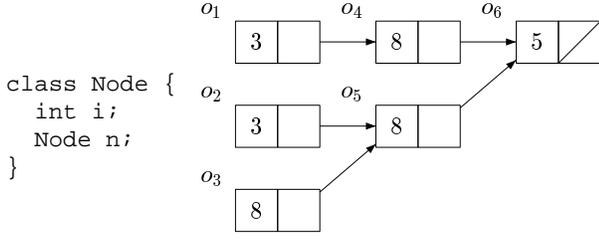


Figure 2: Example heap consisting only of six Node objects

In general, $E_{s^R}(o, o') \wedge C_t^R(o) \wedge C_t^R(o')$ does not imply $M_t^R(o, o')$, because after step t , R may modify an object referenced by o or o' , although it does not directly modify o or o' . However, in the final state, two objects are mergeable iff they are observably equivalent. If two objects are not observably equivalent in the final state, they are not mergeable in any state. Our OEP tool uses graph partitioning to partition objects according to observable equivalence in the final state (Section 4.4).

Figure 2 shows an example final heap that consists of only six objects of the class Node. Each object has an `int` value and a field that is either a reference to a Node object (such as o_1 to o_4) or `null` (in the object o_6). In this example, the observably equivalent objects are only (o_4, o_5) and (o_1, o_2) ; no other pair of distinct objects are observably equivalent.

3.3 Mergeability metrics

We report potential savings due to merging by supposing that, at each time t when a live object o became mergeable with a different live object o' , we actually merged o with o' . We can easily compute the space saved as a result of this merging, but we also need to compute the time interval during which this space saving would have been in effect. Furthermore, merging two objects causes the lifetime of the merged object to become the maximum of the lifetimes of the two objects, and we also take this into account. Thus we must compute when merging can happen and how it affects the lifetimes of objects. Then we can compute the projected heap size at each point in time during the program's (modified) execution.

Conceptually, our profiler reports a limit study of what could be achieved by a merging oracle.

Figure 3 illustrates some mergeability situations for four objects (A , B , C , and D) that are observably equivalent in the final state. The intervals indicate when $C_t^R(o)$ holds for those objects. In this example, the object B gets merged into A , and the merged object has its lifetime extended to the end of the lifetime of B . Similarly D is merged into C . As mentioned, this papers presents OEP only in the “hash consing” mode with no lifetime extensions that measure potential for caching. To measure that potential, the tool would extend the lifetime of the objects (as if they were in a cache) and thus find that D can be merged into A .

3.3.1 Formal definitions

Suppose we have a total ordering $<$ on object identifiers. When merging o with o' , we choose to keep the smaller of o or o' and discard the other. If $o' < o$ we say that o is *merged into* o' . When o is merged into o' , the lifetime of o' may be extended to ensure it lives at least as long as o .

DEFINITION 4. The GC time $G^R(o)$ for an object o is the time at which the object was actually garbage collected (or the time at which the program ended, if the object was never collected).

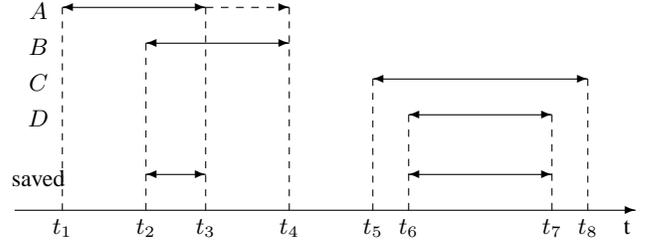


Figure 3: Savings From Merged Objects

DEFINITION 5. The merge time $M^R(o)$ is the time at which o is merged into some other object, or ∞ if o is never merged into another object.

DEFINITION 6. The merge target $T^R(o)$ is the object into which o is merged, or ∞ if o is never merged into another object.

DEFINITION 7. The death time $D^R(o)$ is the time at which o would be discarded when merging is taken into account.

In Figure 3, $G^R(A) = t_3$, $G^R(B) = t_4$, $G^R(C) = t_7$, and $G^R(D) = t_8$; $M^R(B) = t_2$ and $M^R(D) = t_6$, but is ∞ for A and C ; $D^R(A) = t_4$, $D^R(B) = t_2$, $D^R(C) = t_8$, and $D^R(D) = t_6$.

The GC time is measured directly by observing the program run. The rest of the functions are computed as follows.

The following definition says we only merge with objects that have not yet been discarded or themselves merged away:

$$M^R(o) = \min(\{t \mid \exists o'. M_t^R(o, o') \wedge o' < o \wedge D^R(o') \geq t \wedge M^R(o') \geq t\}, \infty)$$

The following definition chooses the minimum suitable o' for o to merge with:

$$T^R(o) = \min\{o' \mid M_t^R(o, o') \wedge o' < o \wedge D^R(o') \geq t \wedge M^R(o') \geq t\}$$

where $t = M^R(o) \wedge t \neq \infty$

$$T^R(o) = \infty \text{ where } M^R(o) = \infty$$

The following definition ensures that o persists at least as long as it did in the actual program run, and also persists as long as all the objects that were merged into it, unless it is merged into some other object first:

$$D^R(o) = \min(M^R(o), \max(\{G^R(o)\} \cup \{D^R(o') \mid T^R(o') = o\}))$$

These recursive functions have a solution which can be computed in a straightforward way. The key idea is to simulate forward through time keeping track of which objects get merged into which other objects. We start by setting $M^R(o)$ and $T^R(o)$ to ∞ and $D^R(o)$ to $G^R(o)$ for all o . We also create a worklist of the pairs (o, o') where $M_t^R(o, o')$ and $o' < o$, sorted by increasing t (the minimum t for which $M_t^R(o, o')$ holds) and o' . We iterate through the worklist; each element can update $M^R(o)$ and $T^R(o)$, but it can be shown that they will only be set once for a given o .

It can be shown that the solution thus obtained is unique.

Using the “death times” instead of each object’s original garbage collection time, it is easy to predict the heap size with optimal merging for any point in time. For example, in Figure 3, we can see that one object is saved between times t_2 and t_3 , and one object is saved between times t_6 and t_7 .

4. MEASURING MERGEABILITY

This section describes our OEP tool. It first instruments a Java bytecode program to record the information about all objects created during a program run. It then runs the program and builds a trace file with the relevant information about the objects. It next partitions the objects into equivalence classes, using an algorithm for graph partitioning. It finally uses collected timing information to determine when objects from an equivalence class could have been safely collapsed into a single representative object.

4.1 Instrumentation

Our OEP tool uses the IBM Shrike framework to instrument Java bytecode programs. We monitor program execution and record the following information about each object:

- the object’s class
- the array length, if the object is an array
- the values of all the fields (or array elements, if the object is an array)
- the time of the last read
- the time of the last write
- the time of the last identity-based operation (identity hash code, lock acquisition or release, the object’s reference being tested for equality against some other reference)
- an upper bound on the time at which the object was garbage collected
- whether we observed the object’s creation

This metadata is associated with each object by a hash table whose keys are weak references [38] to the objects, and whose values are the metadata. Our instrumentation tool and its run time support are written in pure Java and are independent of the underlying Java virtual machine. (However, many JVMs have problems when certain library classes are instrumented, such as `java.lang.Object`, so we sometimes have to use VM-specific workarounds for these problems.)

Using real time for our timestamps would be too expensive. Instead we maintain an internal “event clock” counter which we increment each time the program reads or writes to an object.

Because our approach is at the pure Java level, we have no way to detect certain kinds of accesses to objects. In particular we do not detect accesses performed via reflection or by native code using the Java Native Interface to manipulate Java objects. We do instrument certain native methods commonly used to manipulate objects, such as `System.arraycopy`, and simulate their effects on our metadata. We have performed experiments to compare the writes we observe with the actual contents of objects, and we have found that the number of unobserved writes that actually change values is negligible. For programs that heavily use reflection, we could instrument calls to the methods for reflection.

We insert instrumentation at the following program points:

- allocation sites for objects and arrays, including the `ldc` instruction which can load `String` objects from a constant pool; on IBM’s JVM, we also instrument calls to `ExtendedSystem.newArray`
- field writes
- array element writes

- calls to `System.arraycopy`, and on IBM’s JVM `ExtendedSystem.resizeArray`
- field reads
- array element reads
- array length reads
- identity-dependent operations:
 - lock acquisition and release
 - reference comparisons
 - calls to `System.identityHashCode`
 - calls to the dynamic `hashCode` method (we check the receiver class to see if the `Object.hashCode` implementation is called, which depends on the object’s identity)

All our instrumentation ensures thread safety using a combination of locks and thread-local storage.

Our tool instruments the program and the libraries it uses offline, before it is run (and not during class loading as some other tools). Online instrumentation using a custom class loader is not possible, because we want to track objects of all classes, and many library classes (e.g., `String`) must be loaded by the system class loader.

An important issue in instrumentation is to avoid stability problems with instrumenting the libraries. We do not activate instrumentation until the program starts its `main` method. (VMs bring up classes such as `String` before the VM is ready to run arbitrary code, and so our instrumentation in `String` must remain quiescent until the VM is ready.) This means there are many built-in objects which have already been created and manipulated before our instrumentation starts. These objects are marked “untracked” and they are always treated as unmergeable. We do this for all objects whose creation site we did not instrument, such as objects created by native code or using reflection. The numbers of untracked objects are given in Section 5. Untracked objects are included in all our heap size calculations (except for untracked objects which were never accessed by any Java library code or application code; we have no way of even detecting whether any such objects exist).

Note that our instrumentation uses the same instrumented library classes for its own purposes, e.g., we use `java.io` classes to write the trace file. This could cause infinite recursion: while processing some program event, the instrumentation executes another instrumented program point in a library, and reenters itself. We avoid this scenario using a per-thread re-entrancy protection flag. The manipulation of this flag is carefully implemented to avoid ever entering code that could be instrumented.

4.2 Measuring object reachability

Section 5 reports results based on knowing when objects became unreachable. Computing the precise time when an object becomes unreachable is difficult [23], especially in the context of a full-fledged Java virtual machine, where it is difficult to determine all roots. We compute upper and lower bounds on the time at which an object became unreachable, using two different techniques.

Computing an upper bound on the time an object became unreachable is simple. We already maintain a weak reference to each application object, as described above. When an object is about to be garbage collected, the collector places its weak reference on a queue which we poll every time our metadata table is accessed. When a weak reference is found on the queue, we remove its entry from the metadata table, and note the current time as an upper

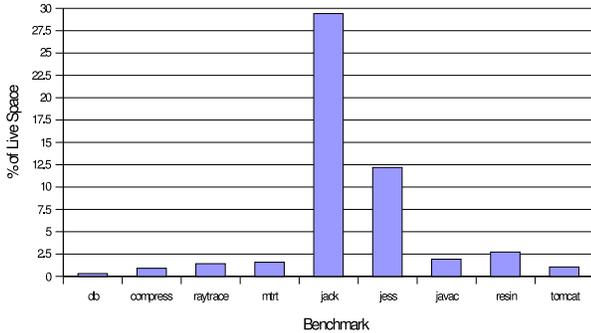


Figure 4: Tightness of bounds on “last reachable” times

bound on the object’s time of last reachability. At this point we write the object’s metadata to a trace file.

We assume that the just-collected object was collected by the very latest garbage collection, and therefore must have survived the second-to-last garbage collection. (We did experiments to verify that the IBM JDK places dead objects on the “dead weak reference queue” as soon as the garbage collection that declared the object dead has finished.) Thus the start time of the second-to-last garbage collection is a lower bound on the last reachability time of the dead object. We may observe an access to the object after the lower bound on reachability computed by this method, in which case we update the lower bound to the time of last observed access to the object. We shrink the gap between the upper and lower bounds by injecting a thread which forces a full garbage collection every 100 milliseconds. This thread also records the time it initiated each collection.

Figure 4 shows the difference between the bounds on last reachability time for our benchmarks. See Section 5.1 for information about the benchmarks. This graph reports the average over the program run of the reachable data size according to the upper bounds, divided by the average over the program run of the reachable data size according to the lower bounds (minus one). The results show that with the exception of `jack` and `jess`, the error is within a few percent.

4.3 Measuring object size

We cannot measure object size directly with our Java-based instrumentation — it depends on the implementation of the virtual machine. We assume that every object has an 8-byte header. We assume that an array object uses 4 bytes to store the length, that array elements of non-boolean primitive type use their natural size as defined by Java, that boolean array elements use 1 byte each, and that reference array elements use 4 bytes each. We assume that a non-array object uses 4 bytes for each field except for fields of type `double` and `long`, which use 8 bytes.

4.4 Partitioning objects

Our OEP tool uses the information from the trace to partition the objects into equivalence classes based on mergeability. As pointed out in Section 3, it is sufficient to consider only observable equivalence of objects in the final state of the run. The trace contains the final state for all objects that the program ever created (except for the “untracked” objects).

Recall that two objects o and o' are *observably equivalent* in final state s (notation $E_s(o, o')$) iff $o = o'$ or

$$\text{class}(o) = \text{class}(o') \wedge (\forall f \in \text{fields}(o). E_s(s[o.f], s[o'.f]))$$

This is an instance of the *graph partitioning* problem [4]. The worst-case time complexity of some simple partitioning algorithms is $O(n^2)$, where n is the number of nodes in the graph, i.e., the number of objects in the final state. Aho, Hopcroft, and Ullman present an efficient partitioning algorithm (“AHU”), based on the Hopcroft’s algorithm for minimizing a deterministic finite automaton [24]. The AHU algorithm is typically said to run in time $O(n \log n)$. For example, this algorithm has been used to detect equivalent computations [6].

Two issues complicate our implementation. First, the final heaps (which include every object the program run ever created) comprise millions of objects for our programs (see Table 1). OEP thus processes graphs many orders of magnitude larger than graphs processed in the context of compilers; the data structures required for OEP do not always fit in RAM. Second, the AHU algorithm runs in time $O(n \log n)$, but only if the number of distinct edge labels is a constant. More precisely, the algorithm runs in time $O(l \cdot n \log n)$, where l is the number of distinct labels. In our graphs, each field name and each value of an array index constitute a distinct edge label, and so l grows with the static size of the program and also with the dynamic size of the largest array, which can make l very large.

The latter problem is easier to address. Cardon and Crochemore [15] show how to modify the AHU algorithm to run in time that does not depend on the number of distinct labels. The AHU algorithm maintains a worklist of partitions that need to be processed. The key change is to store in the worklist, along with each partition, the set of edges incoming to that partition that need to be processed. The modified algorithm runs in time $O(e \log n)$, where e is the actual number of edges in the graph; typically $e \ll l \cdot n$.

We use additional techniques to scale our analysis to millions of objects. Our variant of the AHU algorithm is seeded with an “initial partition” that fully takes account of primitive field values. Thus the expensive graph algorithm does not need to keep in memory field values of primitive type or fields with null references. We also break the heap graph into strongly connected components (SCCs) and process one SCC at a time. Between algorithm phases, data is stored in temporary files structured so that the reads of the next phase are mostly sequential.

4.4.1 Initial partition

We seed our graph partitioning algorithm with an “initial partition” that takes into account each object’s class and the values of its primitive (i.e., non-reference) fields. Two objects are placed in the same initial partition if and only if

- they have the same class
- and, if they are arrays, they have the same length
- and, corresponding fields (or array elements) of non-reference type have the same value
- and, corresponding fields (or array elements) of reference type are both null or both non-null

Having formed the initial partition, the class, primitive fields, and null reference fields of the object are no longer needed.

The program instrumentation produces a trace with a record for each object, in the order in which the objects were garbage collected. To build the initial partition we first scan the trace, computing a hash code for each object based on the above equality relation, and counting how many objects hash to a given value. Most of the hash values that occur are hashed to by just one object; each such

Example	Code (bytes)	Description	Allocated objects		References
			total	untracked	
Java library	15,052,844	Standard Java library (IBM JDK 1.3.1)			
db	168,017	small database management program	3213922	3846	8497930
compress	172,801	Java port of LZW (de)compression	10282	4127	10868
raytrace	222,429	ray-tracing program	6375211	3763	2454931
mtrt	223,572	multi-threaded ray-tracing program	6639901	2944	2747718
jack	286,262	Java parser generator	5690095	17348	5754814
jess	485,805	expert shell system	7927585	3760	28597623
javac	2,096,408	JDK 1.0.2 Java compiler	5868126	273511	8722822
resin	9,727,578	Web application server (v2.1.2) from Caucho Technology	2449706	339960	5214929
tomcat	11,419,565	Web application server (v4.0.4) from Apache Foundation	1980341	210747	4188529

Table 1: Benchmark programs and allocation statistics

object can only be observationally equivalent to itself. We place each such object into its own final partition, and ignore them for the rest of the algorithm. Then we scan the trace again, selecting the objects whose hashes correspond to more than one object, and grouping these into initial partitions using the equality test above.

Next, we project references between objects onto references between initial partitions; i.e., there is an edge between partition p_1 and p_2 if some object in p_1 has a reference to an object in p_2 . We topologically sort the initial partitions according to these edges. We scan the trace again and copy each object (minus the fields of primitive type or reference fields referring to a “singleton” object or null) to a new file, ordering the objects according to the topological order of the SCC containing the object’s initial partition. This step requires two passes over the data: a pass splitting the objects into n temporary files, each individually sorted, followed by a merge pass reading all n files and producing the final sorted file; n is chosen to ensure that the objects in each temporary file can be sorted in memory.

4.4.2 Graph partitioning

Now we read the sorted file and apply the graph partitioning algorithm independently to each strongly connected component of initial partitions. The invariant is that after processing each component, we know the final partitions of all the objects involved.

For each component, we build a new refined initial partition of the objects. The topological ordering guarantees that a reference from object o in the component points to either an object in the same component or an object in a component that has already been processed. In the latter case, we know the target object’s final partition; such object references are treated as “primitive” and used to build the new initial partition. More specifically, two objects in the same new initial partition if and only if

- they were in the same original initial partition
- and, if corresponding fields (or array elements) of reference type refer to objects o_1 and o_2 , then either o_1 and o_2 have been assigned to the same final partition, or o_1 and o_2 are both in the current component

Thus, after building the new initial partition, the only references left to be treated by the graph partitioning algorithm are references between objects in the current component. At this point we run the $O(e \cdot \log n)$ graph partitioning algorithm. It is important to note that at this point, the number of nodes in the graph is only a tiny fraction of the total number of objects in the final heap.

4.5 Computing mergeability metrics

The above algorithm partitions objects according to observable equivalence in the final state. The remaining problem is to compute the merge and death times as described in Section 3.3. Optimizations for efficiency make the full details of our implementation too complex for presentation here.

Our implementation is based on the algorithm described in Section 3.3. That algorithm requires a list of (o, o') pairs sorted by the minimum t for which $M_t^R(o, o')$ holds. As previously discussed, $M_t^R(o, o')$ can only hold for some t when o and o' are observably equivalent in the final state. Thus the partitions computed by the previous stage constrain the set of (o, o') pairs we need to consider for the list. For each object o we precompute the minimum t for which $C_t^R(o)$ holds; these are the times at which $M_t^R(o_1, o_2)$ can hold for the first time, where $o = o_1$ or $o = o_2$, and also when o_1 or o_2 directly or indirectly reference o . Our algorithm steps through the list of all such t s in increasing order, checking $M_t^R(o_1, o_2)$ for all not-yet-merged objects o_1 and o_2 where o_1 and o_2 are in the same partition, and $o = o_1$ or $o = o_2$ or o_1 or o_2 directly or indirectly reference o . The check for each (o_1, o_2) pair mostly uses brute force heap traversal.

5. MERGEABILITY RESULTS

5.1 Benchmarks

We applied the tool to the programs shown in Table 1. The first seven programs are from the SpecJVM98 benchmark suite [36]. (We omitted the `mpegaudio` benchmark because it caused problems with our instrumentation tool.) The last two programs are Web application servers. Every program was used in conjunction with the Java library. The sizes are the sum of the sizes of all class files, often grouped in JARs, that are part of the benchmark. The sizes of the Web servers are somewhat misleading because our tests only exercise a fraction of the server functionality.

To benchmark `resin` and `tomcat`, we configured them with their respective default Web sites and Web application examples, then used the `wget` tool to crawl the default site and retrieve a list of accessible URLs. Then we modified the Web servers to add a “harness” thread which scans through the list 10 times, loading each URL in turn. Each server’s persistent caches were cleared at the end of each benchmark run. All the other benchmarks come with their own input data sets. We used input size ‘100’ for all the SpecJVM98 benchmarks.

All tests were performed using the IBM JDK 1.3.1 (service release 3) on a 2GHz Pentium 4 machine with 1.5GB of memory, running IBM’s variant of Red Hat Linux 7.1.

Example	Normal		Profiling		Analysis	
	Time	Space	Time	Space	Time	Space
db	16.5s	28M	2274s	1122M	530s	710M
compress	14.5s	25M	1985s	273M	117s	159M
raytrace	4.8s	22M	1686s	1093M	183s	260M
mtrt	5.0s	26M	1441s	1123M	215s	234M
jack	9.2s	17M	403s	622M	729s	460M
jess	9.0s	18M	1298s	1077M	234s	219M
javac	17.6s	44M	1327s	1140M	729s	649M
resin	75.3s	47M	867s	1032M	377s	98M
tomcat	24.2s	31M	281s	562M	263s	83M

Table 2: Performance of the OEP tool

Example	Average			Peak		
	Base	Merge	Save	Base	Merge	Save
db	7.60M	4.38M	0.42	9.10M	5.90M	0.35
compress	4.83M	4.77M	0.01	6.79M	6.79M	0.00
raytrace	3.38M	1.91M	0.43	3.70M	2.09M	0.44
mtrt	5.44M	2.25M	0.59	6.87M	2.65M	0.61
jack	0.44M	0.25M	0.42	0.74M	0.41M	0.45
jess	0.91M	0.81M	0.10	1.23M	1.08M	0.12
javac	4.77M	4.35M	0.09	7.11M	6.62M	0.07
resin	6.23M	3.03M	0.51	8.29M	4.60M	0.45
tomcat	2.18M	1.63M	0.25	2.64M	2.13M	0.19

Table 3: Mergeability Results

5.2 Profiling performance

Table 2 shows the performance of our profiler on these benchmarks. The memory numbers are the maximum of SIZE value reported by `top` (“the size of the task’s code plus data plus stack space”) during one run of the profiler and post-mortem analysis. To a large extent the memory numbers are at the mercy of the heap growth heuristics of the IBM Java virtual machine.

This data shows high variation across benchmarks. Profile gathering is very memory-intensive, because we have a large metadata object associated with every normal object. Profiling is also very slow. Interestingly, the smallest overheads seem to apply to the most “real” applications. These applications create smaller numbers of objects than the SpecJVM98 benchmarks even though they run much longer. (Small numbers of objects improve the performance of profiling and post-mortem analysis.) One possible explanation is that the virtual machine optimizes the SpecJVM98 applications particularly well in their unmodified configurations, and these optimizations are inhibited by our profiling instrumentation, but the VM is not optimizing `resin` or `tomcat` as well in the normal case so the slowdown is less severe.

5.3 Mergeability

We measured the ideal space savings obtainable by an oracle merging algorithm as described in Section 3.3. Table 3 and Figure 5 show the space used by live objects, in the original program run and with merging enabled, averaged over the lifetime of the program. We computed the merging benefits using the algorithm outlined in Sections 3.3 and 4.5. Instead of using the actual garbage collection times, for both cases we idealized the garbage collector by using our lower bound on the “last reachable” time for an object as the time of garbage collection. Thus, the benefits shown in the graph are independent of the garbage collection strategy (provided a tracing collector is used) and would apply even with a perfect tracing collector. With a real garbage collector, the benefits would be even greater.

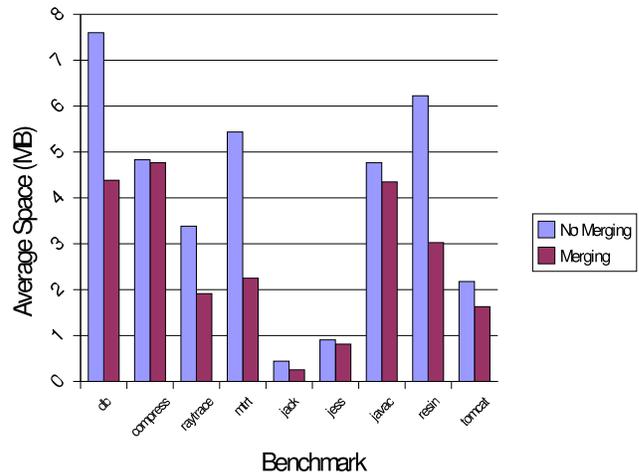


Figure 5: Object Mergeability

Table 3 also shows the peak space used by live objects in the original program run and with merging enabled.

The numbers show that mergeability varies across applications, but in many applications there are significant numbers of objects with overlapping lifetimes which are identical. In particular, in 5 of our 9 applications at least 40% of the average space would have been saved by oracular merging. The results for peak are very similar to the results for average.

The space usages recorded here are smaller than the actual memory used by the applications. These numbers record only the actual data used by the application and do not include virtual machine overheads such as space for holding classes, space for generated code, or heap management overhead. Our results focus on the space usage that the application programmer can control.

5.4 Mergeability by class

We focus on two applications with high mergeability and easy source code access: `db` and `mtrt`. Figures 6 and 7 show the classes of objects in these applications that account for 95% of the savings from merging. (Note that `C[]` denotes “array of `char`”.) For each class, we show the average space used by objects of that class over the program run, and the average space that would be saved by oracular merging.

These figures show that a small number of classes are responsible for the vast majority of savings. They also show that (unsurprisingly) the most mergeability savings are usually (but not always) found where the most memory is used.

To ensure the methodological soundness of the study in Section 6, the per-class and per-site data in these figures were actually computed by running `db` and `mtrt` on a small “training data set”, not the full data set used to produce the benchmarks in the previous subsections. For `db` the training dataset is configured by running the benchmark with “speed=10”, which selects a different, (about 75%) smaller database file than the default configuration. For `mtrt`, the SpecJVM98 “speed=10” configuration just renders the same scene data into a smaller number of pixels. This does not significantly change the input data or the heap configuration. Furthermore, there appear to be no other valid input files for `mtrt/raytrace` on the Internet. Therefore we made our own “training data set” by removing the last 90% of the polygons in the scene file. In both benchmarks, the results were very similar to the results for the full data sets.

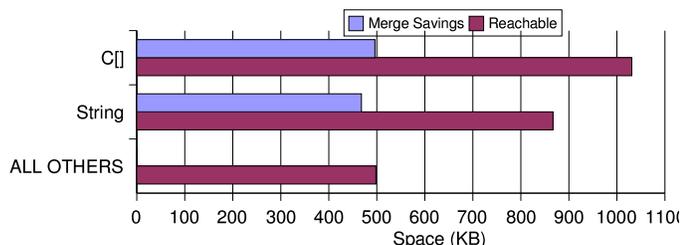


Figure 6: Savings By Class: db

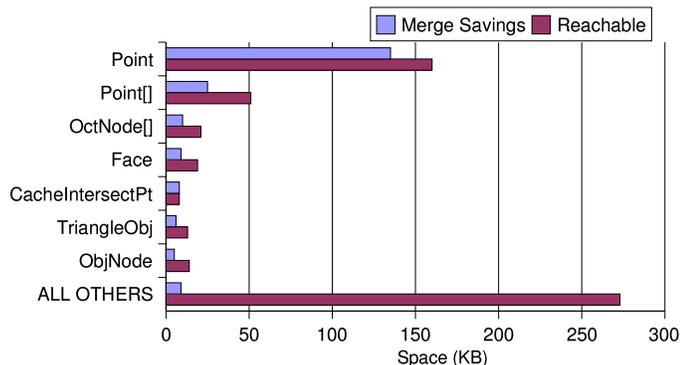


Figure 7: Savings By Class: mtrt

5.5 Mergeability by site

Figures 8 and 9 break down the classes further, distinguishing between objects allocated at different sites. We show the top 10 sites. The figures show that a small number of allocation sites are responsible for the majority of savings in db. However, in mtrt the situation is different; there are many allocation sites for Point that are all equally mergeable.

The data shows that using a profiler to find opportunities for mergeability optimization is feasible, because the opportunities are concentrated in a few classes. The data also shows that if mergeability is present, it may suffice to focus on the classes responsible for the most space consumption to find the savings; this holds true for all our benchmarks which have significant mergeability. Even if it is true across all types of applications, OEP is still useful for identifying the particular applications where mergeability-based optimizations are applicable, and in each application it tells us how many classes are worth looking at, and which allocation sites for each class should be examined.

6. CASE STUDY

In this section we present a case study of using OEP results to help manually optimize applications. We focus on mtrt and db. As mentioned above, the results used in this section were obtained by running OEP on a “training input set”. The resulting improvements shown below were measured against a separate input set.

6.1 Optimizing db

Figure 6 showed that only two classes matter for mergeability. We immediately suspected that the mergeable char arrays were simply part of the mergeable Strings. Figure 8 told us that the relevant allocation site for the mergeable Strings was site 43655. Profiler output tables told us that site 43655 is line 191 in the file Database.java, in the method Database.readDb:

```
entry.items.addElement{
    new String(buffer, 0, s, e-s);
```

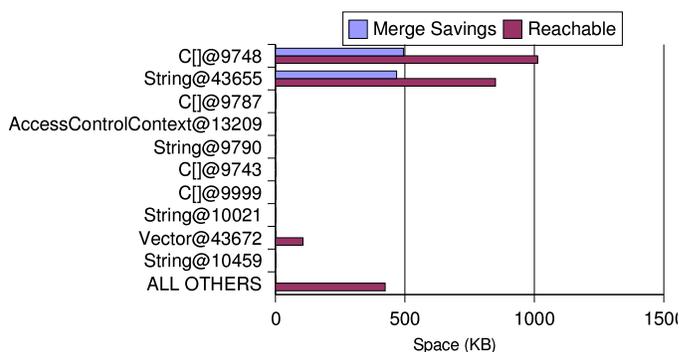


Figure 8: Savings By Site: db

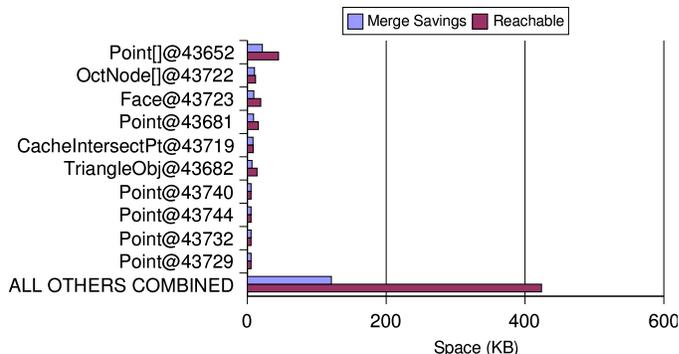


Figure 9: Savings By Site: mtrt

Java’s `String.intern()` method let us convert the string to a canonical `String` object, which can then be shared amongst all sites that get this character sequence:

```
entry.items.addElement(
    (new String(buffer, 0, s, e-s)).intern());
```

Java strings are immutable, so the only way this could affect program semantics is if db did something tricky using identity-based operations, such as locking the strings or testing references to these strings for equality. Even with no prior knowledge of the program, we can tell by inspection (or testing) that this is not so, but a static analysis would be very helpful here.

6.2 Optimizing mtrt

Figure 7 told us that we should focus on the Point class, of which most instances are mergeable with previously existing instances. This class was a little harder to optimize than the strings in db, because its interface allows mutation (although we knew from our profiles that “most of the time” there is no mutation, because otherwise there would be little mergeability).

Our approach here was to let the compiler help. We declared a new class, `ImmutablePoint`, which has only the immutable methods of `Point`. `ImmutablePoint` is not a subclass of `Point`, nor is `Point` a subclass of `ImmutablePoint` (either would be incorrect [25]). `ImmutablePoint` has a private constructor; instead of `new ImmutablePoint`, the programmer calls a factory [19] method `ImmutablePoint.make`. This factory method looks up its parameters in a cache of 2000 recently allocated `ImmutablePoints`. Our implementation of `ImmutablePoint` is 67 lines of code.

Then we looked at our profile data to see which call sites to new Point always get objects which are not mutated after construction. The call site with the most mergeability was

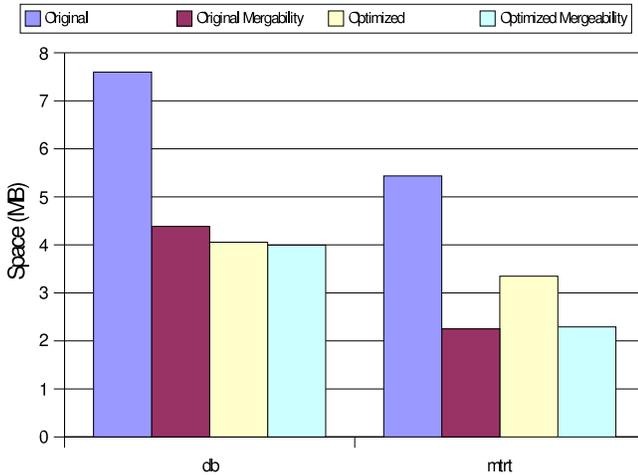


Figure 10: Average Space Before and After Optimization

in `Scene.ReadPoly`, which fills in a vertex array to be used to construct a 3D object. We replaced `new Point` with `ImmutablePoint.make` and then followed the type errors. Whenever the program expected `Point` and had `ImmutablePoint`, we changed it to expect `ImmutablePoint`. After making 10 lines of such changes, we found that the method `Vector.Sub` took `Points` as parameters. Instead of changing them, we added an overloaded `Vector.Sub` which can take `ImmutablePoints` (another 7 lines of changes). The program is then type correct, and therefore (modulo the identity operations mentioned above), we have confidence that we have not changed the behavior.

In the next step we looked at the next most mergeable allocation site for `Point`. It was actually one of a set of more than 20 calls to new `Point` from `OctNode.createChildren`. We replaced them all with `ImmutablePoint.make` and chased the type errors, eventually changing another 37 lines of code. That accounted for almost all the mergeable sites constructing `Points`.

6.3 Results

Figure 10 shows the results of our optimizations. We ran the optimized programs and the original programs on the full input set. We show the average object space usage for both versions of each program. We also show what the average object space usage would be if oracle merging was applied to each version of each program.

The oracle merging results show that in `db` we have captured practically all the mergeability with our optimization. In fact, we have saved more space than with oracular merging. This happened because `String.equals` is optimized to perform a reference equality test on its parameters before comparing their contents. This reference equality test is an identity operation that delays mergeability for objects involved in the test. (It is a “fast path”; the code gives exactly the same behavior if the test was skipped, thus our optimization cannot have affected its behavior.) Thus our call to `intern()` merges the string before the oracle is able to merge it.

In `mtrt` we have captured much of the mergeability, but not all of it. There are some classes (mostly arrays) which are still significantly mergeable.

Overall for these two benchmarks we have reduced average space utilization by objects by 47% and 38% respectively. Including all the overhead of the virtual machine, we reduced peak memory usage as recorded by `top`’s `SIZE` from 32600K to 28952K for

`mtrt`, and from 30164K to 28344K for `db`. We even improved the run time for `mtrt` from 5.4s to 5.0s and the run time for `db` from 19.1s to 18.8s, although these programs run for too short a time for us to claim significant speedup.

These optimizations were designed, implemented and measured in just a few hours. The total changes amounted to one line of code in `db` and 121 lines of code in `mtrt`. In a more realistic environment we might have spent more time retesting the programs after the changes, but we would also have benefited from prior knowledge of the programs.

6.4 Discussion

In `db` and `mtrt`, as well as in all the other examples we looked at, it was easy to transform the code to realize the mergeability opportunities. But, in theory, OEP could report “false positives” where merging cannot be achieved with simple code transformations. For example, if objects became mergeable after the program had already created many references to the objects, it would be hard to update the references to all point to a new merged object. In such cases, our tool helps programmers to evaluate whether it is worthwhile to perform more elaborate code transformations.

In `db` and `mtrt`, the transformations did not involve cyclic structures. In fact, the vast majority of mergeability opportunities found across all our benchmarks do not depend on cyclic structures — the results would be very similar if OEP considered all distinct cyclic structures as not mergeable. Wherever this fact holds true, a much simpler algorithm can be used to determine observable equivalence and mergeability: each object in a cycle is assigned a singleton final partition during initial partitioning of the heap (Section 4.4); the initial partitions form a tree, and a simple bottom-up traversal suffices to find all final partitions. However, for research purposes it is useful to support merging of cyclic structures, to verify whether other kinds of applications have mergeable cyclic data.

7. RELATED WORK

Object Equality Profiling (OEP) discovers opportunities for reusing existing objects in runs of Java programs in order to avoid redundant computation and/or reduce space requirements. To our knowledge, OEP is the first analysis that considers reusing whole objects for an object-oriented language, such as Java. There is an active body of research on reducing space requirements for Java, but the focus is typically on reducing per-object space, reducing the overhead of garbage collection (GC), or reducing heap requirements using stack/static allocation. There has been an extensive work on reusing whole objects, but for functional languages, not for Java. Almost all this work is focused on automatically performing simple transformations; we are not aware of other tools that provide the information for programmers to identify and attempt more difficult transformations. Note that some profitable transformations may not even be strictly semantics-preserving in all situations and will therefore be very difficult to apply automatically.

7.1 Space optimizations for Java

In an object-oriented language, each object is represented in memory with some number of header fields and some number of instance fields. The header fields in Java contain information about the object’s type/class, default hash code, locking status, and GC status. These fields in a non-optimized implementation require two or three memory words. Bacon et al. [9] present a set of techniques that reduce the headers to only one word per object. Their optimizations compress the representation of all header fields.

Several projects offer compressed representation of some header fields for Java objects. Shuf et al. [33] use profiling to discover a

set of “prolific” types that have a large number of instantiated objects. The header fields are then optimized for these prolific types. Thin locking [10], meta locking [2], and lightweight monitors [41] are techniques for improving performance of locking operations on Java objects. Although mainly designed to reduce time for these operations, these techniques also reduce the space required for locking status in object’s header.

Other projects optimize representation of instance fields. Sweeney and Tip [35] show that a significant number of instance fields are never used in many C++ programs, because not every program uses library classes in their full generality. These instance fields can be removed at compile-time and do not take any space at run-time. Further improvements in the space requirements of C++ objects can be obtained through class hierarchy specialization [37]. Bit-width analyses [14, 30, 34] determine the range of values that each variable in a program can take. These results can then be used to compress and pack the instance fields.

Ananian and Rinard [7] combine several techniques for optimizing header and instance fields: compressing type representations, eliminating or reducing the size of instance fields, and field “externalization” (removing mostly-constant fields from their classes and storing them in a hash table). Their results show up to 40% reduction in the maximum heap size. Our OEP tool can be used in addition to all these techniques to achieve even further reductions by manually eliminating whole objects.

Dieckman and Hoelzle [18] present a detailed study of the memory usage of the SpecJVM98 benchmarks. Their study quantifies potential for several space reductions, but it does not offer insights into object mergeability. Shaham et al. [31] present a study that focuses on measuring “drag” time, i.e., the time difference between the last use of an object and its collection time, in several Java applications. Røjemo and Runciman [29] present a similar study for Haskell, a lazy functional language. In a later work [32], Shaham et al. use their results to manually transform Java applications to invoke GC in a more timely manner and thus save space.

Agesen et al. [3] describe how to improve GC for Java using type-precise root set and liveness analysis. They also give an extensive list of references for work on reducing and measuring GC overhead. Chen et al. [16] use compression and lazy allocation (combined with splitting objects into subobjects) to reduce heap requirement for Java applications. They present a family of GC strategies; the best strategy reduces heap requirement by more than 20% on average for a set of benchmarks targeted for handheld devices.

Several escape analyses [13, 17, 20, 40] determine objects whose lifetime is included in the lifetime of the stack frame of the method (or one of its callers) that allocates the objects. These objects can then be allocated on the stack instead on the heap. Gheorghioiu et al. [21] present an analysis that determines allocation sites such that of all objects allocated at each such site, there is at most one live object at any point in the execution. All objects from such a site can then reuse statically preallocated memory space instead of using heap. Stack and static (pre)allocation for Java aim at reducing GC overhead and thus saving time. These optimizations reduce space requirement for heap, but increase for stack or static data. In some situations, the optimizations reduce the total space requirement, but in others, they increase it.

7.2 Reusing existing objects

Hash consing [5, 22] and memoization [11, 12, 26] are major techniques for reusing existing objects. They have been widely used in functional languages, such as LISP and ML. An extensive list of references is given by Acar et al. [1] who present a memoization

system that gives the programmer control over comparisons and caching. They argue that fully automatically applying memoization and hash consing can have negative effects [8, 27, 28]. OEP can help the programmer decide when to apply these techniques.

Appel and Goncalves did a study [8] on automatic hash-consing during GC for SML/NJ. They modified a generational GC to perform hash-consing while copying objects from the young generation to the old generation. They evaluated their implementation on a set of small and medium sized programs. Most programs had only a small reduction on their data size. Moreover, some programs were slowed down by up to 20%, and only one program was sped up by 10%.

There are a few reasons that could account for the difference between their results and our own. First, ML programs mostly manipulate immutable objects, and thus ML programmers may explicitly use sharing much more than Java programmers. In general, the languages and programming styles are very different. Second, the modern Java programs that we analyze are much larger and more complex than the ML programs that they analyzed. These Java benchmarks create more objects, which can expose more sharing because there is a greater probability that any given object is identical to some existing object. Additionally, our tool uses knowledge of the future to find mergeable data which their collector does not have. Thus, when we apply profiling results to implement actual optimizations, we can restructure the code and data in ways that their collector cannot.

8. CONCLUSIONS AND FUTURE WORK

We have demonstrated *Object Equality Profiling* (OEP), a technique for finding opportunities for space optimizations when multiple identical objects can be replaced with one. The results show that OEP is practical and can be applied to large, complex Java applications without requiring virtual machine support. The largest instrumented programs take less than an hour to run and build the trace. Most of the cost appears to be the incessant garbage collections; relaxing the accuracy of the reachability statistics, or finding another way to compute them, would be very helpful. The post-mortem analysis of the trace takes less than 15 minutes for each of the programs. The analysis uses a carefully implemented algorithm for partitioning objects and computing mergeability metrics, which makes it require less space than running the program while building the trace.

Some, but not all, of our benchmark applications exhibit considerable mergeability of the sort we are looking for. Our case studies show how the OEP results allow us to pinpoint applications, and sites within applications, where mergeability exists. In the context of two benchmarks, we have described some simple manual optimization techniques for exploiting the mergeability, and we have shown that they lead to significant space savings.

Our tool only identifies opportunities for optimizations, but does not apply them automatically. It would be useful to have static analysis tools that could verify the soundness of optimizations, or even perform them automatically. The analysis would need to determine when an object can no longer be modified and when there can be no more identity operations performed on it. Given automatic optimizations, our profiling would help focus a compiler or virtual machine on the profitable opportunities. OEP also motivates different language design choices. The immutability and identity operations cannot be constrained by current Java’s type system, so it is difficult to verify their use. Supporting them as types/annotations would both benefit from and be beneficial to OEP.

Acknowledgments

This research was done while Darko Marinov had an internship at the IBM T. J. Watson Research Center. We would like to thank Glen Ammons, Alexandr Andoni, Chandrasekhar Boyapati, Sarfraz Khurshid, Viktor Kuncak, Derek Rayside, Bill Thies, Karen Zee and anonymous reviewers for comments on an earlier draft of this paper.

9. REFERENCES

- [1] U. A. Acar, G. E. Blelloch, and R. Harper. Selective memoization. In *30th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 14–25, New Orleans, LA, Jan. 2003.
- [2] O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, Y. S. Ramakrishna, and D. White. An efficient meta-lock for implementing ubiquitous synchronization. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 207–222. ACM Press, 1999.
- [3] O. Agesen, D. Detlefs, and J. E. Moss. Garbage collection and local variable type-precision and liveness in Java virtual machines. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 269–279, 1998.
- [4] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
- [5] J. R. Allen. *Anatomy of LISP*. McGraw-Hill, NY, 1978.
- [6] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *15th Annual ACM Symposium on the Principles of Programming Languages*, pages 1–11, January 1988. San Diego, CA.
- [7] C. S. Ananian and M. C. Rinard. Data size optimizations for Java programs. In *2003 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES '03)*, San Diego, CA, June 2003.
- [8] A. W. Appel and M. J. R. Goncalves. Hash-consing garbage collection. Technical Report TR-412-93, Princeton University, Computer Science Department, Feb. 1993.
- [9] D. F. Bacon, S. J. Fink, and D. Grove. Space- and time-efficient implementation of the Java object model. In B. Magnusson, editor, *Proceedings of the Sixteenth European Conference on Object-Oriented Programming*, volume 2374 of *Lecture Notes in Computer Science*, pages 111–132, Málaga, Spain, June 2002. Springer-Verlag.
- [10] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: Featherweight synchronization for Java. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 258–268, June 1998.
- [11] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.
- [12] R. S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 3–42. Springer, 1987.
- [13] B. Blanchet. Escape analysis for object oriented languages: Application to Java. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Denver, Colorado, November 1999.
- [14] M. Buidi, M. Sakr, K. Walker, and S. C. Goldstein. BitValue inference: Detecting and exploiting narrow bitwidth computations. In *Proceedings of the 2000 Europar Conference*, volume 1900 of *Lecture Notes in Computer Science*. Springer Verlag, 2000.
- [15] A. Cardon and M. Crochemore. Partitioning a graph in $O(|A| \log_2 |V|)$. *Theoretical Computer Science*, 19(1):85–98, 1982.
- [16] G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, B. Mathiske, and M. Wolczko. Heap compression for memory-constrained java environments. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2003.
- [17] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Denver, Colorado, Nov. 1999.
- [18] S. Dieckman and U. Hölzle. The allocation behaviour of the SPECjvm98 Java benchmarks. In R. Eigenman, editor, *Performance Evaluation and Benchmarking with Realistic Applications*, chapter 3, pages 77–108. MIT Press, 2001.
- [19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [20] D. Gay and B. Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *International Conference on Compiler Construction (CC'2000)*, pages 82–93, 2000.
- [21] O. Gheorghioiu, A. Salcianu, and M. C. Rinard. Interprocedural compatibility analysis for static object preallocation. In *Proc. of the 30th Annual ACM Symposium on the Principles of Programming Languages*, pages 273–284, Jan. 2003.
- [22] E. Goto. Monocopy and associative algorithms in an extended Lisp. Technical Report 74-03, Information Science Laboratory, University of Tokyo, Tokyo, Japan, May 1974.
- [23] M. Hertz, S. M. Blackburn, J. E. B. Moss, K. S. McKinley, and D. Stefanovic. Error-free garbage collection traces: how to cheat and not get caught. *ACM SIGMETRICS Performance Evaluation Review*, 30(1):140–151, 2002.
- [24] J. E. Hopcroft. An $n \log n$ algorithm for minimizing the states in a finite automaton. In *The Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.
- [25] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, 1994.
- [26] D. Michie. "memo" functions and machine learning. *Nature*, 218, Apr. 1968.
- [27] T. Murphy, B. Harper, and K. Crary. The wizard of tilt: Efficient², convenient and abstract type representations. Technical Report CMU-CS-02-120, School of Computer Science, Carnegie Mellon University, Mar. 2002. <http://www-2.cs.cmu.edu/~tom7/papers/wizard.ps.gz>.
- [28] W. Pugh. An improved replacement strategy for function caching. In *Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 269–276. ACM Press, 1988.
- [29] N. Rojemo and C. Runciman. Lag, drag, void and use–heap profiling and space-efficient compilation revisited. In *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming*, pages 34–41, 1996.
- [30] R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In

- Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 182–195. ACM Press, 2000.
- [31] R. Shaham, E. K. Kolodner, and M. Sagiv. On effectiveness of GC in Java. In *Proceedings of the Second International Symposium on Memory Management*, pages 12–17, 2000.
- [32] R. Shaham, E. K. Kolodner, and M. Sagiv. Heap profiling for space-efficient Java. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 104–113, 2001.
- [33] Y. Shuf, M. Gupta, R. Bordawekar, and J. P. Singh. Exploiting prolific types for memory management and optimizations. In *29th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 295–306, 2002.
- [34] M. Stephenson, J. Babb, and S. Amarasinghe. Bidwidth analysis with application to silicon compilation. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 108–120. ACM Press, 2000.
- [35] P. Sweeney and F. Tip. A study of dead data members in C++ applications. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 324–332, June 1998. *SIGPLAN Notices*, 33(5).
- [36] The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. <http://www.spec.org/osg/jvm98/>, 1998.
- [37] F. Tip and P. F. Sweeney. Class hierarchy specialization. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 271–285, 1997.
- [38] Package `java.lang.ref`. <http://java.sun.com/j2se/1.4/docs/api/java/lang/ref/package-summary.html>.
- [39] WebSphere application server development best practices for performance and scalability. White paper available from http://www-3.ibm.com/software/webervers/appserv/ws_bestpractices.pdf, Sept. 2000.
- [40] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Denver, Colorado, Nov. 1999.
- [41] B.-S. Yang, J. Lee, J. Park, S.-M. Moon, K. Ebcioglu, and E. Altman. Lightweight monitor for Java VM. *ACM SIGARCH Computer Architecture News*, 27(1):35–38, 1999.