Reuse, recycle to de-bloat software

Suparna Bhattacharya[1], Mangala Gowri Nanda[2], K Gopinath[1], and Manish Gupta[2]
suparna@csa.iisc.ernet.in,mgowri@in.ibm.com,
gopi@csa.iisc.ernet.in,manishgupta@in.ibm.com

[1] Indian Institute of Science
[2] IBM Research

Abstract. Most Java programmers would agree that Java is a language that promotes a philosophy of "create and go forth". By design, temporary objects are meant to be created on the heap, possibly used and then abandoned to be collected by the garbage collector. Excessive generation of temporary objects is termed "object churn" and is a form of software bloat that often leads to performance and memory problems. To mitigate this problem, many compiler optimizations aim at identifying objects that may be allocated on the stack. However, most such optimizations miss large opportunities for memory reuse when dealing with objects inside loops or when dealing with container objects.
In this paper, we describe a novel algorithm that detects bloat caused by the creation of temporary container and String objects within a loop. Our analysis determines which objects created within a loop can be reused. Then we describe a source-to-source transformation that efficiently reuses such objects. Empirical evaluation indicates that our solution can reduce upto 40% of temporary object allocations in large programs, resulting in a performance improvement that can be as high as a 20% reduction in the run time, specifically when a program has a high churn rate or when the program is memory intensive and needs to run the GC often.

1 Introduction

There are many forms of software bloat [1, 2]. The creation (and deletion) of many temporary objects in Java programs is known as temporary object churn; this is the form of software bloat that we address in this paper. As illustrated by Jack Shirazi [3], creating too many temporary objects results in higher garbage collection overhead, object construction costs and higher memory system stress resulting in an increase in processing time and memory consumption. At the end of the chapter on object creation in his book, Shirazi gives a long list of performance improvement strategies of which we reproduce a few here:

- Reduce the number of temporary objects being used, especially in loops.
- Avoid creating temporary objects within frequently called methods.
- Reuse objects where possible.
- Empty collection objects before reusing them. (Do not shrink them unless they are very large.)

```
  class Klass {
    Hashtable  ftab;
    void foo(int num, Hashtable tab) {
 5    HashSet seen = new HashSet();
 6    Stack work = new Stack();
 7    Vector heap = new Vector();
 8    doSomething(work, num);
 9    while ( !work.isEmpty() ) {
10      Object w = work.pop();
11      if ( seen.contains(w) )
          continue;
12      seen.add(w);
13      heap.add(w);
      }
      Integer inum = new Integer(num);
14    if ( init() ) {
15      ftab.put(inum, heap);
      }
      else {
16      tab.put(inum, heap);
      }
    }

    void bar(int num) {
32    Hashtable tab = new Hashtable();
33    for ( int n=0; n<num; n+=10 ) {
37      foo(n, tab);
      }
43    dumpTabContent(tab);
    }

    void driver() {
44    for ( int num=0; num<100; num+=5 ) {
45       bar(num);
      }
    }
  }
```

(a) sample code

```
  class Klass {
    Hashtable  ftab;
    void foo(int num, Hashtable tab) {
 5    HashSet seen = REUSE.REUSE_01();
 6    Stack work = REUSE.REUSE_02();
 7    Vector heap = new Vector();
 8    doSomething(work, num);
 9    while ( !work.isEmpty() ) {
10      Object w = work.pop();
11      if ( seen.contains(w) )
          continue;
12      seen.add(w);
13      heap.add(w);
      }
      Integer inum = new Integer(num);
14    if ( init() ) {
15      ftab.put(inum, heap);
      }
      else {
16      tab.put(inum, heap);
      }
    }

    void bar(int num) {
32    Hashtable tab = new Hashtable();
33    for ( int n=0; n<num; n+=10 ) {
37      foo(n, tab);
      }
43    dumpTabContent(tab);
    }
  }
  class REUSE {
52  static HashSet hs_01 = new HashSet();
53  HashSet REUSE_01() {
54    hs_01.clear(); return hs_01;
    }
57  static Stack st_02 = new Stack();
58  Stack REUSE_02() {
59    st_02.clear(); return st_02;
    }
  }
```

(b) Code reused

Fig. 1. Sample code.

However, this is easier said than done, especially for Java programmers who have grown up with the luxury of creating and discarding temporary objects, on the assumption that the discards would be efficiently garbage collected. Consider, for example, a typical piece of Java code as shown in Figure 1(a):

In this program, `foo()` calls `doSomething()` which loads several objects into a stack `work`. Then `foo()` picks up each element in the stack, checks for and discards any duplicates using `seen` and loads the unique objects into `heap`. At the end, based on some condition, `heap` is stored into either the field variable `this.ftab` or into the `Hashtable tab`[3] passed in as a parameter[4]. The method `bar()` calls `foo()` iteratively and then dumps the contents of `tab` while the method `driver()` calls `bar()` iteratively.

---

[3] For ease of exposition we model a hashtable as directly containing the key and value fields *e.g.*, `tab.value` instead of containing the fields only indirectly *e.g.*, `tab.bucket`$[i]$`.element`$[j]$`.value`.

[4] This code was modified from Xylem code (refer Section 5), the only modification being the addition of `ftab` and the corresponding lines of code at lines 14 and 15 to highlight that an object may be reusable along one path but not another

Here we observe that `foo()` is called from inside a loop. Hence `HashSet seen`, `Stack work` and `Vector heap` will be created once for every iteration of the loop. Also, it is intuitively clear that `seen` and `work` can be reused, but `heap` may not be reusable.

- Consider `Stack work`: it is created locally and is not accessible outside `foo()`—that is, it does not escape `foo()`. It may be reused as shown in Figure 1(b). Note, however, that the enclosing loop is in a different method than the objects being reused and thus requires interprocedural analysis. Nevertheless, `work` is reusable within the innermost enclosing loop and hence is termed a "Level 1" reusable object.
- Consider `Vector heap`: it is created locally but it is accessible outside `foo()`—that is, it escapes `foo()`.
  - Consider the case when it escapes via `tab`: `heap` does not escape the method `bar`, but it does "escape" the loop inside `bar`. Going further back up the call flow graph, we find that `bar` is called from within a loop. Since `heap` does not escape from this loop, it is potentially reusable. In this case, `heap` is not reusable within the innermost enclosing loop, but it is reusable within the next enclosing loop and hence is termed a "Level 2" reusable object.
  - When it escapes via `ftab`: `ftab` is accessible outside `bar` and `driver` and hence so is `heap`. Therefore, `heap` is not reusable along this path.

When we reused `seen` and `work` as shown in Figure 1(b), we observed a 9% reduction in execution time (on a dual core Intel(R) Core(TM)2 Duo system with 2GB RAM running Java Hotspot(TM) Server VM on Linux).

Thus we see that objects may be reused within the immediately enclosing loop or a higher level loop. The same object may be reusable along one path but not another. Similarly, the same object may be reusable at different levels along different paths. Besides these, there are many issues related to this kind of code transformation:

1. How do we determine automatically which variable can be reused and which one cannot be
2. Which data structures do we target and how do we know how to "clear" the structure before reuse
3. How do we determine when to perform the allocation and the "clear". In the example, we have given a trivial solution which does not always work
4. Where do we insert the reuse code so that it does not become an overhead in itself

Although, in principle, it is possible to reuse any data structure, in our implementation, we address only certain Collection classes—specifically `HashSet, Vector, Stack, PriorityQueue, LinkedList, ArrayList` and `TreeMap`. This makes it easy to clear the objects using the `clear()` method from the Collection class. We also reuse memory in Strings (here we are referring to the reuse of the underlying arrays and not reuse of the string representation by string interning). This is far more complex and the details are given in Section 4.

Contributions In this paper we give a novel algorithm for automatically finding sources of software bloat and then we give a solution to transform the code to reduce the bloat. The main contributions are:

- An algorithm that can detect objects created within a loop and determine whether an object created within a loop can be reused at the end of each iteration. In the case of nested loops, the algorithm will tell us the innermost enclosing loop in which the object can be reused.
- A solution that can automatically transform the source code to reuse the object such as to mitigate the effects of software bloat.
- An implementation that validates our claims and shows that we can get upto 40% reduction in bytes of temporary objects generated and 20% improvement in speed of execution.

Organization We start off with some definitions and a description of escape analysis used in this paper in Section 2. In Section 3 we explain how to find safe reusable allocations and in Section 4 we give an algorithm that achieves the reuse through a source-to-source transformation. In Section 5 we report the empirical justification for using our analysis, Section 6 positions our work with respect to related work and we conclude in Section 7.

2 Preliminaries

The *control-flow graph* (CFG) for a method M contains *nodes* that represent statements in M and *edges* that represent potential flow of control among the statements.

We define here some terms used in the paper.

Definition 1. Dominator*: A node S_i* dominates *a node S_j* iff *$S_i \neq S_j$ and S_i is on every path from Entry to S_j.*

Definition 2. Postdominator*: A node S_j* postdominates *a node S_i* iff *$S_i \neq S_j$ and S_j is on every path from S_i to Exit.*

Definition 3. Control dependence*: A node S_j postdominates a branch of a predicate S_i* iff *S_j is the successor of S_i in that branch or S_j postdominates the successor of S_i in that branch.*

A node S_j is control dependent on a predicate S_i iff *S_j postdominates a branch of S_i but S_j does not postdominate S_i. A node can be directly control dependent on itself. Note that a node with only one successor can never be the source of a control dependence edge.*

Definition 4. Backedge*: A backedge in the CFG is an edge where the destination of the edge dominates the source of the edge.*

Definition 5. Data Dependence*: A node S_j is data dependent on a node S_i, if S_i defines some variable x, S_j uses the variable x, and there exists a path from S_i to S_j without intervening definitions of x.*

Definition 6. Loop Carried Data Dependence*: A node S_j is data dependent on a node S_i, if S_i defines some variable x, S_j uses the variable x, and there exists a path from S_i to S_j without intervening definitions of x and the path contains a backedge.*

Escape Analysis To locate reuse possibilities, we use escape analysis which is a method for determining the dynamic scope of pointers. After constructing the control-flow graph of each method, our solution uses flow- and context-sensitive pointer analysis and escape analysis[5]. The escape analysis computes the escape-in and escape-out sets for each method.

- The *formal-in* set for a method M contains the set of formal parameters. The implicit *this* parameter (in non-static methods) is also a formal-in.
- The *formal-out* set for a non-void method M contains a single parameter R, the designated return value. The *formal-out* set is empty for a void method.
- The *escape-in* set for a method M contains direct and indirect fields of the formal parameters of M that are used, before possibly being defined, in M. These represent the upwards-exposed uses in M.
- The *escape-out* set for M contains direct or indirect fields of the formal parameters of M and the return value of M that are defined in M.
- At each *Call* site c that calls method M, the algorithm uses the escape-in and escape-out information, to compute the actual-in and actual-out sets, where
- we generate an *actual-in* for each formal-in and each escape-in and
- we generate an *actual-out* for each formal-out and escape-out in M.

The algorithm associates escape-in and formal-in sets with the *Entry* node of the CFG, and escape-out and formal-out sets with the *Exit* node of the CFG; likewise, the actual-in and actual-out sets are associated with call sites.

In the example shown in Figure 2, `num` (node 2) and `tab` (node 3) are formal-in parameters in the method `foo` as is the `this` parameter although it is not shown explicitly in the figure. `this.ftab` (node 4) is an escape-in parameter where `ftab` is a field of the formal-in parameter `this`. There is no formal-out parameter as both the functions are void functions, but `this.ftab.key` (node 20) and `this.ftab.value` (node 21) are escape-out parameters generated from the `put` method of the Hashtable. Similarly, `tab.key` (node 22) and `tab.value` (node 23) are escape-out parameters since they are fields of the formal-in `tab`.

In method `bar` at the call site for `foo` (node 37) we have generated actual-ins and actual-outs and mapped them appropriately to the formal-in and escape-out parameters in the called function.

[5] A *context-sensitive* analysis propagates states along interprocedural paths that consist of valid call–return sequences only—the path contains no pair of call and return that denotes control returning from a method to a call site other than the one that invoked it. A *flow-sensitive* analysis, on the other hand, takes into account the order of statements in a program.

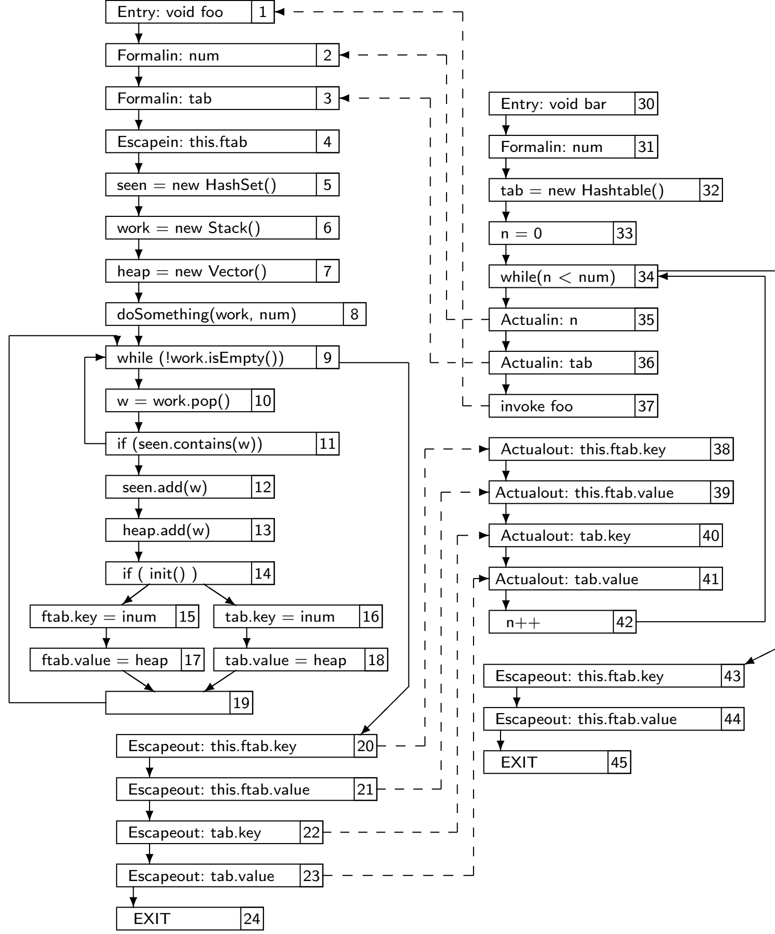

Fig. 2. Escape Analysis.

3 Finding potential sources of bloat

In this section, we describe how to locate object allocation sites within loops that can be reused. Our analysis first identifies whether an allocation site is within a loop. This allocation site can be converted into a reuse site on the condition that

1. it does not have a loop carried dependence
2. it is not accessed outside the loop. To determine whether it is local to an enclosing loop, we need to check if it escapes the scope of the loop.

3.1 The Problem with Loop Carried Data Dependence

Consider the following piece of Java code

```
Vector vprev = new Vector();
while ( cond ) {
    vsucc = new Vector();
    process(vsucc);
    if ( vsucc.size() <= vprev.size() ) {
    }
    vprev = vsucc;
}
```

Here there is a loop carried dependence from vsucc to vprev. So if we reset and reuse vsucc inside the loop, then after the first iteration vsucc and vprev will always point to the same Vector, which is not correct.

Knowing that vsucc can be reused after N cycles, it is possible to design reuse as follows:

```
Vector tmp[] = new Vector()[N];
for (int j=0; j<N; j++) {
    tmp[j] = new Vector();
}
Vector vprev = new Vector();
int i=0;
while ( cond ) {
    tmp[i].clear();
    vsucc = tmp[i];
    i++;
    if ( i == N ) {
      i = 0;
    }
    process(vsucc);
    if ( vsucc.size() <= vprev.size() ) {
    }
    vprev = vsucc;
}
```

Finding loop carried dependence is relatively simple. However, it is not always possible to determine statically after how many cycles vsucc would be reusable, as shown in the example below.

```
Vector vprev = new Vector();
while ( cond ) {
    vsucc = new Vector();
    process(vsucc);
    if ( vsucc.size() >= vprev.size() ) {
      vprev = vsucc;
      ...
    }
}
```

Hence, we conservatively ignore reuse when there is a loop carried data dependence.

3.2 The Basic Algorithm

The preliminary analysis consists of the following steps

- We compute flow and context sensitive data and control dependence, wherein the data dependence includes points-to and escape analysis described in the previous section. Each data dependence that is a loop carried dependence is flagged appropriately.
- We determine which conditionals in the Java bytecode are loop conditionals. We define a loop header as the destination of an edge in the CFG such that the node at the destination of the edge dominates the node at the source of the edge.
- We find all allocation sites in the code. For each allocation site S_{new} we compute the transitive closure of control dependencies. This process is performed interprocedurally. If S_{new} is not directly or transitively control dependent on a loop header, then we can discard it as being uninteresting from the point of reuse.

Computing transitive closure of control dependencies Intra-procedurally speaking, every node is eventually control dependent on the ENTRY node of the method. The ENTRY node is inter-procedurally control dependent on the CALL node from where the method is invoked. The transitive closure thus includes all nodes that the CALL node is control dependent upon.

In Figure 2, `HashSet seen = new HashSet()` is control dependent on the ENTRY node `void foo()`. The ENTRY node is inter-procedurally control dependent on the CALL node `foo(n, tab)` in the method `bar()`. The CALL node is control dependent on the for conditional `n<num`. Hence, the allocation `seen = new HashSet()` is inter-procedurally and transitively control dependent on a loop header.

Removing Unnecessary Loop Header Dependencies Note that a node that is control dependent on a loop header is not necessarily within the loop. However, we are only interested in finding allocations that are within a loop and hence need to perform additional computation.

All nodes within the loop are directly or transitively control dependent on the loop header. However there may be nodes outside the loop that are also control dependent on the loop header. This happens when there is a return from within the loop or when there is an exception flow edge from within the loop. Since we are interested only in nodes within a loop, we need to filter out these external-to-the-loop nodes. We do this by simply checking if there is a path from the node to the loop header that ends with a back edge.

Having found an allocation site that lies within a loop, we perform the algorithm given in Figure 1.

The algorithm takes as input S_{new}, the allocation site $v =$ `new Collection()`, where Collection is one of the classes mentioned in Section 1. It then computes the forward slice for S_{new} as explained at lines 27–36. The forward slice consists

Algorithm 1 Locating reusable allocations within a loop

```
1: INPUT: S_new
2: φ_esc ← new Collection()
3: φ_reg ← new Collection()
4: computeForwardSlice({S_new}, φ_esc, φ_reg)
5: N_CD ← controlDepPred(S_new)
6: while N_CD ≠ null do
7:    if N_CD is a loop header then
8:       level++
9:       if φ_esc = {} and contains(N_CD, φ_reg) and noLoopDD(φ_reg) then
10:         OUTPUT(level, S_new)
11:      end if
12:   else if N_CD is an Entry node then
13:      for all N_invoke a call site of Entry do
14:         newset ← map(N_invoke, φ_esc)
15:         φ_esc ← new Collection()
16:         φ_reg ← new Collection()
17:         computeForwardSlice(newset, φ_esc, φ_reg)
18:      end for
19:   else
20:      reached the top of the call graph
21:      report and exit
22:   end if
23:   N_CD ← controlDepPred(N_CD)
24: end while
25:
26: computeForwardSlice(newset, φ_esc, φ_reg) {
27: while !newset.empty() do
28:   N ← newset.removeLast()
29:   for all N_dd such that N_dd is data dependent on N do
30:      if N_dd is a formal-out or an escape-out then
31:         φ_esc ← φ_esc ∪ N_dd
32:      else
33:         φ_reg ← φ_reg ∪ N_dd
34:      end if
35:   end for
36: end while
37: }
```

of the transitive closure of all def-use sets starting with the definition at S_{new}. The nodes in the slice are separated into two bags, one called the "Escape" bag ϕ_{esc} that contains any `formal-outs` or `escape-outs` in the slice and the other bag ϕ_{reg} that contains all other nodes.

Next we track backward along the control dependence edges.

- If we come to a loop conditional we check if ϕ_{esc} is empty and every node in ϕ_{reg} lies within the loop and does not have a loop carried dependence. If yes, then we have found the closest enclosing loop inside which S_{new} can be reused—along this particular path. We record this path and stop traversing the control flow graph any further for this path. For all other loop conditionals or branch conditionals, continue climbing up the control dependence graph.
- If we come to the `Entry` node of a method S_M, then for each invocation site, S_{call}, we map each node in the ϕ_{esc} set to the corresponding `actual-out` nodes. The old ϕ_{esc} and ϕ_{reg} sets are discarded and fresh sets are computed as the union of the forward slices of the `actual-out` nodes at the given invocation site. Then the analysis continues up the control dependence graph,
- If we come to the top of the call flow graph, we conclude that S_{new} may not be reusable along this path.

An illustrative example Consider the example in Figure 2.

1. We determine that the conditional nodes `n < num` (node 34) in `bar` and `!work.isEmpty()` (node 9) in `foo` are loop headers.
2. Consider the statement `seen = new HashSet()` (node 5) in method `foo` in Figure 1. It is an allocation site for the Collection class HashSet. This node is control dependent only on the entry node `void foo` (node 1). This node is call dependent on the `invoke foo` node (node 37) which in turn is control dependent on the loop header `n < num`. Hence the allocation statement is interprocedurally called from inside a loop and has potential to be reused.
3. The forward slice is computed as $\phi_{reg} = \{$ `seen.contains(w)`, `seen.add(w)` $\}$ and ϕ_{esc} is empty as none of the nodes are formal-outs or escape-outs.
4. Now we traverse the control dependence path. At the entry node there is nothing to be mapped to the `invoke foo` site as ϕ_{esc} is empty. We discard the current ϕ sets and enter the method `bar` with empty sets. Next the `invoke foo` is control dependent on the loop header `n < num`. Here the requisite conditions are trivially true. Hence the allocation may be converted to reuse.

If we consider the allocation site `heap = new Vector()`, it has four escape-outs in its ϕ_{esc}; these map into actual-out nodes in the calling function `bar`. These actual-out statements are inside the loop but their forward slice contains nodes that are outside the scope of the loop. Two of these escape out of `bar` as well as its caller `Driver()`. Hence, this node is correctly not marked for reuse.

3.3 Multiple Control Dependence Paths

The basic analysis algorithm described above records the closest enclosing loop along a control dependence path where reuse may be implemented safely, if at

all. Since there may be multiple paths to an allocation site, several situations may arise:

1. The site is not reusable along any control dependence path
2. The site is reusable along some control dependence paths, but not reusable along other control dependence paths.
3. The site is reusable along all its control dependence paths, but the closest enclosing loop where reuse can be implemented is not the same for all control dependence paths
4. The site is reusable inside the same closest enclosing loop along all its control dependence paths

One could take a conservative approach where only Case 4 is assumed to be safe for reuse conversion. However, this tends to miss several sites with potentially large churn (as we observe experimentally). A second approach is to introduce extra code to perform runtime tracking of the conditions for safe reuse in all situations. While this can enable more opportunities for reuse, it can become fairly complicated and invasive. For example, in the worst case, this might involve interprocedurally tracking path history along every branch leading to an allocation site from enclosing loops located several call levels away.

Instead, we use a simpler scheme that achieves greater precision than the conservative analysis but only exploits runtime state that needs to be introduced anyway for implementing object reuse.

Let us define the *height* h of a loop L along a control dependence path from an allocation site as the number of enclosing loop headers along that path upto and including L. Then, the reuse level k for an allocation site along a particular control dependence path is defined as the height of the closest enclosing loop where the site is reusable for that path. This means that object reuse state for that allocation site must be maintained across iterations of all the inner loops upto height $k-1$, and can only be reset across iterations of the loop at height k or above. As long as this condition can be met across all control dependence paths for the site without conflict, the object can be safely converted for reuse along certain paths (where it is found to reusable) without affecting correctness along its other control dependence paths. This logic can be extended to address not just Case 3, but Case 2 as well, since a path that does not support reuse can be treated as a path with a very high reuse level. In other words, at some outer loop level we can setup one control flow path to reuse and the another to not reuse, provided the two paths do not intersect within the same outer loop iteration.

Illustration: Consider the following variation of example in Figure 2, without lines 14-15, so that the allocation to `heap` no longer escapes directly via `ftab`. Now, suppose we add a couple of routines as follows:

```
void barPersist(int num) {
    for (int n=0; n<num; n+=10) {
        foo(n, ftab);
    }
}
```

```
void driverPersist() {
    for (int num=0; num<100; num+=5) {
        barPersist(num);
    }
}
void mainDriver() {
    if (init()) {
        driverPersist()
    } else {
        driver();
    }
}
```

The allocation site `heap = new Vector()` in `foo()` is reusable at level $k = 2$, the loop in `driver()` that calls `bar()`, along one control dependence path, but is not reusable along another path that goes through `driverPersist()` and `barPersist()`. In this case, we notice that there is no conflict between these two cases as the corresponding loops do not intersect.

Now, let us say the routines `driverPersist()` and `mainDriver()` were removed and instead, the routine `driver()` modified as follows:

```
void driver() {
    for (int num=0; num<100; num+=5) {
        if (init()) {
            barPersist(num);
        } else {
            bar(num);
        }
    }
}
```

This time, the outer loop is common to the two paths, which indicates a potential conflict.

We perform an analysis of the loop sharing structure across control dependence paths to eliminate such potential conflicts.

Figure 3 illustrates some examples of loop header sharing across control dependence paths starting from two distinct nodes P and Q respectively. The loop header nodes are numbered according to the height of the loop with respect to the allocation site. In (a) the loops are embedded along both paths, hence there is a conflict if a reuse site in the the inner loop is not reusable along either P or Q. In (b), the loops are disjoint and both inner loops invoke the method containing the reuse site. In this case, for reuse level $k = 2$ and above, the the $k - 1$ loop of one path never falls inside the other. Hence if the site is reusable at level 2 (or higher) along paths from P, then, even if it is not reusable from Q, our transformation can be set up to safely exploit object reuse along the former. In (c), the innermost loop is shared, but the outer loops are disjoint. Thus a reuse transformation upto level 2 would be unsafe unless both paths share the same reuse level. However, if the site is reusable at level 3 or higher along one path, then, even if it isn't reusable along the other, our transformation can be set up safely to exploit object reuse at the appropriate level along that path.

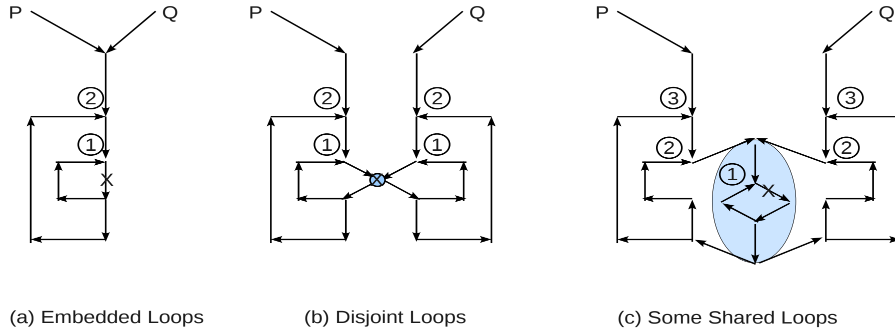

Fig. 3. Loop header sharing for multiple control dependence paths

4 Object reuse, recycle transformations

In the previous section we described an approach for finding allocation sites that are candidates for object reuse and the closest enclosing loops where they can be safely reused. Now we discuss our automated code transformations for implementing object reuse.

Object reuse optimizations may involve object memory reuse or object content reuse. The former recycles the memory and structural representation state of objects of the same type, instead of allocating fresh objects each time. The latter reuses at least some part of the actual object content (a form of memoization/caching) to save repeated content construction costs[6]. Our static analysis based detection technique mainly identifies the first kind of opportunities, hence this forms the focus of our implementation. Our code transformations can, however, be used to support the second category of reuse as well, with slight modifications[7].

4.1 Basic reuse-recycle algorithm

The static analysis phase reports the safe reuse level and the corresponding loops for each allocation site identified for reuse (hereafter referred to briefly as a reuse

[6] object canonicalization is an extreme example of content reuse; object pooling involves memory and sometimes partial content reuse

[7] e.g. steps like clearing the object or simulating effects of a constructor may be skipped when reusing object content, thus simplifying the implementation

site). We use this information to implement a basic reuse/recycle algorithm for these reuse sites.

The simple reuse transformation illustrated in the introduction is efficient but does not work in many situations. The allocation has been moved to a static initializer where constructor parameters that are specified at the allocation site may not be available. The conversion is only applicable for level 1 reuse, i.e. for objects allocated in an inner loop which can be recycled at the next iteration of this loop. In this case, it suffices to allocate a single reusable slot for an allocation site, e.g. the variable `REUSE.st_01` for `Stack work`. However, when allocated objects need to be preserved across iterations of an inner loop, and can only be recycled at a subsequent iteration of an outer loop, multiple reusable slots must be maintained for the same allocation site. This happens for a level k reuse with $k > 1$, (where k is the height of the closest enclosing loop at which the object can be reused), e.g. $k = 2$ for `Vector heap` when `foo()` is invoked via `bar()`. In this case, static initialization cannot be used as the number of distinct slots required (minimum outstanding allocations) may not be known until the inner loop completes its first iteration sequence. It could even change dynamically. The number of inner loop iterations and hence reuse slots maintained for `Vector heap` varies with the loop upper bound num, e.g. when $num = 50$, 10 reuse slots are used. We note that this also means that the number of reuse slots created must be bounded to avoid causing memory overhead due to a blowup in the number of inner loop iterations.

Therefore in our generalized implementation (Algorithm 2), the allocation statement is not moved, but instead, tracked during the first iteration of the level k loop by creating reuse slots as needed and initializing them with the result of the allocations in inner loop iterations (lines 9-12). Note that `Reusevar.numslots` is statically initialized to zero. It is incremented (line 10) each time a reuse slot is created for this site, using `Reusevar.addslot()` (line 11). The objects from these slots are then reused sequentially (lines 16-17) during subsequent iterations of the level k loop. If the inner loop iterations exceed the number of available reuse slots `Reusevar.numslots` (e.g. due to a varying loop bound), then additional slots are created as required (upto a maximum allowed capacity) (lines 9-12). If the maximum capacity of reuse slots is exceeded for a given allocation site, then allocations required beyond the capacity simply fall back to a non-reusable mode (lines 13-14).

This approach has the downside of an extra check in every inner loop iteration to distinguish the first iteration[8] of the level k loop from iterations which reuse previous allocations. The overhead may be optimized using loop peeling and specialization for common scenarios (like level 1 reuse for collection objects which do not require a constructor parameter).

To enable an existing object to be recycled instead of issuing a fresh allocation, some type specific steps need to be executed to re-initialize the object for reuse. In general, this may require simulating (a part of) its constructor functionality. We focus on reusing collection objects and strings.

[8] and checks for dynamic expansion of slots

Algorithm 2 General level K reuse transformation

```
1: while condition K do
2:    processing for K^th loop
3:    slot ← 0; maxslots ← MAXSLOTS {ADDED TO ENABLE REUSE}
4:    while condition K − 1 do
5:       processing loops for K − 2 to 2
6:       while condition 1 do
7:          processing for inner loop
8:          BEGIN: Transformed allocation statement {TO ENABLE REUSE}
9:          if Reusevar.numslots ≤ slot < maxslots then
10:            Reusevar.numslots ← Reusevar.numslots + 1
11:            Reusevar.addslot() ← new TYPE(params)
12:         end if
13:         if Reusevar.numslots ≤ slot then
14:            var ← new TYPE(params)
15:         else
16:            var ← Reusevar.getslot(slot)
17:            slot ← slot + 1
18:         end if
19:         END : transformed allocation statement {TO ENABLE REUSE}
20:         some more processing for inner loop
21:      end while
22:      some more processing for loops for K − 2 to 2
23:   end while
24:   some more processing for K^th loop
25: end while
```

4.2 Reusing collections

Preparing a collection object for reuse is particularly simple. Most collections provide a clear() method to reset a collection to zero entries while keeping the capacity of the collection intact. The larger the collection being reused, the greater the benefit as it saves a large portion of object construction costs.

4.3 Reusing strings

Recycling String objects requires simulating a part of its constructor functionality to re-populate the underlying character array with new content. Since a String object is an immutable data structure, this can be implemented efficiently only with special extension support from the class library or the JVM. For our experimental evaluation, we use reflection to access/clear/overwrite the array as required. This incurs a performance penalty, which is mitigated to some extent by caching the reflection results when the object is first allocated to avoid the overhead on every iteration. Therefore our results provide a conservative estimate of performance gain that can be attained through object reuse in this case.

4.4 Implementation Details

We used a source to source transformation approach to evaluate the feasibility of our automated object reuse/recycle conversion. Simplicity and clarity were our primary motivation for choosing this approach, e.g. ability to perform a visual inspection of the changes in source code after transformation. The conversion may also be implemented using byte-code manipulation and JVM level optimizations as discussed later.

The inputs required for the transformation are the output of the static analysis stage, and the source files of the application to convert.

Figure 4 illustrates our running example before and after automatic reuse conversion. The transformations are performed in a single-pass over source. This is a slightly modified version of our running example from previous sections, without the `ftab` field. As `heap` no longer escapes via `ftab`, it is now reusable at level 2 (i.e. the loop starting at lineno 38 in `driver()`).

```
   class Klass {
     void foo(int num, Hashtable tab) {
14:    HashSet seen = new HashSet();

15:    Stack work = new Stack();

16:    Vector heap = new Vector();

       doSomething(work, num);
       while ( !work.isEmpty() ) {
         Object w = work.pop();
         if ( seen.contains(w) )
           continue;
         seen.add(w);
         heap.add(w);
       }
       Integer inum = new Integer(num);
       tab.put(inum, heap);
     }

     void bar(int num) {
       Hashtable tab = new Hashtable();
31:    for ( int n=0; n<num; n+=10 ) {
         foo(n, tab);
       }
       dumpTabContent(tab);
     }

     void driver() {
38:      for ( int num=100; num > 0; num-=5 ) {
         bar(num);
       }
     }
   }
```

(a) Before transformation

```
     class Klass {
     void foo(int num, Hashtable tab) {
14:    HashSet seen = REUSE.ReuseHashSet_14();

15:    Stack work = REUSE.ReuseStack_15();

16:    Vector heap = REUSE.ReuseVector_16();

       doSomething(work, num);
       while ( !work.isEmpty() ) {
         Object w = work.pop();
         if ( seen.contains(w) )
           continue;
         seen.add(w);
         heap.add(w);
       }
       Integer inum = new Integer(num);
       tab.put(inum, heap);
     }

     void bar(int num) {
       Hashtable tab = new Hashtable();
31:    REUSE.idxVector_16 = 0;
       REUSE.maxVector_16 = MAX_SLOTS;
       for ( int n=0; n<num; n+=10 ) {
         foo(n, tab);
       }
       dumpTabContent(tab);
     }

     void driver() {
38:      for ( int num=100; num > 0; num-=5 ) {
         bar(num);
       }
     }
   }
```

(b) After transformation

At each listed allocation site to convert for reuse (lines 14,15,16), we replace the call to `new` with a call to an allocation site specific reuse method which performs allocation tracking and reuse. At the statement preceding a listed allo-

```
public class REUSE{
  static HashSet HashSet_14;
  public static HashSet ReuseHashSet_14() {
    if (HashSet_14 != null) {
      REUSEUtil.clearHashSet(HashSet_14);
    } else {
      HashSet_14 = new HashSet();
    }
    return HashSet_14;
  }

  static Stack Stack_15;
  public static Stack ReuseStack_15() {
    if (Stack_15 != null) {
      REUSEUtil.clearStack(Stack_15);
    } else {
      Stack_15 = new Stack();
    }
    return Stack_15;
  }
  ...
}
```

```
static int idxVector_16;
static ArrayList<Vector> Slot_Vector_16 =
                  new ArrayList<Vector>();
static Vector Vector_16;
public static Vector ReuseVector_16() {
  if (idxVector_16 < Slot_Vector_16.size()) {
    Vector_16 =
     Slot_Vector_16.get(idxVector_16++);
    REUSEUtil.clearVector(Vector_16);
  } else {
    Vector_16 = new Vector();
    if (idxVector_16 < maxVector_16) {
      Slot_Vector_16.add(Vector_16);
      idxVector_16++;
    }
  }
  return Vector_16;
}
```

Fig. 4. Code transformation example

cation site's level $k-1$ loop header, we insert code to reset the reuse slot index for the allocation site and specify the maximum slots that may be created. Line 31 is the level 1 loop header corresponding to the level 2 reusable allocation of `heap` (`idxVector_16` is the corresponding reuse slot index).

A reuse context area and allocation site specific reuse methods are generated by the transformation. The reuse context fields maintain state corresponding to every allocation site that is converted for reuse. `Stack_15` maintains a reference to the level 1 reusable allocation for `work`. The reuse methods encapsulate allocation tracking and reuse logic specific to these allocation sites, e.g. `ReuseVector_16()` uses `Slot_Vector_16` to keep track of the reuse slots for the allocation of `heap` at line 16. The size of the arraylist `Slot_Vector_16` thus corresponds to `Reusevar.numslots` in Algorithm 2. For level 1 reuse, as in the case of `seen` and `work`, there is a single reuse slot which is accessed directly from `HashSet_14` and `Stack_15` respectively. Before returning the reusable reference, these methods invoke a type-specific utility method to enable reuse for that object (`clearHashSet()` for `seen`, `clearStack` for `work` and `clearVector` for `heap`).

Reuse context entries are typically stored in a thread local reuse context area. For single-threaded programs like the above example, we maintain a global reuse context, to avoid the overhead of thread local context accesses in the interprocedural case.

4.5 Dynamic analysis guided filtering of candidate reuse sites

A purely static analysis based detection scheme has insufficient information to prioritize allocation sites to convert based on an estimate of expected savings. We complement it with a dynamic analysis that profiles allocation sites with high object churn to guide the selection of statically identified candidate reuse

sites that are worth converting. We then apply our object reuse transformation for those reuse sites.

The dynamic analysis phase takes as input the reuse sites reported by static analysis and the output of an allocation profiler that captures the volume of allocated and live vs freed bytes generated at each allocation site under a typical run of the program. It then generates statistics about the proportion of churn generated by reuse sites which use collections or strings, and selects the top sites with significant contribution to overall volume of temporary objects generated.

4.6 Discussion

Alternatives to full source to source transformation Instead of using a pure source to source transformation approach, object reuse transformations could also be implemented using byte code manipulation and JVM level optimizations. A JVM can avoid costs of reflection and thread local accesses that we incur and optimize the overhead of the check required in each iteration to distinguish first time allocation and reuse iterations. It can also enable profile guided object reuse conversion to be applied at runtime for the reusable allocation sites that exhibit the potential for highest savings.

Extending the technique to non-collection objects The technique may be generalized further to any object type that is designed to support a special reuse interface with a type specific reuse method. This method provides an alternative to the constructor that is called to clear a previous instance of the object or re-populate it with new content. Such an approach can also be used to enable partial content reuse by implementing the reuse method to selectively preserve the content of some fields of the object.

5 Empirical evaluation

We apply our analysis to a few large applications, the SPECjbb2005 benchmark and the DaCapo benchmarks [4] lusearch, ps, pmd, antlr. We also apply it to Xylem [5], a proprietary tool that has been built to statically detect null dereferences in Java. In this paper we analyze only a subset of Xylem. Table 1 lists a brief description of the benchmarks used. The freed memory was measured from the garbage collection (GC) logs saved during execution of the applications and represents the total bytes freed over all GC cycles.

We apply the static analysis to all these applications, and use our dynamic analysis to select the applications and candidate reuse sites to convert from the safe reuse sites found. As shown in Table 3, SPECjbb2005, xylem and lucene indicate the greatest potential for savings from object reuse for collections (including Strings and arrays). Hence we apply our automatic transformation to these applications and report the results in Table 4.

The following section presents our experimental results and analysis.

Application	Description	Freed Memory (Object Churn)
SPECjbb2005	Server-side Java Benchmark	8 KB/txn
xylem	Proprietary tool to detect null references	1203MB
DaCapo lusearch	A text search tool	4913 MB
DaCapo pmd	A source code analyzer for Java	1178 MB
DaCapo ps	A postscript interpreter	2366 MB
DaCapo antlr	A parser generator and translator generator	884 MB

Table 1. Benchmarks analysed

	SPECjbb	xylem	lusearch	pmd	ps	antlr
functions analysed	864	1679	2614	5587	1022	2486
statements analysed	864	33924	69688	126312	19078	100359
total analysis time	23s	33s	55s	8m 15s	20s	3m 16s
prelim analysis time	18s	25s	41s	3m 41s	16s	2m 29s
Results	SPECjbb	xylem	lucene	pmd	ps	antlr
no. of alloc sites	1014	1853	1549	2252	1013	2712
no. of alloc sites in loops	784	1456	776	1076	146	1852
no. of (safe) reuse sites found	251	400	688	577	77	375
no. of collection reuse sites	84	220	266	125	12	97
no. of string reuse sites	27	0	9	4	2	51
no. of sites reusable only along some paths	273	148	657	507	67	401
pure level 1 reuse sites	90	274	197	166	28	79
pure level 2 reuse sites	4	3	2	0	0	0
min level 1 reuse	280	416	766	640	93	477
min level 2 reuse	15	6	9	1	0	2

Table 2. Reuse site detection statistics

5.1 Reuse site detection statistics (static analysis)

Table 2 summarizes the results of from the static analysis phase to find safe reuse sites and the closest enclosing loop where they may be reused. We notice that most opportunities exist at level 1 or level 2 reuse, and that a significant number of sites are only reusable along some paths and not others. Less than half of the safe reuse sites found are reusable at a single level along all paths. Except for ps, most benchmarks have a significant number of collection or string reuse sites.

Discussion: Analysis Time and Scalability Table 2 also reports the times for analysis, and how much of that is spent on the preliminary analysis. We rely on an underlying context-sensitive flow analysis. This is, in general, slow, however, with suitable engineering, it can be reasonably scalable. Our analysis

	SPECjbb	xylem	lusearch	pmd	ps	antlr
%churn at (safe) reuse sites	54%	18.4%	77.5%	16.4%	6%	14.6%
%churn at collection (and string) reuse sites	46.6%	16.5%	63%	5%	3.7%	4.25%
%churn at sites reusable only along some paths	6.8%	3.28%	77.2%	15.9%	6%	14%
no of reuse sites with more than 1% churn	7	3	22	9	5	12
no of collection reuse sites with more than 1% churn	5	2	8	1	2	1
%churn at top 3 reuse sites	48%	16%	46%	10%	5.8%	5.8%
Distribution of reuse levels	SPECjbb	xylem	lucene	pmd	ps	antlr
%churn at level 1 reuse sites	81%	99.98	36%	87.2%	50%	84%
%churn at level 2 reuse sites	18%	0.02	64%	12.4%	50%	16%
Distribution of reuse levels for collections	SPECjbb	xylem	lucene	pmd	ps	antlr
%churn at level 1 reuse sites	89%	99.98%	25%	42.8%	100%	99.3%
%churn at level 2 reuse sites	11%	0.02%	75%	57.2%	0%	0.7%

Table 3. Reuse site object churn statistics

is built on top of a basic slicer which we have previously run on programs that are larger than 450,000 lines of code and the preliminary analysis took less than 10 minutes as reported in [5].

The additional analysis that we apply does an all-path exploration to determine the reusability of an object. This is clearly an exponential algorithm. pmd, at a little over 126K bytecode instructions analyzed, took 8 minutes and 15 seconds to analyze, of which the preliminary analysis took 3 minutes and 41 seconds. Here again, we use standard engineering tactics to contain the exponential state space exploration. If, for a given object, the analysis takes too long (currently curtailed at 30 seconds), we abort analysis of the object and conservatively mark it as not reusable.

5.2 Reuse site object churn statistics (dynamic analysis)

Table 3 captures some of the statistics gathered during the dynamic analysis phase based on simple allocation profiling to identify reuse sites that generate more temporary objects.

We observe that in many cases, a few potentially reusable sites cause a perceptible amount of object churn, particularly in SPECjbb2005, lucene and xylem. The results also reflect the importance of being able to handle reuse sites which are safely reusable along some paths but not others. In some benchmarks, e.g. lusearch, the sites that are the top contributors to temporary objects bloat are of this nature.

5.3 Performance impact statistics

Object reuse conversion was applied only to the reuse sites that are indicated by dynamic analysis to have a major contribution to object churn. The performance comparisons between the original and converted application are presented in Table 4.

In general, the performance impact of reducing object allocations depends on the workload, choice of JVM used and both JVM and system parameters. For example, the JVM heap size, the garbage collection algorithm, system memory bandwidth characteristics (esp. on multi-core systems [6]) and workload specific tuning can affect results of comparisons. However, in our evaluation we focus on the effectiveness of our technique rather than characterization of the degree of performance improvement expected from reducing object churn under different conditions. Hence we directly use default configurations instead of explicitly varying/tuning JVM and system parameters.

System Configuration Our performance measurements were taken on a dual core Intel(R) Core(TM)2 Duo T7500, 2.2 GHz with 2GB RAM running Linux, Java HotSpot(TM) Server VM (build 14.3-b01, mixed mode). For the SPECjbb2005 measurements, we used an 8-core Intel server (Intel(R) Xeon(R) X5460, 3.16 GHz) with 16GB RAM, running Linux, Java HotSpot(TM) Server VM (build 1.6.0-b105, mixed mode).

JVM settings For Xylem, we used a heap size of 1.6GB. We used out-of-the-box configuration parameters for the other benchmarks. In the case of SPECjbb2005, the heap size specified in the default benchmark properties file was 256MB. For the DaCapo benchmarks, the default heap size was as determined by the JVM. In all cases, the default garbage collection policy was determined by the specified JVM.

Since the execution time impact of reducing object creation can be highly dependent on the JVM and system parameters, we also measure other metrics like the percentage reduction in bytes of temporary objects used estimated from garbage collection statistics and relative scaling with larger input sizes. This enables us to evaluate whether our transformation is efficient enough to exploit potential for performance gains where opportunities exist.

We observe 20-40% reduction in object churn with our transformation. The execution time improvements range between 6-20%.

In SPECjbb2005, a single heavy allocation site dominates the reuse counts. Despite the fact that this is a string object and there are overheads due to reflection and accessing thread local context, we see significant benefits from object reuse automation. These improvements appear to be consistent with those reported for a manual implementation of object reuse by researchers of [6]; their results were for a well-tuned setup (large heap, GC tuning)[9].

[9] [6] also reports results of experiments conducted across a whole range of JVM settings (heap size, GC policies) to show that performance degradation from excessive object allocation in this case is not a mere artifact of GC algorithm or JVM parameters.

SMALL INPUT SIZE	SPECjbb	xylem	lucene
No. of objects reused	24/txn	144851	232881
No. of element allocations reused	1920/txn	-	125750
% Reduction in temporary objects generated	41	22	24
% improvement in execution time or throughput	7.9	16.4	6.6
LARGE INPUT SIZE	SPECjbb	xylem	lucene
No. of objects reused	21/txn	342651	448787
No. of element allocations reused	1723/txn	-	250739
% Reduction in temporary objects generated	41	27	24
% improvement in execution time or throughput	21.6	19.9	6.2

Table 4. Performance impact statistics: The percentages are baselined against the correponding results for the original benchmark without object reuse conversion; for example 100% reduction in temporary objects generated would mean that all object allocations were eliminated, 100% improvement in throughput would mean that throughput doubled.

We note that execution time improvements do not uniformly reflect the percentage reduction in objects. As observed by previous researchers [7, 6, 8] the relationship between percentage reduction in objects and performance is complex and depends on many factors ranging from workload and program characteristics, object construction costs, JVM tuning and hardware/system characteristics.

6 Related work

Object churn analysis, impact and solutions Many compiler and runtime optimizations like escape analysis [9, 10, 11, 12], escape detection and improvements in memory management and garbage collection techniques [13] have been developed to reduce the overheads of allocating and reclaiming temporary objects. As part of their work on escape analysis, Blanchet [10] consider the problem of stack size limitations in using stack allocation for loop objects and implement a simple liveness check to enable reuse of stack allocated space in loops. Their solution however does not consider higher levels of reuse in nested loops. They also rely on the use of inlining in case the loop header and allocation site are not within the same method, which is not practical in framework based applications where the allocation may lie several levels deep in the call chain from the closest enclosing loop.

Shankar et al [7] found that even a sophisticated escape analysis implementation in a high performance production JVM typically eliminates less than 10% of allocations in component based applications. They experimented with the use of aggressive guided inlining of regions with high object churn to enable the JIT to detect more opportunities for stack allocation of objects. In contrast to their approach we use static analysis approach to perform source code transformations for object reuse, which enables us to detect additional opportunities without incurring a runtime overhead.

Performance understanding techniques have been proposed [14, 15] for guiding programmers in eliminating excess temporaries that cannot be automatically detected by runtime optimizers. For example Buytaert et al [15] identify locations where code refactoring can be applied to reduce object creations. While their goal is similar to ours, they do not propose automated transformations. Their detection scheme uses dynamic traces unlike our static analysis approach where the dynamic analysis phase is only used to estimate potential benefits from the conversion.

Other approaches that help reduce the impact of excess temporary objects include advancements in memory management techniques for ensuring faster reclamation or reuse of temporary objects, e.g taking better advantage of allocation phases in the application [16], or combining the benefits of explicit object release [17, 18, 8] with garbage collection or scoped batch reclamation. Our technique is complementary to these efforts as it avoids the creation of objects wherever possible.

Zhao et al [6] analysed the implications of object allocation on scalability and performance. They proposed the notion of an allocation wall that limits multi-core scalability programs that perform high volumes of temporary objects. For their experimentation they perform manual code modifications to implement a form of object pooling for objects that are allocated very frequently and showed significant benefits for SPECjbb and SPECjvm derby. In their paper they observe that the process of manually converting an application for object reuse is time consuming and hence impractical for application developers to use. Our work succeeds in efficiently automating such optimizations for collection objects and strings.

Analysis and measurement of software bloat Mitchell, Sevitsky and Srinivasan [19] define metrics based on modeling runtime information flow to classify and characterize the nature and volume of data transformations executed, though these measures have not been automated till date. The notion of data structure health signatures proposed by Mitchell and Sevitsky [20] has been used very effectively in characterization and automated measurement [21] of Java memory bloat in long lived heap objects. This is a relative measure of total memory bytes consumed by actual data vs associated representational memory overhead. For some categories of bloat, including the problem of temporary objects bloat which we address in this paper, an explicit model may not always be available for distinguishing overhead from necessary data or activity. Researchers have therefore used different measures of excesses like excessive volumes of temporary objects, data copies and heavy object creation costs to recognize the presence of bloat. For example, Xu et al use an instrumented JVM to summarize chains of runtime data copies [22] and an abstract thin dynamic slicing technique to identify data structures with high cost-benefit ratios [23]. Most approaches for detecting bloat have employed dynamic analysis. [24] applies a static analysis scheme to detect inefficient uses of container objects, particularly for underpopulated and overpopulated containers. All of these techniques are focused on aiding the process

of reducing bloat, however, they are intended for interpretation by experts, not as fully automated solutions to de-bloat software like ours.

Dufour et al [14] apply blended static and dynamic analysis techniques to runtime traces for characterizing the usage of temporaries. Their results show that a significant number of temporary objects may be used several call levels away from their allocation site, which makes them particularly difficult to optimize. This motivates the need for techniques like ours.

7 Conclusion and future work

We presented an analysis technique to automatically detect and convert opportunities for object reuse in Java programs where there is significant potential for benefit from reuse. This is a challenging problem because an object may be reusable in loops that may be several levels above it in the callgraph. Further, as our empirical results show, very often objects may be reusable only along certain paths and not others. In this situation a conservative analysis can miss most opportunities for reuse. We are able to improve precision in such situations by checking whether the conditions required for the correctness of our runtime transformation are met in the event these paths share the same loop header. Our results show that this solution can detect such opportunities in real large programs and reduce the generation of temporary objects significantly.

Further improvements in scalability and precision of our solution can be attained by incorporating feedback from our dynamic analysis to focus static analysis on the allocations sites that are likely to yield most benefits. Other future work includes extending the applicability of our automated transformation to other types of objects and using a combination of byte code manipulation and JVM level optimizations to improve the performance of the transformed code.

Acknowledgments

We thank Gary Sevitsky, Matt Arnold, Kazuaki Ishigaki, Dibyendu Das, Vijay Mann and Prasanna Kalle for their help, particularly their contribution to discussions on the problem of Java temporary objects bloat that motivated this work. We also thank Rupesh Nasre, and our anonymous reviewers for their excellent feedback on the paper.

References

[1] Mitchell, N., Schonberg, E., Sevitsky, G.: Four trends leading to java runtime bloat. IEEE Software **27**(1) (2010) 56–63

[2] Xu, G.e.a.: Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In: Future of Software Engineering Research '10. (2010)

[3] Shirazi, J.: Java performance tuning, O'Reilly (2003)

[4] Blackburn, S.M.e.a.: The DaCapo benchmarks: Java benchmarking development and analysis. (In: Object-Oriented Programming, Systems, Languages and Applications (OOPSLA))

[5] Nanda, M.G., Sinha, S.: Accurate interprocedural null-dereference analysis for Java. In: Proc. of the 31st Intl. Conf. on Softw. Eng. (2009) 133–143

[6] Zhao, Y., Shi, J., Zheng, K., Wang, H., Lin, H., Shao, L.: Allocation wall: a limiting factor of java applications on emerging multi-core platforms. In: Object-Oriented Programming, Systems, Languages and Applications (OOPSLA). (2009)

[7] Shankar, A., Arnold, M., Bodik, R.: Jolt: lightweight dynamic analysis and removal of object churn. In: Object-Oriented Programming, Systems, Languages and Applications (OOPSLA). (2008)

[8] Guyer, S., McKinley, K., Frampton, D.: Free-me: A static analysis for automatic individual object reclamation. In: Programming Language Design and Implementation (PLDI). (2006)

[9] Choi, J.D., Gupta, M., Serrano, M., Sreedhar, V.C., Midkiff, S.: Escape analysis for java. SIGPLAN Not. **34**(10) (1999) 1–19

[10] Blanchet, B.: Escape analysis for object-oriented languages: application to java. SIGPLAN Not. **34**(10) (1999) 20–34

[11] Gay, D., Steensgaard, B.: Fast escape analysis and stack allocation for object-based programs. In: International Conference on Compiler Construction. (2000)

[12] Whaley, J., Rinard, M.: Compositional pointer and escape analysis for java programs. SIGPLAN Not. **34**(10) (1999) 187–206

[13] Bacon, D.F., Cheng, P., Rajan, V.T.: A unified theory of garbage collection. SIGPLAN Not. **39**(10) (2004) 50–68

[14] Dufour, B., Ryder, B.G., Sevitsky, G.: A scalable technique for characterizing the usage of temporaries in framework-intensive java applications. In: SIGSOFT '08/FSE-16. (2008) 59–70

[15] Buytaert, D., Beyls, K., De Bosschere, K.: Hinting refactorings to reduce object creation in java. In: ACES. (2005) 73–76

[16] Xian, F., Srisa-an, W., Jiang, H.: Microphase: an approach to proactively invoking garbage collection for improved performance. In: Object-Oriented Programming, Systems, Languages and Applications (OOPSLA). (2007)

[17] Cherem, S., Rugina, R.: Uniqueness inference for compile-time object deallocation. In: ISMM. (2007)

[18] Inoue, H., Komatsu, H., Nakatani, T.: A study of memory management for web-based applications on multicore processors. In: Programming Language Design and Implementation (PLDI). (2009)

[19] Mitchell, N., Sevitsky, G., Srinivasan, H.: Modeling runtime behaviour in framework based applications. In: European Conference on Object-Oriented Programming (ECOOP). (2006)

[20] Mitchell, N., Sevitsky, G.: The causes of bloat, the limits of health. In: Object-Oriented Programming, Systems, Languages and Applications (OOPSLA). (2007)

[21] Mitchell, N., Schonberg, E., Sevitsky, G.: Making sense of large heaps. In: European Conference on Object-Oriented Programming (ECOOP). (2009)

[22] Xu, G., Arnold, M., Mitchell, N., Rountev, A., Sevitsky, G.: Go with the flow: profiling copies to find runtime bloat. In: Programming Language Design and Implementation (PLDI). (2010)

[23] Xu, G.e.a.: Finding low-utility data structures. In: Programming Language Design and Implementation (PLDI). (2010)

[24] Xu, G.e.a.: Detecting inefficiently-used containers to avoid bloat. In: Programming Language Design and Implementation (PLDI). (2010)