Go with the Flow: Profiling Copies To Find Runtime Bloat

Guoqing Xu
Ohio State University
xug@cse.ohio-state.edu

Matthew Arnold
IBM T.J. Watson Research Center
marnold@us.ibm.com

Nick Mitchell
IBM T.J. Watson Research Center
nickm@us.ibm.com

Atanas Rountev
Ohio State University
rountev@cse.ohio-state.edu

Gary Sevitsky
IBM T.J. Watson Research Center
sevitsky@us.ibm.com

Abstract

Many large-scale Java applications suffer from runtime bloat. They execute large volumes of methods, and create many temporary objects, all to execute relatively simple operations. There are large opportunities for performance optimizations in these applications, but most are being missed by existing optimization and tooling technology. While JIT optimizations struggle for a few percent, performance experts analyze deployed applications and regularly find gains of 2× or more.

Finding such big gains is difficult, for both humans and compilers, because of the diffuse nature of runtime bloat. Time is spread thinly across calling contexts, making it difficult to judge how to improve performance. Bloat results from a pile-up of seemingly harmless decisions. Each adds temporary objects and method calls, and often copies values between those temporary objects. While data copies are not the entirety of bloat, we have observed that they are excellent indicators of regions of excessive activity. By optimizing copies, one is likely to remove the objects that carry copied values, and the method calls that allocate and populate them.

We introduce *copy profiling*, a technique that summarizes runtime activity in terms of chains of data copies. A flat copy profile counts copies by method. We show how flat profiles alone can be helpful. In many cases, diagnosing a problem requires data flow context. Tracking and making sense of raw copy chains does not scale, so we introduce a summarizing abstraction called the *copy graph*. We implement three clients analyses that, using the copy graph, expose common patterns of bloat, such as finding hot copy chains and discovering temporary data structures. We demonstrate, with examples from a large-scale commercial application and several benchmarks, that copy profiling can be used by a programmer to quickly find opportunities for large performance gains.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Debugging aids; D.3.4 [*Programming Languages*]: Processors—Memory management, optimization, run-time environments

General Terms Languages, Measurement, Performance

Keywords Memory bloat, profiling, heap analysis, copy graph

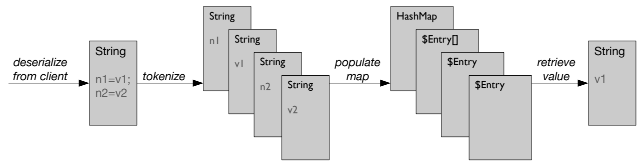
1. Introduction

As a community, we have adopted guiding principles — code quickly, favor reuse and dynamicity, integrate legacy functionality rather than rewrite it — under the assumption that the compiler and garbage collector will take care of the resulting runtime mess. This situation is especially common in large-scale Java applications developed using many layers of custom and third party frameworks. After initial tuning has found the low-hanging fruit, these applications still consume excessive resources for what they accomplish.

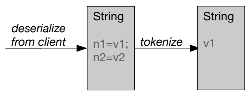
Java applications regularly suffer from systemic runtime bloat [18, 17]. Bloat consists of operations that, while not strictly necessary for forward progress, are executed nonetheless. For example, we have worked with a commercial document management server, deeply diving into its inefficiencies. We found that, to perform the seemingly simple task of inserting a single small document in the database, this application invokes 25,000 methods and creates 3000 temporary objects. This is after the Just In Time (JIT) compiler's best efforts. With less than one person-week of manual tuning, work that only scratched the surface, a performance expert was able to reduce the object creation rate by 66%. Vast improvements are possible, if only tuning were easier, or more automated.

Consider a specific example where the server extracts name-value pairs from a cookie that the client transmits in a serialized, string form. The methods that use these name-value pairs expect Java objects, not strings. They invoke a library method to decode the cookie string into a Java `HashMap`, yet another transient form of this very simple data. In the common case, the caller extracts one or two elements from the 8-element map, and never uses that map again. Figure 1 illustrates the steps necessary to decode a cookie in this application. Decoding a single cookie, an operation that occurs repeatedly, costs 1000 method invocations and 35 temporary objects, after JIT optimizations. A hand-optimized specialization for the common case that only requires one name-value pair invokes 4 invocations and constructs 2 temporary objects.

Today's JITs have sophisticated optimizers that offer important performance improvements, but they are often unable to remove the penalty of systemic bloat. One problem is that the code in large applications is relatively free of hot spots. Table 1 shows a breakdown of the top ten methods from the document management server. This application executes over 60,000 methods, with no single method contributing more than 3.19% to total application time, and only 14 methods contributing more than 1%. JITs are faced with a plurality of important methods. The burden, with



(a) Original version.



(b) Specialized version.

Figure 1. The steps a commercial document management server uses to decode a cookie; the original version tokenizes and returns the entire map, even if the caller needs only one name-value pair.

method	*CPU time*
`HashMap.get`	3.19%
`Id.isId`	2.84%
`String.regionMatches`	2.12%
`CharToByteUTF8.convert`	2.04%
`String.hashCode`	1.77%
`String.charAt`	1.70%
`SimpleCharStream.<init>`	1.65%
`ThreadLocalMap.get`	1.32%
`String.toUpperCase`	1.30%

Table 1. In a commercial document management server, there is no hot method that can be optimized for an easy big gain.

current JIT technology, is on the method inliner to bundle together code into larger, hopefully optimizable, regions.

Forming perfect code regions, and then optimizing them, is an immensely challenging problem [24]. Optimizations that can be easily performed by a programmer, such as a moving a call to a side-effect-free method out of a loop, can require heroic JIT effort to achieve the same effect. That call may ultimately perform thousands of method invocations with call stacks hundreds deep, and allocate many objects. Automatically performing such a transformation requires dozens of powerful analyses to work together flawlessly; a single missed opportunity can render the call immovable. Add in language features that restrict optimization, such as precise exceptions, and there is little hope for a fully automated solution.

Our work advocates a new approach that is not intended to replace JIT optimization, but to complement it. Through a combination of metrics and analyses focused on bloat, we hope to quickly guide developers to the problematic areas of the application, allowing them to refactor to avoid the problem. A small handful of performance experts are already capable of performing this task manually; our goal is to automate as much of it as possible, thus lowering the bar for tuning systemic bloat. The burden cannot remain solely on the shoulders of experts: the problems of excessive bloat will become increasingly painful as cores become simpler, bandwidth per core goes down, and we can't rely on clock speed increases to ameliorate ever-increasing levels of inefficiency.

Bellwethers of Bloat The inefficiencies at the heart of the cookie decoding example are common to many bloated implementations. In these implementations, there is often a chain of information flow that carries values from one storage location to another, often via temporary objects [18]; e.g. as visualized in Figure 1. Bloat of this form manifests itself in a number of ways: temporary structures to carry values, and a sea of method invocations that allocate and initialize these structures, and copy data between them.

In our experience, it is this data copying activity that is an excellent bellwether of bloat. This is not to say that one must only tune copying activity to lessen the burden of good software engineering. Rather, by tuning in a way that reduces the need for copying, one also reduces the attendant object creation and initialization, and method invocation activity. The specialized cookie decoding process avoids not only most of the copies, but also the construction and population of the temporary `HashMap` data structure, and its many unused key and value `String`s.

In Section 2, we introduce a way of profiling information flows, rather than control flows. These profiles distinguish copies from other activities, such as arithmetic operations. We specifically use the volume of copies as a way to quantify bloat. We show how copies are not targeted by current production Java JIT compilers, despite being highly concentrated. This is in contrast to the lack of concentration in execution time: a small number of methods explain most of the copies, but not most of the time. Copies are concentrated, even in the more complex applications. For example, in the document management server, the top fifty consumers of time explain only 24% of total execution time; the top fifty copying methods explain 82% of copies.

Flat summaries that count copies are a good first step, but they are not sufficient to help programmers alleviate bloat. Copies, by their nature, span methods and classes in cross-cutting ways. The specialized cookie decoding process shown in Figure 1(b) is neither the result of tuning the `HashMap` `put` or `get` methods, nor of tuning the `HashMap` storage structure. To specialize this scenario requires understanding the *chains* of copies: which storage locations carry values, and which methods enact the copies. During program execution, there will be billions of copy chains. To combat this blowup, in Section 3, we introduce an abstraction, the *copy graph*, that concisely summarizes chains of copies.

We have implemented a dynamic information flow framework in the JIT compiler of the IBM J9 commercial JVM, which computes copy profiles and forms a copy graph as the program runs. Section 4 provides implementation details that were necessary to make it possible to track information flow in a way that scales to production applications. The copy graph itself consumes a small amount of memory, plus two words for every live object in the application. To facilitate our initial implementation, we store these two words in a side "shadow" heap, to avoid modifying the object header in the VM. The current implementation imposes a slowdown of about 37 times, on average, across the test programs. Although this overhead is significant, it has not limited our ability to collect data from large production applications. The focus of this work is on the results of the analysis, not how cheaply they can be collected.

In Section 5, we introduce three client analyses that use the copy graph to generate useful reports: hot copy chains, a clone detection analysis, and an approximation of escape analysis. In Section 6, we provide examples of using these client analyses to find real problems of bloat. For example, we very quickly found a performance defect in the DaCapo [4] bloat benchmark, and quickly implemented a fix that reduced object creation rate by 65%, and execution time by 30%. We also quickly found an issue in the DaCapo Eclipse benchmark, and implemented a fix that resulted in a 9.3% performance improvement to that benchmark.

The contributions of this paper are:

- A methodology, *copy profiling*, that identifies high-overhead activity in terms of copies and chains of copies.
- A runtime framework, implemented in the JIT compiler of the J9 VM, that generates profiles of information flow, and also tracks the details of those flows during program execution.

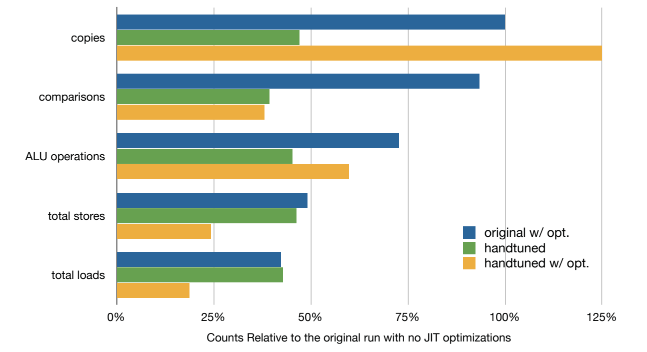


Figure 2. A breakdown of activity in a document processing server application. The baseline, at 100%, is the original code run with JIT optimizations disabled; we compare this to the original code with JIT optimizations enabled, and to an implementation with a dozen hand-tunings. The JIT optimizer does not tune the number of copies or comparison instructions, and in some cases makes things worse.

- The *copy graph*, an abstraction of chains of copies.
- Three client analyses of the copy graph: one that identifies hot copy chains, a second that finds pairs of allocation contexts whose allocated data structures are deep clones of each other, and a third that finds temporary data structures by identifying the tops of structures that do not flow through the heap.

2. Copy Profiling

Bloat often stems from excessive work done along data flows. In this section, we introduce the notion of profiling operations along these flows, with a focus on copies. A copy is a load-store pair that transfers a value, unmodified, from one storage location to another. A flat *copy profile* is the analog of execution time profile, except that it counts copies rather than time.

The copy profiles, and the rest of the reports described in this paper, are in terms of heap locations and the methods that copy between them. In the profiles, a copy operation is associated with the method that performed the write to the heap. Though it is necessary to track through stack locations (as described in Section 4), in order to determine whether a store is the second half of a copy, the reports do not include that level of detail. Since stack variables will likely be assigned to registers, chains of copies between stack locations will usually involve only register transfer operations. They are also more likely to be optimized by conventional dataflow analysis.

A simple count of the number of copies is a useful way to gauge the goodness of an implementation, and the effectiveness of the JIT at tackling certain classes of bloat. Figure 2 shows a comparison of four scenarios of the document management server. The baseline, at 100%, represents the performance of the original code with JIT optimizations disabled, in terms of the number of copy operations that were performed during a 10 minute load run. We compare this baseline to the original code with JIT optimizations enabled, and to a version of the code that had been hand-tuned to improve overall performance (both with and without optimizations). We also show the number of comparison operations, the number of ALU operations, and the total number of loads and stores. Observe that the JIT is good at what you'd expect: reducing ALU operations, and the total number of loads and stores; common subexpression elimination probably explains much of these effects. On the other hand, the JIT does not greatly affect the number of copies; it also has no great affect on the number of comparison instructions. Comparisons are often a sign of over-protective or over-general implementations. These coding practices also lead to bloat. The hand-tuned implementation greatly lowers both the number of copies and the number of comparison operations.

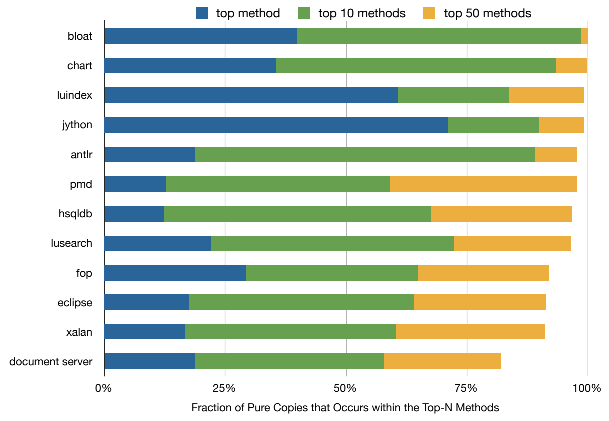


Figure 3. Copy concentration: a small number of methods explain most of the copies in the DaCapo benchmarks, version 2006-10-MR2, and the document management server.

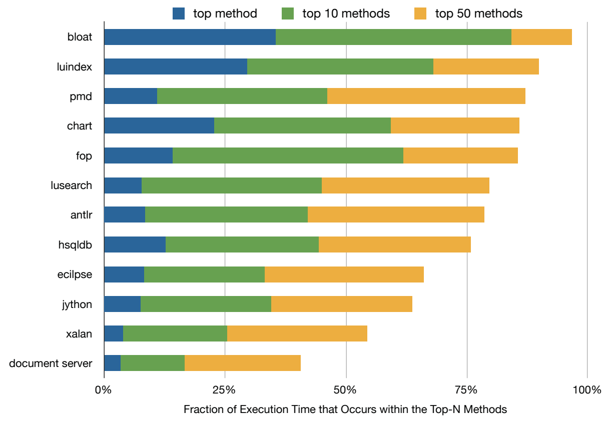


Figure 4. Time concentration: in contrast to copies, which are concentrated even for complex applications, the time spent in methods is only concentrated for the simpler benchmarks.

Flat copy profiles show that copies serve as good indicators of problems. From them, we learn that copy activity is concentrated in a small number of methods. From the copy profiles for the DaCapo benchmark suite[1] [4] and the document management server, we observe the concentration of copies. Figure 3 shows that, across the board, a small number of methods explain most of the copy activity in these programs. Even just the top method explains at least 12% of the copies, often much more. For comparison, Figure 4 shows the concentration of execution time in methods. As expected, the more complex applications, such as the Eclipse DaCapo benchmark and the document management server, have very flat execution time method profiles; this is in contrast to the highly concentrated copy profiles for those same programs.

3. Profiling Copy Chains

Individual copies are usually part of longer copy chains. Optimizing for bloat requires understanding the chains as a whole, as they

[1] We used DaCapo version 2006-10-MR2.

```
1 class List{
2    Object[] elems; int count;
3    List(){ elems = new Object[1000]; }
4    List(List l){ this(); // call default constructor
5       for(Iterator it = l.iterator(); it.hasNext();)
6          { add(it.next()); } }
7    void add(Object m){
8       Object[] t = this.elems;
9       t[count++] = m;
10   }
11   Object get(int ind){
12      Object[] t = this.elems;
13      Object p = t[ind]; return p;
14   }
15   Iterator iterator(){
16      return new ListIterator(this);
17   }
18 }
19 class ListIterator{
20    int pos = 0; List list;
21    ListIterator(List l){
22       this.list = l;
23    }
24    boolean hasNext(){ return pos < list.count - 1;}
25    Object next(){ return list.get(pos ++);}
26 }
27 class ListClient{
28    List myList;
29    ListClient(List l){ myList = l; }
30    ListClient slowClone(){
31       List j = new List(myList);
32       return new ListClient(j);
33    }
34    ListClient fastClone(){
35       return new ListClient(myList);
36    }
37 }
38 static void main(String[] args){
39    List data1 = new List();
40    for(int i = 0; i < 1000; i++)data1.add(new Integer(i));
41    List data2 = new List();
42    for(int i = 0; i<5; i++){data2.add(new String(args[i]));
43       System.out.println(data2.get(i));}
44    ListClient c1 = new ListClient(data1);
45    ListClient c2 = new ListClient(data2);
46    ListClient new_c1 = c1.slowClone();
47    ListClient new_c2 = c2.fastClone();
48 }
```

Figure 5. Running example.

may span large code regions that need to be examined and transformed. We now show how to form an abstraction, the copy graph, that can be used to identify chains of copies.

DEFINITION 1. *A* copy chain *is a sequence of copies that carry a value through two or more heap storage locations. Each copy chain node is a heap location. Each edge represents a sequence of copies that transfers a value from one heap location to another, abstracting away the intermediate copies via stack locations, parameter passing, and value returns.*

The heap locations of interest are fields of objects and elements of arrays. A copy chain ends if the value it carries is the operand of a computation, which produces a new value, or is an argument to a native method. It is important to note that, in a copy chain, each maximal-length subsequence of stack copies is abstracted by a single edge directly connecting two heap locations.

3.1 A Motivating Example

The code in Figure 5 is used for illustration throughout the paper. The example is based on a common usage scenario of Java collections. A simple implementation of a data structure `List` is used by a client `ListClient`. `ListClient` declares two clone methods `fastClone` and `slowClone`, which return a new `ListClient` object by reusing the old backing list and by copying list elements, respectively. The entry method `main` creates two lists `data1` and `data2` and initializes them with 1000 Integer and 5 String objects (lines 40 and 42). The two lists are then passed into two `ListClient` objects and eventually two new `ListClient` objects are created by calling `slowClone` and `fastClone`. For simplicity, the approach is described at the level of Java source code, although our implementation works with a lower-level virtual machine intermediate representation (IR).

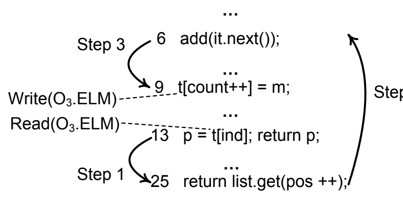


Figure 6. A copy chain due to `ListClient.slowClone`. Line numbers 6, 9, 13, and 25 correspond to the code in Figure 5.

Figure 6 depicts the steps in the creation of a single-edge copy chain. This chain results from the invocation of `slowClone` (line 46) which copies Integer object references from the array referenced by field *elems* of one `List` to the array referenced by field *elems* of another `List`. The source array and the target array will be denoted by O_3 since they are created at line 3 in the code. (For now, the reader can ignore the naming scheme; it will be discussed shortly.) The copy chain in Figure 6 is $O_3.ELM \rightarrow O_3.ELM$, where ELM represents any array element.

To represent the source and the sink of the data propagated along a copy chain, we can augment the chain with two nodes: a *producer* node added at the beginning, and a *consumer* node added at the end. The producer node can be a constant value, a `new X` expression, or a computation operation representing the creation of a new value. The consumer node has only one instance (denoted by C) in the copy graph, showing that the data goes to a computation operation or to a native method. These two types of nodes are not heap locations, and are added solely for the purpose of subsequent client analyses. Note that not every chain has these two special nodes. For the producer node, we are interested only in reference-typed values because they are important for further analysis and program understanding. Thus, chains that propagate values of primitive types do not have producer nodes. Not every piece of data goes to a consumer and therefore not every chain has a consumer node. The absence of a consumer is a strong symptom of bloat and can be used to identify performance problems. An example of a full augmented copy chain starting from producer O_{42} (i.e., `new String`) is $O_{42} \rightarrow O_3.ELM \rightarrow C$. This chain ends in consumer node C because the data goes into method `println` which eventually calls native method `write`.

Profiling copy chains can be extremely space expensive, because it requires maintaining a distinct node for each heap location on each copy chain, regardless of whether chains have shared heap locations. In addition, for each heap location, it is necessary to maintain the history information regarding all chains that go through this location, which may incur significant running time overhead. To make the analysis scale to large applications, we apply a series of abstractions on copy chains. These abstractions are also essential for producing summarized reports that do not overwhelm the tool user with millions of chains. The first abstraction is to merge all copy chains in a *copy graph*, so that nodes shared among chains do not need to be maintained separately. In addition, the copy graph construction algorithm can be designed to profile only graph edges (i.e., one-hop heap copy), which is much more efficient than profiling of entire chains.

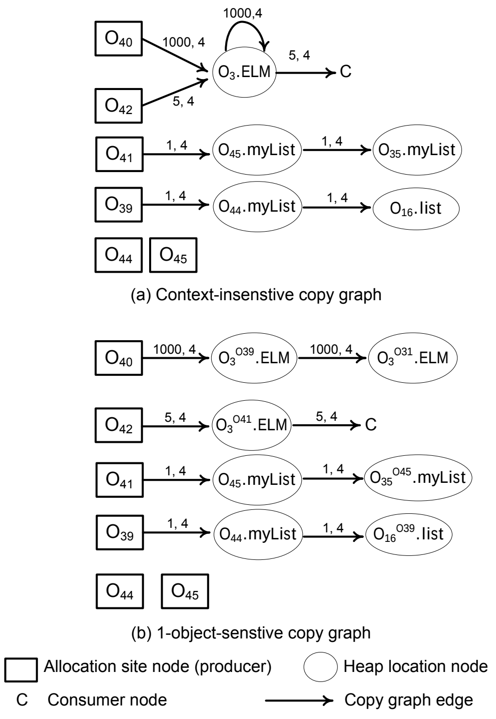


Figure 7. Partial copy graph with context-insensitive and context-sensitive object naming.

DEFINITION 2. *A copy graph $G = (\mathcal{N}, \mathcal{E})$ has node set $\mathcal{N} \subseteq \mathcal{AL} \cup \mathcal{IF} \cup \mathcal{SF} \cup \{C\}$. Here $\mathcal{AL}$ is the domain of allocation sites O_i which serve as producer nodes and do not have any incoming edges. $\mathcal{IF}$ is the domain of instance field nodes $O_i.f$. $\mathcal{SF}$ is the domain of static field nodes. C is the consumer node; it has only incoming edges. The edge set is $\mathcal{E} \subseteq \mathcal{N} \times Integer \times Integer \times \mathcal{N}$. Each edge is annotated with two integer values: the frequency of the heap copy and the number of copied bytes (i.e., 1, 2, 4, or 8).*

There could be many different ways to map the run-time execution to these abstractions. The rest of this section describes the mapping used in our current work; future work could explore other choices with varying cost, precision, and usefulness for tool users.

Object naming scheme. Following an abstraction technique widely adopted in static analysis, an allocation site is used to represent the set of run-time instances that it creates. Similarly, all heap locations that an instance field dereference expression $a.f$ represents are projected to a set of nodes $\{O_i.f\}$ such that the objects that a points to are projected to set $\{O_i\}$. Applying this abstraction reduces the number of allocation site nodes $\mathcal{AL}$ and instance field nodes $\mathcal{IF}$. Each element of an array a is represented by a special field node $O_a.ELM$, where O_a denotes the allocation site of a and ELM represents the field name. Individual array elements are not distinguished: considering each element separately may introduce infeasible time and space overhead.

For illustration, consider the partial copy graph in Figure 7(a). The figure shows only paths starting from nodes in method `main` in the running example. An allocation site is named O_i, where i is the number of the code line containing the site. Each copy graph edge is annotated with two numbers: its frequency and the number of bytes it copies. For example, edge $O_{40} \xrightarrow{1000,4} O_3.ELM$ copies the `Integer` objects created at line 40 into the array referenced by `data1`'s `elems` field. This edge consists of a sequence of copies via parameter passing (line 40 and line 9). This sequence of copies occurs 1000 times, and each time 4 bytes of data are transferred. Both $O_{40} \xrightarrow{1000,4} O_3.ELM$ and $O_3.ELM \xrightarrow{1000,4} O_3.ELM$ are hot edges: their frequencies and the total number of bytes copied are much larger than those of other edges. When there exists a performance problem in the program, a better design might be needed to eliminate these copies.

It is important to note again that nodes that represent different objects may be merged due to the employed abstraction. For example, although variable `t` at line 9 points to different objects at run time, the array element node `t[count++]` is represented by a single node $O_3.ELM$, regardless of the `List` object that owns the array. Consider the self-pointing edge $\xrightarrow{1000,4}$ at node $O_3.ELM$. The edge captures the data flow illustrated in Figure 6. This sequence of copies moves object references from the array pointed-to by $O_{39}.elems$ to the array pointed-to by $O_{31}.elems$. Since both arrays are represented by O_3, their elements are merged into $O_3.ELM$ in the copy graph and this self-pointing edge is generated.

Merging of nodes could lead to spurious copy chains that are inferred from the copy graph. For example, from Figure 7(a), one could imprecisely conclude that both O_{40} and O_{42} will eventually be consumed, because both edges $\xrightarrow{1000,4}$ and $\xrightarrow{5,4}$ can lead to consumer node C. The cause of the problem is the *context-insensitive* object naming scheme, which maps each run-time object to its allocation site, regardless of the larger data structure in which the object appears. In order to model copy chains more precisely, we introduce a context-sensitive object naming scheme.

3.2 Context Sensitivity

When naming a run-time object, a context-sensitive copy graph construction algorithm takes into account both the allocation site and the calling context of the method in which the object is allocated. Existing static analysis work proposes two major types of context sensitivity for object-oriented programs: call-chain-based context sensitivity (i.e., k-CFA) [25], which considers a sequence of call sites invoking the analyzed method, and object-sensitivity [16], in which the context is the sequence of static abstractions of the objects (i.e., allocation sites) that are run-time receivers of methods preceding the analyzed method on the call stack. Of particular interest for our work is the object-sensitive naming scheme because, to a large degree, it reflects object ownership and is suitable for improving the analysis precision for real-world applications making use of a large number of object-oriented data structures.

Figure 7(b) shows the 1-object-sensitive version of the copy graph, in which an object is named using its allocation site together with the allocation site of the receiver object of the method in which the object is created. For objects created in a constructor, the context is usually their run-time owner. By adding context sensitivity, paths that start from O_{40} and O_{42} do not share any nodes. Note that there are no contexts for nodes $O_{39}, \ldots, O_{45}$ because they are created in static method `main` which does not have a receiver object. Although longer context strings may increase precision, our tool limits the length of the context to 1 since it could be prohibitively expensive (both in time and space) to employ longer contexts in a dynamic analysis.

4. Runtime Information Flow Tracking

This section presents the details of the copy profiling technique. We modified J9, a production virtual machine developed by IBM, to support dynamic information flow profiling: all memory locations in the program have a corresponding *shadow* location (c.f. [19]). This allows a dynamic analysis to tag all application data with dataflow metadata information, which we refer to as *tracking data*.

As the program executes and application data is read or written, the information flow analysis updates the corresponding tracking data. Although we focus on profiling of copies in this paper, various other client analyses can be implemented in this framework by redefining the shadow data initialization and flow transfer functions.

4.1 Shadow Locations

Our information flow infrastructure supports shadowing of all memory in the application, including local variables, static fields, arrays, and object fields. Local variables are shadowed simply by introducing an extra location on the stack. Tracking data is also passed interprocedurally through parameters and return values. A *tracking stack* is maintained for passing shadow variables for parameters, as well as return values.

Shadowing of object fields is supported by use of a *shadow heap* [19]. The shadow heap is a contiguous region of memory equal in size to the Java heap. Scratch space for every byte of data in the Java heap can be referenced quickly by adding a constant offset to the address location. A non-moving garbage collector is used so the address of objects does not change during the execution. A moving collector could be used as long as it is modified to move the corresponding shadow data when moving an object.

Doubling size of the heap is a significant space overhead, but is not a limitation in practice, even for large applications. With a 1-gigabyte Java heap and a 1-gigabyte shadow heap, we were able to successfully run all programs we encountered, including large production web server applications.

4.2 Copy Graph Construction

The copy graph construction algorithm consists of two main components: (1) "compile time" instrumentation, which occurs at run time during JIT compilation, and (2) run-time profiling. To avoid having to modify both the interpreter and the JIT, we run the VM in a JIT-only mode such that all methods in the program are compiled by the JIT prior to their first invocation, allowing the tool to track data flow throughout the entire program.

Copy graph construction requires the ability to tag an object with its allocation site, so the allocation site information can be efficiently looked up at run time. To perform this lookup quickly we rely on the shadow heap. When an object is allocated, we store its allocation site ID and its context allocation ID in its shadow location, so it can be referenced through operation $*(objAddr + distance)$. This provides the ability to quickly store and retrieve the allocation site for every object.

4.3 Data Structure Design

The data structure design for the copy graph is important for minimizing overhead. The goal of the design is to allow efficient mapping from a run-time heap location to its name (which in our analysis is a copy graph node address). Figure 8 shows an overview of the data structures for the copy graph. Static field nodes are stored in a singly-linked-list that is constructed at instrumentation time. The node address is hard-coded in the generated executable code, so that the retrieval of nodes does not contribute to running time (thus, the analysis does not need to use the shadow locations for static fields). Each node has an edge pointer, which points to a linked list of copy graph edges that leave this node. Edge adding occurs at run time. If an existing edge is found for a pair of a source node and a target node, a new edge is not added. Instead, the frequency field of the existing edge is incremented. The size field (i.e., number of bytes) can be determined at compile time by inspecting the type of data that the copy transfers.

Allocation site nodes and instance field nodes are implemented using arrays to allow fast access. For each allocation site, a unique integer ID is generated at compile time (the IDs start from 0). The

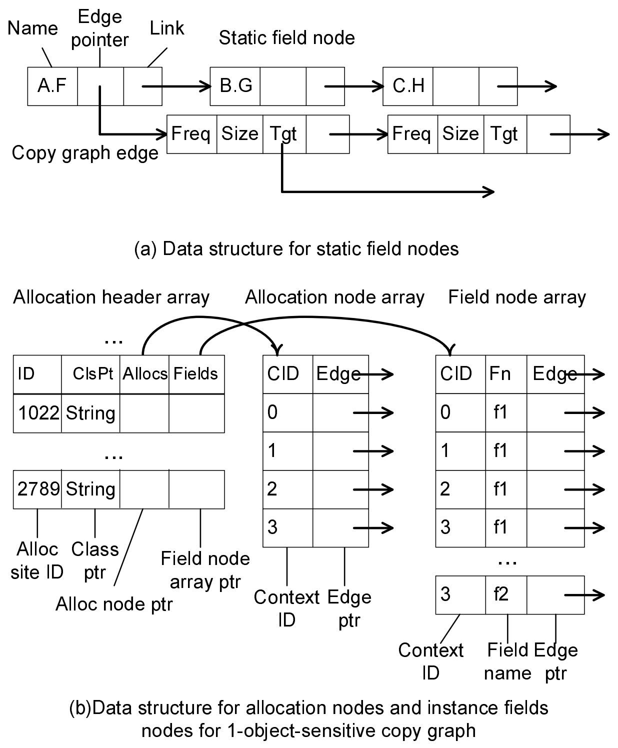


Figure 8. Data structure overview.

ID is used as the index into an array of *allocation headers*. Each allocation header corresponds to one ID, and points to an array of *allocation nodes* and to an array of *field nodes*, both specific to this ID. For a context-insensitive copy graph, the allocation node array for the ID has only one element. For the context-sensitive copy graph that requires a unique allocation node for each calling context (i.e., the allocation site ID of the receiver object of the surrounding method), each element of the allocation node array corresponds to a different calling context. In the current implementation the array does not grow dynamically, thus the number of calling contexts for each allocation site is limited to a pre-defined value c. We have experimented with different values of c and these results are reported in Section 7. An encoding function maps an allocation site ID representing a context to a value in $[0, c-1]$; currently, we use a simple mod operation $contextAllocId \,\%\, c$ to encode contexts. As reported in Section 7, very few contexts for an object have conflicts (i.e., they map to the same value) when using this function. A default c value of 4 was used for the studies described in Section 6.

The field node array is created similarly. The order of different fields in the array is dependent on the offsets of these fields in the class. We build a class metadata table at the time the class is resolved by the JIT. The table sorts fields based on their offsets, and maps each field to a unique ID (starting from 0) indicating its order in the field node array. For each instance field declared in the type (and all its supertypes) instantiated at the allocation site, there are 1 (i.e., for context-insensitive naming) or c (i.e., for 1-object-sensitive naming) entries in the field node array. For example, consider an instance field dereference $a.f$ for which the allocation site ID of the object pointed-to by a is 1000, the corresponding context allocation ID is 245, the offset of f is 12, and this offset (at compile time) is mapped to field ID $i = class\_metadata[12]$. The corresponding copy graph node address can be obtained from the element with index $c * i + 245 \,\%\, c$ in the array pointed-to by column $Fields$ of $alloc\_headers[1000]$.

[1. local=alloc]

$$\frac{\begin{array}{l} SH(i) = (AllocID(new\ O), SH(this)\&0xFFFFFFFF) \\ CG' = CG \cup CreateAllocHeaderEntry(O, AllocID(new\ O)) \\ \mathbb{E} \vdash SH(i) : addr_{rhs} \\ shadow_i = addr_{rhs} \end{array}}{CG \Rightarrow^{i=new\ O} CG'}$$

[2. local=static]

$$\frac{\begin{array}{l} \mathbb{E} \vdash F : addr_{rhs} \\ CG' = CG \\ shadow_i = addr_{rhs} \end{array}}{CG \Rightarrow^{i=F} CG'}$$

[3. local=instance field dereference]

$$\frac{\begin{array}{l} \mathbb{E} \vdash (SH(a), Offset(f)) : addr_{rhs} \\ CG' = CG \\ shadow_i = addr_{rhs} \end{array}}{CG \Rightarrow^{i=a.f} CG'}$$

[4. local=local]

$$\frac{\begin{array}{l} CG' = CG \\ shadow_i = shadow_j \end{array}}{CG \Rightarrow^{i=j} CG'}$$

[5. static=local]

$$\frac{\begin{array}{l} \mathbb{E} \vdash F : addr_{lhs} \\ CG' = CG \cup CreateEdge(shadow_i, addr_{lhs}) \end{array}}{CG \Rightarrow^{F=i} CG'}$$

[6. instance field dereference=local]

$$\frac{\begin{array}{l} \mathbb{E} \vdash (SH(a), Offset(f)) : addr_{lhs} \\ CG' = CG \cup CreateEdge(addr_{lhs}, shadow_i) \end{array}}{CG \Rightarrow^{a.f=i} CG'}$$

[7. local=computation]

$$\frac{\begin{array}{l} edge_c = CreateEdge(shadow_c, C) \\ edge_d = CreateEdge(shadow_d, C) \\ CG' = CG \cup edge_c \cup edge_d \end{array}}{CG \Rightarrow^{i=c+d} CG'}$$

Figure 9. Run-time effects of instrumentation.

4.4 Instrumentation Relation CG $\Rightarrow^a$ CG$'$

Our instrumenter takes the assembly-like J9 intermediate representation (IR) as input, and feeds the instrumented IR to the code generator. The goal of the instrumentation is to insert code to propagate the address of a copy graph node at run time. The copy graph node represents the heap location from which a piece of data comes.

The intraprocedural instrumentation is illustrated at a high-level in Figure 9. Based on the techniques described earlier, the name environment $\mathbb{E}$ maps each heap location to the address of its corresponding copy graph node. Function SH (i.e., shadow heap) returns, for each object, its allocation site ID and its context allocation site ID. For example, $\mathbb{E} \vdash SH(i) : addr_{rhs}$ in rule 1 says that given the (allocation ID, context ID) pair for the heap object pointed-to by local variable i, $\mathbb{E}$ maps this pair to the copy graph node at address $addr_{rhs}$. Here $addr_{rhs}$ and $addr_{lhs}$ represent the addresses of the copy graph nodes for the heap locations corresponding to the right/left-hand-side expressions of an instruction. Each rule describes the update of the copy graph (i.e., CG) for a type of instruction, with unprimed and primed symbols representing the copy graph before and after the instruction is executed.

In rule 1, the shadow of local variable i is assigned the address of the copy graph allocation node representing the newly-created heap object. If the method containing the allocation site is an instance method, the context object is the object referenced by `this`. The bit operation (& *0xFFFFFFFF*) retrieves the lower 4 bytes from the shadow heap location, which stores the allocation site ID for `this` itself (while the higher 4 bytes contain the allocation site ID of `this`'s context). A static method does not have a context.

Before each call site in a caller, the shadow variables for the actual parameters are pushed on the tracking stack, and they are popped at the entry of the callee method. Similarly, at the exit of the callee method, the shadow variable for the returned value is pushed, and it is popped after the call site in the caller. Data carried by exception flow is not tracked by the tool.

Once a heap load operation is seen (rules 2 and 3), the address of the node representing the heap location is stored in the shadow variable. Upon a heap store (rules 5 and 6), an edge with the source node address (contained in the shadow variable) and target node address (obtained from the heap location) is created, and the graph is updated with this new edge. In rule 7, once data comes to a computation instruction, we create edges to connect the copy graph node for each participating variable with the consumer node C.

5. Copy Graph Client Analyses

This section presents three client analyses implemented in J9. These clients analyze the copy graph and generate reports that are useful for understanding run-time behavior and pinpointing performance bottlenecks. Due to space limitations, the analyses are described informally, without low-level details.

5.1 Hot Copy Chains

Given a copy chain with frequency n and data size s, its copy volume is $n \times s$. The copy volume of a chain is the total amount of data transmitted along that chain. Chains with large copy volumes are more likely to be sources of performance problem. Another important metric is chain length—the longer a copy chain is, the more wasteful memory operations it contains. Considering both factors, we compute a *waste factor* (WF) for each chain as the product of length and copy volume. The goal of the hot chain analysis is to find copy chains that have large WF values.

The first issue is how to recover chains from copy graph edges. We use a brute-force approach which traverses the copy graph and computes the set of all distinct paths whose length is smaller than a pre-defined threshold value. If a path is a true copy chain, all its edges should have the same frequency. Based on this observation, the WF for each path is computed by using its smallest edge frequency as the path frequency. The resulting copy graph paths are ranked based on their WF values, and the top paths are reported. An example of a chain reported for benchmark antlr from DaCapo is as follows:

```
(355162, 2):
array[antlr/PreservingFileWriter:61].ELM
        — [java/io/BufferedWriter.write:198, 177581, 2] →
array[java/io/BufferedWriter:108].ELM
        — [sun/io/CharToByteUTF8.convert:262, 177759, 2] →
array[sun/nio/cs/StreamEncoder$ConverterSE:237].ELM
```

The chain contains three nodes connected by two edges. The pair (355162,2) shows the WF and the chain length. Each node in this example is an array element node. For instance field nodes and array element nodes, the allocation site of the base object is also shown. In this example, line 61 in class antlr.PreservingFileWriter creates the array whose elements are the sources of the copy chain. An edge shows the method where its last copy operation occurs (e.g., line 198 in method java.io.BufferedWriter.write), the edge frequency (e.g., 177581), and the data size (e.g., 2 bytes).

5.2 Clone Detector

Many applications make expensive clones of objects. A cloned object can be obtained via field-to-field copies from another object (e.g., as usually done in `clone` methods), or by adding data held by another object during initialization (e.g., many container classes have constructors that can initialize an object from another container object). Although clones are sometimes necessary, they indicate the existence of wasteful operations and redundant data. For instance, in our running example, `slowClone` initializes a new list by copying data from an existing list. Invoking this method many times may cause performance problems. The goal of this analysis is to find pairs of allocation sites, each of which represents the top (i.e., root) of a heap object subgraphs, such that a large amount of data is copied from one subgraph to the other.

For each copy graph edge $O_1.f \xrightarrow{a,b} O_2.g$, where f and g are instance fields, the value of $a \times b$ is counted as part of the direct flow from O_1 to O_2. The total direct flow for pair (O_1, O_2) shows how many bytes are copied from fields of O_1 to fields of O_2. Next, the analysis considers the indirect flow between objects. Suppose that some field of O_1 points to an object O_3, and some field of O_2 points to an object O_4. Furthermore, suppose that there is direct flow (i.e., some copy volume) from O_3 to O_4. In addition to attributing this copy volume to the pair (O_3, O_4), we want to also attribute it to the pair (O_1, O_2). This is done because O_1 may potentially be the root of an object subgraph for a data structure containing O_3. Similarly, O_2 may be the root of a data structure containing O_4. If copying is occurring for the entire data structures, the copy volume reported for pair (O_1, O_2) should reflect this.

The analysis considers all objects O_i reachable from O_1 along reference chains of a pre-defined length (length 3 was used for the experiments). Similarly, all objects O_j reachable from O_2 along reference chains of this length are considered. The copy volume reported for (O_1, O_2) is the sum of the direct copy volumes for all such pairs (O_i, O_j), including the direct flow from O_1 to O_2. To determine all relationships of the form "O' points to O", the analysis considers chains such that O is the producer node—that is, the value propagated along the chain is a reference to O. For any field node $O'.h$ in such a chain, object O' points to object O.

In the running example, `slowClone` illustrates this approach. At line 31, a new `List` object is created. Its field `elems` points to an array which is initialized with the contents of the array pointed to by the `List` created at line 39. In the first step of the analysis, volume 4000 is associated with the two array objects (1000 copies of 4-byte references to `Integer` objects). This volume is then also attributed to the two `List` objects, represented by pair (O_{39}, O_{31}), and to the two `ListClient` objects that own the lists, represented by pair (O_{44}, O_{32}). Ultimately, the reason for this entire copy volume is the cloning of a `ListClient` object, even though it manifests in the copying of the array data owned by this `ListClient`. Reporting the pair (O_{44}, O_{32}) highlights this underlying cause.

5.3 Not Assigned To Heap (NATH)

The third client analysis detects allocation sites that are instantiated many times and whose object references do not flow to the heap. For instance, O_{44} and O_{45} in the running example represent objects whose references are never assigned to any heap object or static field. These allocation sites are likely to represent the tops of temporary data structures that are constructed many times to provide simple services. For example, we have observed an application that creates GregorianCalendar objects inside a loop. These objects are used to construct the date fields of other objects. This causes significant performance degradation, as construction of GregorianCalendar objects is very expensive. In addition, these objects are usually temporary and short-lived, which may lead to frequent garbage collection. A simple fix that moves the object construction out of the loop can solve the problem. The escape analysis performed by a JIT usually does not remove this type of bloat, because many such objects escape the method where they are created, and are eventually captured far away from the method. Using copy graph, this analysis can be easily performed by finding all allocation nodes that do not have outgoing edges. These nodes are ranked based on the numbers of times that they are instantiated. Using the information provided by this analysis, we have found in Eclipse 3.1 a few places where NATH objects are heavily used. Running time reduction can be achieved after a simple manual optimization that avoids the creation of these objects.

5.4 Other Potential Clients

There are a variety of performance analyses that can take advantage of the copy graph. For example, one can measure and identify useless data by finding nodes that cannot reach the consumer node, and by aggregating them based on the objects that they belong to. As another example, developers of large applications usually maintain a performance regression test suite, which will be executed across versions of a program to guarantee that no performance degradation results from the changes. However, these performance regression tests can easily fail due to bug fixes or the addition of new features that involve extra memory copies and method invocations. It is labor-intensive to find the cause of these failures. Differentiating the copy graphs constructed from the runs of two versions of the program with the same input data can potentially help pinpoint performance problems that are introduced by the changes. A possible direction for future work is to investigate these interesting copy-graph-based analyses.

6. Using Copy Profiles to Find Bloat

This section presents three case studies of using copy profiles, both flat and ones derived from the copy graph, to pinpoint sources of useless work.

6.1 DaCapo Bloat

Recall from Figure 4 that most of the simple benchmarks, including the DaCapo bloat benchmark, have highly concentrated method profiles: a few methods explain most of the time. However, it is difficult to know whether these methods are important or merely excessive. Inspecting the total copy count of the DaCapo bloat benchmark, we found a high volume of data copies. Averaged across all method invocations, 28% of all operations were copies from one memory location to another. This lead us to suspect that there were big opportunities for optimizing away excessive computations and temporary object construction.

When inspecting the cumulative copy profile (i.e. a copy profile that counts copies in a method and any methods it invokes), we found that approximately 50% of all data copies came from a variety of `toString` and `append` methods. Inspecting the source code, we found that most of these calls centered around code of the form: `Assert.isTrue(cond, "bug: " + node)`. This benchmark was written prior to the existence of the Java `assert` keyword. This coding pattern meant that debugging logic resulted in entire data structures being serialized to strings, even though most of the time the strings themselves were unused; the `isTrue` method does not use the second parameter, if the first parameter is `true`. We made a simple modification to eliminate the temporary strings created during the most important copying methods[2]. This resulted in a 65% reduction in objects created, and a 29–35% reduction in execution time (depending on the JVM used; we tried Sun 1.6.0_10 and IBM 1.6.0 SR2).

The DaCapo suite is geared towards JVM and hardware designers. Therefore, it is important to reevaluate the benchmarks so as to

[2] We commented out the `toString` methods of `Block`, `FlowGraph`, `RegisterAllocator`, `Liveness`, `Node`, `Tree`, `Label`, `MemberRef`, `Instruction`, `NameAndType`, `LocalVariable`, `Field`, and `Constant`.

distinguish inefficiencies that a JIT could possibly eliminate from ones that require a programmer with good tooling. It may be better for such a benchmark suite not to contain an excess of the latter.

6.2 Java 5 GregorianCalendar

A recurring problem with the Java 1.5 standard libraries is the slow performance of calendar-related classes [27]. Many users experienced a 50× slowdown when upgrading from Java 1.4 to Java 1.5. The problems centered around methods in class `GregorianCalendar`, which is an important part of date formatting and parsing. We ran the test case provided by a user and constructed a context-sensitive copy graph. The test case makes intensive calls of the `before`, `after`, and `equals` methods. The report of hot copy chains includes a family of hot chains with the following structure:

array[Calendar:907].ELM
    — [Calendar.clone:2168,510000] →
array[Calendar:2169].ELM

This chain (and others similar to it, for the fields of a calendar) suggests that `clone` is invoked many times to copy values from one Calendar to another. To confirm this, we ran the clone detector and the top four pairs of allocation sites were as follows:

340000: (GregorianCalendar[GregorianCalendarTest:11],array[Calendar:2168])
340000: (array[Calendar:906],array[Calendar:2168])
340000: (array[Calendar:907],array:[Calendar:2169])
340000: (array[Calendar:908],array[Calendar:2170])

The first pair shows that an array created at line 2168 of `Calendar` gets a large amount of data from the `GregorianCalendar` object created in the test case. The remaining three pairs of allocation sites also suggest the occurrence of clones, because the first group of objects (i.e., at lines 906, 907, 908) are arrays created in the constructor of `Calendar`, while the second group (i.e., at lines 2168, 2169, and 2170) are arrays created in `clone`. By examining the code, we found that `clone` creates a new object by deep copying all array fields from the old `Calendar` object. These copies also include the cloning of a time zone from the *zone* field of the existing object. Upon further inspection, we found the cause of the slowdown: methods `before`, `after`, and `equals` invoke method `compareTo` to compare two `GregorianCalendar` objects, which is implemented by comparing the current times (in milliseconds) obtained from these objects. However, `getMillisof` does not compute time directly from the existing calendar object, but instead makes a clone of the calendar and obtains the time from the clone.

The JDK 1.4 implementation of `Calendar` does not clone any objects. This is because the 1.4 implementation of `getMillisof` mistakenly changes the internal state of the object when computing the current time. In order to avoid touching the internal state, the implementers of JDK 1.5 made the decision to clone the calendar and get the time from the clone. Of course, it is not a perfect solution as it fixes the original bug at the cost of introducing a significant performance problem. Our tool highlighted the useless work being done in order to work around the `getMillisof` issue.

6.3 DaCapo Eclipse

As a large framework-based application, Eclipse suffers from performance problems that result from the pile-up of wasteful operations in its plugins. These problems impact usability, and even programmers' choice when comparing Java development tools [12]. We ran Eclipse 3.1 from the DaCapo benchmark set and used the NATH analysis to identify allocation sites whose run-time objects are never assigned to the heap. The top nine allocation sites are shown below:

(1) 295004: org/eclipse/jdt/internal/compiler/ISourceElementRequestor$MethodInfo [SourceElementParser:968]
(2) 161169: .../SimpleWordSet[SimpleWordSet:58]
(3) 145987: .../ISourceElementRequestor$FieldInfo[SourceElementParser:1074]
(4) 46603: .../ContentTypeCatalog$7[ContentTypeCatalog:523]
(5) 46186: .../ISourceElementRequestor$TypeInfo[SourceElementParser:1190]
(6) 45813: .../Path[PackageFragment:309]
(7) 44703, .../Path[CompilationUnit:786]
(8) 37201, .../ContentTypeHandler[ContentTypeMatcher:50]
(9) 30939, .../HashtableOfObject[HashtableOfObject:123]

Each line shows an allocation site and the number of times it is instantiated. For example, the first line is for an allocation site at line 968 in class `SourceElementParser`, which creates 295004 objects of type `ISourceElementRequestor$MethodInfo`. Sites 4 and 8 are from plugin org.eclipse.core.resources. The remaining sites are located in org.eclipse.jdt.core. Because the Eclipse 3.1 release does not contain the source code for org.eclipse.core.resources, we inspected only the seven sites in the JDT plugin.

The first site is located in class `SourceElementParser`, which is a key part of the JDT compiler. JDT provides many source code manipulation functionalities that can be used for various purposes, such as automated formating and refactoring. The observer pattern is used to provide source code element objects when a client needs them. Method `notifySourceElementRequestor`, which contains this site, plays the observer role: once a requestor (i.e., a client) asks for a compilation unit node (i.e., a class), the method notifies all child elements (i.e., methods) of the compilation unit by calling method `enterMethod`, which will subsequently notify source code statements in each method. Method `enterMethod` takes a `MethodInfo` object as input; this object contains all necessary information for the method that needs to be notified.

The site creates `MethodInfo` objects which are then provided to `enterMethod`. Because `enterMethod` is defined in an interface, we checked all implementations of the method. Surprisingly, none of these implementations invoke any methods on this parameter object. They extract all information about the method to be notified from fields of the object; these fields are previously set by `notifySourceElementRequestor`. The third and the fifth allocation sites from above tell the same story: these hundreds of thousands of objects are created solely for the purpose of carrying data across one-level method invocations. It is expensive to create and reclaim these objects, and to perform the corresponding heap copies. We modified the interface and all related implementations to pass data directly through parameters. This modification reduces the number of allocated objects by millions and improves the running time by 2.8%. In large applications with no single hot spot, significant performance improvements are possible by accumulating several such "small" improvements, as illustrated below.

Table 2 shows a list of several problems we identified with the help of the analyses. For each problem, the table shows the problematic class (*Class*), our code modification, the number of allocated objects (*#Objs*), the total number of GC invocations (*#GCs*), and the running times. Row *Original* characterizes the original execution. Each subsequent row shows the cumulative improvements due to our changes in the JDT plugin. The second row corresponds to allocation sites 1, 3, and 5 listed above, the third row is for sites 6 and 7, the fourth row is for site 2, and the last row is for site 9.

Class	*Modification*	*#Objs*	*#GCs*	*Time(s)*
Original	—	273991250	478	143.6
MethodInfo, FieldInfo, TypeInfo	Directly pass the data	272461138	460	139.6
PackageFragment	Get IResource directly from String	272429471	448	138.3
SimpleWordSet	In-place rehash	272395776	430	136.8
HashtableOfObject	In-place rehash	272320499	424	134.0

Table 2. Eclipse 3.1 performance problems, fixes, and performance improvements.

By modifying the code to eliminate redundant copies and the related creation of objects, we successfully reduced the number of GC runs, the number of allocated objects, and the total running time. With the help of the tool, it took us only a few hours to find these problems and to make modifications in a large application we had never studied before.

It is important to note that this effort just scratches the surface: significant performance improvement may be possible if a developer or a performance expert carefully examines the tool reports (with different tests and workloads) and eliminates the identified useless work. This is the kind of manual tuning that is already being done today for large Java applications with performance problems that cannot be attributed to a single hot spot. This tedious and labor-intensive process can be made more efficient and effective by the dynamic analyses proposed in our work. Future studies should investigate such potential performance improvements for a broad range of Java applications.

7. Copy Graph Characteristics

This section presents characteristics of the copy graph and its construction. All experiments were performed on a machine with a 1.99GHz Dual Core AMD Opteron processor, running Linux 2.6.9. The programs were run with JIT optimizations turned off to collect a copy graph of the unmodified source program. The maximum heap size specified for each run was 500Mb. Hence, the size of shadow for each run was 500Mb. IBM DMS is the IBM document management server mentioned in the first section, which is run on top of a J2EE application server. Each DaCapo benchmark was run with large workload for two iterations, and the running time for the second iteration is shown. SPECjbb and IBM DMS are server applications that report throughput, not total running time; both were run for 30 minutes with a standard workload.

Table 3 presents the time and space overhead of context-insensitive copy graphs. The second column, labeled T_{orig}, presents the original running times in seconds. The remaining columns show the total numbers of nodes N_0 and edges E_0, the amount of memory M_0 needed by the analysis (in megabytes), the running times T_0 (in seconds), and the performance slowdowns (shown in parentheses). The slowdown for each program is T_0/T_{orig}. Because the shadow heap is 500Mb, the space overhead of the copy graph is M_0–500.

In Table 4, the same measurements are reported for 1-object-sensitive copy graphs. To understand the impact of the number of context slots (i.e., parameter c from Section 4.3), we experimented with values 4, 8 and 16 when constructing the 1-object-sensitive copy graph. The slowdown for each program was calculated as T_i/T_{orig} (the original time from Table 3), where $i \in \{4, 8, 16\}$.

The copy graph itself consumes a relatively small amount of memory. Other than for IBM DMS, the space overhead of the copy graph does not exceed 27Mb even when using 16 context slots. As expected, a context-sensitive copy graph consumes more memory than the context-insensitive one, and using more context slots leads to larger space overhead.

The running time overheads for profiling the context-insensitive copy graph and the three versions of 1-object-sensitive copy graphs are, on average, 36×, 37×, 37×, and 37× respectively. This overhead is not surprising because the analysis tracks the execution of every instruction in the program. The overhead also comes from synchronization performed by the instrumentation of allocation sites, which sequentially executes the allocation handler to create allocation header elements. The current implementation provides a general facility for mapping an object address to a context ID. This is done even for the context-insensitive analysis, where the ID is always 0. Since the cost of this mapping is negligible, we have not created a specialized context-insensitive implementation. Hence, the difference between the running times of profiling context-insensitive and context-sensitive copy graphs is noise. The only significant difference between context-insensitive and context-sensitive analysis is the space overhead.

Program	*Original*	Context-insensitive			
	$T_{orig}(s)$	$\#N_0$	$\#E_0$	$M_0(Mb)$	$T_0(s)\,(\times)$
antlr	8.9	12516	56703	503.7	284.2 (31.9)
bloat	157.5	14058	14471	502.2	9812.2 (62.4)
chart	32.5	18113	12810	502.5	1053.2 (32.4)
fop	3.6	12419	7675	501.8	38.2 (10.6)
pmd	46.6	11289	8418	501.7	1542.4 (33.1)
jython	74.7	25653	21893	503.2	2826.1 (37.8)
xalan	64.8	13505	28678	502.6	3030.5 (46.8)
hsqldb	13.5	12294	9102	501.7	350.0 (25.9)
luindex	12.1	10154	10227	501.6	583.4 (48.2)
lusearch	19.2	8390	13849	501.5	662.8 (34.5)
eclipse	124.7	34074	52957	506.5	4343.8 (34.8)
SPECjbb	1800*	17146	12637	502.4	1800*
IBM DMS	1800*	147517	87531	519.6	1800*

Table 3. Copy graph size and time/space overhead, part 1. Shown are the original running time T_{orig}, as well as the total numbers of graph nodes N_0 and edges E_0, the total amount of memory consumed M_0, the running time T_0, and the slowdown (shown in parentheses) when using a context-insensitive copy graph.

Although significant, these overheads have not hindered us from running the tool on any programs, including real world large-scale production applications. It was an intentional design decision *not* to focus on the performance of the analysis, but instead focus on the content collected and on demonstrating that the results are useful for finding performance problems in real programs. Now that the value of the tool has been established, a possible future direction is to use sampling-based profiling to obtain the same or similar results. Another possibility is to employ static pre-analyses that reduce the cost of the subsequent dynamic analysis.

Table 5 shows measurements for the copy chains obtained from a context-insensitive copy graph, including the total number of generated chains (*#Chains*) and the average length of these chains (*Length*). The table also shows the number of NATH allocation sites and NATH run-time objects. The significant numbers of NATH objects indicate that eliminating such objects may be a worthwhile goal for future work on manual and automatic optimizations.

The first part of Table 6 lists the average node *fan-out* for the context-insensitive copy graph (*CIFO*) and the three versions of context-sensitive copy graphs (*CSFO-i*, where i is the number of context slots for each object). A node's fan-out is the number of its outgoing edges. The average fan-out indicates the degree of node sharing among paths in the graph. Note that *CIFO* and *CSFO-i* are small, because there exist a large number of producer nodes (allocation site) that do not have outgoing edges. In addition, the more slots are used to represent contexts, the smaller the average fan-out, because more nodes are created to avoid path sharing.

In addition, for each context-sensitive copy graph, the table reports the average *context conflict ratio* (*CCR-i*). The CCR for an object o is defined as follows:

$$\textit{CCR-i}(o) = \begin{cases} 0 & \max_{0 \le k \le i}(nc[k]) = 1 \\ \max(nc[k]) / \sum nc[k] & \text{otherwise} \end{cases}$$

Here $nc[k]$ represents the number of distinct contexts that fall into context slot k. The CCR value captures the degree to which our encoding function (i.e., $id \,\%\, k$) causes distinct contexts to be merged in the copy graph. For example, the CCR is 0 if each context slot represents at most one distinct context; the CCR is 1 if all contexts for the object fall into the same slot. The table reports the average CCR for all allocation sites in the copy graph. As expected, the average CCR decreases with an increase in the

Program	1-object-sensitive ($c = 4$)				1-object-sensitive ($c = 8$)				1-object-sensitive ($c = 16$)			
	#N_4	*#E_4*	*M_4(Mb)*	*T_4(s) (×)*	*#N_8*	*#E_8*	*M_8(Mb)*	*T_8(s) (×)*	*#N_{16}*	*#E_{16}*	*M_{16}(Mb)*	*T_{16}(s) (×)*
antlr	48556	112907	506.9	294.8 (33.1)	96609	159042	510.2	300.7 (33.8)	192713	210522	515.2	309.5 (34.8)
bloat	54960	35678	504.3	10182.9 (64.7)	109494	48840	506.5	10147.4 (64.4)	218558	60483	510.5	10068.2 (63.9)
chart	69438	25951	504.6	1079.4 (33.2)	137945	39133	507.3	1054.4 (32.4)	274903	45071	511.9	1056.5 (32.5)
fop	47893	11985	503.1	37.4 (10.4)	95180	13509	504.6	37.2 (10.3)	189757	14388	507.7	36.8 (10.2)
pmd	43740	15576	503.0	1586.7 (34.0)	86980	19568	504.5	1568.5 (33.7)	173484	21339	507.3	1555.5 (33.4)
jython	95493	32256	505.8	2865.6 (38.4)	188583	37005	509.0	2879.9 (38.6)	374791	41027	515.0	2861.4 (38.3)
xalan	52485	55367	504.9	2983.3 (46.0)	85751	88001	507.7	3067.6 (47.3)	208119	117760	512.2	3067.6 (47.3)
hsqldb	47666	13432	503.0	358.0 (26.5)	94846	15201	504.6	346.7 (25.7)	189183	17190	507.7	345.9 (25.6)
luindex	39319	17695	502.8	568.7 (47.0)	78232	22912	504.3	581.1 (48.0)	156033	28333	507.0	564.8(46.7)
lusearch	32354	22163	502.6	643.5 (33.5)	64280	26629	503.8	651.6 (33.9)	128152	32544	506.1	658.4(34.3)
eclipse	131065	124043	512.3	4521.5 (36.3)	259168	154004	517.4	4545.3 (36.4)	516030	174846	526.4	4746.4 (38.1)
SPECjbb	66102	23909	503.3	1800*	131413	27660	507.2	1800*	261915	29017	511.0	1800*
IBM DMS	193707	180187	533.7	1800*	381072	242049	571.2	1800*	755829	304759	652.3	1800*

Table 4. Copy graph size and time/space overhead, part 2. The columns report the same measurements as Table 3, but for 1-object sensitive copy graph with 4, 8 and 16 context slots.

Program	*#Chains*	*Length*	*#NATH Sites*	*#NATH Objects*
antlr	250680	2.60	811	411536
bloat	6955316	4.00	1160	31217025
chart	29490	1.16	1652	15080848
fop	275835	3.36	1282	167808
pmd	436397	2.96	1062	54103059
jython	6827057	4.00	493	35926287
xalan	93263	2.60	1218	6186112
hsqldb	8595	1.80	828	3059666
luindex	30183	2.24	749	5543579
lusearch	10640	3.8	302	4200325
eclipse	10070910	1.24	3030	3494187
SPECjbb	21468	2.00	575	722800
IBM DMS	1937646	3.75	4695	1413528

Table 5. Copy chains and NATH objects. All copy graph paths with length ≤ 5 are traversed to compute hot chains. The columns show the total number of generated chains, the average length of these chains, and the number of NATH allocation sites and NATH run-time objects.

Program	Average fan-out				Context conflict ratio		
	CIFO	*CSFO* 4	*CSFO* 8	*CSFO* 16	*CCR* 4	*CCR* 8	*CCR* 16
antlr	4.66	2.33	1.64	1.09	0.237	0.131	0.081
bloat	1.03	0.64	0.44	0.27	0.199	0.090	0.068
chart	0.70	0.36	0.28	0.16	0.118	0.059	0.028
fop	0.62	0.25	0.15	0.08	0.134	0.060	0.043
pmd	0.75	0.35	0.22	0.12	0.131	0.059	0.051
jython	0.80	0.31	0.18	0.10	0.079	0.071	0.024
xalan	2.13	1.79	0.81	0.56	0.128	0.067	0.040
hsqldb	0.74	0.28	0.16	0.09	0.169	0.080	0.051
luindex	1.02	0.45	0.29	0.18	0.148	0.073	0.051
lusearch	1.68	0.68	0.41	0.25	0.127	0.082	0.052
eclipse	1.53	0.91	0.57	0.33	0.193	0.114	0.071
SPECjbb	0.75	0.36	0.21	0.11	0.144	0.065	0.026
IBM DMS	0.76	0.32	0.17	0.09	0.112	0.047	0.027

Table 6. Average node *fan-out* for context-insensitive (*CIFO*) and context-sensitive (*CSFO-i*) copy graphs, as well as average *context conflict ratios* (*CCR-i*) for the context-sensitive copy graphs.

number of context slots. Note that very few context conflicts occur even when $c = 4$, because a large number of objects have only one distinct context during theirs lifetimes.

8. Related Work

Dynamic Data Flow Dynamic taint analysis [20, 30, 21, 7] tracks input data from untrusted channels to detect potential security attacks. Debugging, testing, and program understanding tools track dynamic dataflow for other specialized purposes (e.g., [15]). The work of [6] tracks the origin of undefined values. Dynamic slicing [34, 31, 32] generates a trace of control and data flow during an execution, in order to enable postmortem analyses for bug detection. Dynamic slicing is expensive in both time and space, as it records complete data and control flow dependencies. Our analysis records only copy flow. In addition, it applies static abstractions on dynamic information flow, and thus makes it possible to scale to large and long-running applications such as IBM Websphere.

Profiling When profiling to find performance problems, all techniques that we are aware of concentrate on control flow, rather than data flow, from path profiling [3, 14, 5, 28] to feedback-directed profiling [2], all to identify heavily-executed paths for further optimization. The work of [1] develops a dynamic analysis tool to explore calling context trees in order to find performance bottlenecks. The work of [26] uses a dynamic analysis technique that identifies important program components, also by inspecting calling context trees. Xu and Rountev propose container-centric profiling [29], and Rayside and Mendel propose object ownership profiling [23], both to detect memory leaks in Java programs. Zhang and Gupta propose *whole execution traces* [33] that includes complete data information of an execution to enable the mining of program profiles that require understanding of relationships among various profiles.

Bloat Dufour *et al.* propose dynamic metrics for Java [8], which provide insights by quantifying run-time bloat. Many memory profiling tools have been developed to take heap snapshots for understanding memory usage [13], and identify objects of suspicious types [22, 11] that consume a large amount of memory. However, none of these tools attempt to understand the underlying causes of memory bloat, and thus cannot help programmers pinpoint the problematic areas. The research in [18] structures behavior according to the flow of information, though using a manual technique. Their aim is to allow programmers to place judgments on whether certain classes of computations are excessive. Our work is in this same spirit, and automates an important component of it. The work of [17] introduces a way to find data structures that consume excessive amounts of memory. Recent work finds excessive use of temporary data structures [9, 10] and summarizes the shape of the temporary structures. In contrast to the purely dynamic approximation introduced here, it employs a blended escape analysis, which applies static analysis to a region of dynamically collected calling structure with observed performance problem. By approximating object effective lifetimes, the analysis has been shown to be useful in classifying the usage of newly created objects in the problematic program region. Another recent work also identifies regions that make heavy use of temporary objects, in order to guide aggressive method inlining [24]. Our paper addresses the challenge of automatically detecting bloated computations that fall out of the purview of conventional JIT optimization strategies.

9. Conclusion

Programmers must have some level of trust in the optimizations that will be performed on their behalf. This is especially true, because software is now assembled from so many abstractions. We trust compilers enough to avoid low-level tuning, in the hope that automated optimizations will take care of those details. At every juncture, we make such assumptions, and add delegation to our data models, and employ over-general libraries. What's one extra method call or object? At some point, however, these decisions pile up, and our assumptions are no longer true: the JIT can no longer clean up the mess. Most of the temporary strings in DaCapo bloat benchmark are never used beyond their construction, and yet state of the art JITs do not identify this fact.

Compilers and tools are currently focused on control flow, and largely make optimization decisions based on time spent in control dependence regions. In large-scale systems, what makes code suspicious, and worthy of a focused optimization, is not time, but other signs of excess — such as data copying. In this paper, we introduced a way to help developers find larger performance improvements that are beyond the scope of local calling contexts. We are hopeful that this analysis can also expose unexplored opportunities for the JITs to optimize more globally, such as specializing across multiple components, or hoisting complex, many-layered computations.

This attention to the typical patterns of excess, i.e. that which should be optimizable at some level, can also help with benchmark design and validation. Benchmark designers can use metrics such as the ones we have presented to ensure that the benchmarks mimic the kinds of bloat we see in real applications. If benchmark suites came with copy profiles for each benchmark, then researchers could use these same metrics to evaluate how well their techniques target bloat. We would like to avoid a situation where, for example, hardware designers add features to help an application better digest its short-lived objects, all so that it runs 5% faster. We know that there are larger opportunities, if we optimize at the right level.

Acknowledgments

We would like to thank the PLDI reviewers for their valuable and thorough comments and suggestions. This research was supported in part by the National Science Foundation under CAREER grant CCF-0546040 and by an IBM Software Quality Innovation Faculty Award. Guoqing Xu was supported in part by a summer internship at the IBM T.J. Watson Research Center.

References

[1] G. Ammons, J.-D. Choi, M. Gupta, and N. Swamy. Finding and removing performance bottlenecks in large systems. In *ECOOP*, pages 172–196, 2004.

[2] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. A survey of adaptive optimization in virtual machines. *Proc. IEEE*, 92(2):449–466, 2005.

[3] T. Ball and J. Larus. Efficient path profiling. In *MICRO*, pages 46–57, 1996.

[4] S. M. Blackburn and et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, 2006.

[5] M. D. Bond and K. S. McKinley. Continuous path and edge profiling. In *MICRO*, pages 130–140, 2005.

[6] M. D. Bond, N. Nethercote, S. W. Kent, S. Z. Guyer, and K. S. McKinley. Tracking bad apples: reporting the origin of null and undefined value errors. In *OOPSLA*, pages 405–422, 2007.

[7] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *ISSTA*, pages 196–206, 2007.

[8] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for Java. In *OOPSLA*, pages 149–168, 2003.

[9] B. Dufour, B. G. Ryder, and G. Sevitsky. Blended analysis for performance understanding of framework-based applications. In *ISSTA*, pages 118–128, 2007.

[10] B. Dufour, B. G. Ryder, and G. Sevitsky. A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications. In *FSE*, pages 59–70, 2008.

[11] ej-technologies. JProfiler. `www.ej-technologies.com`.

[12] Java Development Blog. `dld.blog-city.com`.

[13] Java Heap Analyzer Tool (HAT). `hat.dev.java.net`.

[14] J. Larus. Whole program paths. In *PLDI*, pages 259–269, 1999.

[15] W. Masri and A. Podgurski. An empirical study of the strength of information flows in programs. In *WODA*, pages 73–80, 2006.

[16] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *TOSEM*, 14(1):1–41, 2005.

[17] N. Mitchell and G. Sevitsky. The causes of bloat, the limits of health. *OOPSLA*, pages 245–260, 2007.

[18] N. Mitchell, G. Sevitsky, and H. Srinivasan. Modeling runtime behavior in framework-based applications. In *ECOOP*, pages 429–451, 2006.

[19] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *VEE*, pages 65–74, 2007.

[20] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005.

[21] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *MICRO*, pages 135–148, 2006.

[22] Quest Software. JProbe memory debugging. `www.quest.com/jprobe`.

[23] D. Rayside and L. Mendel. Object ownership profiling: a technique for finding and fixing memory leaks. In *ASE*, pages 194–203, 2007.

[24] A. Shankar, M. Arnold, and R. Bodik. JOLT: Lightweight dynamic analysis and removal of object churn. In *OOPSLA*, pages 127–142, 2008.

[25] O. Shivers. Control-flow analysis in Scheme. In *PLDI*, pages 164–174, 1988.

[26] K. Srinivas and H. Srinivasan. Summarizing application performance from a component perspective. In *FSE*, pages 136–145, 2005.

[27] Sun Java Forum. `forums.java.net/jive/thread.jspa?messageID=180784`.

[28] K. Vaswani, A. V. Nori, and T. M. Chilimbi. Preferential path profiling: Compactly numbering interesting paths. In *POPL*, pages 351–362, 2007.

[29] G. Xu and A. Rountev. Precise memory leak detection for Java software using container profiling. In *ICSE*, pages 151–160, 2008.

[30] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In *USENIX Security*, pages 121–136, 2006.

[31] X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. In *PLDI*, pages 169–180, 2006.

[32] X. Zhang and R. Gupta. Cost effective dynamic program slicing. In *PLDI*, pages 94–106, 2004.

[33] X. Zhang and R. Gupta. Whole execution traces. In *MICRO*, pages 105–116, 2004.

[34] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *ICSE*, pages 319–329, 2003.