Finding Low-Utility Data Structures

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

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

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

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

Edith Schonberg
IBM T.J. Watson Research Center
ediths@us.ibm.com

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

Abstract

Many opportunities for easy, big-win, program optimizations are missed by compilers. This is especially true in highly layered Java applications. Often at the heart of these missed optimization opportunities lie computations that, with great expense, produce data values that have little impact on the program's final output. Constructing a new date formatter to format every date, or populating a large set full of expensively constructed structures only to check its size: these involve costs that are out of line with the benefits gained. This disparity between the formation costs and accrued benefits of data structures is at the heart of much runtime bloat.

We introduce a run-time analysis to discover these *low-utility* data structures. The analysis employs dynamic thin slicing, which naturally associates costs with value flows rather than raw data flows. It constructs a model of the incremental, hop-to-hop, costs and benefits of each data structure. The analysis then identifies suspicious structures based on imbalances of its incremental costs and benefits. To decrease the memory requirements of slicing, we introduce *abstract dynamic thin slicing*, which performs thin slicing over bounded abstract domains. We have modified the IBM J9 commercial JVM to implement this approach.

We demonstrate two client analyses: one that finds objects that are expensive to construct but are not necessary for the forward execution, and second that pinpoints ultimately-dead values. We have successfully applied them to large-scale and long-running Java applications. We show that these analyses are effective at detecting operations that have unbalanced costs and benefits.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Memory management, optimization, run-time environments; F.3.2 [*Logics and Meaning of Programs*]: Semantics of Programming Languages—Program analysis; D.2.5 [*Software Engineering*]: Testing and Debugging—Debugging aids

General Terms Languages, Measurement, Performance

Keywords Memory bloat, abstract dynamic thin slicing, cost-benefit analysis

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

PLDI'10, June 5–10, 2010, Toronto, Ontario, Canada.
Copyright © 2010 ACM 978-1-4503-0019/10/06. . . $10.00

1. Introduction

The family of dataflow optimizations does a good job in finding and fixing local program inefficiencies. Through an iterative process of enabling and pruning passes, a dataflow optimizer reduces a program's wasted effort. Inlining, scalar replacement, useless store elimination, dead code removal, and code hoisting combine in often powerful ways. However, our experience shows that a surprising number of opportunities of this ilk, ones that would be easy to implement for a developer, remain untapped by compilers [19, 34].

Despite the commendable accomplishments of the commercial JIT teams, optimizers seem to have reached somewhat of an impasse [16, 22]. In a typical large-scale commercial application, we routinely find missed opportunities that can yield substantial performance gains, in return for a small amount of developer time. This is only true, however, if the problem is clearly identified for them. The sea of abstractions can easily hide what would be immediately obvious to a dataflow optimizer as well as a human expert.

In this regard, developers face the very same impasse that JITs have reached. We write code that interfaces with a mountain of libraries and frameworks. The code we write and use has been explicitly architected to be overly general and late bound, to facilitate reuse and extensibility. One of the primary effects of this style of development is to decrease the efficacy of inlining [29], which is a lynchpin of the dataflow optimization chain. A JIT often can not, in a provably safe way, specialize across a vast web of subroutine calls; a developer should not start optimizing the code until definitive evidence is given that the generality and pluggability cost too dearly.

However, if we wish to enlist the help of humans, we cannot have them focus on every tiny detail, as a compiler would. Inefficiencies are the accumulation of minutiae, including a large volume of temporary objects [11, 29], highly-stale objects [5, 26], excessive data copies [34], and inappropriate collection choices [28, 35]. The challenge is to help the developers perform dataflow-style optimizations by providing them with relatively small problems.

Very often, the required optimizations center around the formation and use of data structures. Many optimizations are as simple as hoisting an object constructor out of a loop. In other cases, code paths must be specialized to avoid frequent expensive conditional checks that are always true, or to avoid the expense of computing data values hardly, if ever, used. In each of these scenarios, a developer, enlisted to tune these problems, needn't be presented with a register-transfer dataflow graph. There may exist larger opportunities if they focus on high-level entities, such as how whole data structures are being built up and used.

المنارة للاستشارات
www.manaraa.com

Cost and benefit A primary symptom of these missed optimization opportunities is an imbalance between costs and benefits. The cost of forming a data structure, of computing and storing its values, are out of line with the benefits gained over the uses of those values. For example, the DaCapo `chart` benchmark creates many lists and adds thousands of data structures to them, for the sole purpose of obtaining list sizes. The actual values stored in the list entries are never used. These entries have non-trivial formation costs, but contain values of zero benefit to the end goal of the program.

In this paper, we focus on these *low-utility data structures*. A data structure has low utility if the cumulative costs of creating its member fields outweighs a weighted sum over the subsequent uses of those fields. For this paper, we compute these based on the dynamic flows that occur during representative runs. The cost of a member field is the total number of bytecode instructions transitively required to produce it; each instruction is treated as having unit cost. The weighted benefit of this field depends on how it is used; we consider three cases: a field that reaches program output has infinite weight; a field that is used to (transitively) produce a new value written into another field has a weight equal to the work done (on the stack) to create that new value; finally, a field that is used for neither has zero weight. We show how these simple, heuristic assignments of cost and benefit are sufficient to identify many important, large chunks of suboptimal work.

Abstract thin slicing formulation A natural way of computing cost and benefit for a value produced by an instruction i is to perform dynamic slicing and calculate the numbers of instructions that can reach i (i.e., for computing cost) and that are reachable from i (i.e., for computing benefit) in the dynamic dependence graph.

Conventional dynamic slicing [33, 37, 38, 40] does not distinguish flows along pointers from flows along values. Consider an example assignment `b.f = g(a.f)`. The cost of the value `b.f` should include the cost of computing the field `a.f`, plus the costs of the `g` logic. A dynamic slicing approach would also include the cost of computing the `a` pointer. An approach based on thin slicing [30] would exclude any pointer computation cost. When analyzing the utility of data structures, this is a more useful approach. If the object referenced by `a` is expensive to construct, then its cost should be associated with the objects involved in facilitating that pointer computation. For example, had there existed another assignment `c.g = a`, `c` would be the object to which `a`'s cost should be attributed, not `b`. We therefore formulate our solution on top of dynamic thin slicing. By separating pointer computations from value flow, this approach naturally assigns formation costs to data structures.

Dynamic thin slicing, in general, suffers from a problem common to most client analyses of dynamic slicing [2, 9, 34, 41]: large memory requirements. While there exists a range of strategies to reduce these costs [15, 27, 33, 38, 39, 41], it is still prohibitively expensive to perform whole-program dynamic slicing. The amount of memory consumed by these approaches is a function of the number of instruction instances (i.e., depending completely on dynamic behavior), which is very large, and also unbounded.

To improve the scalability of thin slicing, we introduce *abstract dynamic thin slicing*, a technique that applies thin slicing over *bounded abstract domains* that divide equivalent classes among instruction instances. The resulting dependence graph contains abstractions of instructions, rather than their runtime instances. The key insight here is that, having no knowledge of the client that will use the profiles, traditional slicing captures every single detail of the execution, much of which is not needed at all by the client. By taking into account the semantics of the client (encoded by the bounded abstract domain) during the profiling, one can record only the abstraction of the whole execution profiles that satisfies the client, leading to significant time and space overhead reduction. For example, with abstract dynamic thin slicing, the amount of memory required for the dependence graph is bounded by the number of abstractions.

Section 2 shows that, in addition to cost-benefit analysis, such slicing can solve a range of other backward dynamic flow problems that exhibit bounded-domain properties.

Relative costs and benefits How far back should costs accumulate, and how far forward should benefits accrue? A straightforward application of dynamic (traditional or thin, concrete or abstract) slicing would suffer from the *ab initio, ad finem* problem: costs would accumulate from the beginning, and benefits would accrue to the end of the program's run. Values produced later during the execution are highly likely to have larger costs than values produced earlier. In this case, there would not exist a correlation between high cost and actual performance problems, and such an analysis would not be useful for performance analysis. To alleviate this problem, we introduce the notion of *relative* cost and benefit. The relative cost of a data structure D is computed as the amount of work performed to construct D from other data structures that already exist in the heap. The relative benefit of D is the flip side of this: the amount of work performed to transform data read from D in order to construct other data structures. We show how to compute the relative costs and benefits over the abstractions; these are termed the *relative abstract cost* and *relative abstract benefit*.

Implementation and experiments To help the programmer diagnose performance problems, we introduce several cost-benefit analyses that take as input the abstract dynamic dependence graph and report information related to the underlying causes. Section 3 defines in detail one of these analyses, which computes relative costs and benefits at the level of data structures by aggregating costs and benefits for individual heap locations.

These analyses have been implemented in the IBM J9 commercial JVM and successfully applied to large-scale and long-running applications such as `derby`, `tomcat` and `trade`. Section 4 presents an evaluation of analysis expenses. A shadow heap is used to record information about fields of live objects [24]. The dependence graph consumes less than 20 megabytes across the applications and benchmarks tested. The current prototype implementation imposes an average slowdown of 71 times when whole-program tracking is enabled. We show that it is possible to reduce overhead significantly if one tracks only important phases. For example, by tracking only the steady-state portion of a server's run, overheads are reduced by up to 10×. This overhead, admittedly high, has nonetheless proved acceptable for performance tuning.

Section 4 describes six case studies. Using the tool, we found hundreds of performance problems, and eliminating them resulted in 2% – 37% performance improvements. These problems include inefficiencies caused by common programming idioms, repeated work whose results need to be cached, computation of redundant data, and choices of unnecessary expensive operations. Some of these finding also provide useful insights for automatic code optimization in compilers.

The contributions of this work are:

- *Cost and benefit profiling*, a methodology that identifies runtime inefficiencies by understanding the cost of producing values and the benefit of consuming them.
- *Abstract dynamic thin slicing*, a general technique that performs dynamic thin slicing over bounded abstract domains. It produces much smaller and more relevant slices than traditional dynamic slicing and can be used for cost-benefit analysis and for other dynamic analyses.
- *Relative cost-benefit analysis* which reports data structures that have unbalanced cost-benefit rates.

المنارة للاستشارات
www.manaraa.com

- A J9-based implementation and six case studies using real-world programs, demonstrating that the tool can help a programmer to find opportunities for performance improvements.

2. Cost Computation Using Abstract Slicing

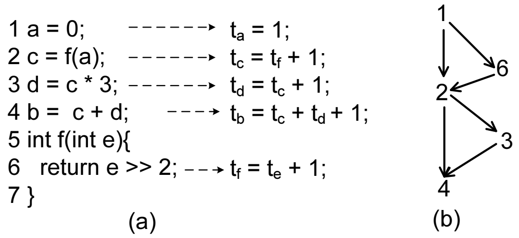
This section formalizes the proposed technique for abstract dynamic thin slicing, and shows several clients that can take advantage of this technique. Next, the definition of cost is presented together with a run-time profiling technique that constructs the necessary abstract data dependence graph. While our tool works on the low-level JVM intermediate representation, the discussion of the algorithms uses a three-address-code representation of the program. In this representation, each statement corresponds to a bytecode instruction (i.e., it is either a copy assignment `a=b` or a computation `a=b+c` that contains only one operator). We will use terms *statement* and *instruction* interchangeably, both meaning a statement in the three-address-code representation.

2.1 Abstract Dynamic Thin Slicing

The computation of costs of run-time values appears to be a problem which can be solved by associating a small amount of *tracking data* with each storage location and by updating this data as the corresponding location is written. A similar technique has been adopted, for example, in taint analysis [25], where the tracking data is a simple taint mark. In the setting of cost computation, the tracking data associated with each location records the cumulative cost of producing the value that is written into the location. For an instruction s, a simple approach of updating the tracking data for the left-hand-side variable is to store in it the sum of the tracking data for all right-hand-side variables, plus the cost of s itself.

However, this simple approach may double-count cost. To illustrate this, consider the example code shown in Figure 1. Using the taint-like flow tracking (shown in Figure 1 (a)), the cost t_b of producing the value in b is 8, which must be incorrect as there are only 5 instructions in the program. This is because the cost of c is included in the costs of both d and b. As the costs of c and d are added to obtain b's cost, c's cost is considered twice. While this difference is small for the example, it can quickly accumulate and become significant when the size of the program increases. For example, as we have observed, this cost can quickly grow and cause a 64-bit integer to overflow for even moderate-size applications. Furthermore, such dynamic flow tracking does not provide any additional information about the data flow and its usefulness for helping diagnose performance problems is limited.

To avoid double-counting in the computation of cost t_b, one could record all instruction instances before variable b is written and their dependences (as shown in Figure 1 (b)). Cost t_b can then be computed by traversing backward the dependence graph from instruction 4 and counting the number of instructions. There are many other dynamic analysis problems that have similar characteristics. These problems, for example, include `null` value propagation analysis [9], dynamic object type-state checking [2], event-based execution fast forwarding [41], copy chain profiling [34], etc. One common feature of this class of dynamic analyses is that they require additional trace information recorded for the purpose of diagnosis or debugging, as opposed to simpler approaches such as taint analysis. Because the solutions these analyses need to compute are history-related and can be obtained by traversing backward the recorded traces, we will refer to them as *backward dynamic flow* (BDF) problems. In general, BDF problems can be solved by dynamic slicing [14, 33, 37, 38, 40]. In our example of cost computation, the cost of a value produced by an instruction is essentially the size of the data-dependence-based backward dynamic slice starting from the instruction.



```
1 a = 0;        ------>  t_a = 1;
2 c = f(a);     ------>  t_c = t_f + 1;
3 d = c * 3;    ------>  t_d = t_c + 1;
4 b =  c + d;      --->  t_b = t_c + t_d + 1;
5 int f(int e){
6   return e >> 2; -->  t_f = t_e + 1;
7 }
```

Figure 1. An example illustrating the *double-counting* problem: (a)Example code and the step-wise updating of tracking data. t_i denotes the tracking data for variable i; (b)its dynamic data dependence graph where instructions are represented by their line numbers. An edge represents a def-use relationship between two instructions.

In dynamic slicing, the instrumented program is first executed to obtain an execution trace with control flow and memory reference information. At a pointer dereference, both the data that is referenced and the pointer value (i.e., the address of the data) are captured. Our technique considers only data dependences and the control predicates are treated in a special way as described later. Based on a dynamic data dependence graph inferred from the trace, a slicing algorithm is executed. Let $\mathcal{I}$ be the domain of static instructions and $\mathcal{N}$ be the domain of natural numbers.

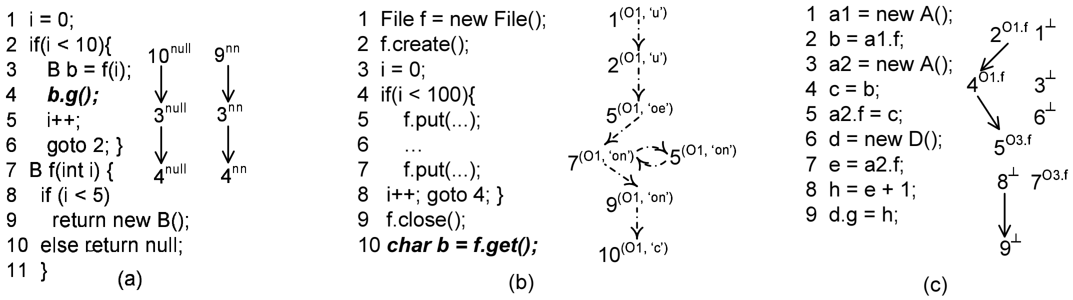
DEFINITION 1. (Dynamic Data Dependence Graph). *A dynamic data dependence graph* $(\mathcal{V}, \mathcal{E})$ *has node set* $\mathcal{V} \subseteq \mathcal{I} \times \mathcal{N}$*, where each node is a static instruction annotated with an integer* j*, representing the* j*-th occurrence of this instruction in the trace. An edge from* a^j *to* b^k $(a, b \in \mathcal{I}$ *and* $j, k \in \mathcal{N})$ *shows that the* j*-th occurrence of* a *writes a location that is then used by the* k*-th occurrence of* b*, without an intervening write to that location. If an instruction accesses a heap location through* $v.f$*, the reference value in stack location* v *is also considered to be used.*

Thin slicing [30] is a static technique that focuses on statements that flow values to the seed, ignoring the uses of base pointers. In this paper, the technique is re-stated for dynamic analysis. A *thin data dependence graph*, formed from the execution trace, has exactly the same set of nodes as its corresponding dynamic data dependence graph. However, for an access $v.f$, the base pointer value in v is not considered to be used. This property makes thin slicing especially attractive for computing costs for programs that make extensive use of object-oriented data structures—with traditional slicing, the cost of each data element retrieved from a data structure would include the cost of producing the object references that form the layout of the data structure, resulting in significant imprecision. A thin data dependence graph contains fewer edges and leads to smaller slices. Both for standard and thin dynamic slicing, the amount of memory required for representing the dynamic dependence graph cannot be bounded before or during the execution.

For some BDF problems, there exists a certain pattern of backward traversal that can be exploited for increased efficiency. Among the instruction instances that are traversed, equivalence classes can usually be seen. Each equivalence class is related to a certain property of an instruction from the program code, and distinguishing instruction instances in the same equivalence class (i.e., with the same property) does not affect the analysis precision. Moreover, it is only necessary to record one instruction instance online as the representative for that equivalence class, leading to significant space reduction of the generated execution trace. Several examples of such problems will be discussed shortly.

To solve such BDF problems, we propose to introduce the semantics of a target analysis into profiling, by defining a problem-

المنارة للاستشارات
www.manaraa.com



```
1  i = 0;
2  if(i < 10){
3     B b = f(i);
4     b.g();
5     i++;
6     goto 2; }
7  B f(int i) {
8    if (i < 5)
9      return new B();
10  else return null;
11  }
```

```
1  File f = new File();
2  f.create();
3  i = 0;
4  if(i < 100){
5      f.put(...);
6      ...
7      f.put(...);
8   i++; goto 4; }
9   f.close();
10 char b = f.get();
```

```
1  a1 = new A();
2  b = a1.f;
3  a2 = new A();
4  c = b;
5  a2.f = c;
6  d = new D();
7  e = a2.f;
8  h = e + 1;
9  d.g = h;
```

Figure 2. Data dependence graphs for three BDF problems. Line numbers are used to represent the corresponding instructions. Arrows with solid lines are def-use edges. (a) Null origin tracking. Instructions that handle primitive-typed values are omitted; (b) Typestate history recording; arrows with dashed lines represent "next-event" relationships. (c) Extended copy profiling; O_i denotes the allocation site at line i.

specific bounded abstract domain $\mathcal{D}$ containing identifiers that define equivalence classes in $\mathcal{N}$. An unbounded subset of elements in $\mathcal{N}$ can be mapped to an element in $\mathcal{D}$. For a particular instruction $a \in \mathcal{I}$, an abstraction function $f_a : \mathcal{N} \to \mathcal{D}$ is used to map a^j, where $j \in \mathcal{N}$, to an abstracted instance a^d. This yields an abstraction of the dynamic data dependence graph. For our purposes we are interested in thin slicing. The corresponding dependence graph will be referred as an *abstract thin data dependence graph*.

DEFINITION 2. (Abstract Thin Data Dependence Graph). *An abstract thin data dependence graph ($\mathcal{V}'$, $\mathcal{E}'$, $\mathcal{F}$, $\mathcal{D}$) has node set $\mathcal{V}' \subseteq \mathcal{I} \times \mathcal{D}$, where each node is a static instruction annotated with an element $d \in \mathcal{D}$, denoting the equivalence class of instances of the instruction mapped to d. An edge from a^j to b^k ($a, b \in \mathcal{I}$ and $j, k \in \mathcal{D}$) shows that an instance of a mapped to a^j writes a location that is used by an instance of b mapped to b^k, without an intervening write to that location. If an instruction accesses the heap through $v.f$, the base pointer value v is* not *considered to be used. $\mathcal{F}$ is a family of abstraction functions f_a, one per instruction $a \in \mathcal{I}$.*

For simplicity, we will use "dependence graph" to refer to the abstract thin data dependence graph defined above. Note that for an array element access, the index used to locate the element is still considered to be used. The number of static instructions (i.e., the size of $\mathcal{I}$) is relatively small even for large-scale programs, and by carefully selecting domain $\mathcal{D}$ and abstraction functions f_a, it is possible to require only a small amount of memory for the graph and yet preserve necessary information needed for a target analysis.

Many BDF problems exhibit bounded-domain properties. Their analysis-specific dependence graphs can be obtained by defining the appropriate abstraction functions. The following examples show a few analyses and their formulations in our framework.

Propagation of `null` values When a `NullPointerException` is observed in the program, this analysis locates the program point where the `null` value starts propagating and the propagation flow. Compared to existing `null` value tracking approaches (e.g., [9]) that track only the origin of a `null` value, this analysis also provides information about how this value flows to the point where it is dereferenced, allowing the programmer to quickly track down the bug. Here, $\mathcal{D}$ contains two elements *null* and *not_null*. Abstraction function $f_a(j)$ = *null* if a^j produces `null` and *not_null* otherwise. Based on the dependence graph, the analysis traverses backward from node a^{null} where $a \in \mathcal{I}$ is the instruction whose execution causes the `NullPointerException`. The node that is annotated with *null* and that does not have incoming edges represents the instruction that created the `null` value originally. Figure 2 (a) shows an example of this analysis. Annotation `nn` denotes *not_null*. A `NullPointerException` is thrown when line 4 is reached.

Recording typestate history Proposed in QVM [2], this analysis tracks the typestates of the specified objects and records the history of state changes. When the typestate protocol of an object is violated, it provides the programmer with the recorded history. Instead of recording every single event in the trace, a summarization approach is employed to merge these events into DFAs. We show how this analysis can be formulated as an abstract slicing problem, and the DFAs can be easily derived from the dependence graph.

Domain $\mathcal{D}$ is $\mathcal{O} \times \mathcal{S}$, where $\mathcal{O}$ is a specified set of allocation sites (whose objects need to be tracked) and $\mathcal{S}$ is a set of predefined states $s_0, s_1, \ldots, s_n$ of the objects created by the allocation sites in $\mathcal{O}$. Abstraction function $f_a(j)$ = (`alloc`(a^j), `state`(a^j)) if instruction instance a^j invokes a method on an object $\in \mathcal{O}$, and the method can cause the object to change its state. The function is undefined otherwise (i.e., all other instructions are not tracked). Here `alloc` is a function that returns the allocation site of the receiver object at a^j, and function `state` returns the state of this object immediately before a^j. The state can be stored as a tag of the object, and updated when a method is invoked on this object.

An example is shown in Figure 2 (b). Consider the object O_1 created at line 1, with states 'u' (uninitialized), 'oe' (opened and empty), 'on' (opened but not empty), and 'c' (closed). Arrows with dashed lines denote the "next-event" relationships. These relationships are added to the graph for constructing the DFA described in [2], and they can be easily obtained by memorizing the last event on each tracked object. When line 10 is executed, the typestate protocol is violated because the file is read after it is closed. The programmer can easily identify the problem when she inspects the graph and finds that line 10 is executed on a closed file. While the example shown in Figure 2 (b) is not strictly a dependence graph, the "next-event" edges can be conceptually thought of as def-use edges between nodes that write and read the object state tag.

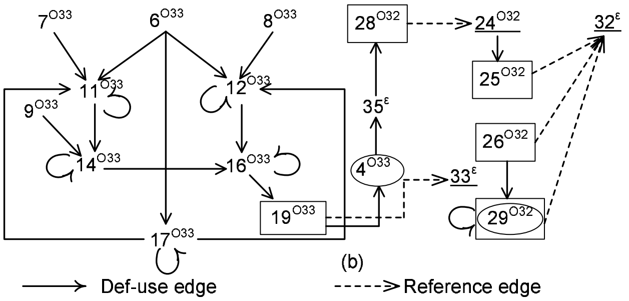
Extended copy profiling Work in [34] describes how to profile copy chains that represent the transfer of the same data without any computation. Nodes in a copy chain are fields of objects represented by their allocation sites (e.g., $O_i.f$). An edge connects two field nodes, abstracting away intermediate stack copies. Each stack variable has a shadow stack location, which records the object field from which its value originated. A more powerful version of this analysis is to include intermediate stack nodes along copy chains, because they are important for understanding the methods through which the values are transferred.

Domain $\mathcal{D}$ is $\mathcal{O} \times \mathcal{P}$, where $\mathcal{O}$ is the set of allocation sites and $\mathcal{P}$ is the set of field identifiers. A special element $\perp \in \mathcal{D}$ shows that the data does not come from any field (e.g., it is a constant or

المنارة للاستشارات
www.manaraa.com

```
1 class A{
2   int t = 0;
3   int getTotal(){
4       return this.t; }
5   void foo(int n){
6     int i = 0;
7     int a = 0;
8     int b = 0;
9     int total = 0;
10    if(i < n){
11       a = a + i;
12       b = b * i;
13       if(i < 500){
14        total = total + a;
15       else
16         total = total + b;
17       i = i + 1;
18      goto 10; }
19    this. t = total; }}
20 class IntList{
21   int[] arr;
22   int size;
23   List() {
24     int[] tmp = new int[100];
25     this.arr = tmp;
26     size = 0; }
27   void add(int i){
28     arr[size] = i;
29     this.size = this.size + 1;
30  }}
31 void main(){
32   IntList l = new IntList();
33   A o = new A();
34   o.foo(1000);
35   int s = o.getTotal();
36   l.add(s);
37   }
38  }
```

(a)



Node	*Freq*	*AC*		*Node*	*Freq*	*AC*
6^{O33}	1	1		12^{O33}	1000	2003
7^{O33}	1	1		14^{O33}	500	2504
8^{O33}	1	1		16^{O33}	500	5007
9^{O33}	1	1		17^{O33}	1000	1001
11^{O33}	1000	2003		19^{O33}	1	5008

(c)

Node	*1-RAC*	*1-RAB*	*2-RAC*	*2-RAB*
$O_{24}{}^{O32}$	2	0	2	0
$O_{32}{}^{\varepsilon}$	4	1	6	1
$O_{33}{}^{\varepsilon}$	5008	4	5008	4

(d)

Figure 3. (a) Code example. (b) Corresponding dependence graph G_{cost}; nodes in boxes/circles write/read heap locations; underlined nodes create objects. (c) Nodes in method `A.foo` (*Node*), their frequencies (*Freq*), and their abstract costs (*AC*); (d) Relative abstract costs i-*RAC* and benefits i-*RAB* for the three allocation sites; i is the level of reference edges considered.

a reference to a newly-created object). Abstraction function f_a (j) = `map(shadow(`a^j`))`, if a is a copy instruction, and $\perp$ otherwise. Function `shadow` maps an instruction instance to the tracking data (i.e., the object field $o.g$ from which the value originated) contained in the shadow location of its left-hand-side variable, and function `map` maps $o.g$ to its static abstraction $O_i.g$. An example is shown in Figure 2 (c). For instance, to identify the intermediate stack locations `b` and `c` in the copy chain between $O_1.f$ and $O_3.f$, one can traverse backward the dependence graph from $5^{O3.f}$ (which writes to f in the object created by O_3). The traversal follows nodes that are annotated with $O_1.f$, until it reaches the node $2^{O1.f}$ which reads that field.

Similarly to how a data flow analysis or an abstract interpreter employs static abstractions, abstract dynamic slicing uses abstract domains for dynamic data flow, recognizing that it is only necessary to distinguish instruction instances that are important for the client analysis. The rest of the paper focuses on using this approach to perform cost-benefit analyses that target a class of bloat problems.

2.2 Cost Computation

DEFINITION 3. *(*Absolute Cost*). Given a non-abstract thin data dependence graph G and an instruction instance a^j $(a \in \mathcal{I}, j \in \mathcal{N})$ that produces a value v, the* absolute cost *of v is the number of nodes that can reach a^j in G.*

Absolute costs are expensive to compute and it does not make much sense to present them to the programmer, unless they are aggregated in some meaningful way across instruction instances. In our approach the instructions are abstracted based on dynamic calling contexts. The contexts are represented with object sensitivity [17], which is well suited for modeling of object-oriented data structures.

A calling context is represented by a chain of static abstractions (i.e., allocation sites $O_i \in \mathcal{O}$) of the receiver objects for the invocations on the call stack. Domain $\mathcal{D}_{cost}$ contains all possible chains of allocation sites. Abstraction function $f_a(j)$ = `objCon(cs(`a^j`))`, where function `cs` takes a snapshot of the call stack when a^j is executed, and function `objCon` maps this snapshot to the corresponding chain of allocation sites O_i for the run-time receiver objects. $\mathcal{D}_{cost}$ is not finite in the presence of recursion, and even for a recursion-free program its size is exponential. We limit the size of $\mathcal{D}_{cost}$ further to be a fixed number s (short for "slots"), specified by the user as a parameter of the profiling tool. Now the domain is simply the set of integers 0 to $s-1$. An encoding function `h` is used to map an allocation site chain to such an integer; the description of `h` will be presented shortly. With this approach, the amount of memory required for the analysis is linear in program size.

Each node in the dependence graph is annotated with an integer, representing the execution frequency of the node. Based on these frequencies, an *abstract cost* for each node can be computed as an approximation of the total costs of values produced by the instruction instances represented by the node.

DEFINITION 4. (Abstract Cost). *Given a dependence graph G_{cost}, the abstract cost of a node n^k is $\Sigma_{a^j | a^j \rightsquigarrow n^k}$ freq(a^j), where $a^j \rightsquigarrow n^k$ if there is a path from a^j to n^k in G_{cost}, or $a^j = n^k$.*

Example Figure 3 shows a code example and its dependence graph for cost computation. While some statements (line 29) may correspond to multiple bytecode instructions, they are still considered to have unit costs. These statements are shown for illustration purposes and will be broken into multiple ones by our tool.

All nodes are annotated with their object contexts (i.e., elements of $\mathcal{D}_{cost}$). For ease of understanding, the contexts are shown in their original forms, and the tool actually uses the encoded forms (through function `h`). Nodes in boxes represent instructions that write heap locations. Dashed arrows represent reference edges; these edges can be ignored for now. The table shown in part (c) lists nodes for the execution of method `A.foo` (invoked by the call site at line 34), their frequencies, and their abstract costs.

The abstract cost of a node computed by this approach may be larger than the exact sum of absolute costs of the values produced by the instruction instances represented by the node. This is because for a node a such that $a \rightsquigarrow n$, there may not exist any dependences between some instruction instances of a and some instruction instances of n. This difference can be large when the abstract cost is computed after traversing long dependence graph paths, and the imprecision gets magnified. More importantly, this cost represents the *cumulative effort* that has been made from the very beginning of the execution to produce the values. It may still not make much sense for the programmer to diagnose problems using abstract costs, as it is almost certain that nodes representing instructions executed later have larger costs than those representing instructions executed earlier. In Section 3, we address this problem

المنارة للاستشارات

by computing a *relative abstract cost*, which measures execution bloat at the object level by traversing dependence graph paths connecting nodes that read and write object fields.

Special nodes and edges in G_{cost} To measure execution bloat, we augment the graph with two special kinds of nodes: *predicate* nodes and *native* nodes, both representing the consumption of data. A predicate node is created for each `if` statement, and a native node is created for each call site that invokes a native method. These nodes do not have associated contexts. In addition, we mark nodes that allocate objects (underlined in Figure 3 (b)), that read heap locations (nodes in circles), and that write heap locations (nodes in boxes). These nodes are later used to identify object structures.

Reference edges are used to represent reference relationships. For each heap store $a.f = b$, a reference edge is created to connect the node representing this store (i.e., a boxed node) and the node allocating the object that flows to a (i.e., an underlined node). For example, there exists a reference edge from $28^{O_{32}}$ to $24^{O_{32}}$, because $24^{O_{32}}$ allocates the array object and $28^{O_{32}}$ stores an integer to the array (which is similar to writing an object field). These edges will be used to aggregate costs for individual heap locations to form costs for objects and data structures.

2.3 Construction of G_{cost}

Selecting encoding function $\hbar$ There are two steps in mapping an allocation site chain to an integer $d \in \mathcal{D}_{cost}$ (i.e., $[0, \ldots, s–1]$). The first step is to encode the chain into a *probabilistically unique value* that will accurately represent the original object context chain. An encoding function proposed in [6] is adapted to perform this mapping: $\mathtt{g}_i = 3 * \mathtt{g}_{i-1} + o_i$, where o_i is the i-th allocation site ID in the chain and $\mathtt{g}_{i-1}$ is the probabilistic context value computed for the chain prefix with length $i-1$. While simple, this function exhibits very small context conflict rate, as demonstrated in [6]. In the second step, this encoded value is mapped to an integer in the range $[0, \ldots, s-1]$ using a simple `mod` operation.

Profiling for constructing G_{cost} Instead of recording the full execution trace and building a dependence graph offline, we use an online approach that combines dynamic flow tracking and slicing. The key issue is to identify the data dependences online, which can be done using shadow locations [23, 24]. For each location l, a shadow location l' contains the address of the dependence graph node representing the instruction instance that wrote the last value of l. When an instruction is executed, the node n to which this instruction instance is mapped is retrieved, and all nodes m_i that last wrote the locations read by the instruction are identified. Edges are then added between n and each m_i.

For a local variable, its shadow location is a new local variable on the stack. For heap locations we use a *shadow heap* [24] that has the same size as the Java heap. To enable quick access, there is a predefined distance *dist* between the starting addresses of these two heaps. For a heap location l, the address of l' can be quickly obtained as l + *dist*. A *tracking stack* is maintained in parallel with the call stack to pass data dependence relationships across calls. For each invocation, the tracking stack also passes the receiver object chain for the caller. The next context is obtained by concatenating the caller's chain and the allocation site of `this`.

Instrumentation semantics Figure 4 shows a list of inference rules defining the instrumentation semantics. Each rule is of the form $\mathsf{V}, \mathsf{E}, \mathsf{H}, \mathsf{S}, \mathsf{P}, \mathsf{T} \Rightarrow^{a:i=\cdots} \mathsf{V}', \mathsf{E}', \mathsf{H}', \mathsf{S}', \mathsf{P}', \mathsf{T}'$ with unprimed and primed symbols representing the state before and after the execution of statement a. In cases where a set does not change (e.g., when $\mathsf{S} = \mathsf{S}'$), it is omitted. Node domain V contains nodes of the form $a^{h(c)}$, where a denotes the instruction and $h(c)$ denotes the encoded integer of the object context c. Edge domain $\mathsf{E} : \mathsf{V} \times \mathsf{V}$ is a relation containing dependence relationships of the form $a^l \triangleright k^n$, which represents that an instance of a abstracted as a^l is data dependent on an instance of k abstracted as k^n. Shadow environment $\mathsf{S} : \mathsf{M} \rightarrow \mathsf{V}$ maps a run-time storage location to the content in its corresponding shadow location (i.e., to its tracking data). Here M is the domain of memory locations. For each location, its shadow location contains the (address of the) node that performs the most recent write to this location. Rules ASSIGN, COMPUTATION, PREDICATE, LOAD STATIC, and STORE STATIC update the environments in expected ways. In rule PREDICATE, instruction instances are not distinguished and the node is represented by a^{ϵ}.

ASSIGN

$$\frac{\mathsf{V}' = \mathsf{V} \cup \{a^{\mathtt{h}(c)}\} \quad \mathsf{S}' = \mathsf{S}[i \mapsto a^{\mathtt{h}(c)}] \\ \mathsf{E}' = \mathsf{E} \cup \{a^{\mathtt{h}(c)} \triangleright \mathsf{S}(k)\}}{\mathsf{V}, \mathsf{E}, \mathsf{S} \Rightarrow^{a:i=k} \mathsf{V}', \mathsf{E}', \mathsf{S}'}$$

COMPUTATION

$$\frac{\mathsf{V}' = \mathsf{V} \cup \{a^{\mathtt{h}(c)}\} \quad \mathsf{S}' = \mathsf{S}[i \mapsto a^{\mathtt{h}(c)}] \\ \mathsf{E}' = \mathsf{E} \cup \{a^{\mathtt{h}(c)} \triangleright \mathsf{S}(k)\} \cup \{a^{\mathtt{h}(c)} \triangleright \mathsf{S}(l)\}}{\mathsf{V}, \mathsf{E}, \mathsf{S} \Rightarrow^{a:i=k \oplus l} \mathsf{V}', \mathsf{E}', \mathsf{S}'}$$

PREDICATE

$$\frac{\mathsf{V}' = \mathsf{V} \cup \{a^{\epsilon}\} \quad \mathsf{S}' = \mathsf{S} \\ \mathsf{E}' = \mathsf{E} \cup \{a^{\epsilon} \triangleright \mathsf{S}(i)\} \cup \{a^{\epsilon} \triangleright \mathsf{S}(k)\}}{\mathsf{V}, \mathsf{E}, \mathsf{S} \Rightarrow^{a:if\ (i>k)\{\ldots\}} \mathsf{V}', \mathsf{E}', \mathsf{S}'}$$

LOAD STATIC

$$\frac{\mathsf{V}' = \mathsf{V} \cup \{a^{\mathtt{h}(c)}\} \quad \mathsf{S}' = \mathsf{S}[i \mapsto a^{\mathtt{h}(c)}] \\ \mathsf{E}' = \mathsf{E} \cup \{a^{\mathtt{h}(c)} \triangleright \mathsf{S}(A.f)\}}{\mathsf{V}, \mathsf{E}, \mathsf{S} \Rightarrow^{a:i=A.f} \mathsf{V}', \mathsf{E}', \mathsf{S}'}$$

STORE STATIC

$$\frac{\mathsf{V}' = \mathsf{V} \cup \{a^{\mathtt{h}(c)}\} \quad \mathsf{S}' = \mathsf{S}[A.f \mapsto a^{\mathtt{h}(c)}] \\ \mathsf{E}' = \mathsf{E} \cup \{a^{\mathtt{h}(c)} \triangleright \mathsf{S}(i)\}}{\mathsf{V}, \mathsf{E}, \mathsf{S} \Rightarrow^{a:A.f=i} \mathsf{V}', \mathsf{E}', \mathsf{S}'}$$

ALLOC

$$\frac{\mathsf{V}' = \mathsf{V} \cup \{a^{\mathtt{h}(c)}\} \quad \mathsf{S}' = \mathsf{S}[i \mapsto a^{\mathtt{h}(c)}] \\ \mathsf{H}' = \mathsf{H}[a^{\mathtt{h}(c)} \mapsto ('U', (new\ X)^{\mathtt{h}(c)}, '\ ')] \\ \mathsf{P}' = \mathsf{P}[o \mapsto (new\ X)^{\mathtt{h}(c)}]}{\mathsf{V}, \mathsf{H}, \mathsf{S}, \mathsf{P} \Rightarrow^{a:i=new\ X} \mathsf{V}', \mathsf{H}', \mathsf{S}', \mathsf{P}'}$$

LOAD FIELD

$$\frac{\mathsf{V}' = \mathsf{V} \cup \{a^{\mathtt{h}(c)}\} \quad \mathsf{S}' = \mathsf{S}[i \mapsto a^{\mathtt{h}(c)}] \\ \mathsf{E}' = \mathsf{E} \cup \{a^{\mathtt{h}(c)} \triangleright \mathsf{S}(o_v.f)\} \\ \mathsf{H}' = \mathsf{H}[a^{\mathtt{h}(c)} \mapsto ('C', \mathsf{P}(o_v), f)]}{\mathsf{V}, \mathsf{E}, \mathsf{H}, \mathsf{S} \Rightarrow^{a:i=v.f} \mathsf{V}', \mathsf{E}', \mathsf{H}', \mathsf{S}'}$$

STORE FIELD

$$\frac{\mathsf{V}' = \mathsf{V} \cup \{a^{\mathtt{h}(c)}\} \quad \mathsf{S}' = \mathsf{S}[o_v.f \mapsto a^{\mathtt{h}(c)}] \\ \mathsf{E}' = \mathsf{E} \cup \{a^{\mathtt{h}(c)} \triangleright \mathsf{S}(i)\} \\ \mathsf{H}' = \mathsf{H}[a^{\mathtt{h}(c)} \mapsto ('B', \mathsf{P}(o_v), f)]}{\mathsf{V}, \mathsf{E}, \mathsf{H}, \mathsf{S} \Rightarrow^{a:v.f=i} \mathsf{V}', \mathsf{E}', \mathsf{H}', \mathsf{S}'}$$

METHOD ENTRY

$$\frac{\mathsf{S}' = \mathsf{S}[t_i \mapsto \mathsf{T}(i)] \text{ for } 1 \leq i \leq n \\ \mathsf{T}' = (\mathsf{T}(n+1) \circ \textsc{AllocId}\,(\mathsf{P}(o_{this})), \mathsf{T}(n+1), \mathsf{T}(n+2), \ldots)}{\mathsf{S}, \mathsf{T} \Rightarrow^{a:m(t_1, t_2, \ldots, t_n)} \mathsf{S}', \mathsf{T}'}$$

RETURN

$$\frac{\mathsf{T}' = (\mathsf{S}(i), \mathsf{T}(2), \mathsf{T}(3), \ldots)}{\mathsf{T} \Rightarrow^{a:return\ i} \mathsf{T}'}$$

Figure 4. Inference rules defining the run-time effects of instrumentation.

المنارة للاستشارات

www.manaraa.com

Rules ALLOC, LOAD FIELD and STORE FIELD additionally update heap effect environment H, which is used to construct reference edges in G_{cost}. $\mathsf{H} : \mathsf{V} \rightarrow \mathsf{Z}$ maps a node $a^l \in \mathsf{V}$ to a heap effect triple (*type*, *alloc*, *field*) $\in$ domain Z of heap effects. Here, *type* can be $'U'$ (i.e., underlined) representing the allocation of an object, $'B'$ (i.e., boxed) representing a field store, or $'C'$ (i.e., circled) representing a field load. *alloc* and *field* denote the object and the field on which the effect occurs. For instance, triple $('U', O, ' ')$ means that a node contains an allocation site O, while triple $('B', O, f)$ means that a node writes to field f of an object created by allocation site O. A reference edge can be added between a (store) node with effect $('B', O, *)$ and another (allocation) node with effect $('U', O, ' ')$, where $*$ represents any field name. In order to perform this matching, we need to provide access to the allocation site ID for each run-time object. This is done using tag environment P that maps a run-time object to its allocation site ID.

However, the reference edge could be spurious if the store node and the allocation node are connected using only allocation site ID O, because the two effects (i.e., $'B'$ and $'U'$) could occur on different instances created by O. To improve the precision of the client analyses, object context is used again to annotate allocation sites. For example, in rule ALLOC, H is updated with effect triple $('U', (new\ X)^{\mathtt{h}(c)}, ' ')$, where the allocation site $new\ X$ is annotated with the encoded context integer $\mathtt{h}(c)$. This triple matches only (store) node with effect $('B', (new\ X)^{\mathtt{h}(c)}, *)$, and many spurious reference edges can thus be eliminated. In rule ALLOC, $(new\ X)^{\mathtt{h}(c)}$ is used to tag the newly-created run-time object o (by updating tag environment P), and this information will be retrieved later when o is dereferenced. In rules LOAD FIELD and STORE FIELD, o_v denotes the run-time object that variable v points to. $\mathsf{P}(o_v)$ is used to retrieve the allocation site (annotated with the context) of o_v, which is previously set as o_v's tag upon its allocation.

The last two rules show the instrumentation semantics at the entry and the return site of a method, respectively. At the entry of a method with n parameters, tracking stack T contains the tracking data for the actual parameters of the call, as the n top elements $\mathsf{T}(1), \ldots, \mathsf{T}(n)$, followed by the receiver object chain for the caller of the method (as element $\mathsf{T}(n+1)$). In rule METHOD ENTRY, the tracking data for a formal parameter t_i is updated with the tracking data for the corresponding actual parameter (stored in $\mathsf{T}(i)$). The new object context is computed by applying concatenation operator $\circ$ to the old chain $\mathsf{T}(n+1)$ and the allocation site of the run-time receiver object o_{this} pointed to by *this* (or an empty string if the current method is static). Function ALLOCID removes the context annotation from the tag of o_{this}, leaving only the allocation site ID. The stack is updated by removing the tracking data for the actuals, and storing the new context on the top of the stack. This new context is available for use by all rules applied in the body of the method (denoted by c in those rules). At the return site, T is updated to remove the current context and to store the tracking data for the return variable i.

The rule for call sites is not shown in Figure 4, as it requires splitting a call site into a *call* part and a *return* part, and reasoning about both of them. Immediately before the call, the tracking data for the actual parameters is pushed on tracking stack T. Immediately after the call, the tracking data for the returned value is popped from T and used to update the dependence graph and the shadow location for the left-hand-side variable at the call site. If the method invoked at the call site is a native method, we create a node (without context) for it, and add edges between each node contained in the shadow locations of the actual parameters and this node, representing that the values of parameters are consumed by this native method.

Implementation of P   A natural idea of implementing object tagging is to save the tag in the header of each object (i.e., the header usually has unused space). However, in the J9 VM that we use, this 64-bit header cannot be modified. To solve this problem, the corresponding 64 bits on the shadow heap are used to store the object tag. Hence, although environments P and S have different mathmatical meanings, both are implemented using shadow locations.

3. Relative Object Cost-Benefit Analysis

This section describes a novel diagnosis technique that identifies data structures with high cost-benefit rates. As discussed in Section 4, this analysis effectively uncovers significant optimization opportunities in six large real-world applications. We propose to compute a *relative abstract cost* for an object, which measures the effort of constructing the object from data already available in fields of other objects (rather than the cumulative effort from the beginning of the execution). Similarly, we compute a *relative abstract benefit* for an object, which explains how the data contained in the object is used to construct other objects. These metrics can help a programmer pinpoint specific objects that are expensive to construct (e.g., there are large costs of computing the data being written into this object) but are not very useful (e.g, the only use of this object is to make a clone of it and then invoke methods on the clone).

We first develop an *object cost-benefit analysis* that aggregates relative costs and benefits for individual fields of an object in order to compute the cost and benefit for the object itself. Next, the cost and benefit for a *higher-level data structure* is obtained in a similar manner, by gathering costs and benefits of lower-level objects/data structures accessible through reference edges.

3.1 Analysis Algorithm

DEFINITION 5. (Relative Abstract Cost). *Given G_{cost}, the heap-relative abstract cost (HRAC) of a node n^k is $\Sigma_{a^j\,|\,a^j \rightharpoonup n^k}$ $freq(a^j)$, where $a^j \rightharpoonup n^k$ if $a^j \rightsquigarrow n^k$ and there exists a path from a^j to n^k such that no node on the path reads from a static or object field. The relative abstract cost (RAC) for an object field represented by $O^d.f$ is the average HRAC of store nodes n^k that write to $O^d.f$.*

Consider the entire flow of a piece of data (from the input of the program to its output) during the execution. This flow consists of multiple hops of data transformations among heap locations. Each hop performs the following three steps: reading values from heap locations, performing stack copies and computations on them, and writing the results to other heap locations. Consider one single hop with multiple sources and one target along the flow, which reads values from heap locations $l_1, l_2, \ldots, l_n$, transforms them to produce a new value, and writes it back to heap location l'. The RAC of l' measures the amount of work needed (on the stack) to complete this hop of transformations.

The computation of HRAC for a node n^k requires a backward traversal from n^k, which finds all nodes on the paths between each heap-reading node and n^k, and calculates the sum of their frequencies. For example, the HRAC for node 35^{ϵ} is only 1 (instead of 5010), because the node depends directly on a node (i.e., $4^{O_{33}}$) that reads heap location `this.t`. The RAC for a heap location is the average HRAC of the nodes that can write this location. For example, the RAC for $O_{33}^{\epsilon}.t$ is the HRAC for $19^{O_{33}}$, which is 5008. The RAC for $O_{24}^{O_{32}}.ELM$ (i.e., the elements of the array object) is 2, which equals the HRAC of node $28^{O_{32}}$ that writes this field.

DEFINITION 6. (Relative Abstract Benefit). *Given G_{cost}, the heap-relative abstract benefit (HRAB) of a node n^k is $\Sigma_{a^j\,|\,n^k \rightharpoondown a^j}$ $freq(a^j)$, where $n^k \rightharpoondown a^j$ if $n^k \rightsquigarrow a^j$ and there exists a path from n^k to a^j such that no node on the path writes to a static or object field. The*

www.manaraa.com

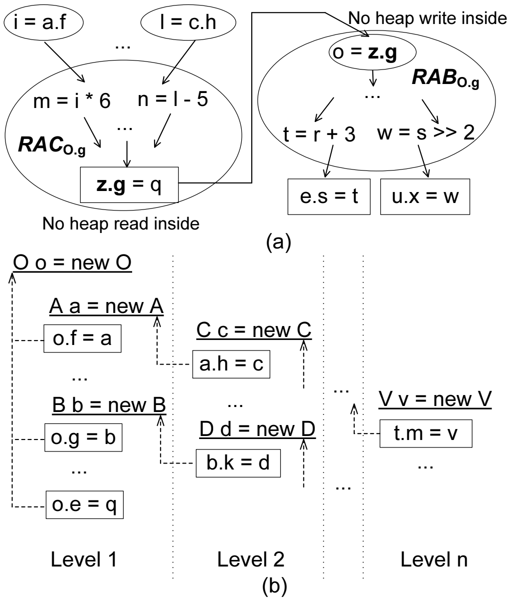
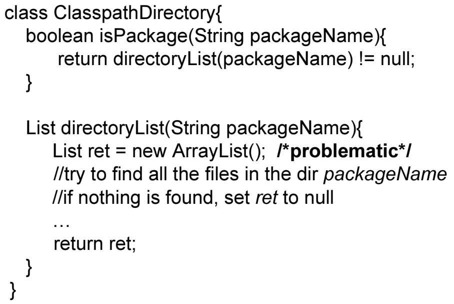


Figure 5. (a) Relative abstract cost and benefit. Nodes considered in computing RAC and RAB for $O.g$ (where O is the allocation site for the object referenced by z) are included in the two circles, respectively; (b) Illustration of n-RAC and n-RAB for the object created by o = *new* O; dashed arrows are reference edges.

relative abstract benefit (RAB) for an object field represented by $O^d.f$ *is the average HRAB of load nodes* n^k *that read from* $O^d.f$.

Symmetric to the definition of RAC that focuses on how a heap value is *produced*, the RAB for l explains how a heap value is *consumed*. Consider again one single hop (but with one source and multiple targets) along the flow, which reads a value from location l, transforms this value (together with values read from other locations), and writes the results to a set of other heap locations $l'_1, l'_2, \ldots, l'_n$. The RAB of l measures the amount of work performed (on the stack) to complete this hop of transformations. For example, the RAB for $O_{33}^{\epsilon}.t$ is the HRAB of node $4^{O_{33}}$ that reads this field, which is 2 (because the relevant nodes a^j are only $4^{O_{33}}$ and 35^{ϵ}). Figure 5 (a) illustrates the computation of RAC and RAB.

This definition of benefit captures both the frequency and the complexity of data use. First, the more target heap values that the value read from l is used to (transitively) produce, the larger benefit location l can have for the construction of these other objects. Second, the more effort is made to transform the value from l to other heap values, the larger benefit l can have. This is because the purpose of writing a value into a heap location is, intuitively, to keep the value so that it can be reused later and the (heavy) cost of re-computing it can be avoided. Whether to store a value in a heap location is essentially a decision involving space-time tradeoffs. If l's value v can be easily converted to some other value v' and v' is immediately stored in another heap location (i.e., little computation performed), the benefit of keeping v in l becomes less obvious, since v and v' may differ slightly and it may not be necessary to use two different heap locations to cache them. In the extreme case where v' is simply a copy of v, the RAB for l is 1 and storing v is not desirable at all if the RAC for l is large. Special treatment is applied to consumer nodes: we assign a large RAB to a heap location if the value it contains can flow to a predicate or a native node. This means the value contributes to control decision making or is used by the JVM, and thus benefits the overall execution.


```
class ClasspathDirectory{
   boolean isPackage(String packageName){
        return directoryList(packageName) != null;
   }

   List directoryList(String packageName){
        List ret = new ArrayList();  /*problematic*/
        //try to find all the files in the dir packageName
        //if nothing is found, set ret to null
        …
        return ret;
   }
}
```

Figure 6. Real-world example that our analysis found from `eclipse`.

DEFINITION 7. (n-RAC and n-RAB). *Consider an object reference tree* RT_n *of height* n *rooted at* O^d. *The* n*-RAC for* O^d *is the sum of the RACs for all fields* $O_i^k.f$, *such that both* O_i^k *and the object* $O_i^k.f$ *points to are in* RT_n. *Similarly, the* n*-RAB for* O^d *is the sum of the RABs for all such fields* $O_i^k.f$.

The object reference (points-to) tree can be constructed by using reference edges in the dependence graph, and by removing cycles and nodes more than n reference edges away from O^d. We aggregate the RACs and RABs for individual fields through the tree edges to form the RACs and RABs for objects (when $n = 1$) and high-level data structures (when $n > 1$). Figure 5 (b) illustrates n-RAC and n-RAB for an object created by o = *new* O. The n-RAC(RAB) for this object includes the RAC(RAB) of each field written by a boxed node (i.e., heap store) shown in the figure. For all case studies and experiments, n = 4 was used as this is the reference chain length for the most complex container classes in the Java collection framework (i.e., `HashSet`).

Table (d) in Figure 3 shows examples of 1- and 2- RACs and RABs. Both the 1-RAB and the 2-RAB for $O_{24}^{O_{32}}$ are 0, because the array element is never used in the code. Objects O_{32}^{ϵ} and O_{33}^{ϵ} have large cost-benefit rates, which indicates the existence of wasteful operations. This is indeed the case in this example: for O_{32}^{ϵ}, there is an element added but never retrieved; for O_{33}^{ϵ}, there is a large cost of computing the value stored in its field `t`, and the value is copied to another heap location (in `IntList`) immediately after it is calculated. The creation of object O_{33}^{ϵ} is not beneficial at all because this value could have been stored directly to the array.

Finding bloat Several usage scenarios are intended for this cost-benefit analysis. First, it can find long-lived objects that are written much more frequently than being read. Second, it can find containers that contain many more objects than they should. These containers are often the sources of memory leaks. The analysis can find that they have large RAC/RAB rates because few elements are retrieved and assigned to other heap locations. Third, it can find allocation sites that create large volumes of temporary (short-lived) objects. These objects are often created simply to carry data across method invocations. Data that is computed and written into them is read somewhere else and assigned to other object fields. This simple use of the data causes these objects to have large cost-benefit rates. The next section shows that our tool finds all three categories of problems in real-world Java applications.

Real-world example Figure 6 shows a real-world example that illustrates how our analysis works. An object with high costs and low benefits is highlighted in the figure. The code in the example is extracted from `eclipse` 3.1.2, a popular Java development tool. Method `isPackage` returns true/false based on whether the given package name corresponds to an actual Java package. This method is implemented by calling (reusing) `directoryList` which in-

www.manaraa.com

vokes many other methods to compute a list of files and directories under the package specified by the parameter. `isPackage` then returns whether the list computed by `directoryList` is `null`. While the reference to list `ret` is used in a predicate, its fields are not read and do not participate in computations. Hence, when the RACs and RABs for its fields are aggregated based on the object hierarchy, the imbalance between the cost and benefit for the entire `List` data structure can be seen. To optimize this case, we created a specialized version of `directoryList`, which returns immediately when the package corresponding to the given name is found.

3.2 Comparison to Other Design Choices

Cost/benefit for computation vs cost/benefit for cache Note that the relative cost and benefit for an object are essentially measured in terms of the *computations* that produce values written into the object. The goal of this analysis is to find objects such that they contain (relatively) useless values and these values are produced by (relatively) expensive operations. Upon identifying these operations, the user may find more efficient ways to achieve the same functionality. This is orthogonal to measuring the usefulness of a data structure as a *cache*, where the cost of the cache should include only the instructions executed to create the data structure itself (i.e., without the cost of computing the values being cached) and the benefit should be (re-)defined as a function of the amount of work cached and the number of times the cached values are used. It would be interesting to investigate, in future work, how these new definitions of cost and benefit can be used to find inappropriately-used caches.

Single-hop cost/benefit vs multi-hop cost/benefit The analysis limits the scope of tracked data flow to one single hop—that is, reading data from the heap, transforming it through stack locations, and writing the results back to the heap. While this design choice can produce easy-to-understand reports, it could miss problematic data structures because of its "short-sightedness". For example, our tool may consider a piece of data that is ultimately-dead to be appropriately used, because it is indeed involved in complex computations within the one hop seen by the analysis. To alleviate this problem, we have developed an additional analysis, based on G_{cost}, which identifies computations that can reach ultimately-dead values. Section 4 presents measurements of redundant computations based on this analysis.

A different way of handling this issue is to consider multiple hops when computing costs and benefits based on graph G_{cost}, so that more detailed information about data production and consumption can be obtained. For example, costs and benefits for an instruction can be recomputed by traversing multiple heap-to-heap hops on G_{cost} backward and forward, respectively, starting from the instruction. Of course, extending the inspected region of the data flow would make the report hard to verify as the programmer has to inspect larger program scopes to understand the detected problems. In future work, it would be interesting to compare empirically problems found using different scope lengths, and to design particular tradeoffs between the scope length considered and the difficulty of explaining the report.

Considering vs ignoring control decision making Our analysis does not consider the effort of making control decisions as part of the costs of computing values under these decisions. The major reason is that by doing so we could potentially include the costs of computing many values that are irrelevant to the value of interest into the cost of that value, leading to imprecise and hard-to-understand reports. However, ignoring this effort of control decision making could lead to information loss. For example, the tool may miss problematic objects due to the underestimation of the cost of constructing them. In future work, we will also be interested in accounting for this effort, and investigating the relationship between the scope of control flow decisions considered (e.g., the closest n predicates on which an instruction is control-dependent) and the usefulness of the analysis output.

Other analyses Graph G_{cost} (annotated with other information) can be used as basis for discovering a variety of performance-related program properties. For example, we have implemented a few clients that can answer specific performance-related queries. These clients include an analysis that computes method-level costs (i.e., the cost of producing the return value of a method relative to its inputs), an analysis that detects locations that are rewritten before being read, an analysis that identifies nodes producing always-true or always-false predicate conditions, and an analysis that searches for problematic collections by ranking collection objects based on their RAC/RAB rates. While these analyses are currently implemented inside a JVM, they could be easily migrated to an offline heap analysis tool that provides user-friendly interfaces and improved operability (i.e., the JVM only needs to write G_{cost} to external storage).

4. Evaluation

We have performed a variety of studies with our technique using the DaCapo benchmark set [4], which contains 11 programs in its original version (from `antlr` to `eclipse` in Table 1) and an additional set of 7 programs in its new (beta) release (from `avrora` to `tradesoaps`). We were able to run our tool on all these 18 large programs, including both client and server applications. 16 programs (except `tradesoap` and `tradebeans`) were executed with their large workloads. `tradesoap` and `tradebeans` were run with their default workloads, because these two benchmarks are not stable enough and running them with large workloads can fail even without our tool. All experiments were conducted on a 1.99GHz Dual Core machine. The evaluation has several components: cost graph characteristics, evaluation of the time and space overhead of the tool, the measurement of bloat based on nodes producing dead values, and six case studies that describe problems found by the tool in real applications.

4.1 G_{cost} characteristics and bloat measurement

Parts (a) and (b) in Table 1 report, for two different values of s (the number of slots for each object used to represent context), the numbers of nodes and edges in G_{cost}, as well as the space overheads and the time overheads of the tool. Note that all programs can successfully execute when we increase s to 32, while the offline traversal of the graph (to generate statistics) can make the tool run out of memory for some large programs. The space overhead does not include the size of shadow heap, which is 500Mb for all programs. Note that the shadow heap is *not* compulsory for using our technique. For example, it can be replaced by a global hash table that maps each object to its tracking data (and an object entry is removed when the object is garbage collected). The choice of shadow heap in our work is just to allow quick access to the tracking information. When the number of context slots s grows from 8 to 16, the space overhead increases while the running time is almost not affected. The instrumentation significantly increases the running times (i.e., 71× slowdown on average for $s = 8$ and 72× for $s = 16$ when the whole-program tracking is enabled). This is because (1) G_{cost} is updated at each instruction instance and (2) the creation of G_{cost} nodes and edges needs to be synchronized to guarantee that the tool is race-free. It was an intentional decision *not* to focus on the performance of the profiling, but instead to focus on the collected information and on demonstrating that the results are useful for finding bloat in real-world programs. One effective way of reducing overhead is to choose only relevant components

المنارة للاستشارات
www.manaraa.com

Program	(a) $s = 8$					(b) $s = 16$					(c) Bloat measurement for $s = 16$			
	#N(K)	*#E*(K)	*M*(Mb)	*O*(×)	*CR*(%)	*#N*(K)	*#E*(K)	*M*(Mb)	*O*(×)	*CR*(%)	*#I*(B)	*IPD*(%)	*IPP*(%)	*NLD*(%)
antlr	183	689	10.2	82	0.066	355	949	16.1	77	0.041	4.9	3.7	96.2	17.5
bloat	201	434	9.8	78	0.089	396	914	17.4	76	0.051	91.2	26.9	69.9	19.3
chart	288	306	13.2	76	0.068	567	453	22.6	76	0.047	9.4	8.0	91.7	30.0
fop	195	120	8.4	45	0.067	381	162	14.0	46	0.045	0.2	28.8	60.9	30.5
pmd	184	187	8.0	55	0.075	365	313	13.6	96	0.052	5.6	7.5	92.1	27.0
jython	288	275	12.6	28	0.065	666	539	26.1	27	0.042	14.6	13.1	81.9	26.8
xalan	168	594	8.5	75	0.066	407	1095	18.1	74	0.044	25.5	17.8	82.0	19.4
hsqldb	192	110	8.0	88	0.072	379	132	13.7	86	0.050	1.3	6.4	92.4	31.0
luindex	160	177	6.7	92	0.073	315	331	11.5	86	0.040	3.5	4.6	93.0	24.6
lusearch	139	110	5.5	48	0.079	275	223	11.0	52	0.053	9.1	9.3	65.2	29.1
eclipse	525	2435	28.8	47	0.072	1016	5724	53.1	53	0.047	28.6	21.0	78.3	22.0
avrora	189	108	7.9	67	0.086	330	125	11.2	56	0.034	3.3	3.2	94.8	34.5
batik	361	355	15.8	85	0.086	662	614	24.9	89	0.049	2.4	27.1	71.1	26.7
derby	308	314	13.9	63	0.080	425	530	22.1	57	0.049	65.2	5.0	94.0	23.7
sunflow	206	152	8.2	92	0.076	330	212	10.3	91	0.040	82.5	32.7	43.7	31.7
tomcat	533	1100	25.4	94	0.098	730	2209	48.6	92	0.063	29.1	24.2	72.2	23.1
tradebeans	825	1010	38.2	89/8*	0.053	1568	1925	58.9	82/8*	0.036	15.1	14.9	80.0	22.3
tradesoap	860	1370	41	82/17*	0.062	1628	2536	63.6	81/16*	0.040	41.0	24.5	59.4	20.1

Table 1. Characteristics of G_{cost}. Reported are the numbers (in thousand) of nodes (*N*) and edges (*E*), the memory overhead (in megabytes) excluding the size of the shadow heap (*M*), the running time overhead (*O*), and the context conflict ratio (*CR*). Part (c) reports the total number (in billion) of instruction instances (*I*), the percentages of instruction instances (directly and transitively) producing values that are ultimately dead (*IPD*), the percentages of instruction instances (directly or transitively) producing values that end up only in predicates (*IPP*), and the percentages of G_{cost} nodes such that all the instruction instances represented by these nodes produce ultimately-dead values (*NLD*) .

to track. For example, for the two transaction-based applications `tradebeans` and `tradesoap`, there is 5-10× overhead reduction when we enable tracking only for the load runs (i.e., the application is not tracked for the server startup and shutdown phases). Hence, it is possible for a programmer to identify suspicious program components using lightweight profiling tools such as a method execution time profiler or an object allocation profiler, and run our tool on the selected components for detailed diagnosis. It is also possible to employ various sampling-based or static pre-processing techniques (e.g., from [38]) to reduce the dynamic effort in data collection.

A small amount of memory is required to store the graph, and this is achieved primarily by employing abstract domains. The space reduction resulting from abstract slicing can also be seen from the comparison between the number of nodes in the graph (*N*) and the total number of instruction instances (*I*), as *N* represents the size of the abstract domain employed in the analysis while *I* represents the size of the actual concrete domain that fully depends on the run-time behavior of the application. *CR* measures the degree to which distinct contexts are mapped to the same slots by our encoding function `h`. Following [34], *CR-s* for an instruction i is defined as:

$$CR\text{-}s(i) = \begin{cases} 0 & \max_{0 \leq j \leq s}(dc[j]) = 1 \\ \max(dc[j]) / \sum dc[j] & \text{otherwise} \end{cases}$$

where $dc[j]$ represents the number of distinct contexts that fall into context slot j. *CR* is 0 if each context slot represents at most one distinct context; *CR* is 1 if all contexts fall into the same slot. The table reports the average *CR* for all instructions in G_{cost}. Note that both *CR-8* and *CR-16* show very small numbers. This is because many methods in a program only have a small number of distinct object chains throughout the execution.

Columns *IPD* and *IPP* in part (c) report the measurements of inefficiency for $s = 16$. *IPD* represents the percentage of instruction instances that produce only dead values. Suppose D is a set of non-consumer nodes in G_{cost} that do not have any outgoing edges (i.e., no other instructions are data-dependent on them), and D^* is a set of nodes that can lead only to nodes in D. Hence, D^* contains nodes that ultimately produce only dead values. *IPD* is calculated as the ratio between the sum of execution frequencies of the nodes in D^* and the total number of instruction instances during the execution (shown in column *I*). Similarly, suppose P^* is the set of nodes that can lead only to predicate consumer nodes, and *IPP* is calculated as the ratio between the sum of execution frequencies of the nodes in P^* and *I*. Programs such as `bloat`, `eclipse` and `sunflow` have large *IPD*s, which indicates that there may exist large optimization opportunities. In fact, these three programs are the ones for which we have achieved the largest performance improvement after removing bloat (as discussed shortly in case studies). Clearly, a significant portion of the set of instruction instances is executed to produce only control flow conditions. While this does not help performance diagnosis directly, a high *IPP* indicates the program performs a large amount of comparisons-related work, which may be a sign of over-protective or over-general implementations.

Column *NLD* in part (c) reports the percentage of nodes in D^*, relative to the total number of graph nodes. The higher *NLD* a program has, the easier it is for a programmer to find problems from G_{cost}. Despite the merging of a large number of instruction instances in a single graph node, there are on average 25.5% nodes in the graph that have this property. Large performance opportunities may be found by inspecting the report to identify these wasteful operations.

4.2 Case studies

We have carefully inspected the tool reports for the following six large applications: `bloat`, `eclipse`, `sunflow`, `derby`, `tomcat`, and `trade`. These applications have large code bases, and are representatives of various kinds of real-world applications, including program analysis tools (`bloat`), Java development tools (`eclipse`), image renders (`sunflow`), database servers (`derby`), servlet containers (`tomcat`), and transaction-based enterprise applications (`trade`). We have found significant optimization opportunities for unoptimized programs, such as `bloat` (37% speedup). For the other five applications that have been well maintained and tuned, the removal of the bloat detected by our tool can still result in considerable performance improvement (2%-15% speedup). More insightful changes could have been made if we were familiar with the overall design of functionality and data models. We use the DaCapo versions of these programs, because the server applications are converted to run fixed loads, and the performance can be measured simply by using running time rather than other metrics such as throughput and the number of concurrent users. It took us

المنارة للاستشارات
www.manaraa.com

about 2.5 weeks to find the problems and implement the fixes for these six applications that we had never studied before.

sunflow Because it is an image rendering tool, much of its functionality is based on matrix and vector computations, such as *transpose* and *scale*. However, each such method in class Matrix and Vector starts with cloning a new Matrix or Vector object and assigns the result of the computation to the new object. Our tool reported that these newly created (short-lived) objects have extremely large unbalanced costs and benefits, as they serve primarily the purpose of carrying data across method invocations. Another few lines of the report directed us to an int array where some slots of the array are used to contain float values. These float values are converted to integers using method `Float.floatToIntBits` and assigned to the array elements. Later, the encoded integers are read from the array and converted back to float values. These operations occur in the most-frequently executed methods in the program and are therefore are expensive to perform. By eliminating unnecessary clones and bookkeeping the float values that need to be passed across method boundaries (to avoid the back-and-forth conversions), we observed 9%-15% running time reduction.

eclipse Some of the allocation sites that have the largest cost-benefit rates create objects of inner classes and Iterators, which implement visitor patterns to traverse the workspace. These visitor objects do not contain any data and are passed into iterators, where their `visit` method is invoked to process individual children elements of the workspace. However, the Iterator used here is a stack-based class that provides general functionality for traversing different types of data structures (e.g., graph, tree, etc.), while the workspace has a very simple tree structure. We replaced the visitor implementation with a worklist implementation, and this simple specialization eliminated millions of run-time objects. The second major problem found by the tool is with the hash computation implemented in a set of Hashtable classes in the JDT plugin. One of the most frequently used classes in this set is called `HashtableOfArrayToObject`, which uses arrays of objects as keys. Every time the Hashtable is expanded, its `rehash` method needs to be invoked and the hash codes of all existing entries have to be recomputed. Because the key can be a big object array, computing its hash code can trigger invocations of the `hashcode` method in many other objects, and can thus take considerably large amount of time. We created an int array field in the Hashtable class to cache the hash codes of the entries, and the recorded hash codes are used when `rehash` is executed. To conclude, by removing these high-cost-low-benefit operations, we have managed to reduce the running time by 14.5% (from 151s to 129s), and the number of objects by 2% (5.5 million).

bloat Previous work [34] has found that `bloat` suffers from excessive string creations. This finding is confirmed by our tool report. 46 allocation sites out of the top 50 that have the largest cost-benefit rates are `String` and `StringBuffer` objects created in the set of `toString` methods. Most of these objects eventually flow into methods `Assert.isTrue` and `db`, which print the strings when certain debugging-related conditions hold. However, in production runs where most bugs have been fixed, such conditions can rarely evaluate to true, and there is no benefit in constructing these objects. Another problem exposed by our tool (but not reported in [34]) is the excessive use of objects of an inner class `NodeComparator`, which contains no data but methods to compare a pair of AST nodes. The comparison starts with the given root nodes, and recursively creates `NodeComparator` objects to compare children nodes. Comparing two large trees usually requires the allocation (and garbage collection) of hundreds of objects, and such comparisons occur in almost all methods related to ASTs, even including `hashcode` and `equals`. Eliminating the unnecessary `String` and `StringBuffer` objects and replacing the visitor pattern with a breadth-first search algorithm result in 37% reduction in running time, and 68% reduction in the number of objects created.

derby The tool report shows that an int array in class `FileContainer` has large cost-benefit rates. After inspecting the code, we found it is an array containing the information of a file-based container. Every time the (same) container is written into a page, the array needs to be updated. Hence, it is written much more frequently (with the same data) than being read. To solve the problem, we modify the code to update this array only before it is read. Another set of objects that were found to have unbalanced cost-benefit rates are the strings representing IDs for different ContextManagers. These strings are used to retrieve the ContextManagers in a variety of ways, but mostly serve as HashMap keys. Because the database contexts are frequently switched, clear performance improvement can be seen when we replaced these strings with integer IDs. Eventually, the running time of the program was reduced by 6%, and the number of objects created was reduced by 8.6%.

tomcat `tomcat` is a well-tuned JSP and servlet container. There are only a few objects that have large cost-benefits according to the tool report. One set of such objects is arrays used in util.Mapper, representing the (sorted) list of existing contexts. Once a context is added or removed from the manager, an update algorithm is executed. The algorithm creates a new array, inserts the new context at the right position in this new array, copies the old context array to the new one, and discards the old array. To remove this bloat, we maintain only two arrays, using them back and forth as the main context list and the backing array used for the update algorithm. Another problem reported by our tool pointed to string comparisons in various `getContents` and `getProperty` methods. These methods take a property name and a `Class` object (representing the type of the property) as input, and return the value corresponding to the property using reflection. To decide the type of the property, the implementations of these methods first obtain the names of the argument classes and compare them with the embedded names such as ”Integer” and ”Boolean”. Because a property can have only a few types, we remove such string comparisons and insert code to directly compare the `Class` objects. After the modifications, the program could run 3 seconds faster (about 2% reduction).

tradebeans `tradebeans` is an EJB application that performs database queries to simulate a stock trading process. One problem that our tool reported was with the use of `KeyBlock` and its iterators. This class represents a range of integers that will be given as IDs for the accounts and holdings when they are requested. We found that for each ID request, the class needs to perform a few redundant database queries and updates. In addition, a simple int array can suffice to represent IDs since the `KeyBlock` and the iterators are just wrappers over integers. By removing the additional database queries and using directly the int array, we have manged to make the application run 9 seconds faster (from 350s to 341s, 2.5% reduction). The number of objects created was reduced by 2.3%. DaCapo has another implementation (`tradesoap`) of `trade`, which uses the SOAP protocol to perform client-server communication and runs much slower than `tradebeans`. An interesting comparison between these two benchmarks is that the major high-cost-low-benefit objects reported for `tradesoap` are the bean objects created in the set of `convertXBean` methods. As part of the SOAP protocol, these methods perform large volumes of copies between different representations of the same bean data, resulting in significant performance slowdown. Developers can quickly find such (design) issues leading to the low performance after inspecting the tool reports.

Summary With the help of the cost-benefit analyses, we have found various performance problems in these large applications

المنارة للاستشارات
www.manaraa.com

with which we do not have any experience. These problems include inefficiencies caused by common programming idioms such as visitor patterns, repeated work whose result needs to be cached (e.g., the hash code example in `eclipse`), computation of data not necessarily used (e.g., strings in `bloat`), and choices of expensive operations (e.g., string comparison in `tomcat` and the use of SOAP in `tradesoap`). For specific bloat patterns such as the use of inner classes, it is also possible for the compiler/optimizer designers to take them into account and develop optimization techniques that can remove the bloat while not having to restrict programmers from using these patterns.

5. Related Work

There is a very large body of work related to dynamic slicing, dynamic flow analysis, and bloat detection. Due to space limitations, we only discuss techniques closely related to the work presented in this paper.

Performance measurement for optimizations Performance measurement is key to optimization choices made in modern runtime systems. Existing approaches for performance measurement attribute cost to some coarse-grained program entities, such as methods and calling contexts [31, 42]. The cost (e.g., frequency) is then used to identify hot spots and guide optimizations. On the contrary, our work computes finer-grained cost at the instruction level and uses it to understand performance and detect program regions that are likely to contain wasteful operations. The work from [12] proposes a technique that constructs models of *empirical computational complexity*, which can be used to predict the execution frequency of a basic block as a function of the program's workloads. While the profiling results can expose performance problems in general, they do not provide any information about the flow of data. In many cases, data flow information can be more useful in detecting performance bugs than execution frequencies [26, 34].

Dynamic slicing A general description of slicing technology and challenges can be found in Tip's survey [32]. Recently, the work by Zhang *et al.* [37, 38, 39, 40, 41] has significantly advanced the state of the art of dynamic slicing. This work includes, for example, a set of cost-effective dynamic slicing algorithms [38, 40], a slice-pruning analysis that computes confidence values for instructions to select those that are most related to errors [37], a technique that performs online compression of the execution trace [39], and an event-based approach that reduces the cost by focusing on events instead of individual instruction instances [41]. Sridharan *et al.* propose thin slicing [30], a technique that improves the relevance of the slice by focusing on the statements that compute and copy a value to the seed. Although this technique is originally proposed for static analysis, it fits naturally in our dynamic analysis work.

Our work is fundamentally different from these existing techniques in the following ways. Orthogonal to the existing profile summarization techniques such as [1, 3, 10, 39], abstract slicing achieves efficiency by introducing analysis semantics to profiling, establishing a foundation for solving a range of backward dynamic flow problems. As long as an analysis can be formulated in our framework, the profiled information is sufficiently precise for the analysis. Hence, although our approach falls into the general category of lossy compression, it can be lossless for the specific analysis formulated. The work from [41] is more related to our work in that the proposed event-based slicing approach is a special case of abstract slicing with the domain $\mathcal{D}$ containing a set of pre-defined events. In addition, all the existing work on dynamic slicing targets automated program debugging, whereas the goal of our work is to understand performance and find bottlenecks.

Leak/bloat detection A body of work has been devoted to manually [22] or automatically [5, 7, 8, 11, 13, 18, 20, 21, 35] detecting memory leaks and bloat in large-scale Java applications. Work from [21, 22] proposes metrics to provide performance assessment of use of data structures. Recent work by Shankar *et al.* [29] measures *object churn* and then enables aggressive method inlining over the regions that contain excessive temporary objects. The research from [26] proposes a false-positive-free technique that identifies stale objects based on data sampling. The work from [28] presents a collection-centric technique that makes appropriate collection choices based on a dynamically collected trace on collection behaviors. Work from [36] proposes a static analysis that can automatically discover container structures and find inefficiencies with use of containers. Our previous work [34] proposes a copy profiling technique that identifies bloat by profiling copy activities. Similarly to our G_{cost} which contains dependence relationships, these activities are recorded in a copy graph, which is used later for offline analysis.

The major difference between the core techniques in our work and all the existing bloat detection work lies in the different symptom definitions used to locate bloat. For example, existing approaches find bloat based on various kinds of suspicious symptoms (e.g., excessive copies [34], redundant collection space [28], large numbers of temporary objects [11, 29], and object staleness [5, 26]). On the contrary, we detect bloat by capturing directly the core pf the problem, that is, by looking for the objects that have high cost-benefit rates. In fact, unbalanced cost and benefit can be a more general indicator of bloat than specific symptoms, because it is common for the wasteful operations to produce high-cost-low-benefit values while these operations may exhibit different observable symptoms.

6. Conclusions and Future Work

What is the cost of constructing this object? Is that really worth performing such a heavyweight operation? We hear these questions all the time during software development. They represent the most natural and explicit form in which a programmer can express her concern on performance. Tuning could have been much easier if there existed a way so that these questions can be (even partially) automatically answered. As a step toward achieving the goal, this paper introduces a dynamic analysis of data flow, given the fact that much functionality of a large application is about massaging data. Optimizations based on pure control flow information (e.g., execution frequency) can no longer suffice to capture the redundancy that accumulates during data manipulation. This work is not only about the measurement of the cost of generating a piece of data, but more importantly, it computes an assessment of the way this data is used. Instead of focusing on bytes and integers, this assessment is made at the data structure level as object-oriented data structures are extensively used and programmers often do not need to be aware of their internal implementations during development.

In future work, we plan to extend the notions of cost and benefit (defined in terms of computations in this paper) in many other ways to help performance tuning. One example is to measure the effectiveness of the data structures used as caches. The way of redefining costs and benefits for caches was discussed in Section 3. As another example, one can adapt the cost and benefit for data presented in this paper to measure performance of control-flow entities, such as methods, components, and plugins. Faced with a large and complex application, a developer would need to first identify such coarser-grained program constructs that can potentially cause performance issues, in order to track down a performance bug through subsequent more detailed profiling. In addition, it is possible in future work to consider the space of other design choices that are discussed in Section 3. Other than the work of bloat detection, we are

المنارة للاستشارات
www.manaraa.com

also interested in investigating future extensions and applications of abstract slicing as a general technique, so that it could potentially benefit a wider range of dynamic analyses.

Acknowledgments We would like to thank the anonymous reviewers for their valuable and thorough 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] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *PLDI*, pages 246–256, 1990.

[2] M. Arnold, M. Vechev, and E. Yahav. QVM: An efficient runtime for detecting defects in deployed systems. In *OOPSLA*, pages 143–162, 2008.

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

[4] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, pages 169–190, 2006.

[5] M. D. Bond and K. S. McKinley. Bell: Bit-encoding online memory leak detection. In *ASPLOS*, pages 61–72, 2006.

[6] M. D. Bond and K. S. McKinley. Probabilistic calling context. In *OOPSLA*, pages 97–112, 2007.

[7] M. D. Bond and K. S. McKinley. Tolerating memory leaks. In *OOPSLA*, pages 109–126, 2008.

[8] M. D. Bond and K. S. McKinley. Leak pruning. In *ASPLOS*, pages 277–288, 2009.

[9] 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.

[10] B. Calder, P. Feller, and A. Eustace. Value profiling. In *MICRO*, pages 259–269, 1997.

[11] 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.

[12] S. F. Goldsmith, A. S. Aiken, and D. S. Wilkerson. Measuring empirical computational complexity. In *FSE*, pages 395–404, 2007.

[13] M. Jump and K. S. McKinley. Cork: Dynamic memory leak detection for garbage-collected languages. In *POPL*, pages 31–38, 2007.

[14] B. Korel and J. Laski. Dynamic slicing of computer programs. *J. Syst. Softw.*, 13(3):187–195, 1990.

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

[16] J. Larus. Spending Moore's dividend. *Commun. ACM*, 52(5):62–69, 2009.

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

[18] N. Mitchell, E. Schonberg, and G. Sevitsky. Making sense of large heaps. In *ECOOP*, pages 77–97, 2009.

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

[20] N. Mitchell and G. Sevitsky. Leakbot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In *ECOOP*, pages 351–377, 2003.

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

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

[23] V. Nagarajan and R. Gupta. Architectural support for shadow memory in multiprocessors. In *VEE*, pages 1–10, 2009.

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

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

[26] G. Novark, E. D. Berger, and B. G. Zorn. Efficiently and precisely locating memory leaks and bloat. In *PLDI*, pages 397–407, 2009.

[27] 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.

[28] O. Shacham, M. Vechev, and E. Yahav. Chameleon: Adaptive selection of collections. In *PLDI*, pages 408–418, 2009.

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

[30] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. In *PLDI*, pages 112–122, 2007.

[31] N. R. Tallent, J. M. Mellor-Crummey, and M. W. Fagan. Binary analysis for measurement and attribution of program performance. In *PLDI*, pages 441–452, 2009.

[32] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[33] C. Wang and A. Roychoudhury. Dynamic slicing on Java bytecode traces. *ACM Transactions on Programming Languages and Systems*, 30(2):1–49, 2008.

[34] G. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky. Go with the flow: Profiling copies to find runtime bloat. In *PLDI*, pages 419–430, 2009.

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

[36] G. Xu and A. Rountev. Detecting inefficiently-used containers to avoid bloat. In *PLDI*, 2010.

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

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

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

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

[41] X. Zhang, S. Tallam, and R. Gupta. Dynamic slicing long running programs through execution fast forwarding. In *FSE*, pages 81–91, 2006.

[42] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi. Accurate, efficient, and adaptive calling context profiling. In *PLDI*, pages 263–271, 2006.

المنارة للاستشارات

www.manaraa.com