Cachetor: Detecting Cacheable Data to Remove Bloat

Khanh Nguyen and Guoqing Xu
University of California, Irvine, CA, USA
{khanhtn1, guoqingx}@ics.uci.edu

ABSTRACT

Modern object-oriented software commonly suffers from runtime bloat that significantly affects its performance and scalability. Studies have shown that one important pattern of bloat is the work repeatedly done to compute the same data values. Very often the cost of computation is very high and it is thus beneficial to memoize the invariant data values for later use. While this is a common practice in real-world development, manually finding invariant data values is a daunting task during development and tuning. To help the developers quickly find such optimization opportunities for performance improvement, we propose a novel run-time profiling tool, called Cachetor, which uses a combination of *dynamic dependence profiling and value profiling* to identify and report operations that keep generating identical data values. The major challenge in the design of Cachetor is that both dependence and value profiling are extremely expensive techniques that cannot scale to large, real-world applications for which optimizations are important. To overcome this challenge, we propose a series of novel abstractions that are applied to run-time instruction instances during profiling, yielding significantly improved analysis time and scalability. We have implemented Cachetor in Jikes Research Virtual Machine and evaluated it on a set of 14 large Java applications. Our experimental results suggest that Cachetor is effective in exposing caching opportunities and substantial performance gains can be achieved by modifying a program to cache the reported data.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Metrics—*Performance measures*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*

General Terms

Performance, Reliability, Experimentation

Keywords

Runtime bloat, performance optimization, cacheable data, dynamic dependence analysis

ESEC/FSE'13, August 18–26, 2013, Saint Petersburg, Russia
Copyright 2013 ACM 978-1-4503-2237-9/13/08...$15.00
http://dx.doi.org/10.1145/2491411.2491416

1. INTRODUCTION

Many applications suffer from chronic runtime bloat—excessive memory usage and run-time work to accomplish simple tasks—that significantly affects scalability and performance. Our experience with dozens of large-scale, real-world applications [33, 35, 36, 37] shows that a very important source of runtime bloat is the work repeatedly done to compute identical data values—if the computation is expensive, significant performance improvement can be achieved by memoizing[1] these values and avoiding computing them many times. In fact, caching important data (instead of recomputing them) is already a well-known programming practice. For example, in the white paper "WebSphere Application Server Development Best Practices for Performance and Scalability" [3], four of the eighteen best practices are instructions to avoid repeated creation of identical objects. While finding and caching identical data values is critical to the performance of many large-scale software systems, the task is notoriously challenging for programmers to achieve during development and tuning. A large, long-running program may contain millions of instructions, and each instruction may be executed for an extremely large number of times and produce a sea of data values. It would be extremely difficult, if not impossible, to find identical run-time data values and understand how to cache them without appropriate tool support.

Motivation To illustrate, consider the following code example, adapted from `sunflow`[2], an open-source image rendering system.

```
float[] fValues = {0, 1.0, 2.3, 1.0, 1.0, 3.4, 1.0, 1.0,
                                                ..., 1.0};
int[] iValues = new int[fValues.length] ;
for (int i = 0; i < fValues.length; i++){
   iValues[i] = Float.floatToIntBits(fValues[i]);
}
```

This simple program encodes each float value in array *fValues* using a bit array (represented by an Integer), which can then be stored in an Integer array. In this example, most of the values in *fValues* are 1.0, and it is unnecessary to invoke method `Float.floatToIntBits` (which is quite expensive) to compute the bit array for each of them. The program would run more efficiently if `Float.floatToIntBits` can be invoked the first time 1.0 is seen , and the result can be cached and reused for its future occurrences. However, this information may not be available to the programmer during development, as *fValues* may be a dynamically computed array whose content is unknown at compile time, or the fact that most of its elements are the same is specific to a certain kind of input image being processed. As a result, it is necessary to develop techniques and tools that can help the programmer

[1]Terms "memoize" and "cache" are used interchangeably.
[2]http://sunflow.sourceforge.net/.

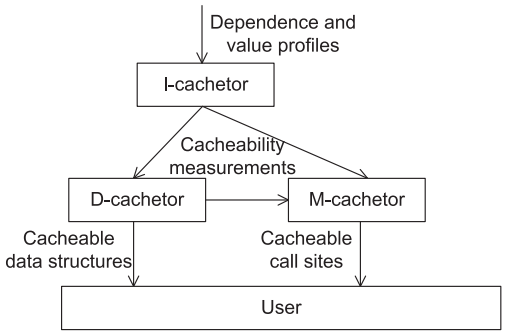


Figure 1: An overview of Cachetor.

find such missed optimization opportunities (e.g., report method `floatToIntBits` is frequently executed with the same input and produces the same output), especially in a situation where a significant performance issue is observed and tuning must be undertaken to make the application reach its performance goal.

Our proposal In this paper, we propose a dynamic analysis tool, called *Cachetor*, that profiles large-scale applications to pinpoint cacheable data values and operations producing these values. Cachetor has three major components, which are illustrated in Figure 1. At the lowest level of the tool is an instruction-level cacheable value detector called *I-Cachetor*, which identifies (bytecode) instructions whose executions produce identical (primitive-typed) values. Finding only instructions that always produce the same values may significantly limit the amount of optimization opportunities that can be detected. To improve usefulness, we propose to compute a *cacheability measurement* for each instruction, that captures the percentage of the most frequently-occurring value among all values produced by the instruction. For example, although the call instruction that invokes `floatToIntBits` does not always return the same value, this instruction has a high CM and will thus be recognized by the developer during inspection.

I-Cachetor is of limited usefulness by itself—it is often not possible to cache values produced by specific instructions. We develop two higher-level detectors, namely *D-Cachetor* and *M-Cachetor*, that detect *data structures* containing identical values and *method calls* producing identical values, respectively, to help developers understand and fix inefficiencies at a logical level. D-Cachetor queries I-Cachetor for the CMs of the heap store instructions that write into a data structure and aggregate these instruction-level CMs to compute the CM of the data structure. M-Cachetor focuses on the value returned from each call site: it queries I-Cachetor for the CM of the call instruction if the return value is of primitive type, or, otherwise, queries D-Cachetor for the CM of the returned object, so as to compute the CM of the call site. Eventually, allocation sites and method calls are ranked based on their respective CMs and the lists are reported to the developer for manual inspection.

Fixing problems reported by Cachetor Because Cachetor relates optimization opportunities with high-level program entities, the reported problems can be easily understood and fixed. For example, D-Cachetor reports allocation sites that create identical objects and data structures. To fix the reported problems, one may create a singleton pattern for such an allocation site to enforce the use of one single instance throughout the execution, or develop a `clone` method in its class to directly copy values between the objects instead of re-computing the (same) values from the scratch. As another example, M-Cachetor reports call sites whose executions always produce the same results. One may easily create a (static or instance) field and cache the result of such a call site in the field, so that the frequent invocation of the method can be avoided.

Technical challenges and our solution The biggest challenge that stands in the way of implementing Cachetor is how to find these identical data values in a *scalable* way so that Cachetor can be applied to large, real-world applications. In particular, the implementation of I-Cachetor requires the comparison of run-time values produced by different executions of the same instruction. To do this, a natural idea is to perform whole-program *value profiling* [10], which records run-time values for all instruction executions. These values are compared offline to identify cacheable instructions. In addition, in order to compute data-structure-level CMs, a *dynamic dependence analysis* may be needed to understand which data structure an instruction may write into at run time. However, both whole-program value profiling and dependence analysis are extremely expensive techniques that cannot scale to real-world applications. To improve the practicality of our analysis, we propose a novel approach that combines value profiling with dynamic dependence analysis in a way so that they mutually improve the scalability of each other. Specifically, we use distinct values produced at run time to abstract dependence graph nodes, yielding a *value-abstracted dynamic dependence graph*. This graph contains the value information necessary for our analysis, and yet is much smaller and easier to compute than a regular dynamic dependence graph. We then propose a series of further abstractions based on this dependence analysis to scale Cachetor to the real world.

We have implemented this combined dependence and value profiling technique in Jikes Research Virtual Machine [16], and then built the three detectors based on the abstract representation. An evaluation of the tool on a set of 14 large-scale Java applications shows that Cachetor incurs an overall 201.96× running time overhead and 1.98× space overhead. While these overheads are very large, they have not prevented us from collecting any run-time data for real-world applications. In fact, it could have been impossible to implement such a heavyweight analysis on real-world applications (such as those in the DaCapo benchmark set [7]) without the proposed abstractions. We have carefully inspected the analysis reports; fixing the reported problems has led to significant performance improvements for many large-scale applications. This experience is described in the five case studies in Section 5.

The major contributions of the paper are:

- A novel approach that uses distinct values to abstract instruction instances and dependence relationships, leading to significantly increased efficiency;
- Three cacheability detectors that are built on top of the value-abstracted dependence graph to find caching opportunities;
- An implementation of Cachetor in Jikes Research Virtual Machine;
- An evaluation of Cachetor on a set of large Java applications that shows (1) Cachetor incurs a high but acceptable overhead and (2) large optimization opportunities can be quickly found from Cachetor's reports. These initial results suggest that Cachetor is useful in practice in helping developers find caching opportunities, and our combined dependence and value profiling may be employed to improve the efficiency of a variety of dynamic analysis techniques.

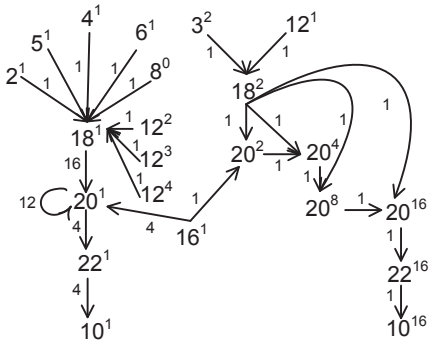
2. VALUE-ABSTRACTED DYNAMIC DEPENDENCE ANALYSIS

The naive implementation of either dynamic dependence analysis or value profiling cannot scale to large, real-world applications for which optimizations are important. To overcome this scalability challenge, we propose a novel technique, called *value-abstracted*

```
1  int[] input = new int[6];
2  input[0] = 1;
3  input[1] = 2;
4  input[2] = 1;
5  input[3] = 1;
6  input[4] = 1;
7  input[5] = 5;
8  int i = 0;
9  if(i < intput.length){
10   int result = f(i, input);
11   print(result);
12   i = i + 1;
13   goto 9;
14 }
```

```
15 int f(int i, int[] arr) {
16     int j = 1;
17     int k = 0;
18     int p = arr[i];
19     if(k < 4) {
20         j = j * p;
21         k = k + 1;
22     } return j;
23 }
```

(a) A simple program



(b) Part of the value-abstracted dynamic dependence graph

Figure 2: An example of value-abstracted data dependence graph.

dependence analysis, that uses distinct run-time values an instruction produces to define a set of equivalence classes to abstract dependence graph nodes (i.e., instruction instances), leading to significantly reduced analysis time and space consumption.

2.1 Value-Abstracted Dependence Graph

To formally define the abstraction, we first give our definition of dynamic dependence graph. Since our goal is to detect caching opportunities, we are interested only in data dependence.

Definition 1 (Dynamic Data Dependence Graph) *A dynamic data dependence graph $(\mathcal{N}, \mathcal{E})$ has node set $\mathcal{N} \subseteq \mathcal{S} \times \mathcal{I}$, where each node is a static instruction ($\in \mathcal{S}$) annotated with a natural number i ($\in \mathcal{I}$), representing the i-th execution of this instruction. An edge $s_1^j \rightarrow s_2^k$ ($s_1, s_2 \in \mathcal{S}$ and $j, k \in \mathcal{I}$) shows that the j-th execution of instruction s_1 writes a (heap or stack) location that is then used by the k-th execution of s_2, 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.*

While Cachetor works on the low-level JVM intermediate representation, the discussion of the algorithms uses a three-address-code representation of the program (e.g., an assignment $a = b$ or a computation $a = b + c$). We will use terms statement and instruction interchangeably, both meaning a statement in the three-address-code representation. We divide the static instruction set $\mathcal{S}$ into $\mathcal{SP}$ and $\mathcal{SR}$, which contain instructions that process primitive-typed and reference-typed data, respectively. The first step of our analysis targets $\mathcal{SP}$, because we are interested in finding instructions that produce identical primitive-typed values. $\mathcal{SR}$ will be considered in the next stage when the cacheability information of instructions in $\mathcal{SP}$ needs to be aggregated to compute the cacheability information for objects and data structures (in D-Cachetor).

For an instruction $s \in \mathcal{SP}$, we use $\mathcal{V}_s$ to denote the set of distinct values that the instances of s produce at run time. We use each value $v \in \mathcal{V}_s$ as an identifier to determine an equivalence class including a set of instances of s that produce the same value v. The definition of the value-abstracted dependence graph is given as follows:

Definition 2 (Value-Abstracted Data Dependence Graph) *A value-abstracted data dependence graph $(\mathcal{N}', \mathcal{E}')$ has node set $\mathcal{N}' \subseteq \mathcal{SP} \times \mathcal{V}$, where each node is a pair of a static instruction $s \in \mathcal{SP}$ and a value $v \in \mathcal{V}_s$ (represented as s^v), denoting the set of instances of s that produce the same value v. An edge $s_1^w \rightarrow s_2^x$ ($s_1, s_2 \in \mathcal{SP}$ and $w, x \in \mathcal{V}$) shows that an instance of s_1 that produces a value w writes a location that is used by an instance of s_2 that produces a value x, 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.*

In the value-abstracted dependence graph, instances of an instruction that produce the same run-time value are merged and represented by a single node. The advantage of developing such an abstraction is two-fold: (1) the number of distinct values produced by an instruction is often much smaller than the total number of executions of the instruction; this is especially true for (cacheable) instructions that frequently produce identical values, leading to reduced dependence graph size and profiling cost; (2) the merged nodes and edges are very likely to represent computation paths that ultimately produce the same results; maintaining one single copy of the path would often suffice to help us understand how these results are computed; computation paths leading to different results are still distinguished.

Example To illustrate, consider the example in Figure 2. Part (a) shows a simple program where each integer in the array $input$ is passed into function `f`, which simply computes the integer to the power of 4 (i.e., $arr[i]^4$). Note that among the six integers in $input$, four of them are 1, and hence, it is highly beneficial to cache and reuse the result of function f for the specific input value 1. Shown in Figure 2 (b) is an important part of the value-abstracted dependence graph for the program execution. Each edge in the graph is annotated with the number of occurrences of the dependence relationship the edge represents during execution. Each node in Figure 2 (b) is a static instruction annotated with a (distinct) value it produces. The static instruction is represented by its line number in the program. The left part of the value-abstracted dependence graph combines the computations for i = 0, 2, 3, and 4 (i.e., $input[i]$ = 1)—for these four input values, all instructions except those at line 8 and 12 produce the same output value 1. Edges annotated with high frequencies represent common computations that may be reused to improve performance. Dependence relationships for i = 1 (i.e., $input[i]$ = 2) are shown in the right part, where each dependence edge occurs only once. Dependence relationships for i = 5 are similar to those for i = 2 and are thus omitted from the figure.

While using distinct values to abstract dynamic dependence graph can significantly reduce the graph size and the profiling cost, the numbers of distinct values can be very large for some instructions, such as those that increase loop variables (e.g., line 12 in Figure 2). Because each instance of such an instruction produces a different

value, its instances can never be merged. In addition, the execution frequency of the instruction depends completely on the number of loop iterations, which is *input-sensitive* and *unbounded*, and thus, it can still be difficult to collect the value-abstracted dependence graph for large-scale, long-running applications.

Our experience shows that key to developing a scalable dynamic analysis is to *statically bound the amount of information to be collected dynamically*. In other words, the size of the profile should have an upper bound before the execution; it cannot depend on dynamic behaviors of the program. In Cachetor, we propose to further limit the number of equivalence classes for each instruction to a fixed number k, so that at most k dependence graph nodes can be recorded regardless of how many times the instruction is executed. To do this, we define a hash function $h(v) = v\%k$ that uses a simple modulo operation to map each distinct run-time value v produced by the instruction into an integer in the set [0, k). If v is a floating point value, it is cast to an integer before the hash function is performed. This abstraction results in a hash-value-based dependence graph, defined as follows:

Definition 3 (Hash-Value-Abstracted Data Dependence Graph) *A hash-value-abstracted data dependence graph ($\mathcal{N}''$, $\mathcal{E}''$) has node set $\mathcal{N}'' \subseteq \mathcal{SP} \times [0, k)$, where each node is a pair of a static instruction $s \in \mathcal{SP}$ and an integer $m \in [0, k)$ (represented as s^m), denoting the set of instances of s whose results are mapped to the same number m by the hash function h. An edge $s_1^{m_1} \rightarrow s_2^{m_2}$ ($s_1, s_2 \in \mathcal{SP}$ and $m_1, m_2 \in [0, k)$) shows that an instance of s_1 whose result is mapped to m_1 writes a location that is used by an instance of s_2 whose result is mapped to m_2 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.*

Note that the hash-value-abstracted dependence graph is a *lossy representation*, where the parameter k defines a tradeoff framework between analysis precision and scalability. k can be provided by the user as a tuning parameter to find the sweetspot for a particular program. We associate a frequency count with each graph node, representing the number of instruction instances that are mapped to the node. It is clear to see that instructions whose executions are dominated by one graph node are more likely to create identical values than those whose execution frequencies are thinly spread among multiple nodes. In our experiments, we have evaluated Cachetor using different k's. We find that (1) a prime number preserves more information than a composite number and (2) a relatively small number can often be very effective to distinguish truly cacheable instructions from those that are not. Details of our evaluation can be found in Section 5.

2.2 Adding Calling Context Abstraction

In order to compute cacheability for an object (in D-Cachetor), we need to aggregate the cacheability information for instructions that write into the object. A common abstraction for modeling a heap object is its allocation site. However, using only allocation sites to aggregate run-time information can cause significant imprecision, leading to reduced usefulness of the tool. This is especially true for large-scale object-oriented applications that make heavy use of data structures. For example, each HashMap has an internal entry array and all these array objects are created by the same allocation site. Failing to distinguish array objects based on the HashMap objects they belong to would cause all HashMaps to have similar cacheability measurements.

Object contexts [23] have been widely used in static analysis to distinguish objects that belong to different data structures. An object context is represented by a chain of allocation sites of the receiver objects for the method invocations on the call stack. We propose to add object contexts into the dependence graph so that instructions that write into different data structures can be distinguished and the cacheability of an instruction can be appropriately attributed to the cacheability of the data structure that the instruction writes into. Details about the cacheability computation will be discussed shortly in the next section.

It can be extremely expensive to record a chain of allocation sites for each dependence graph node. To solve the problem, we encode an object context into a *probabilistic unique value*. An encoding function proposed in [8] is adapted to perform this computation: c_i = 3 * c_{i-1} + a_i, where a_i is the i-th allocation site ID in the chain and c_{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 [8]. Similarly to the handling of distinct values, we bound the number of object contexts allowed for each instruction with a user-defined parameter k', and map each context c_i to a number in [0, k') using the same hash function $c_i\%k'$. Note that this modeling is performed for both instructions that manipulate primitive-typed values (i.e., $\mathcal{SP}$) and those that manipulate objects (i.e., $\mathcal{SR}$). The addition of object contexts results in a new dependence graph, which we refer to as *value-and-context (VC)-abstracted data dependence graph*: each instruction $s \in \mathcal{SP}$ has a pair annotation $\langle m, n\rangle$, where m is a hash value $\in [0, k)$ and n is a hash context $\in [0, k')$; each instruction $s \in \mathcal{SR}$ has only one (context) annotation $n \in [0, k')$. The VC-abstracted data dependence graph is defined as follows:

Definition 4 (VC-Abstracted Data Dependence Graph) *A VC-abstracted data dependence graph ($\mathcal{N}'''$, $\mathcal{E}'''$) has node set $\mathcal{N}''' \subseteq \mathcal{SP} \times [0, k) \times [0, k') \cup \mathcal{SR} \times [0, k')$, where each node is either a triple $s_1^{\langle m,n\rangle}$ ($s_1 \in \mathcal{SP}$, $m \in [0, k)$, $n \in [0, k')$) , denoting the set of instances of s_1 whose results are mapped to the same number m and whose object contexts are mapped to the same number n, or a pair s_2^n ($s_2 \in \mathcal{SR}$, $n \in [0, k')$), denoting the set of instances of s_2 whose object contexts are mapped to the same number n. Each edge can have one of the three forms: $s_1^{n_1} \rightarrow s_2^{n_2}$, $s_1^{\langle m_1,n_1\rangle} \rightarrow s_2^{\langle m_2,n_2\rangle}$, or $s_1^{n_1} \rightarrow s_2^{\langle m,n_2\rangle}$. The first two forms represent the propagation of a reference-typed and a primitive-typed value, respectively. The third form represents a pointer dereferencing operation.*

Cachetor profiles a program execution to compute a VC-abstracted data dependence graph. This graph will be used by a series of *offline analyses* discussed in the next section to compute cacheability measurements and rank data structures/call sites. The profiling details can be found in Section 4.

3. CACHEABILITY COMPUTATION

This section presents three offline analyses that take a VC-abstracted dependence graph as input and compute cacheability measurements (CM) for instructions, data structures, and call sites. These measurements are subsequently used to rank the corresponding program entities to facilitate user inspection.

3.1 Computing Instruction Cacheability Measurements

The first analysis, I-Cachetor, computes CMs for static instructions. The higher CM an instruction has, the more identical values the instruction produces during execution. As discussed in Section 2.1, each dependence graph node $s^{\langle m,n\rangle}$ is associated with an execution frequency count, recording the number of instruction instances that are merged into this node. Using $freq_s^{m,n}$ to represent

the frequency associated with node $s^{\langle m,n\rangle}$, we give the definition of the instruction CM as follows:

Definition 5 (Instruction Cacheability Measurement (ICM)) *For each static instruction $s \in \mathcal{SP}$, its ICM is defined as*

$$ICM_s = Avg_{0 \le n < k'} \frac{Max_{0 \le m < k}(freq_s^{\langle m,n \rangle})}{Sum_{0 \le m < k}(freq_s^{\langle m,n \rangle})}$$

over all VC-abstracted data dependence graph nodes of the form $s^{\langle m,n\rangle}$.

ICM is computed only for instructions that manipulate primitive-typed data. For each VC-abstracted dependence graph node $s^{\langle m,n\rangle}$, we first fix its context slot n, and compute a ratio between the maximum of the frequencies and their sum over the nodes with different hash values m. The ICM of s is finally computed as the average of these ratios for all contexts n. It is clear to see that $\frac{1}{k} \le \text{ICM}_s \le 1$. If the instruction always produces the same value during execution, its ICM is 1. On the other hand, if the values the instruction produces are spread evenly in the k hash value slots, its ICM is $\frac{1}{k}$.

ICM is not particularly useful by itself, because it can be very difficult, if not impossible, for the developer to cache the value of a particular instruction in the program. To further improve Cachetor's usefulness, we develop two high-level cacheability detectors to help the developer make sense of the heap and execution information at a high, logical level. ICM will be used later by the two detectors to compute high-level CMs.

3.2 Computing Data Structure Cacheability Measurements

The second offline analysis, D-Cachetor, aggregates ICMs for instructions that write into a data structure to compute data structure cacheability measurements (DCMs). Specifically, we compute a DCM for each *allocation site*, summarizing the likelihood of the run-time data structures created by the allocation site containing identical data values. We find that focusing on allocation sites achieves the right balance between the amount of optimization opportunities that can be detected and the difficulty of developing fixes. For example, if an allocation site has a 100% CM, we may simply cache and reuse one single instance for it. Optimization opportunities can still be found for allocation sites with smaller CMs—although their objects are not entirely identical, they may contain identical fields, which may be cached for improved performance.

Note that simply ranking allocation sites based on their execution frequencies cannot reveal caching opportunities. For example, in a typical large application, the most frequently executed allocation site is the one in `HashMap.put` that keeps creating `Map$Entry` objects to store newly-added keys and values. Objects created by this allocation site are not reusable at all. Hence, it is necessary to develop new metrics for allocation sites in our framework.

A data structure often contains multiple levels of objects, and hence, we first consider the computation of cacheability measurements for individual objects (i.e., OCMs). OCMs are aggregated later based on the reference relationships among objects to form the DCM of a data structure. To compute the OCM for an allocation site o, we focus on heap store instructions that access primitive-typed data only. Reference-typed stores will be considered later when OCMs are aggregated.

The OCM computation starts with inspecting each allocation site of the form $o : a = new\ A$. As the allocation site accesses a reference-typed variable, it belongs to the instruction set $\in \mathcal{SR}$ and has a total of k' nodes in the VC-abstracted dependence graph. Each node is of the form o^n, where n is a hash context value

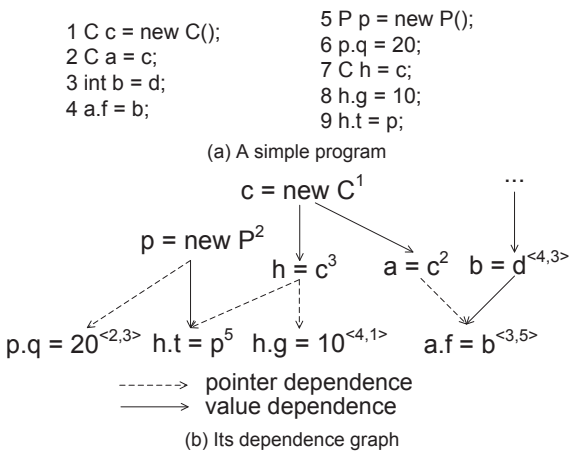


Figure 3: An example program, and its value and pointer dependence.

$\in [0, k')$. The analysis traverses the VC-abstracted dependence graph starting from each such node find a set of nodes of the form $s : a.f = b^{\langle m,n'\rangle}$ such that $s \in \mathcal{SP}$ and a points to an object created at o^n. This can be done by distinguishing *pointer dependence* and *value dependence* in the dependence graph. A pointer dependence relationship occurs between an instruction instance that defines a pointer variable and a subsequent (load or store) instruction instance that dereferences this pointer. A value dependence relationship occurs between an instruction instance that writes a value into a (stack or heap) location and a subsequent instruction instance that reads the value from the location. Figure 3 shows a simple program and its dependence graph. These two dependence relationships are represented by dashed arrows and solid arrows, respectively. Note that nodes that have pair annotations and that have single (context) annotations in Figure 3 (b) represent instruction instances manipulating primitive-typed and reference-typed values, respectively. Suppose Figure 3 (a) shows an *inlined* program—those statements are originally located in different methods. Hence, different statements may have different context encoding in Figure 3 (b).

In order to find the set of nodes that write into an object created by an allocation site, we perform a depth-first traversal from each node o^n representing the allocation site. During the traversal, we are interested in such a pointer dependence edge e that e is reachable from o^n only through value dependence edges. In other words, no other pointer dependence edge exists between e and the root node o^n. Clearly, the target node of e represents an instruction instance that reads/writes an object of the allocation site o^n. Among these (target) nodes, we are interested only in those that write primitive-typed values into the object. This subset of nodes is referred to as the object writing instruction set (OWIS) for the allocation site node o^n. The OWIS for node $c = \texttt{new}\ C^1$ in Figure 3 includes, for example, `a.f = b`$^{\langle 3,5\rangle}$ and `h.g = 10`$^{\langle 4,1\rangle}$. Note that node $h.t = p^5$ writes a reference into an object created by $c = \texttt{new}\ C^1$ (i.e., it has a single context annotation), and thus, is not considered in the OCM computation. Now we give the definition of o^n's object cacheability measurement:

Definition 6 (Object Cacheability Measurement (OCM)) *For each allocation site node o^n, $o \in \mathcal{SR}$, its OCM is defined as*

$$OCM_{o^n} = Avg_{s \in OWIS} ICM_{s,OWIS}$$

Algorithm 1: Finding allocation site nodes that belong to the same data structure.

Input: VC-abstracted data dependence graph g, selection ratio r
Output: Map<Node, Set<Node> > $dsMap$

```
1  Set<Node> visited ← ∅ // Alloc site nodes that have been visited
2  foreach Alloc Site Node o^n in g do
3      List<Node> allocList ← {o^n}
4      Set<Node> ds ← {o^n}
5      dsMap ← dsMap ∪ {⟨o^n, ds⟩}
6      while allocList ≠ ∅ do
7          Alloc Site Node p^c ← removeTop(allocList)
8          if p^c ∈ visited then
9              continue
10         visited ← visited ∪ {p^c}
11         List<Node> reachedNodes ← {p^c}
12         while reachedNodes ≠ ∅ do
13             Node q^d ← removeTop(reachedNodes)
14             foreach outgoing edge e of node q^d do
15                 if e is a pointer dep edge and target(e) writes a reference
                   value then
16                     Node r^e ← target(e)
17                     foreach incoming edge e' of node r^e do
18                         if e' is a value dependence edge then
19                             /*traverse backward*/
20                             Alloc Site Node
                               o'^{n'} ← backwardTraverse(e')
21                             /*if the difference between the OCMs of o^n
                               and o'^{n'} are ≤ r*/
22                             if |OCM_{o'^{n'}} − OCM_{o^n}| / OCM_{o^n} ≤ r then
23                                 ds ← ds ∪ {o'^{n'}}
24                                 allocList ← allocList ∪ {o'^{n'}}
25                 else if e is a value dep edge and target(e) writes a
                   reference value then
26                     reachedNodes ←
                       reachedNodes ∪ {target(e)}
27 return dsMap
```

where OWIS is the object writing instruction set for node o^n, and $ICM_{s,OWIS}$ denotes the ICM of the instruction s computed over the nodes in OWIS.

The OCM of an allocation site node is determined by the ICMs of the instructions that write primitive-typed values into the objects created by the allocation site. However, the ICM for an instruction used here is computed over the nodes in the OWIS of o^n, while the ICM in Definition 5 is computed over all nodes for the instruction. Because a static instruction may write into objects created by different allocation sites during execution, its graph nodes that are unreachable from o^n are not considered towards the computation of o^n's OCM.

DCM computation The OCM computation considers only the primitive-typed values contained in an object. In order to find large optimization opportunities, we compute DCMs for data structures by considering reference-typed store instructions and aggregating OCMs of the objects connected by such instructions. For each allocation site node o^n, the DCM computation first identifies other allocation site nodes p^c that belong to the same logical data structure rooted at o^n. This can be done by traversing the dependence graph from each node o^n and transitively following pointer dependence edges. Algorithm 1 describes the details of such a traversal. The algorithm takes as input a VC-abstracted dependence graph and a selection ratio $r \in [0, 1]$, and eventually computes a map $dsMap$ that contains, for each allocation site node in the graph (e.g., o^n), a set of allocation site nodes (e.g., ds at line 4) that may belong to the same logical data structure.

Initially, ds contains one single element o^n (line 4), and more allocation site nodes will be gradually added into ds as the data structure is being discovered by the analysis. $allocList$ is a list of allocation site nodes that have been identified as part of the data structure. These nodes need further inspection to find those that are transitively reachable from them. In the beginning of the analysis, $allocList$ has one node o^n (line 3). Lines 6–26 show a worklist-based iterative process to discover the data structure. Each iteration of the process retrieves an allocation site node p^c from $allocList$ and attempts to find allocation site nodes that are referenced by p^c. This is achieved by performing a breath-first traversal of the graph starting from node p^c (lines 12–24). For each node q^d reached during the traversal (line 13), the algorithm inspects each of its outgoing edges. If an outgoing edge e is a value dependence edge that propagates a reference value (lines 25–26), the target node of e is added into list $reachedNodes$ for further inspection. We are particularly interested in pointer dependence edges whose target is a store node writing a reference value (lines 15–24), such as $h.t = p^5$ in Figure 3, because such an edge can lead the analysis to find allocation sites that are *referenced* by p^c.

Once such a pointer dependence edge is found, lines 17–24 traverse backward the dependence graph, starting from the target (r^e) of the edge. This backward traversal follows only value dependence edges until it reaches an allocation site node $o'^{n'}$ (line 20), which is the creation point of the object that flows to r^e. It is important to note that we add $o'^{n'}$ into the data structure set ds only when the OCM of $o'^{n'}$ and the OCM of the root of the data structure o^n are close enough (i.e., their difference is $\leq$ the given selection ratio r). In other words, we select objects that have similar cacheability measurements to form a data structure so that the developer can easily find and fix problems related to the data structure.

Example We use the simple example in Figure 3 to illustrate how the algorithm works. In order to identify the data structure rooted at the node $c = \texttt{new}\ C^1$, our analysis traverses the graph until it reaches a pointer dependence edge whose target node (i.e., $h.t = p^5$) is a heap store writing a reference value. Next, Lines 15–24 in Algorithm 1 traverse backward the dependence graph starting from $h.t = p^5$, following only value dependence edges. This traversal leads up to the allocation site node $p = \texttt{new}\ P^2$. There exists a "reference" relationship between this node and node $c = \texttt{new}\ C^1$. The OCMs of these two nodes are then compared (against the selection ratio r) to determine whether $p = \texttt{new}\ P^2$ should be included in the data structure set of $c = \texttt{new}\ C^1$.

Using DS_{o^n} to represent the set of allocation nodes discovered by Algorithm 1 for the root note o^n, we give the definition of DCM as follows. Note that o^n itself is also included in DS_{o^n}.

Definition 7 (Data Structure Cacheability Measurement (DCM)) *For each allocation site node o^n, its DCM is defined as*

$$DCM_{o^n} = Avg_{p^d \in DS_{o^n}} OCM_{p^d}$$

where OCM_{p^d} is the object cacheability measurement for the allocation site node p^d, as defined in Definition 6.

Eventually, allocation site nodes are ranked based on their DCMs and the ranked list is reported to the user for manual inspection. When Cachetor reports an allocation site node o^n, it reports not only o^n itself but also the data structure DS_{o^n} computed by Algorithm 1. This would make it easier for the developer to understand the problem and develop the fix to cache the (invariant part) of the

data structure. Note that different graph nodes for the same static allocation site are reported separately based on their DCMs. The allocation site may be cacheable only under certain calling contexts; reporting them separately (instead of combining them using an average) would potentially reveal more optimization opportunities.

3.3 Computing Call Site Cacheability Measurements

The third offline analysis, M-Cachetor, computes cacheability measurements for call sites. Our target is call sites that have values returned from the callees. We are not interested in those that do not bring values back, because it is often unclear how to avoid re-executing such calls. Given a call site of the form $a = f(\ldots)$, its call site cacheability measurement (CCM) is determined by whether a always receives identical values. It is computed by either querying I-Cachetor for the ICM of the instruction (if a is a primitive-typed variable) or querying D-Cachetor for the DCM(s) of the allocation site(s) that create the objects a can point to. Formally, we give the definition of CCM as follows.

Definition 8 (Call Site Cacheability Measurement (CCM)) *For each call site $c : a = f(\ldots)$, its CCM is defined as*

$$CCM_c = \begin{cases} ICM_c & \textit{a is a primitive-typed var} \\ Avg_{o^n \in Alloc(a)} DCM_{o^n} & otherwise \end{cases}$$

where Alloc(a) is a set of allocation site nodes such that the objects created at these allocation sites may flow to a.

The set of allocation site nodes (*Alloc*(a)) can be obtained by traversing backward the dependence graph from the call site (into the callee) and following only value-dependence edges. This process is similar to what lines 18–24 do in Algorithm 1. Note that to compute CCM for a call site, we consider only whether the call returns identical values, regardless of its arguments. This definition may potentially expose more optimization opportunities—if the call produces the same output under different inputs, there may be some repeated computation inside the method that can be reused for increased efficiency. Eventually, call sites are ranked based on their CCMs and then reported to the user.

Another important optimizability measurement is the execution frequency of each allocation/call site. To take this into account, we take the top 100 allocation/call sites from their respective lists (ranked based on the DCMs and CCMs), and re-rank them based on their frequencies. These newly ranked lists contain information regarding not only cacheability but also execution "hotness"; the developer could thus focus her effort on fixing problems with allocation/call sites that are both cacheable and frequently executed.

4. DEPENDENCE GRAPH PROFILING

We have implemented Cachetor in the Jikes Research Virtual Machine (RVM) version 3.1.1 [16]. JikesRVM contains two Just-In-Time (JIT) compilers: a baseline compiler that directly translates Java bytecode into Assembly code, and an optimizing compiler that recompiles and optimizes hot methods for improved performance. The Cachetor instrumentation is performed on the high-level intermediate representation (HIR) generated by the optimizing compiler, and thus it runs in the optimizing-compiler-only mode.

As the parameters k (i.e., the number of value slots) and k' (i.e., the number of context slots) are determined by the user before the execution, all dependence graph nodes can be created at the compile time. Specifically, we inspect each HIR instruction during the compilation; if the instruction manipulates primitive-typed data, we allocate an $k \times k'$ array and each slot of the array represents a graph node for the instruction; if the instruction manipulates references, we only need to create an array of k' slots. We use a *shadow memory* to perform the dependence graph profiling. For each memory location l in the program, we maintain a shadow location l' that keeps track of the address of the graph node that represents the instruction instance that last writes into l. If l is a stack variable, l' is simply a new (32-bit) variable in the same method stack. To shadow heap locations, we introduce an additional (32-bit) field for each existing field in each class. To shadow an array with s slots, we modify the object allocator in the JikesRVM in a way so that an additional space of $s * 4$ bytes is allocated and appended to the space of the array upon its creation.

At each instruction that reads locations $l_1, l_2, \ldots, l_n$ and writes location l_0, our instrumentation code adds dependence graph edges in the following three steps: (1) the current (encoded) calling context and the value in l_0 are retrieved. These values are used to determine which dependence graph node n this particular instance of the instruction should be mapped to; (2) values contained in the shadow locations of $l'_1, l'_2, \ldots, l'_n$ are retrieved. These values correspond to (the addresses of) the dependence graph nodes that last write into $l_1, l_2, \ldots, l_n$, respectively; (3) we add a dependence graph edge between each node contained in l'_i and node n, representing a dependence relationship. The address of n is then written into the shadow location l'_0; this address will be retrieved later when another instruction uses l_0. Eventually, if the instruction does a pointer deference on l_i, the edge connecting l'_i and n is annotated with pointer dependence; otherwise, it is annotated with value dependence.

We create a tracking stack to propagate tracking information between callers and callees. In addition, instrumentation code is inserted before each call site and in the beginning of each method in order to calculate object contexts. When a method is executed, its current object context is stored in a local variable, which will be retrieved and used later to determine dependence graph nodes. It is important to note that Cachetor is thread-safe. We collect a VC-abstracted dependence graph per thread and eventually combine these graphs to compute various CMs.

5. EVALUATION

We have applied Cachetor to a set of 14 real-world programs, from both the DaCapo benchmark set [7] and the Java Grande benchmark set [2]. We are not aware of any publicly available tool that can provide similar diagnostic information to serve as basis for comparison. Therefore, in this section, we describe the overhead measurements and our experiences with finding and fixing performance problems using Cachetor.

All programs were executed with their large workloads. Experimental results were obtained on a quad-core machine with Intel Xeon E5620 2.40 GHz processor, running Linux 2.6.18. The maximal heap size was 4GB.

5.1 Overhead Measurements

Table 1 shows our overhead measurements. The parameters k and k' used to obtain the data are both 11. It appears that 11 is the largest prime number to which our benchmarks can scale. Increasing either k or k' to 13 causes some programs to run out of memory. Due to space limitations, we report only the worst-case performance measurements (for $k = k' = 11$). Using a smaller k or k' will significantly reduce the time and space overheads. User may use k and k' as tuning parameters to find the balance point between the precision of the report and the scalability of the analysis. Section (a) reports the characteristics of 14 benchmarks in term of the size of their VC-abstracted graphs (i.e., number of nodes and

Table 1: Our benchmarks, the characteristics of their VC-abstracted graphs, and the overhead measurements.

Program	(a) Graph characteristics				(b) Time overhead			(c) Space overhead		
	#Nodes	*#Edges*	*#Classes*	*#Methods*	T_0 *(s)*	T_1 *(s)*	*TO* (×)	S_0 *(MB)*	S_1 *(MB)*	*SO* (×)
antlr	333212	1217324	121	1297	13.48	7654.49	566.90	42.69	426.40	8.99
bloat	339295	371854	252	1780	69.42	2765.86	38.84	119.74	199.64	0.67
fop	188947	85003	671	2581	2.05	40.51	18.80	82.47	135.79	0.65
hsqldb	77484	16317	140	1126	10.50	45.81	3.36	295.06	457.40	0.55
luindex	122914	69281	118	652	13.79	3967.88	286.78	50.12	123.74	1.47
lusearch	109549	35726	110	549	5.52	935.25	168.41	67.91	179.90	1.65
pmd	201069	107503	391	2298	20.24	6438.53	317.09	96.54	174.71	0.81
xalan	246202	122172	353	2192	16.07	1016.12	62.23	169.43	226.77	0.34
avrora	541244	545131	377	1132	29.75	5332.26	178.24	91.70	254.46	1.77
sunflow	419705	1131193	140	537	53.22	12447.50	232.88	82.08	336.80	3.10
euler	62656	3821	8	38	14.11	3637.41	256.79	34.02	81.74	1.40
moldyn	20350	19761	13	76	33.36	8451.03	252.33	14.49	27.49	0.90
montecarlo	12749	14105	18	106	20.90	3088.01	146.75	549.81	824.77	0.50
raytracer	13464	252030	16	67	42.99	12852.51	297.97	12.56	74.76	4.95
GeoMean							201.96			1.98

edges) and source code (i.e., number of classes and methods that are executed and instrumented by Cachetor).

Section (b) and (c) of the table report the overheads of Cachetor. The running time measured for our tool (column T_1) includes both the actual execution time and the time for the offline analyses, which occur before the JVM exits. On average, the tool incurs a 201.96× overhead in execution time. The additional peak memory consumption is 1.98× larger than that of the original program across all benchmarks. This is due to the extra space for memory shadowing as well as the VC-abstracted graph. While both time and space overhead of our tool are too high for production executions, they may be acceptable for performance tuning and debugging. In addition, the high overhead has not prevented us from collecting data from real-world applications. This paper focuses on the demonstration of the usefulness of the technique, and future work may consider various optimization techniques, such as sampling and selective profiling of certain suspicious areas, to reduce Cachetor's overheads.

5.2 Case Studies

We have inspected the analysis reports for all the benchmarks in Table 1. This section describes our studies on five of them: *montecarlo*, *raytracer*, *euler*, *bloat* and *xalan*. It takes us 2 weeks to conduct the studies. We choose to report our experience with these benchmarks partly because they contain interesting (representative) coding patterns leading to repeated computations of identical data values, and partly because large performance improvements have been seen after the implementation of fixes (e.g., 98.7% space reduction for *montecarlo* and 20.5% running time reduction for *euler*). These reports were generated under the same analysis configuration and 30% was chosen to be the selection ratio r (described in Algorithm 1). We have compared the analysis reports generated under three different r (i.e., 10%, 30% and 50%), and find that r = 30% appears to be a balance point between the amount of optimization opportunities that can be found and the difficulty of developing fixes. While r = 10% identifies truly optimizable data structures, the size of each reported data structure is very small. On the other hand, r = 50% identifies large data structures which are often mixes of optimizable and non-optimizable parts of the heap. Even though Jikes RVM is the platform on which Cachetor was implemented and the reports were generated, the performance statistics before and after the fixes were collected using Java Hotspot 64-bit Server VM build 1.6.0_27. Hence, the performance improvements we have achieved are beyond the compiler optimizations even in a commercial JVM. In order to avoid compilation costs and execution noises, each application is run 5 times and the medians are compared and reported.

montecarlo is a financial simulation tool included in the Java Grande benchmark suite [2]. It uses Monte Carlo techniques to price products derived from the price of an underlying asset. The simulation generates 60,000 sample time series with the same mean and fluctuation as a series of historical data. The top two allocation sites from D-Cachetor's report are located at lines 184 and 185 of class `AppDemo`. These allocation sites create seed objects and header objects, respectively, in order to construct a `Task`, which is then saved into a list. After carefully inspecting the code, we find that we are actually able to implement a singleton pattern for these allocation sites—the list contains identical tasks, and hence, one single instance of the task would suffice for the execution. Even though the DCMs are relatively low because only part of the objects contains identical values, the next three allocation sites are located inside loops and their executions create objects with completely disjoint lifetimes. To optimize, we hoist these sites out of the loops and create a clone when required.

Many call sites reported by *M-Cachetor* have high CCMs (1.0). Among the top are call sites at line 155 of method `getResult` and line 104 of `setInitAllTasks` in the class `PriceStock`. These calls return exactly the same objects every time they are executed. In addition, the returned objects are used only as temporary objects that carry multiple values from the callees to the callers. We cache these objects into static fields to avoid re-invoking these calls. Further investigation of method `getResult` reveals that there is even no need for this method to create an object to hold data. Originally the program stores all the returned data into a list and wait to compute the final price. We eliminate the list and compute the final price on the fly every time a new result is returned. After implementing these fixes, we have seen a 98.7% reduction in the peak memory consumption (from 507268KB to 6320KB), a 70.0% reduction in the number of garbage collection runs (from 10 to 3), a 89.2% reduction in the time spent on GC (from 461ms to 50ms), and a 12.1% reduction in the total running time (from 12.146s to 10.678s). Each Java Grande benchmark reports an "efficiency" measurement based on a certain performance metric. For *montecarlo*, the fixed version has gained a 12.5% improvement according to its own efficiency measurement.

raytracer is a Java Grande benchmark that measures the performance of a 3D ray tracer. The scene contains 64 spheres and is rendered at a resolution of 500×500 pixels. Out of the top four allocation sites reported by D-Cachetor, two sites are located in

method shade. We inspect the code and find that objects created at the allocation site at line 337 always contain the same data and are discarded after the method returns. The other allocation site is located at line 335; even though its DCM is 1, we cannot develop a fix for it—objects created by this site are returned by method shade, which is a recursive method. The recursion prevents us from caching an instance of the allocation site in a field.

We additionally find that many allocation sites in the report create large numbers of objects of type Vec, each of which stores the results of certain computations in a method and is then returned to the caller of the method. While the DCMs of these allocation sites are not very high (i.e., only certain fields of the objects contain identical values), we manage to reuse one Vec instance across all these allocation sites and reset its content if necessary. We also find that in class Sphere, objects referenced by its field Vec b always have the same data content, because the field is completely unused after being initialized. Thus, we remove this field from the class. All these fixes have led to a 19.1% reduction in the running time (from 18.595s to 15.034s), a 33.3% reduction in the number of GC runs (15 to 10), and a 30.2% reduction in the GC time (39ms to 27ms). The benchmark-specific "efficiency" (i.e., the number of pixels per second) is improved by 20.3%, from 13453.95 to 16179.14. Only a 1.2% reduction is seen in the peak memory consumption.

euler is a Java Grande benchmark that solves the time-dependent Euler equations for the flow from a channel employing a structured, irregular 96×384 mesh. The top allocation site reported by D-Cachetor is at line 158 of class Tunnel, initializing a matrix with elements typed Statevector. This allocation site has a 1.0 DCM. After inspecting the code, we realize that immediately after this initialization point, the element of this matrix is replaced by another Statevector object created in method calculateDamping. Hence, we can safely remove this allocation site. The matrix element replacement leads us to inspect the method calculateDamping. In this method, three Statevector objects are used as value containers for computations and die after the method returns. While these three allocations sites do not have very high DCMs (i.e., only around 0.57), we come up with a fix by creating three static fields, one for each allocation site to cache its instance. A similar fix is employed for three Vector2 objects in method calculateDummyCells.

The report of M-Cachetor reveals more optimization opportunities. Among the top call sites are those that invoke method svect of class Statevector. svect returns a Statevector object that may be saved in the matrix created in class Tunnel mentioned above. These calls are located inside a loop, and thus they create large numbers of objects at run time. We develop a fix that makes svect return one single Statevector object and create a clone only when it receives different values or is assigned to the matrix. Altogether these fixes have reduced the running time from 5.344s to 4.246s (20.5%), number of GC runs from 5 to 3 (40.0%) and the GC time from 46ms to 25ms (44.8%). No significant reduction is seen in the peak memory consumption. There is a 27.9% "efficiency" improvement after the fixes are implemented.

bloat is a Java byte-code optimizer and analysis tool. The report of D-Cachetor points to the heavy use of the visitor pattern—many of the top allocation sites are related to visitor classes such as TreeVisitor and ComponentVisitor. *bloat* declares a visitor class for each program entity and creates an object of the class to visit each entity object. We find that these visitor objects contain the same auxiliary data and their lifetimes are completely disjoint. To optimize, we manually implement the singleton pattern for each visitor class and use a single visitor object to visit all program entities of the same type. Cachetor also reports that many allocation sites frequently create constant arrays, such as those at lines 1365 and 1400 of class Tree, lines 1658 and 1644 of class CodeGenerator, lines 330 and 336 of class TypeInferenceVisitor, and line 131 of class SSAGraph. These arrays contain offset values and serve no other purposes. Hence, we can safely cache these arrays into static fields. The running time and the peak memory consumption are reduced by 13.1% (from 26239ms to 22809ms) and 12.6% (from 177286KB to 154933KB), respectively. No significant reduction is seen in the number of GC runs and total GC time.

xalan is an XSLT processor for transforming XML documents. The top allocation sites on D-Cachetor's report are at lines 191 and 440 of class NodeSequence, where the content of a newly-created NodeVector object is only used to initialize another object. Lines 862 and 865 of class XPathContext create NodeVector objects whose content are always the same are also among the top results. We fix the problems by using static fields to cache the instances of NodeVector created by these allocation sites. The same fix can be applied to the allocation site at line 721 of class SAX2DTM. We also inspect call sites that have high CCMs from M-Cachetor's report. As these calls have very simple inputs, we perform lightweight profiling to understand what the frequent inputs and outputs are for them. Then we cache these frequent inputs and outputs into a HashMap; upon the execution of the call site, we first query the HashMap to see if the input of the call is in the map. If it is, the result is directly retrieved and used; otherwise, the call still needs to be executed. These fixes have resulted in a 5.2% reduction in the running time (from 8321ms to 7889ms). No reduction is seen in the memory consumption.

Table 2: Numbers of false positives identified in the reports of *D-Cachetor* and *M-Cachetor*.

Program	*D-Cachetor*	*M-Cachetor*
bloat	1	4
xalan	4	3
euler	1 (6)	7 (19)
montecarlo	2 (9)	6 (17)
raytracer	3 (13)	4 (17)

5.3 False Positives

Table 2 lists the numbers of false positives identified among the top 20 allocations sites in the reports of D-Cachetor and M-Cachetor. If the total number of reported allocation/call sites is smaller than 20, that number is shown in parentheses. An allocation/call site is classified as a false positive if either (a) it is clearly not cacheable or (b) we could not develop a fix to cache the data. Based on our studies, we have identified the following four sources of false positives. The first source is the handling of floating point values. Casting floating point values into integers before applying the hash function causes Cachetor to mistakenly classify different run-time values into the same slot. Future work may develop a lossless encoding for floating point values, using, for example, the IEEE 754 floating-point "single format" bit layout [1].

The second source of false positives is the context-sensitive reporting of allocation sites. Associating calling contexts with instructions is extremely important for Cachetor to distinguish objects based on their logical data structures. However, reporting cacheability of allocation sites separately for different contexts makes it difficult for the developer to understand and fix problems. This is the case especially if the DCMs of an allocation site differ significantly under different contexts. Future work may address this prob-

lem by recording richer calling context information which would allow the user to recover a context from the encoded number.

The third source is the missing of the actual values that are frequently produced in our analysis reports. For example, M-Cachetor reports call sites with high CCMs, but fails to provide information regarding what the frequent inputs and outputs are for the call sites. The developer has to do additional profiling to understand these values in order to develop fixes. While some extra effort is needed to find such frequent input and output values, this profiling is very lightweight and can be quickly implemented. We implement such profiling for almost every call site we inspect, and the burden of time is negligible.

Eventually, value hashing may give rise to false positives. In D-Cachetor's report for *montecarlo*, we have found a few allocation sites that have high DCMs but do not really contain identical values. One example is an allocation site that creates seed objects. Our code inspection reveals that a seed object is created based on a sequence of numbers, each of which is a multiple of 11. This hard-coded number 11 is coincidentally the same as the value of k we chose, causing instruction instances producing different values to be mapped into the same slot. However, hashing-induced false positives are not common and *montecarlo* is the only program where such misclassification was found. Future work may alleviate the problem by comparing the reports generated under different ks, and selecting only the common (top) allocation sites.

6. RELATED WORK

Software bloat analysis Dufour *et al.* propose dynamic metrics for Java [12], which provide insights by quantifying runtime bloat. Mitchell *et al.* [25] structure behavior according to the flow of information, though using a manual technique. Their aim is to allow programmers to place judgments on whether certain classes of computations are excessive. Their follow-up work [24] introduces a way to find data structures that consume excessive amounts of memory. Work by Dufour *et al.* finds excessive use of temporary data structures [13, 14] and summarizes the shape of these structures. In contrast to the purely dynamic approximation introduced in our work, they employ a blended escape analysis, which applies static analysis to a region of dynamically collected calling structure with observed performance problem. By approximating object effective lifetimes, the analysis has been shown to be useful in classifying the usage of newly created objects in the problematic program region.

Object Equality Profiling (OEP) [22] is a run-time technique that discovers opportunities for replacing a set of equivalent object instances with a single representative object to save space. JOLT [30] is a VM-based tool that uses a new metric to quantify *object churn* and identify regions that make heavy use of temporary objects, in order to guide aggressive method inlining. Work by Xu *et al.* [35, 36, 37, 38, 34] finds copy- and container-related inefficiencies. Jin *et al.* from [17] studies performance bugs in real-world software systems. These bugs are analyzed to extract efficiency rules, which are then applied to detect problems in other applications. Recent work by Nistor *et al.* [26] detects performance problems using similar memory-access patterns. Xu [33] proposes a technique to find reusable data structures. The technique gives a three-level reusability definition, and encodes instances, shapes, and data content of run-time data structures to find optimization opportunities. Unlike all the existing work on performance problem detection, Cachetor is the first attempt to use an abstracted dynamic dependence graph to find caching opportunities.

Control- and data-based profiling Profiling techniques have been proposed for various optimization and software engineering tasks: These techniques include dynamic dependence profiles [4], control flow profiles [6], and value profiles [10]. Research from [20, 45] studies the compressed representations of control flow traces. Value predictors [9] are proposed to compress value profiles, which can be used to perform various kinds of tasks such as code specialization [10], data compression [46], value encoding [39] and value speculation [21]. Research from [11] proposes a technique to compress an address profile, which is used to help prefetch data [15] and to find cache conscious data layouts [28]. Zhang and Gupta propose *whole execution traces* [42] that include complete data information of an execution, to enable the mining of behavior that requires understanding of relationships among various profiles. Ammons *et al.* [5] develops a dynamic analysis tool to explore calling context trees in order to find performance bottlenecks. Srinivas *et al.* [31] use a dynamic analysis technique that identifies important program components, also by inspecting calling context trees. Chameleon [29] is a dynamic analysis tool that profiles container behaviors to provide advice as to the choices of appropriate containers. The work in [27] proposes object ownership profiling to detect memory leaks in Java programs.

Dynamic dependence analysis and slicing Since first being proposed by Korel and Laski [18], dynamic slicing has inspired a large body of work on efficiently computing slices and on applications to a variety of software engineering tasks. A general description of slicing technology and challenges can be found in Tip's survey [32] and Krinke's thesis [19]. The work by Zhang *et al.* [40, 41, 43] has considerably improved the state of the art in dynamic slicing. The work from [44] is more related to our work in that the proposed event-based slicing approach uses pre-defined events to merge dependence graph nodes. However, this work targets automated program debugging, whereas the goal of the proposed work is to find caching opportunities to improve performance.

7. CONCLUSIONS

This paper presents a novel dynamic analysis tool, called Cachetor, that profiles a program to help the developer find caching opportunities for improved performance. Cachetor contains three different cacheable data detectors: I-Cachetor, D-Cachetor, and M-Cachetor, that find cacheable data at the instruction-, data-structure-, and call-site-level, respectively. To make Cachetor scale to real-world programs, we develop a novel dynamic technique that combines value profiling and dynamic dependence analysis in a novel way so that distinct values are used to abstract instruction instances. Based on the abstracted dependence graph, we develop three offline analyses to compute cacheability measurements that will be used by the three detectors to generate analysis report. We have implemented Cachetor on JikesRVM and evaluated it on a set of 14 large-scale, real-world applications. Our experimental results show that the overhead of Cachetor is large but acceptable, and large optimization opportunities can be quickly found by inspecting its reports.

8. ACKNOWLEDGMENTS

We thank Kathryn McKinley for her helpful comments on an early draft of this paper. We would also like to thank the anonymous reviewers for their constructive comments.

9. REFERENCES

[1] IEEE standard 754 floating-point. http://steve.hollasch.net/cgindex/coding/ieeefloat.html.

[2] The Java Grande benchmark suite. http://www2.epcc.ed.ac.uk/computing/research_activities/

java_grande/index_1.html.

[3] Websphere application server development best practices for performance and scalability. http://www-3.ibm.com/software/webservers/appserv/ws_bestpractices.pdf.

[4] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *PLDI*, pages 246–256, 1990.

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

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

[7] 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.

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

[9] M. Burtscher and M. Jeeradit. Compressing extended program traces using value predictors. In *PACT*, pages 159–169, 2003.

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

[11] T. M. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *PLDI*, pages 191–202, 2001.

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

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

[14] 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.

[15] Q. Jacobson, E. Rotenberg, and J. E. Smith. Path-based next trace prediction. In *MICRO*, pages 14–23, 1997.

[16] *Jikes Research Virtual Machine*. `http://jikesrvm.org`.

[17] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. In *PLDI*, pages 77–88, 2012.

[18] B. Korel and J. Laski. Dynamic slicing of computer programs. *Journal of Systems and Software*, 13(3):187–195, 1990.

[19] J. Krinke. *Advanced Slicing of Sequential and Concurrent Programs*. PhD thesis, University of Passau, 2003.

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

[21] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *MICRO*, pages 226–237, 1996.

[22] D. Marinov and R. O'Callahan. Object equality profiling. In *OOPSLA*, pages 313–325, 2003.

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

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

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

[26] A. Nistor, L. Song, D. Marinov, and S. Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *ICSE*, 2013. to appear.

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

[28] S. Rubin, R. Bodik, and T. Chilimbi. An efficient profile-analysis framework for data-layout optimizations. In *POPL*, pages 140–153, 2002.

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

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

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

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

[33] G. Xu. Finding reusable data structures. In *OOPSLA*, pages 1017–1034, 2012.

[34] G. Xu. CoCo: Sound and adaptive replacement of Java collections. In *ECOOP*, pages 1–26, 2013.

[35] G. Xu, M. Arnold, N. Mitchell, A. Rountev, E. Schonberg, and G. Sevitsky. Finding low-utility data structures. In *PLDI*, pages 174–186, 2010.

[36] 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.

[37] G. Xu and A. Rountev. Detecting inefficiently-used containers to avoid bloat. In *PLDI*, pages 160–173, 2010.

[38] G. Xu, D. Yan, and A. Rountev. Static detection of loop-invariant data structures. In *ECOOP*, pages 738–763, 2012.

[39] J. Yang and R. Gupta. Frequent value locality and its applications. *TOPLAS*, 1(1):79–105, 2002.

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

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

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

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

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

[45] Y. Zhang and R. Gupta. Timestamped whole program path representation and its applications. In *PLDI*, pages 180–190, 2001.

[46] Y. Zhang and R. Gupta. Data compression transformations for dynamically allocated data structures. In *CC*, pages 14–28, 2002.