Performance Problems You Can Fix: A Dynamic Analysis of Memoization Opportunities

Luca Della Toffola
Department of Computer Science
ETH Zurich, Switzerland

Michael Pradel
Department of Computer Science
TU Darmstadt, Germany

Thomas R. Gross
Department of Computer Science
ETH Zurich, Switzerland

Abstract

Performance bugs are a prevalent problem and recent research proposes various techniques to identify such bugs. This paper addresses a kind of performance problem that often is easy to address but difficult to identify: redundant computations that may be avoided by reusing already computed results for particular inputs, a technique called memoization. To help developers find and use memoization opportunities, we present MemoizeIt, a dynamic analysis that identifies methods that repeatedly perform the same computation. The key idea is to compare inputs and outputs of method calls in a scalable yet precise way. To avoid the overhead of comparing objects at all method invocations in detail, MemoizeIt first compares objects without following any references and iteratively increases the depth of exploration while shrinking the set of considered methods. After each iteration, the approach ignores methods that cannot benefit from memoization, allowing it to analyze calls to the remaining methods in more detail. For every memoization opportunity that MemoizeIt detects, it provides hints on how to implement memoization, making it easy for the developer to fix the performance issue. Applying MemoizeIt to eleven real-world Java programs reveals nine profitable memoization opportunities, most of which are missed by traditional CPU time profilers, conservative compiler optimizations, and other existing approaches for finding performance bugs. Adding memoization as proposed by MemoizeIt leads to statistically significant speedups by factors between 1.04x and 12.93x.

Categories and Subject Descriptors C.4 [*Performance of Systems*]: Measurement Techniques; D.1.5 [*Programming Techniques*]: Object-oriented Programming; D.2.5 [*Software Engineering*]: Testing and Debugging

General Terms Algorithms, Experimentation, Languages, Performance, Measurement

Keywords Memoization, caching, profiling, performance bugs

1. Introduction

Performance bugs are a prevalent problem [23, 26, 50] but existing approaches to identify, debug, and fix them are limited. The two most widely used approaches are CPU time profiling [19] and automated compiler optimizations. CPU time profiling reports code that is "hot" but cannot determine whether this code contains optimization potential and how to use this potential. Even worse, profilers may even mislead developers to refactor code with little or no performance improvements [29]. Automated compiler optimizations transform code into more efficient code but often fail to discover latent performance problems because a compiler of its static, over-approximated, view of the program.

Existing research approaches address these limitations through dynamic analyses that search for symptoms of performance bugs, such as unnecessarily high memory and CPU consumption [28, 43, 46], inefficient loops [31], and responsiveness problems [34]. By focusing on particular bottleneck symptoms, these approaches help developers understand the cause of a problem but not necessarily how to address it. In other words, the performance bugs may be relevant but not actionable. Recent work focuses on performance problems that developers are likely to fix [32] but is limited to a loop-related performance bug pattern. Overall, the challenge of pointing developers to actionable performance bugs still waits to be addressed.

To illustrate the challenges of finding easy to address performance bottlenecks, consider Listing 1, which illustrates a performance bug in the document converter tool suite Apache POI [1]. The method in the example analyzes a given string and returns a boolean that indicates whether the string matches one of several possible date formats. The method gets called frequently in a typical workload, possibly repeating the same computation many times. An easy way to optimize the method is *memoization*, i.e., to store the results from earlier calls of the method and to reuse these

```
1  class DateUtil {
2    private static final Pattern date_ptrn = Pattern.compile(..);
3    private static String lastFormat;
4    private static boolean cachedResult;
5    static boolean isADateFormat(String format) {
6      if (format.equals(lastFormat)
7        return cachedResult;
8      String f = format;
9      StringBuilder sb = new StringBuilder(f.length());
10     for (int i = 0; i < f.length(); i++) {
11       // [..] copy parts of f to sb
12     }
13     f = sb.toString();
14     // [..] process f using date patterns
15     cachedResult = date_ptrn.matcher(f).matches();
16     return cachedResult;
17   }
18 }
```

Listing 1: Memoization opportunity in Apache POI that can be used by adding the highlighted code.

results when the same input reappears. The highlighted code in the figure shows a simple implementation of memoization, which leads to speedups ranging from 7% up to 25% in different workloads we experiment with.

Unfortunately, a developer may miss this opportunity with CPU time profiling because the method is one of many hot methods. Likewise, the opportunity is missed by existing compilers [13, 47] because the method has side effects. Existing work on finding repeated computations [30] also misses this opportunity because that work focuses on call sites that always produce the same output, not on methods where multiple input-output pairs occur repeatedly.

To help developers find and fix performance bugs, this paper presents MemoizeIt, an iterative dynamic analysis to discover methods that may benefit from memoization. The key idea is to systematically compare the inputs and outputs of different invocations of the same method with each other. If a method repeatedly obtains the same inputs and produces the same outputs, it is reported as a *memoization candidate*. In addition, MemoizeIt reports hints on how to implement memoization for this method in an efficient way. The hints help the developer to implement a cache but ultimately, it is the developer's responsibility to implement a correct and efficient cache. MemoizeIt strikes a balance between compiler optimizations, which automatically transform the code but must guarantee that the transformation preserves the semantics of the program, and existing profilers, which focus on hot but not necessarily optimizable code.

An important insight of our work is that methods may benefit from memoization even though they have side effects. The cost for considering methods despite side effects is that MemoizeIt may report methods that cannot easily be memoized because it would change the program's semantics. We show that this problem is manageable in practice and that MemoizeIt reports valid memoization opportunities missed by all existing approaches we are aware of.

A major challenge for comparing the inputs and outputs of method calls in an object-oriented language is the high cost of comparing complex objects. We address this challenge with an iterative profiling algorithm that repeatedly executes a program while iteratively increasing the depth up to which it explores the object graph. The approach starts by comparing objects at a shallow level, without following their references, and prunes methods that certainly cannot benefit from memoization. Then, the approach re-executes the program to analyze the remaining methods at a deeper level. This process continues until each method has been found to be either not memoizable or has been fully explored. The iterative approach guarantees to report the same memoization opportunities as a heavyweight (and prohibitive) analysis that compares all method calls with each other in full detail. As a result, MemoizeIt can analyze real-world Java programs with complex heap structures in reasonable time.

We evaluate MemoizeIt by applying it to eleven real-world Java programs. In eight of these programs, the approach finds at least one previously unknown memoization opportunity that results in a statistically significant speedup. For example, MemoizeIt reports the opportunity in Figure 1 and suggests to add a single-element global cache, as shown in the highlighted code. We have reported this fix to the Apache POI developers and they have modified their code accordingly.[1] Adding the caches proposed by MemoizeIt preserves the semantics of the program and reduces the program's execution time by a factor between 1.04x and 1.27x using the profiling input, and gives up to 12.93x speedup with other inputs. MemoizeIt reports a ranked list of memoization opportunities, and it reports these beneficial opportunities as the first, second, or third method for the respective program. In contrast, traditional CPU time profiling reports the methods that contain these opportunities behind many other methods without suggesting any program fixes. Based on our reports, four of the detected memoization opportunities have already been used by the respective developers.

In summary, this paper contributes the following:

- *Profiling of memoization opportunities*. We present the first profiler that focuses on opportunities for method-level caching. The profiler allows developers to find easy to implement optimizations that are difficult to detect with existing profiling approaches.
- *Iterative refinement of dynamic analysis*. We present an algorithm that repeatedly executes a program to iteratively increase the degree of detail of a dynamic analysis while shrinking the set of program locations to analyze. The algorithm iteratively refines the set of memoization candidates, making the profiling approach applicable to large programs.
- *Profile-based cache suggestions*. We present a technique that gives hints on implementing memoization.

[1] Bug 55611 in the Apache-POI bug database.

- *Empirical evaluation*. We implement the approach into a practical tool and show that it effectively finds memoization opportunities in widely used Java programs.

Our implementation and all data required to reproduce our results are publicly available at [3].

2. Overview and Example

This section illustrates the key ideas of our approach with a self-contained example. Figure 1a shows a program that repeatedly calls two methods, `compute` and `append`. One of them, `compute`, redundantly performs a complex computation that can be avoided through memoization. The method computes a result that depends only on the given argument, and calling the method multiple times with equivalent arguments yields equivalent results. In contrast, the other method, `append`, cannot be memoized because it logs messages into a writer and because it increments a counter at each call. A static or dynamic analysis that conservatively searches for memoization opportunities [13, 47], misses the opportunity in `compute` for two reasons. First, the method has side effects because it creates a new object that escapes from `compute`. In general, memoizing a method with side effects may change the semantics of the program. Second, the method's return value has a different object identity at every call. In general, memoizing such a method may change the semantics because the program may depend on object identities. In the given program, however, the side effects of `compute` are redundant because the created `Result` objects are structurally equivalent and the program is oblivious of object identities. Therefore, memoizing the results of `compute` improves performance while preserving the program's semantics.

A key insight of our work is that a method may benefit from memoization even though it has side effects and even though the object identities of its inputs and outputs vary across calls. Instead of conservatively searching for memoization opportunities that can certainly be applied without affecting the semantics, we consider a method m as a potential memoization opportunity if all of the following memoization conditions hold:

- (MC1) The program spends a non-negligible amount of time in m.
- (MC2) The program repeatedly passes structurally equivalent inputs to m, and m repeatedly produces structurally equivalent outputs for these inputs. (We formally define "structurally equivalent" in Section 3.2.)
- (MC3) The hit ratio of the cache will be at least a user-defined minimum.[2] This condition ensures that adding a cache will lead to savings in time because the time saved by reusing already computed results outweighs the time spent for maintaining the cache.

MemoizeIt detects performance bugs that can be addressed through memoization with a dynamic analysis that checks, for each method that is called in the analyzed program execution, whether conditions MC1 to MC3 hold. To check for MC1, MemoizeIt uses a state of the art CPU time profiler to identify methods where a non-negligible amount of time is spent. To check for MC2, MemoizeIt records the inputs and outputs of methods to identify methods that repeatedly take the same input and produce the same output. To check for MC3, MemoizeIt estimates the hit ratio that a cache would have if the developer added memoization to a method. This estimate is based on the inputs and outputs observed in the execution.

To implement this idea, we must address two challenges. First, we must define which inputs and outputs of a method call to consider. Second, we must address the problem that the inputs and outputs of a call may involve complex heap structures that are too large to record in full detail for each call.

Challenge 1: Inputs and outputs A conservative approach to detect memoization opportunities must consider all values that a method execution depends on as the call's input, and all side effects and the return value of the call as the call's output. In this work, we deliberately deviate from this conservative approach to detect memoization opportunities in methods that have redundant side effects. As *input* to a call, MemoizeIt considers the arguments given to the method, as well as those parts of the call's target object that influence the method's execution. As *output* of a call, MemoizeIt considers the call's return value. These definitions of input and output may ignore some state that a method depends on and ignore some side effects that a method may have. E.g., in addition to method arguments and the target object, a method may depend on state reachable via static fields and on the state of the file system. Furthermore, a method may modify state reachable from the passed arguments and the state of the file system. Our hypothesis is that focusing on the inputs and outputs given above summarizes the behavior of many real-world methods well enough to decide whether these methods may benefit from memoization. Our experimental results validate this hypothesis.

Challenge 2: Complex heap structures In object-oriented programs, the inputs and outputs of method calls often involve complex objects. Recording these objects, including all other objects reachable from them, for each call does not scale well to large programs. Consider the inputs and outputs of method `append` in Figure 1a. Since we consider the state of the call target as part of the input, a naive implementation of our approach would have to record the state of `logger` at lines 6 and 8. This state includes the `Writer` object that `logger` refers to, which in turn refers to various

[2] The hit ratio of a cache is the number of times that a result can be reused over the total number of cache lookups. We use 50% as the default minimum hit ratio.

```
1  class Main {
2    static void main() {
3      Main main = new Main();
4      Logger logger = new Logger();
5      Result res1 = main.compute(new Input(23));
6      boolean ok1 = logger.append(res1);
7      Result res2 = main.compute(new Input(23));
8      boolean ok2 = logger.append(res2);
9      if (ok1 && ok2) System.out.println("done");
10   }
11   Result compute(Input inp) {
12     Result r = new Result();
13     // complex computation based on inp
14     r.p.fst = ..
15     r.p.snd = ..
16     return r;
17   }
18 }
19 class Input {
20   int n;
21   Input(int n) { this.n = n; }
22 }
23 class Result {
24   Pair p = new Pair();
25 }
26 class Pair {
27   int fst;
28   int snd
29 }
30 class Logger {
31   Writer wr = ..;
32   int ctr = 0;
33   boolean append(res) {
34     wr.write(ctr+": "+res);
35     ctr++;
36     return true;
37   }
38 }
```

(a) Program with memoization opportunity.

First iteration (depth 1):

Call	Input (target, arguments)	Output (return value)
compute (line 5)	Main , Input: n=23	Result —p→ some Pair
append (line 6)	Logger: ctr=0 —wr→ some Writer , Result —p→ some Pair	true
compute (line 7)	Main , Input: n=23	Result —p→ some Pair
append (line 8)	Logger: ctr=1 —wr→ some Writer , Result —p→ some Pair	true

Second iteration (depth 2):

Call	Input (target, arguments)	Output (return value)
compute (line 5)	Main , Input: n=23	Result —p→ Pair: fst=42, snd=23
compute (line 7)	Main , Input: n=23	Result —p→ Pair: fst=42, snd=23

(b) Iterative input-output profiling.

Potential performance bug in method `Main.compute(Input)`:

- Same input-output pair occurs twice.
- Suggestion: Add a global single-element cache

(c) Report produced by MemoizeIt.

```
1  private static int key = INVALID_CACHE; // Global cache key
2  private static Result cache = null;
   // Global cache value
3  Result compute(Input inp) {
4    if (key != INVALID_CACHE && key == inp.n) {
5      return cache;
6    } else {
7      Result r = new Result();
8      // complex computation based on inp
9      r.p.fst = ..
10     r.p.snd = ..
11     key = inp.n;
12     cache = r;
13     return cache;
14   }
15 }
```

(d) Method `Main.compute(Input)` with cache.

Figure 1: Running example.

other objects. In general, fully recording the state reachable from an object may involve arbitrarily many other objects, in the worst case, the entire heap of the program.

To address the challenge of recording large input and output objects, MemoizeIt uses an iterative analysis approach that gradually refines the set of methods that are considered as memoization candidates. The key idea is to repeatedly execute the program, starting with an analysis that records objects without following their references, and to iteratively increase the level of detail of the recorded inputs and outputs while pruning the methods to consider. After each iteration, MemoizeIt identifies methods that certainly fulfill the memoization conditions MC2 and MC3, and methods that certainly miss a condition. Methods that miss a condition are pruned and not considered in subsequent iterations.

Figure 1b illustrates the iterative profiling approach for our running example. We illustrate the recorded objects as heap graphs, where a node represents an object with its

primitive fields, and where an edge represents a reference. In the first iteration, MemoizeIt records inputs and outputs at depth 1, i.e., without following any references. E.g., the output of the first call to `compute` is recorded as a `Result` object that points to a not further analyzed `Pair` object. The information recorded at depth 1 allows MemoizeIt to decide that `append` cannot fulfill MC2 because there is no recurring input-output pair. The reason is that the value of `ctr` is 0 at line 6 but 1 at line 8. Note that MemoizeIt prunes method `append` without following the reference to the `Writer` object, i.e., without unnecessarily exploring complex heap structures.

After the first iteration, MemoizeIt keeps `compute` in the set of methods that may benefit from memoization and re-executes the program for the second iteration. Now, MemoizeIt records inputs and outputs at depth 2, i.e., it follows references from input and output objects but not references from these references. The lower part of Figure 1b shows the information recorded at depth 2. MemoizeIt now completely records all inputs and outputs of `compute` and determines that both calls to `compute` have structurally equivalent inputs and outputs, i.e., the method fulfills MC2. Furthermore, the method fulfills MC3 because memoizing the results of `compute` would lead to a cache miss at the first call and a cache hit at the second call, i.e., a hit ratio of 50%.

Since MemoizeIt has fully explored all methods after two iterations, it stops the analysis and reports `compute` as a potential memoization opportunity (Figure 1c). To help developers use this opportunity, MemoizeIt suggests how to implement memoization for every reported method. Based on the analyzed execution, the approach suggests to add a global single-element cache, i.e., to store the last observed input and output in static fields of `Main`, and to reuse the already computed output if the input matches the most recently observed input.

Figure 1d shows an implementation of MemoizeIt's suggestion. The optimized code has two additional static fields `key` and `cache` that store the most recently computed result and the corresponding input value `inp.n`. To avoid redundantly recomputing the result, the optimized method reuses the result whenever the same input value appears again. As a result, the program in Figure 1a performs the complex computation in `compute` only once.

3. Approach

This section describes MemoizeIt, an approach to find memoization opportunities through iterative profiling. The input to MemoizeIt is an executable program. The approach executes the program multiple times while applying dynamic analyses and reports performance bugs that can be fixed through memoization. MemoizeIt consists of four parts:

- *Time and frequency profiling.* This part executes the program once and uses traditional CPU time profiling to identify an initial set of methods that may benefit from optimization (Section 3.1).
- *Input-output profiling.* The main part of the analysis. It repeatedly executes the program to identify memoizable methods by analyzing the inputs and outputs of method calls (Section 3.2).
- *Clustering and ranking.* This part summarizes the analysis results and reports a ranked list of potential memoization opportunities to the developer (Section 3.3).
- *Suggest cache implementation*. This part suggests for each memoization opportunity how to implement a cache for the respective method (Section 3.4).

The remainder of this section describes each part in detail.

3.1 Time and frequency profiling

The first part of MemoizeIt uses state of the art CPU time profiling to identify the set of *initial memoization candidates*. We record, for each executed method m, the time t_m spent in m (including time spent in callees) and the number c_m of calls. Furthermore, we also measure the total execution time t_{prgm} of the program. As the initial set of memoization candidates, MemoizeIt considers all methods that fulfill three requirements. First, the average execution time of the method is above a configurable minimum average execution time: $\frac{t_m}{c_m} > t_{min}$. Second, the relative time spent in the method is above a configurable threshold: $\frac{t_m}{t_{prgm}} > r_{min}$. Third, the method must be called at least twice, $c_m \geq 2$. The first two requirements focus MemoizeIt on methods that are worth optimizing (MC1). The third requirement is a necessary condition for MC2 because a method can repeat a computation only if the method is called multiple times.

3.2 Input-output profiling

The core of MemoizeIt is input-output profiling, which computes a set of memoization candidates by comparing method calls with each other to check whether MC2 and MC3 hold.

3.2.1 Representing input-output data

To detect method calls that perform computations redundantly, MemoizeIt records the input and output of each call. For full precision, we could consider the following input and output data:

- Input state (before the call):
  - The state of the target object of the call. (*)
  - The state of each argument passed to the call. (*)
  - All heap state that is reachable via static fields.
  - The environment state outside of the program's heap, e.g., file system and network state.
- Output state (after the call):

- The four kinds of state listed above.
- The state of the return value of the call (if any). (*)

Recording all these inputs and outputs of each method call is clearly infeasible, partly because it would not scale to large programs and partly because acquiring the complete state is practically impossible. Instead, MemoizeIt focuses on those parts of the input and output state that are marked with (*). The rationale for focusing on these part of the input and output state is twofold. First, we made the observation that these parts of the state are sufficient to describe the relevant input and output state for many real-world methods. E.g., most methods read the arguments given to them, but only few methods depend on modifiable heap state reachable through static references. Second, recording some of the state that is ignored by MemoizeIt would lead to redundant information that does not improve the precision of the approach. E.g., recording the state of the target object after each call would often replicate the state recorded for the same target object before the next call. If a method is not memoizable because it depends on the state of the target object, then recording the target object's state once as part of the input is sufficient to identify this method as non-memoizable. To compare input and output data of method calls, the analysis flattens data items into a generic representation. The representation describes the data itself and, in case of complex object structures, the shape of the data item. The representation describes objects structurally and is independent of the memory locations where objects are stored or other globally unique identifiers of objects.

Definition 1 (Flattened data representation)**.** *A data item d is flattened as follows:*

- *If d is a primitive value, it is represented by its string representation.*
- *If d is an object, it is represented as a pair (R_n, F), where*
  - *R_n identifies the object d using its runtime type R and an identifier n that is unique within the flattened data representation,*
  - *F is a list of flattened data representations; each element of F represents the value of one of d's fields.*
- *If d is the `null` value, it is represented by $NULL$.*
- *If d is an object that is already flattened in this representation and that has the identifier R_n, it is represented by $@R_n$. We use this notation to deal with object structures that have cyclic references.*

To ensure that structurally equivalent objects have the same flattened representation, each identifier is unique within its flattened representation but not globally unique. Furthermore, the list F contains fields in a canonical order based on alphabetic sorting by field name.

For example, the return values of the two calls of `compute` in Figure 1 are both represented as:

$$(Result_1, [(Pair_1, [42, 23])])$$

To illustrate how the representation deals with cyclic data structures, consider a double-linked list built from instances of `Item` with fields `next`, `previous`, and `value`. For a two-element list with values `111` and `222`, the flattened representation is:

$$(Item_1, [(Item_2, [null, @Item_1, 222]), null, 111])$$

3.2.2 Comparing input-output data

The goal of input-output profiling is to evaluate MC2 and MC3 by identifying memoizable methods where multiple calls use the same input and produce the same output. To achieve this goal, the analysis summarizes each call into a tuple that represents the inputs and the output of the call.

Definition 2 (Input-output tuple)**.** *The input-output tuple of a call is $T = (d_{tar}, d_{p1}, \ldots, d_{pn}, d_{ret})$, where*

- *d_{tar} is the flattened representation of the target of the call,*
- *$d_{p1}, \ldots, d_{pn}$ are the flattened representations of the parameters of the call, and*
- *d_{ret} is the flattened representation of the call's return value.*

For each method m, the profiler builds a multiset $\mathcal{T}_m$ of input-output tuples observed for m, called the *tuple summary*. The tuple summary maps each observed tuple to the number of times the tuple has been observed during the program execution. When the profiler observes a call of m, the profiler creates an input-output tuple T and adds it to the tuple summary $\mathcal{T}_m$.

Definition 3 (Tuple multiplicity)**.** *The multiplicity $mult(T)$ of a tuple T in the tuple summary $\mathcal{T}_m$ is the number of occurrences of T in $\mathcal{T}_m$.*

E.g., the call in line 5 of Figure 1a gives the following input-output tuple:

$$T = ((Main_1, []), (Input_1, [23]), (Result_1, [(Pair_1, [42, 23])]))$$

This tuple T has $mult(T) = 2$ because the two calls at lines 5 and 7 have the same input-output tuple.

Based on the tuple summary for a method, MemoizeIt computes the potential hit ratio that a cache for the method may have:

Definition 4 (Potential hit ratio)**.** *For a method m with tuple summary $\mathcal{T}_m$, the potential cache hit ratio is:*

$$hit_m = \frac{\sum_{T \in \mathcal{T}_m} (mult(T) - 1)}{\sum_{T \in \mathcal{T}_m} mult(T)}$$

The potential hit ratio indicates how often a method execution could be avoided by reusing an already computed result. The hit ratio estimates how profitable memoization would be, based on the assumption that the return value for

a particular input is stored in a cache when the input occurs the first time and that the stored value is reused whenever the same input occurs again. The ratio is an estimate, e.g., because it ignores that a cache may have to be invalidated, a step that may reduce the number of hits.

Finally, MemoizeIt identifies methods that potentially benefit from caching:

Definition 5 (Memoization candidate)**.** *Method m is a memoization candidate if $hit_m \geq h_{min}$, where h_{min} is a configurable threshold.*

By this definition, memoization candidates fulfill both MC2 and MC3. In particular, a method that is called only once is not a memoization candidate, because the potential hit ratio of such a method is zero.

For the example in Figure 1b, the potential hit ratio of `compute` is 50% because the call in line 5 must compute the return value, but the call in line 7 can reuse the cached value.

3.2.3 Iterative refinement of memoization candidates

The approach described so far is effective but prohibitive for programs with complex heap structures. The reason is that recording input-output tuples requires the analysis to traverse all objects reachable from the target, the arguments, and the return value of a call. In the worst case, the analysis may traverse the entire heap multiple times for each call. To overcome this scalability problem, our analysis executes the program multiple times, while iteratively increasing the *exploration depth* up to which the analysis explores the object graph, and while shrinking the set of memoization candidates. After each iteration, the analysis discards two sets of methods: (i) methods that certainly do not satisfy Definition 5 and (ii) methods for which the input-output tuples include all objects and all their transitive references. We say that a method in the second set has been *fully explored*. To support iterative refinement of memoization candidates, we refine Definitions 1, 2 by including a bound k for the exploration depth:

Definition 6 (k-bounded flattened data representation)**.** *The k-bounded flattened data representation of a data item d is the flattened representation of d, where only objects reachable from d via at most k references are included.*

Similar to the above, we also adapt Definitions 2, 3, 4, and 5 to consider the exploration depth k. The k-bounded input-output tuple of a call is a tuple T_k where each element of the tuple is now k-bounded. For a method m with a k-bounded tuple summary $\mathcal{T}_{m,k}$, the k-bounded potential hit ratio becomes $hit_{m,k}$ and, a method m is a memoization candidate at depth k if $hit_{m,k} \geq h_{min}$.

For example, the previously illustrated call at line 5 of Figure 1a gives the following k-bounded input-output tuple for $k = 1$:

$$T = ((Main_1, []), (Input_1, [23]), (Result_1, [(Pair_1, [])]))$$

Input : Initial method candidate set $\mathcal{C}_{init}$
Output Candidates set $\mathcal{C}$, $\mathcal{T}_m$ for each method $m \in C$
:

```
1  k = 1
2  C = C_init
3  while (stopping condition != true) do
4  |   T_{m_1,k}, ..., T_{m_j,k} = IOprofile(C, k)
5  |   C_next = ∅
6  |   foreach method m ∈ C do
7  |   |   hit_{m,k} = computeHitRatio(T_{m,k})
8  |   |   if hit_{m,k} ≥ h_min then
9  |   |   |   C_next = m ∪ C_next
10 |   C = C_next
11 |   k = nextDepth(k)
```

Algorithm 1: Iterative refinement of memoization candidates.

That is, in the k-bounded flattened data representation an object that is not expanded is represented as the pair (R_n, []), where R is the runtime type and where n is the unique identifier.

Based on these adapted definitions, Algorithm 1 summarizes the iterative algorithm that refines the set of memoization candidates. The algorithm starts with an initial set $\mathcal{C}_{init}$ of memoization candidates provided by time and frequency profiling (Section 3.1). The outer loop of the algorithm iteratively increases the depth k and performs input-output profiling at each depth, which returns a list of k-bounded tuple summaries. Based on the tuple summaries, lines 5 to 10 compute the set $\mathcal{C}_{next}$ of methods to consider in the next iteration as the set of methods with a sufficiently high hit ratio.

The algorithm iterates until one of the following stopping conditions holds:

- all remaining memoization candidates in $\mathcal{C}$ have been fully explored;
- there are no more memoization candidates that satisfy MC3.
- a user-provided timeout is reached;

When reaching the stopping condition, the algorithm returns the set $\mathcal{C}$ of memoization candidates, along with the tuple summary $\mathcal{T}_m$.

For illustration, recall the example in Figure 1b. Initially, both methods `compute` and `append` are in the set $\mathcal{C}$ of candidates. After the first iteration, the check at line 8 finds that the hit ratio of `append` is zero, i.e., below the minimum hit ratio, where as the hit ratio of `compute` is 50%, i.e., above the threshold. Therefore, only `compute` remains as a candidate for the second iteration. After the second iteration, `compute` is still a memoization candidate and the algorithm stops because all calls have been fully explored.

Properties of iterative refinement. It is important to note that Algorithm 1 does not miss any memoization opportunities found by an exhaustive approach that analyzes all calls with unbounded tuples. Specifically, the iterative refinement of caching candidates provides two guarantees:

- If the analysis discards a method as non-memoizable at depth k, it would never find that the method requires memoization at a depth $> k$. That is, discarding methods is sound. The reason is that once two tuples are found to be different, this fact cannot be changed by a more detailed analysis (that is, a larger k), i.e., $\mathcal{C}$ is guaranteed to be a superset of $\mathcal{C}_{next}$.
- When a method is fully explored, the iterative approach yields the same set of methods as with unbounded exploration. This property is an immediate consequence of the first property.

As a result, iteratively refining memoization candidates reduces the complexity of input-output profiling without causing the analysis to miss a potential memoization opportunity.

To increase the depth k, Algorithm 1 uses a function $nextDepth$. This function must balance the cost of repeatedly executing the program against the ability to remove methods from the candidate set. Smaller increments result in more program executions, but they also allow the algorithm to discard methods at a lower k. E.g., suppose a method can be pruned from the candidate set at depth 2. If $nextDepth$ increases k from 1 to 10, the algorithm will unnecessarily explore the method's inputs and outputs at depth 10. In contrast, incrementing the depth by one allows the algorithm to prune the method after exploring it at depth 2. In our experiments we find that doubling the depth at each iteration, i.e., $k = 1, 2, 4, ..$, provides a reasonable tradeoff. To further reduce the runtime of Algorithm 1, future work may adapt $nextDepth$ based on knowledge from previous iterations. For example, the analysis could use larger increases of k if it discovers that many methods have deep reference structures.

3.2.4 Field access profiling

The following describes a refinement of input-output profiling that allows MemoizeIt to discover additional memoization opportunities and that improves the efficiency of MemoizeIt by reducing the size of flattened representations. The approach described so far considers all objects reachable from the input and output objects as part of the input and output, respectively. However, a method may read and write only parts of these objects. E.g., suppose that in Figure 1a, class `Main` has a field `f` that is not accessed by `compute`. Recording the value of `f` as part of the target object of `compute` includes unnecessary data because the memoizability of `compute` is independent of `f`. Even worse, suppose that `f`'s value differs between the two redundant calls in lines 5 and line 7. In this case, MemoizeIt would not report `compute` as a potential memoization opportunity because there would not be any repeated input-output tuple.

To avoid unnecessarily including fields of target objects in input-output tuples, we refine input-output profiling by considering only those fields of the target object as input to a method m that are used in some execution of m. To compute this set of *input fields* for a method m, MemoizeIt executes the program once before input-output profiling to track all field reads and writes. A field f is an input field of m if there exists at least one execution of m, including the callees of m, that reads f before writing to it. Other fields, e.g., a field that is never accessed in m or a field that is written by m before being read by m are not part of m's input.

To further reduce the overhead and improve the precision of the analysis, a similar refinement can be applied for other inputs and outputs, namely, method parameters and return object. In practice, we have not observed many cases where these additional refinements would improve precision.

3.3 Clustering and ranking

MemoizeIt constructs reports about potential memoization opportunities by clustering methods that should be inspected together and by ranking methods. To cluster methods that should be inspected together, MemoizeIt creates a static call graph and assigns two methods m_1 and m_2 to the same cluster if:

- m_1 is a direct caller of m_2, or
- m_1 is an indirect caller of m_2 and both methods are defined in the same class.

The clustering approach is based on the observation that a possibly memoizable method often calls other possibly memoizable methods. In this case, a developer would waste time by inspecting both methods separately. Instead, MemoizeIt presents both methods together.

The analysis ranks method clusters based on an estimate of their potentially saved time $saved_m = t_m * hit_m$ where t_m is the total time spent in the method during the initial time profiling run. For each cluster of methods, the analysis selects the method with the highest potentially saved time $saved_m$ and sorts all clusters by the potentially saved time of this method.

3.4 Suggesting a cache implementation

To use a memoization opportunity identified by MemoizeIt, a developer must implement a cache that stores previously computed results of a method for later reuse. Choosing an appropriate implementation is non-trivial and an inefficiently implemented cache may even reduce the performance of a program. In particular, a developer must take the following decisions. First, how many input-output pairs should the cache store? Common strategies include a single-element cache that remembers the last input-output pair and an associative map of bounded size that maps previously observed inputs to their computation output. Second, what

Table 1: Cache implementations suggested by MemoizeIt.

Size	Scope	Description
Single	Global	Stores the most recently seen input and output of all calls of the method, e.g., in static fields, and reuses the output when the same input appears in consecutive calls.
Single	Instance	For each target object, stores the most recently seen input and output of the method, e.g., in instance fields, and reuses the output when the same input appears in consecutive calls.
Multi	Global	Maps all inputs to the method to the computed output, e.g., in a static map, and reuses outputs whenever a previously seen input occurs.
Multi	Instance	For each target object, maps inputs to the method to the computed output, e.g., in an map stored in an instance field, and reuses outputs whenever an input has already been passed to this instance.

should be the scope of the cache? Common strategies include a global cache that stores input-output pairs for all instances of a class and an instance-level cache that stores input-output pairs for each instance of the class.

To help developers decide on an appropriate cache implementation, MemoizeIt suggests a cache implementation based on the observed execution. To this end, the approach considers four common kinds of caches (Table 1). For each possible cache implementation, the approach simulates the effects of the cache on the analyzed execution based on the recorded input/output tuples, and it computes the following data:

- *Hit ratio.* How often can the method reuse a previously computed result? Depending on the cache implementation, the hit ratio may differ from Definition 4, which assumes that a global, multi-element cache is used.
- *Invalidation.* Does the program have to invalidate the cache because returning a cached value would diverge from the method's actual behavior? The cache simulator determines that the cache needs to be invalidated if there is a cache hit (i.e., a call's input matches a stored input) but the cached output does not match the recorded output of the call.
- *Size.* For multi-element caches, how many input-output pairs does the cache store, assuming that it never evicts cache entries?

Based on these data, MemoizeIt suggests a cache implementation with the following algorithm. First, the approach removes all cache implementations that lead to a hit ratio below a configurable threshold (default: 50%). Second, the approach picks from the remaining cache implementations the top-most implementation as listed in Table 1. The table sorts cache implementations by how simple they are to implement and by the computational effort of inserting and looking up input-output pairs. As a result, MemoizeIt suggests the most simple and most efficient cache implementation that yields a large enough hit ratio.

4. Implementation

We implement MemoizeIt into a performance bug detection tool for Java programs. The implementation combines online and offline analysis and builds upon several existing tools. Time and frequency profiling (Section 3.1) builds on JFluid [14] included in NetBeans 7.3 [4]. Input-output profiling (Section 3.2) uses ASM-based instrumentation [9] to inject bytecode that traverses fields. For some container classes, the flattened data representation (Definition 1) contains internal details that are not necessary to determine whether two objects are conceptually equivalent. To deal with such classes, our implementation provides type-specific representations for arrays and for all classes implementing `java.util.Collection` or `java.util.Map`. Non-map collections and arrays are represented as a list of elements. Maps are represented as a list of key-value pairs. MemoizeIt summarizes flattened data representations into hash values using the `MurmurHash3` hash function [2], writes input-output tuples to a trace file, and analyses the file offline as described in Algorithm 1 and Section 3.4. For clustering optimization opportunities (Section 3.3), MemoizeIt uses Soot [41] to obtain call graphs. Our current implementation assumes that heap objects are not shared between multiple concurrent threads.

5. Evaluation

We evaluate the effectiveness of the MemoizeIt performance bug detection approach by applying it to widely used Java programs. The analysis discovers nine memoization opportunities that give rise to speedups between 1.04x and 1.27x with the profiling input, and up to 12.93x with other inputs. Four of these nine memoization opportunities have already been confirmed by the developers in reaction to our reports. In the ranked list of methods reported by MemoizeIt, the memoization opportunities are first, second, or third for the respective program. In contrast, traditional CPU time profiling often hides these opportunities behind dozens of other methods and fails to bring them to the developer's attention.

5.1 Experimental setup

Table 2 lists the programs and inputs used in the evaluation. We use all single-threaded programs from the DaCapo 2006-10-MR2 [8] benchmark suite (antlr, bloat, chart, fop, luindex, and pmd).[3] In addition we analyze Apache POI (a library for manipulating MS Office documents, 4,538 classes), the content analysis toolkit Apache Tika (13,875 classes), the static code checker Checkstyle (1,261 classes), and the Java optimization framework Soot (5,206 classes).

We apply MemoizeIt to each program with a profiling input. Since the speedup obtained by a cache depends on the input, we also experiment with other inputs to explore

[3] We exclude jython because its use of custom class loading breaks our instrumentation system. The remaining benchmarks are excluded because they are multi-threaded.

Table 2: Programs and inputs used in the evaluation, and a comparison between the iterative and exhaustive approaches. "Time" means the time for running the entire analysis (in minutes). "TO" means timeout. "Opp." is the number of reported opportunities. k_{max} is the maximum depth explored by Algorithm 1. The last column indicates whether the iterative approach outperforms the exhaustive approach.

Program	Description	Profiling input	Other input(s)	Exhaustive		Iterative			Iterative
				Time	Opp.	Time	Opp.	k_{max}	wins
Apache POI 3.9	Convert spreadsheets to text	Grades (40KB)	Statistics (13.5MB)	37	3	23	3	16	✓
Apache Tika 1.3 Excel	Convert spreadsheets to text	Two files (260KB)	Ten files (13.9MB)	56	1	58	1	64	✗
Apache Tika 1.3 Jar	Convert .jar files to text	Checkstyle, Gson	rt.jar, Soot	35	2	41	2	64	✗
Checkstyle 5.6	Check source code	Checkstyle itself	Soot	22	2	6	2	64	✓
Soot ae0cec69c0	Optim. & validate Java proj.	101 classes	Soot itself	TO	-	TO	13	1	✓
DaCapo-antlr	Benchmark input	Default	Large	TO	-	TO	3	16	✓
DaCapo-bloat	Benchmark input	Default	Large	TO	-	TO	4	8	✓
DaCapo-chart	Benchmark input	Default	Large	2	0	2	0	8	✓
DaCapo-fop	Benchmark input	Default	(none)	TO	-	18	2	128	✓
DaCapo-luindex	Benchmark input	Default	Large	TO	-	32	0	4	✓
DaCapo-pmd	Benchmark input	Default	Large	TO	-	TO	1	8	✓

the potential benefit of adding a cache. For DaCapo, we use the "default" inputs for profiling and the "large" inputs (if available) as additional inputs. For the other programs, we use typical inputs, such as a spreadsheet with student grades for the spreadsheet conversion tool Apache POI, or the Java source code of CheckStyle for the static analysis CheckStyle. Columns 3 and 4 of Table 2 summarize the inputs.

For each profiling input, we run MemoizeIt until Algorithm 1 has fully explored all memoization candidates or until a one hour timeout occurs. All experiments are done on an eight-core machine with 3GHz Intel Xeon processors, 8GB memory running 64-bit Ubuntu Linux 12.04.2 LTS, Java 1.6.0_27 using OpenJDK IcedTea6 1.12.5, with 4GB of heap assigned to the VM. We set the minimum average execution time t_{min} to 5μs because it filters most short methods, such as accessor methods, in our environment. To study the overhead of the profiler with a large set of memoization candidates, we set the minimum relative execution time r_{min} to 0.25%; to evaluate the reported memoization opportunities, we use the more realistic $r_{min} = 1\%$. For pruning reports, we set the minimum hit ratio h_{min} to 50%.

To measure the performance of the DaCapo benchmarks, we use their built-in steady-state measurement infrastructure. Because the other programs are relatively short-running applications, we measure startup performance by repeatedly running them on a fresh VM, as suggested by [16]. To assess whether memoization yields a statistically significant speedup, we measure the execution time 30 times each with and without the cache, and compute the confidence interval for the difference (confidence level 95%). We report a performance difference if and only if the confidence interval excludes zero, i.e., the difference is statistically significant.

5.2 Memoization opportunities found

MemoizeIt reports potential memoization opportunities for eight of the eleven programs. We inspect the three highest ranked opportunities for each program and implement a patch that adds a cache for the most promising opportunities (Table 3). The second-to-last and the last column of Table 3 show the speedup when using the profiling input and another input, respectively. When adding a cache, we follow the implementation strategy suggested by MemoizeIt. As we have only limited knowledge of the programs, we add a cache only if it certainly preserves the program's semantics. We use the programs' unit tests to check the correctness of the modified programs.

The memoization opportunities detected by MemoizeIt confirm two important design decision. First, considering complex objects in addition to primitive input and output values is crucial to detect various memoization opportunities. Five of the nine reported methods in Table 3 involve non-primitive values. IDs 1 and 5 have a complex target object; IDs 4, 6, and 8 have non-primitive target objects and non-primitive return values; ID 7 returns an integer array. These results underline the importance of having an analysis able to analyze complex objects. Second, several methods have side effects but nevertheless are valid optimization opportunities (IDs 1, 4, 5, 6, 7, and 8). These methods are memoizable because the side effects are redundant. These examples illustrate how our approach differs from checking for side effect-free methods.

In the following, we describe representative examples of detected memoization opportunities.

Apache POI The motivating example in Listing 1 is a memoization opportunity that MemoizeIt finds in Apache POI (ID 2). The analysis reports the method as memoizable because most calls (99.9%) pass a string that was already passed earlier to the method. The memoization opportunity is also reported when analyzing Apache Tika Excel (ID 3) because Tika builds upon POI. Adding a single element cache that returns a cached value if the string is the same as in the last call of the method gives speedups of 1.12x

Table 3: Summary of memoization opportunities found by MemoizeIt. "Rel. time" is the percentage of execution time spent in the method with the profiling input. The "Rank" columns indicate at which position a method is reported by inclusive CPU time profiling, self-time CPU time profiling, and MemoizeIt, respectively. All speedups are statistically significant, otherwise we write "-". For DaCapo-fop there is only a single input because DaCapo's large and default inputs are the same.

ID	Program	Method	Rel. time (%)	Calls	Pot. hit ratio (%)	Rank: CPU time Incl.	Rank: CPU time Self	Rank: MemoizeIt	Speedup: Profiling input	Speedup: Other input
1	Apache POI	HSSFCellStyle.getDataFormatString()	12.4	7,159	99.8	12	70	1	1.11 ± 0.01	1.92 ± 0.01
2	Apache POI	DateUtil.isADateFormat(int,String)	5.3	3,630	99.9	27	6	2	1.07 ± 0.01	1.12 ± 0.01
3	Apache Tika Excel	DateUtil.isADateFormat(int,String)	1.0	4,256	99.9	189	10	1	-	1.25 ± 0.02
4	Apache Tika Jar	CompositeParser.getParsers(ParseContext)	14.2	4,698	80.9	27	2	1	1.09 ± 0.01	1.12 ± 0.02
5	Checkstyle	LocalizedMessage.getMessage()	1.3	6,138	72.4	110	12	2	-	9.95 ± 0.10
6	Soot	Scene.getActiveHierarchy()	18.4	201	99	23	253	1	1.27 ± 0.03	12.93 ± 0.05
7	DaCapo-antlr	BitSet.toArray()	14.4	19,260	96.3	36	6	1	1.04 ± 0.03	1.05 ± 0.02
8	DaCapo-bloat	PhiJoinStmt.operands()	12.0	6,713	50.9	49	52	3	1.08 ± 0.03	-
9	DaCapo-fop	FontInfo.createFontKey(String,String,String)	2.2	5,429	98.9	105	2	2	1.05 ± 0.01	NA

```
1  class LocalizedMessage {
2    // set at startup
3    private static Locale sLocale = Locale.getDefault();
4    private final String mKey;    // set in constructor
5    private final Object[] mArgs; // set in constructor
6    private final String mBundle  // set in constructor
7
8    String getMessage() {
9      try {
10       // use sLocale to get bundle via current classloader
11       final ResourceBundle bundle = getBundle(mBundle);
12       final String pattern = bundle.getString(mKey);
13       return MessageFormat.format(pattern, mArgs);
14     } catch (final MissingResourceException ex) {
15       return MessageFormat.format(mKey, mArgs);
16     }
17   }
18 }
```

Listing 2: Memoization opportunity in Checkstyle.

and 1.25x for Apache POI and Apache Tika, respectively. We reported this problem and another performance problem (IDs 1 and 2) to the Apache POI developers who confirmed and fixed the problems.

Checkstyle The `LocalizedMessage` class of Checkstyle represents messages about violations of coding standards and its `getMessage` method constructs and returns a message string. Listing 2 shows the implementation of the method. The message string for a particular instance of `LocalizedMessage` is always the same because the message depends only on final fields and on the locale, which does not change while Checkstyle is running. Nevertheless, Checkstyle unnecessarily re-constructs the message at each invocation. MemoizeIt detects this redundancy and suggests to memoize the message. Memoization does not lead to a measurable speedup for the profiling input, which applies Checkstyle to its own source code because the Checkstyle developers adhere to their own coding standards. As an alternative input, we configure Checkstyle to search for duplicate code lines, which leads to many calls of `getMessage`. For this alternative input, adding a cache yields a 2.80x speedup (not reported in Table 3). When applying Checkstyle to a larger code base (the 160 KLoC of the Soot project), the benefits of the cache become even more obvious: the execution time is reduced from 6–7 minutes to about 40 seconds, giving a speedup of 9.95x.

Soot MemoizeIt identifies a memoization opportunity for a method, `Scene.getActiveHierarchy`, that already has a cache that is accidentally invalidated more often than necessary. The method either computes the class hierarchy of the program analyzed by Soot or reuses the already-computed hierarchy. MemoizeIt reveals this opportunity because the method repeatedly recomputes the same hierarchy in one of the last phases of Soot, i.e., at a point in time when the class hierarchy does not change anymore. The problem is that the existing cache is flushed every time before Soot validates the intermediate representation of a method body. To avoid redundantly computing the hierarchy, it suffices to remove the unnecessary flushing of the cache, which gives 1.27x speedup with the profiling input. An alternative, larger input that runs Soot on its own source code yields a speedup of 12.93x. The speedup for the second input is so much higher because the second input triggers significantly more recomputations of the class hierarchy than the profiling input. We reported this performance bug to the Soot developers who confirmed and fixed the problem.

5.3 Suggestions for implementing caches

When implementing caches for the reported optimization opportunities, we follow the implementation strategies suggested by MemoizeIt. Table 4 lists the data that the approach computes to suggest a cache implementation and the kind of cache that MemoizeIt suggests. For example, for memoization opportunity 4 (as listed in Table 3), MemoizeIt's simulation of different kinds of caches shows that a global single-element cache would achieve only 21% hits and require to invalidate the cache, whereas an instance-level single-element cache would have 82% hits and not require inval-

Table 4: Data computed for suggesting a cache implementation and suggested implementation.

ID	Instance-level					Global					Suggestion
	Single		Multi			Single		Multi			
	HR	I	HR	S	I	HR	I	HR	S	I	
1	100	no	100	1	no	88	no	100	12	no	single, global
2	-	-	-	-	-	97	no	100	3	no	single, global
3	-	-	-	-	-	97	no	100	3	no	single, global
4	82	no	79	1	no	21	yes	72	10	yes	single, instance
5	72	no	72	1	no	21	no	72	1,696	no	single, instance
6	99	no	99	1	no	50	no	99	2	no	single, instance
7	96	no	96	1	no	5	no	96	708	no	single, instance
8	60	no	60	1	no	4	no	60	2,655	no	single, instance
9	-	-	-	-	-	41	no	99	57	no	multi, global

HR=hit ratio, I=needs invalidation, S=size

idation. Therefore, MemoizeIt suggests to exploit this memoization opportunity with an instance-level single-element cache. For all but one of the memoization opportunities in Table 4, following MemoizeIt's suggestion yields a profitable optimization. The one exception is memoization opportunity 7, where the first suggestion does not lead to any speedup, but where the second suggestion, a global multi-element cache, turns out to be profitable. Such suboptimal suggestions are possible because the suggestions are based on a simple model of caching behavior, which, for example, ignores the effects of JIT optimizations.

Example of cache implementation. Listing 3 shows a cache implementation that exploits optimization opportunity 1 in method `getDataFormatString()`. Following MemoizeIt's suggestion to implement a global single element cache, we add three static fields (lines 3 to 6): Fields `lastDateFormat` and `lastFormats` store the input key of the cache; field `cache` contains the cached result. We modify the method so that it returns the cached result if the stored inputs match the current inputs (lines 9 to 13). Otherwise, the method executes as usually and fills the cache. MemoizeIt suggests that the cache may not require invalidation because the profiled executions do not trigger any path that requires invalidation. However, inspecting the source code reveals that `cloneStyleFrom(HSSFCellStyle)` writes into fields that may invalidate a previously stored result. To ensure that caching preserves the program's semantics for all execution paths, we invalidate the cache in lines 23 to 25. We reported this cache implementation to the Apache POI developers who integrated the change into their code.

5.4 Precision of the analysis

The effectiveness of an approach to find performance bugs largely depends on how quickly a developer can identify valuable optimization opportunities based on the output of the approach. MemoizeIt reports a ranked list of methods, and we expect developers to inspect them starting at the top. As shown by the "Rank, MemoizeIt" column of Table 3,

```
1  class HSSFCellStyle {
2    // inputs
3    private static short lastDateFormat = INVALID_VALUE;
4    private static List<FormatRecord> lastFormats = null;
5    // output
6    private static String cache = null;
7
8    String getDataFormatString() {
9      if (cache != null &&
10         lastDateFormat == getDataFormat() &&
11         lastFormats.equals(_workbook.getFormats())) {
12       return cache;
13     }
14     lastFormats = _workbook.getFormats();
15     lastDateFormat = getDataFormat();
16     cache = getDataFormatString(_workbook);
17     return cache;
18   }
19
20   void cloneStyleFrom(HSSFCellStyle source) {
21     _format.cloneStyleFrom( source._format );
22     if (_workbook != source._workbook) {
23       lastDateFormat = INVALID_VALUE; // invalidate cache
24       lastFormats = null;
25       cache = null;
26       // ...
27 } } }
```

Listing 3: Cache implementation in Apache POI.

the memoization opportunities that give rise to speedups are quickly found: five are reported as the top opportunity of the program and three are ranked as the second opportunity.

Comparison with CPU time profiling. We compare MemoizeIt to two state-of-the-art profiling approaches: (i) Rank methods by inclusive CPU time, that is, including the time spent in callees, and (ii) rank methods by CPU self-time, that is, excluding the time spent in callees. In Table 3, the "Rank, CPU time" columns show the rank that CPU time profiling assigns to the methods with memoization opportunities. Both CPU time-based approaches report many other methods before the opportunities found by MemoizeIt, illustrating a weakness of CPU time profiling: It shows where time is spent but not where time is wasted [31]. Instead of overwhelming developers with hot but not necessarily optimizable methods, MemoizeIt points developers to a small set of methods that are likely candidates for a simple and well-known optimization.

Non-optimizable methods. MemoizeIt may report methods that are not easily memoizable. We find two causes for such reports. First, some methods already use memoization but are reported because their execution time is relatively high despite the cache. Even though reports about such methods do not reveal new optimization opportunities, they confirm that our analysis finds valid memoization opportunities and helps the developer understand performance problems.

Second, some methods have non-redundant side effects that the analysis does not consider. For example, a method reads a chunk of data from a file and advances the file pointer. Although the method may return the same chunk of data multiple times, its behavior cannot be replaced by

```
1  class CachingBloatContext extends PersistentBloatContext {
2    Map fieldInfos; // Declared and initialized in superclass
3
4    public FieldEditor editField(MemberRef field) {
5      // Lookup in existing cache
6      FieldInfo info = fieldInfos.get(field);
7      if (info == null) {
8        // Compute field infos and store them for reuse
9        FieldInfo[] fields = /* Get declaring class */
10       for (int i = 0; i < fields.length; i++) {
11         FieldEditor fe = editField(fields[i]);
12           if (/* field[i] is field */) {
13             fieldInfos.put(field, fields[i]);
14             return fe;
15           }
16         ...
17       }
18       ...
19     }
20     return editField(info);
21 } }
```

Listing 4: Non-memoizable method in DaCapo-bloat.

returning a cached value because the file pointer must be advanced to ensure reading the expected data in later invocations of the method. In most cases, the analysis effectively addresses the problem of non-redundant side effects by considering the target object as part of the call's input, but the analysis fails to identify some side effects, such as advancing file pointers. A strict side effect analysis [47] could avoid reporting such methods but would also remove six of the nine valid optimization opportunities that MemoizeIt reveals, including the 12.93x-speedup opportunity in Soot.

Another potential cause for reporting non-memoizable methods, which does not occur in our experiments, are nondeterministic methods. MemoizeIt can filter such methods by checking for inputs that lead to multiple outputs.

Listing 4 shows an example of a memoization candidate in Dacapo-bloat, which we cannot further optimize because the method already uses memoization. At line 6 the parameter `field` is used as the key of an instance-level, multi-element cache. If `field` is not yet in the cache, then the associated field information is created and added to the cache. MemoizeIt reports this method because the hit ratio is 57.65%, which confirms the developers' intuition to implement a cache.

5.5 Iterative vs. exhaustive profiling

We compare MemoizeIt's approach of iteratively refining memoization candidates to a more naive approach that exhaustively explores all input-output tuples in a single execution. The last six columns of Table 2 show how long both approaches take and how many opportunities they report. The last column shows that the iterative approach clearly outperforms exhaustive exploration. For six programs, exhaustive exploration cannot completely analyze the execution within one hour ("TO") and therefore does not report any opportunities, whereas the iterative approach reports opportunities even if it does not fully explore all methods within the given time. For three programs, both approaches terminate and the iterative approach is faster. For the remaining two programs both approaches takes a comparable amount of time, while reporting the same opportunities.

5.6 Performance of the analysis

Our prototype implementation of the MemoizeIt approach yields actionable results after at most one hour (Table 2, column "Iterative, Time") for all programs used in the evaluation. Therefore, we consider the approach to be appropriate as an automated tool for in-house performance analysis.

The runtime overhead imposed by MemoizeIt depends on the exploration depth k. Figure 2 shows the profiling overhead (left axes) throughout the iterative refinement algorithm, i.e., as a function of the exploration depth k. The profiling overhead imposed by MemoizeIt is influenced by two factors. On the one hand, a higher depth requires the analysis to explore the input and output of calls in more detail, which increases the overhead. On the other hand, the analysis iteratively prunes more and more methods while the depth increases, which decreases overhead. The interplay of these two factors explains why the overhead does not increase monotonically for some programs, such as DaCapo-fop, and Apache Tika Excel. In general, we observe an increasing overhead for larger values of k because recording large object graphs requires substantial effort, even with a small set of memoization candidates.

Figure 2 also shows the number of memoization candidates (right axes) depending on the exploration depth. The figure illustrates that the iterative approach quickly removes many potential memoization opportunities. For example, MemoizeIt initially considers 30 methods of Apache Tika Excel and reduces the number of methods to analyze to 15, 9, and 6 methods after one, two, and three iterations, respectively.

6. Limitations

Details of cache implementation. MemoizeIt identifies memoization opportunities and outlines a potential cache implementation, but leaves the decision whether and how exactly to implement memoization to the developer. In particular, the developer carefully must decide whether memoization might break the semantics of the program and whether the benefits of memoization outweigh its cost (such as increased memory usage). Furthermore, MemoizeIt's suggestions may not be accurate because they are based on a limited set of executions. In particular, a cache may require invalidation even though MemoizeIt does not suggest it.

Input selection. As all profiling approaches that we are aware of, including traditional CPU time profiling, MemoizeIt requires the developer to provide input that drives the execution of the program. The memoization candidates provided by MemoizeIt are valid only for the given inputs, i.e., other inputs may lead to other sets of memoization candidates.

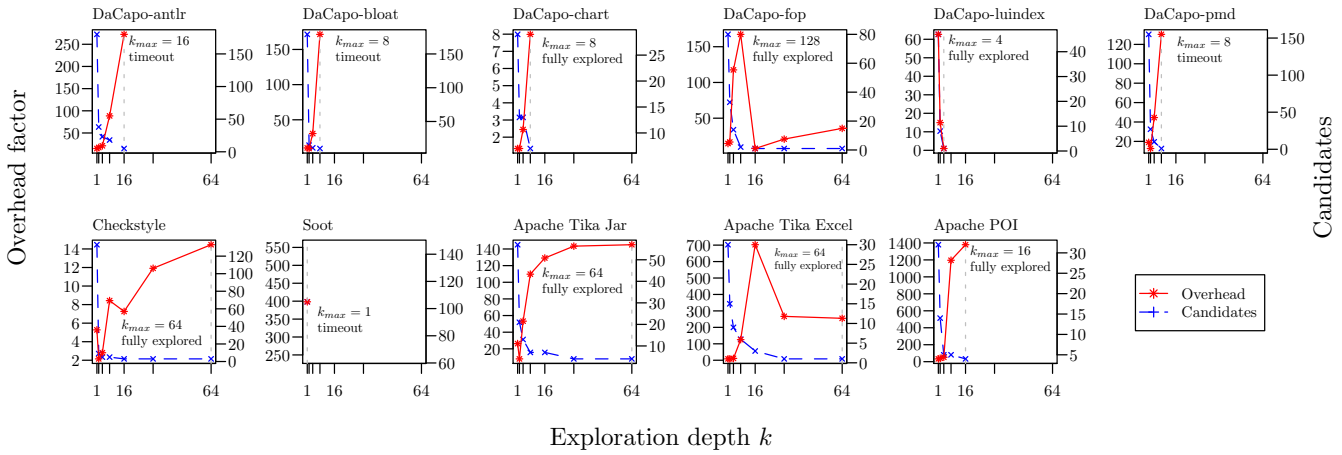

Figure 2: Number of memoization candidates and overhead depending on the exploration depth k. The maximum value for k is indicated as k_{max}.

7. Related Work

7.1 Detecting performance problems

Various dynamic analyses find excessive memory and CPU usage, e.g, by searching for equal or similar objects [28, 43], overuse of temporary structures [15], under-utilized or over-utilized containers [44], unnecessarily copied data [45], objects where the cost to create them exceeds the benefit from using them [46], and similar memory access patterns [31]. Yan et al. use reference propagation profiling to detect common patterns of excessive memory usage [48]. Jovic et al. propose a profiler to identify long latencies of UI event handlers [24]. All these approaches focus on particular "symptoms" of performance problems, which helps developers to identify a problem but not to find a fix for it. In contrast, MemoizeIt focuses on a well-known "cure", memoization, and searches for methods where this cure is applicable.

Several profiling approaches help developers find performance bugs due to asymptotic inefficiencies [11, 17, 51]. Xiao et al. describe a multi-execution profiler to find methods that do not scale well to large inputs [42]. These approaches relate the input of a computation to execution time to help developers reduce the complexity of the computation. Instead, MemoizeIt relates the input of a computation to its output to help developers avoid the computations.

Test generation can drive analyses that reveal performance and scalability problems. Burnim et al. generate tests to trigger the worst-case complexity of a program [10]. SpeedGun [33] creates tests for automated performance regression testing of concurrent software. Pradel et al. propose an algorithm that generates sequences of UI events to expose responsiveness problems in web applications [34]. As all dynamic analyses, our approach relies on inputs to exercise a program, and combining our work with test generation may further increase MemoizeIt's effectiveness.

7.2 Understanding performance problems

Approaches to diagnose performance problems include statistical debugging, which identifies program predicates that correlate with slow executions [39], the analysis of idle times in server applications caused by thread synchronization issues or excessive system activities [5], and dynamic taint analysis to discover root causes of performance problems in production software [6]. Other approaches analyze execution traces [49] or stack traces [21] from a large number of executions to ease performance debugging. These approaches are aimed at understanding the root cause of a performance problem, whereas MemoizeIt discovers problems that a developer may not be aware of.

7.3 Fixing performance problems

Nistor et al. propose a static analysis that detects loop-related performance problems and that proposes source code transformations to fix these problems [32]. MemoizeIt is orthogonal to their work because it addresses a different class of performance bugs.

7.4 Compiler optimizations

Compilers and runtime systems can automatically memoize the results of some computations. Ding et al. propose a profiling-based, static compiler optimization that identifies deterministic code segments and that adds memoization via a source to source transformation [13]. Their approach seems to be limited to primitive input values, whereas we address the problem of summarizing large object structures. Xu et al. propose a dynamic purity analysis and apply it in a compiler optimization that caches method return values of dynamically pure methods [47]. Their analysis considers the parameters of a method as its only input, and adds a global cache if a method is found to be a memoization candidate. Guo et al. propose an optimization that adds caches to long running functions for programs written in a dynami-

cally typed language [20]. They target applications that elaborate data in stages and add caches after each stage. As conservative compiler optimizations, the above approaches can memoize a computation only if it is side effect-free. As shown in Section 5.2, these optimizations miss various memoization opportunities identified by MemoizeIt because the methods have redundant side effects. Shankar et al. propose a dynamic analysis that identifies methods that create many short-lived objects and a runtime optimization that inlines those methods to enable other optimizations [37]. Shankar et al. propose a code specialization technique embedded in a JIT compiler that optimizes execution paths based on runtime values observed during execution [38]. Costa et al. propose a JIT optimization that speculatively specializes functions based on previously observed parameters [12]. Combined with other optimizations, such as constant propagation, their approach can have an effect similar to automatically added memoization. In contrast to MemoizeIt, their work focuses in primitive values instead of complex object structures. Sartor et al. [36] and Huang et al. [22] use a compiler and runtime infrastructure to optimize programs towards better usage of hardware caches. While memoization can indirectly affect hardware caching, our work focuses on software caches where performance is improved by avoiding repetitive expensive computations.

JITProf [18] profiles JavaScript programs to identify code locations that prohibit profitable JIT optimizations. Optimization coaching [40] aims at improving the feedback given by a compiler to the programmer to help the programmer enable additional optimizations. These approaches optimize a program for a particular compiler or execution environment, whereas MemoizeIt identifies platform-independent optimization opportunities.

7.5 Other related work

Ma et al. empirically study the effectiveness of caching web resources loaded in a mobile browser and suggest strategies for improving the effectiveness [27]. Our work shares the idea of analyzing caching opportunities but addresses redundant computations instead of redundant transfers of data over the network.

Depth-first iterative-deepening is a tree search algorithm that repeatedly applies a depth-first search up to an iteratively increasing depth limit [25]. Our iterative refinement of the dynamic analysis shares the idea of iteratively increasing the depth of exploration to reduce the complexity of the problem. Our approach differs from iterative-deepening because we do not perform a tree search and because we do not stop once a solution is found but prune methods from the search space that are guaranteed not to be memoization opportunities.

Incremental computation, a technique to update the results of a computation when the input changes [35], memoizes partial results across program executions. Instead, MemoizeIt focuses on memoization opportunities within a single execution.

Biswas et al. present DoubleChecker, an atomicity checker that executes the program twice, with an imprecise and a precise analysis, respectively [7]. Similar to our approach, DoubleChecker increases precision based on the results of a less precise analysis. Their approach may miss some atomicity violations due to different thread-schedules. In contrast to DoubleChecker, MemoizeIt does not bound the number of iterations a-priori and guarantees that the iterative analysis does not miss any memoization opportunities.

8. Conclusion

Performance bugs are difficult to detect and fix. To help developers with these tasks, we present MemoizeIt, a dynamic analysis that reveals methods that can be optimized through memoization and that gives hints on applying this well-known optimization. The approach reports methods where memoization is likely to be beneficial because the method repeatedly transforms the same inputs into the same outputs. Key to scaling the approach to complex object-oriented programs is an iterative analysis algorithm that increases the degree of detail of the analysis while shrinking the set of methods to analyze. We apply the approach to real-world Java programs and demonstrate that MemoizeIt finds nine previously unknown memoization opportunities. Adding memoization to these methods as suggested by MemoizeIt reduces the program execution time up to a factor of 12.93x. Our work lays the foundation for a practical tool that supports developers in improving their code's efficiency by providing them actionable reports about performance bugs.

Acknowledgements

We thank Michael Bond and the anonymous reviewers for their valuable comments and suggestions. This research is supported, in part, by SNF Grant CRSII2_136225 (FAN: Foundations of dynamic program ANalysis), by the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE, and by the German Research Foundation (DFG) within the Emmy Noether Project "ConcSys".

References

[1] Apache Poi. `http://poi.apache.org`.

[2] Google Guava. `http://code.google.com/p/guava-libraries`.

[3] MemoizeIt - Repository. `https://github.com/lucadt/memoizeit`.

[4] NetBeans. `http://www.netbeans.org`.

[5] E. R. Altman, M. Arnold, S. Fink, and N. Mitchell. Performance analysis of idle programs. In *OOPSLA '10*, pages 739–753.

[6] M. Attariyan, M. Chow, and J. Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production. In *OSDI '12*, pages 307–320.

[7] S. Biswas, J. Huang, A. Sengupta, and M. D. Bond. Doublechecker: Efficient sound and precise atomicity checking. In *PLDI '14*, pages 28–39, 2014.

[8] S. M. Blackburn et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06*, pages 169–190.

[9] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: a code manipulation tool to implement adaptable systems. In *Adaptable and extensible component systems*, Nov. 2002.

[10] J. Burnim, S. Juvekar, and K. Sen. WISE: Automated test generation for worst-case complexity. In *ICSE '09*, pages 463–473.

[11] E. Coppa, C. Demetrescu, and I. Finocchi. Input-sensitive profiling. In *PLDI '12*, pages 89–98.

[12] I. Costa, P. Alves, H. N. Santos, and F. M. Q. Pereira. Just-in-time value specialization. In *CGO '13*, pages 1–11.

[13] Y. Ding and Z. Li. A compiler scheme for reusing intermediate computation results. In *CGO '04*, pages 279–290, 2004.

[14] M. Dmitriev. Design of JFluid: a profiling technology and tool based on dynamic bytecode instrumentation. Technical Report SMLI TR-2003-125, 2003.

[15] B. Dufour, B. G. Ryder, and G. Sevitsky. Blended analysis for performance understanding of framework-based applications. In *ISSTA '07*, pages 118–128.

[16] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *OOPSLA '07*, pages 57–76.

[17] S. Goldsmith, A. Aiken, and D. S. Wilkerson. Measuring empirical computational complexity. In *FSE '07*, pages 395–404.

[18] L. Gong, M. Pradel, and K. Sen. JITProf: Pinpointing JIT-unfriendly JavaScript code. In *FSE '15*, 2015.

[19] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction '82*, pages 120–126.

[20] P. J. Guo and D. Engler. Using automatic persistent memoization to facilitate data analysis scripting. In *ISSTA '11*, pages 287–297.

[21] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie. Performance debugging in the large via mining millions of stack traces. In *ICSE '12*, pages 145–155.

[22] X. Huang, S. M. Blackburn, and K. S. McKinley. The garbage collection advantage: improving program locality. In *OOPSLA '04*, pages 69–80.

[23] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. In *PLDI '12*, pages 77–88.

[24] M. Jovic, A. Adamoli, and M. Hauswirth. Catch me if you can: performance bug detection in the wild. In *OOPSLA '11*, pages 155–170.

[25] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artif. Intell.*, pages 97–109, 1985.

[26] Y. Liu, C. Xu, and S. Cheung. Characterizing and detecting performance bugs for smartphone applications. In *ICSE '14*, pages 1013–1024.

[27] Y. Ma, X. Liu, S. Zhang, R. Xiang, Y. Liu, and T. Xie. Measurement and analysis of mobile web cache performance. In *WWW '15*, pages 691–701.

[28] D. Marinov and R. O'Callahan. Object equality profiling. In *OOPSLA '03*, pages 313–325.

[29] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Evaluating the accuracy of Java profilers. In *PLDI '10*, pages 187–197.

[30] K. Nguyen and G. Xu. Cachetor: Detecting cacheable data to remove bloat. In *FSE '13*, pages 268–278.

[31] A. Nistor, L. Song, D. Marinov, and S. Lu. Toddler: detecting performance problems via similar memory-access patterns. In *ICSE '13*, pages 562–571, 2013.

[32] A. Nistor, P.-C. Chang, C. Radoi, and S. Lu. CARAMEL: Detecting and Fixing Performance Problems That Have Non-Intrusive Fixes. In *ICSE '15*, 2015.

[33] M. Pradel, M. Huggler, and T. R. Gross. Performance regression testing of concurrent classes. In *ISSTA '14*, pages 13–25, 2014.

[34] M. Pradel, P. Schuh, G. Necula, and K. Sen. EventBreak: Analyzing the responsiveness of user interfaces through performance-guided test generation. In *OOPSLA '14*, pages 33–47, 2014.

[35] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *POPL '89*, pages 315–328.

[36] J. B. Sartor, S. Venkiteswaran, K. S. McKinley, and Z. Wang. Cooperative Caching with Keep-Me and Evict-Me. In *INTERACT '05*, pages 1–11.

[37] A. Shankar, M. Arnold, and R. Bodík. Jolt: lightweight dynamic analysis and removal of object churn. In *OOPSLA '08*, pages 127–142.

[38] A. Shankar, S. S. Sastry, R. Bodík, and J. E. Smith. Runtime specialization with optimistic heap analysis. In *OOPSLA '05*, pages 327–343, 2005.

[39] L. Song and S. Lu. Statistical debugging for real-world performance problems. In *OOPSLA '14*, pages 561–578.

[40] V. St-Amour, S. Tobin-Hochstadt, and M. Felleisen. Optimization coaching: optimizers learn to communicate with programmers. In *OOPSLA '12*, pages 163–178.

[41] R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON '99*, pages 125–135. IBM, 1999.

[42] X. Xiao, S. Han, D. Zhang, and T. Xie. Context-sensitive delta inference for identifying workload-dependent performance bottlenecks. In *ISSTA '13*, pages 90–100.

[43] G. Xu. Finding reusable data structures. In *OOPSLA '12*, pages 1017–1034, 2012.

[44] G. Xu and A. Rountev. Detecting inefficiently-used containers to avoid bloat. In *PLDI '10*, pages 160–173, 2010.

[45] G. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky. Go with the flow: profiling copies to find runtime bloat. In *PLDI '09*, pages 419–430, 2009.

[46] G. Xu, N. Mitchell, M. Arnold, A. Rountev, E. Schonberg, and G. Sevitsky. Finding low-utility data structures. In *PLDI '10*, pages 174–186, 2010.

[47] H. Xu, C. J. F. Pickett, and C. Verbrugge. Dynamic purity analysis for Java programs. In *PASTE '07*, pages 75–82.

[48] D. Yan, G. Xu, and A. Rountev. Uncovering performance problems in Java applications with reference propagation profiling. In *ISCE '12*, pages 134–144.

[49] X. Yu, S. Han, D. Zhang, and T. Xie. Comprehending Performance from Real-world Execution Traces: A Device-driver Case. In *ASPLOS '14*, pages 193–206.

[50] S. Zaman, B. Adams, and A. E. Hassan. A qualitative study on performance bugs. In *MSR '12*, pages 199–208.

[51] D. Zaparanuks and M. Hauswirth. Algorithmic profiling. In *PLDI '12*, pages 67–76.