Algorithmic Profiling

Dmitrijs Zaparanuks
University of Lugano
Dmitrijs.Zaparanuks@usi.ch

Matthias Hauswirth
University of Lugano
Matthias.Hauswirth@usi.ch

Abstract

Traditional profilers identify where a program spends most of its resources. They do not provide information about *why* the program spends those resources or about how resource consumption *would change* for different program inputs. In this paper we introduce the idea of *algorithmic profiling*. While a traditional profiler determines a set of measured cost *values*, an algorithmic profiler determines a cost *function*. It does that by automatically determining the "inputs" of a program, by measuring the program's "cost" for any given input, and by inferring an empirical cost function.

Categories and Subject Descriptors C.4 [*Performance of Systems*]: Measurement techniques; D.2.8 [*Software Engineering*]: Metrics—Performance measures

General Terms Performance, Measurement

Keywords Algorithmic Profiling, Algorithmic Complexity

1. Introduction

When developers need to understand and optimize the performance of their code, they use traditional profilers. Since the introduction of gprof [5], profilers have been focused on finding the locations in software that are responsible for excessive resource consumption, be that execution time, memory allocation, usage, or leaks, cache performance, various forms of contention, or even energy consumption. One commonality of such profiling approaches is that they only provide information about a specific program run. Different runs, with different inputs, may lead to different profiles. A profile created by a traditional profiler does not enable the developer to predict how the program might scale to larger inputs, and it provides only limited information for the reasons of the observed resource consumption.

The resource consumption of a software system is affected by three main factors, (1) the *algorithm* used, (2) the *problem size* (the program input), and (3) the *implementation* of that algorithm in a programming language running on a concrete execution platform. Traditional profilers provide measures of resource consumption that conflate all three factors: they only report the overall cost. They do not help in understanding how the cost was affected by the algorithm, the program input, and the underlying implementation.

Our new profiling approach, *Algorithmic Profiling*, addresses this limitation. Instead of providing a single number representing the cost of a given run, we provide a *cost function* that relates program input to algorithmic steps.

Algorithms researchers and advanced practitioners use cost functions when analyzing the complexity of their algorithms. They usually perform asymptotic analysis to bound cost. In contrast, our algorithmic profiling approach automatically determines approximate cost functions based on multiple program runs. Our cost functions are approximations, not bounds. They thus provide no guarantees, but they have the advantage that they represent the *expected* realistic cost, not a possibly loose worst case bound or an idealized, potentially unrealistic, average case cost.

The program in **Listing 1** shows an implementation of the insertion sort algorithm for a linked list. If we run this program under a traditional hotness profiler, we learn that most of the execution time is spent in method `sort`. A finer grained profile would tell us in which lines, statements, or bytecode instructions we spent most time. However, the profile would not provide any information about the algorithmic complexity of the program. An algorithmic profiler, on the other hand, would automatically produce results such as the ones shown in **Figure 1**. The figure shows three examples. In all three examples the program was run on a set of inputs that are representative of the expected application usage. In the top plot (a), that usage corresponds to sorting random lists, in the middle plot (b), it corresponds to sorting lists that are already sorted, while in the bottom plot (c), it correspond to sorting lists that are already sorted in reverse order. The x-axis in each plot shows the input size (corresponding to the length of the list), and the y-axis shows the number of algorithmic steps (corresponding to overall iteration counts). Each dot represents a program run with a different input, and the curves represent the approximate cost functions of the program expected for the given type of inputs.

These plots may look simple, and indeed, they are what a programmer would produce when manually determining the empirical cost function of an algorithm. The programmer would study the code to (1) locate the algorithm, to (2) determine the algorithm's essential operations (e.g., the comparisons or the swaps in the insertion sort), to (3) determine what the input of the algorithm could be (e.g., the linked list passed as an argument to the sort function), and to (4) determine how to quantify the input's size (e.g., by traversing the linked lists to count the number of nodes). He would then (5) instrument the program to count these operations and to measure the input's size. Our algorithmic profiling approach performs all the above steps *automatically* for arbitrary programs. It only needs the original program and a set of representative program executions to produce graphs like those in Figure 1.

These graphs of cost functions provide the developer with much deeper insight into program performance than a simple hotness profile. A hotness profile provides a single number (or a number for each code region). An algorithmic profile uncovers the relationship of execution cost and program input. It provides the programmer with scalability information by pointing out algorithms with high complexity, and it uncovers which inputs are the causes of long

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

PLDI'12, June 11–16, 2012, Beijing, China.
Copyright © 2012 ACM 978-1-4503-1205-9/12/06...$10.00

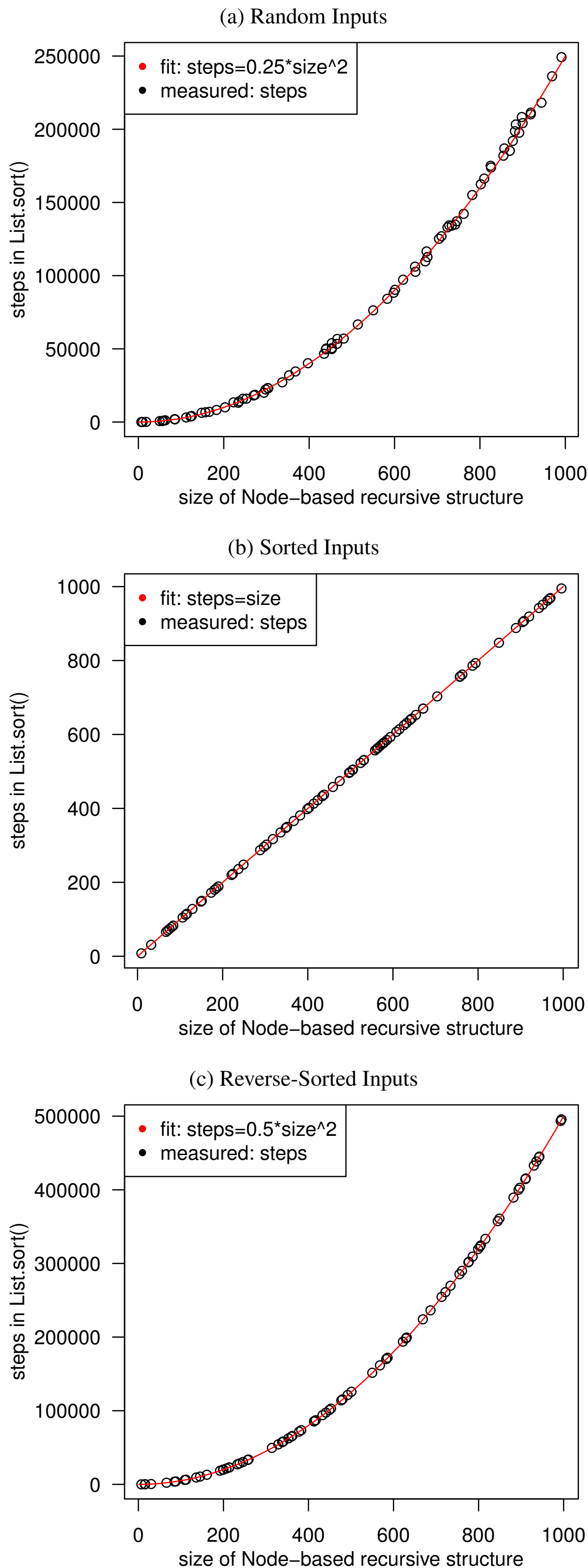

Figure 1. Cost Function of Insertion Sort

Listing 1. Insertion Sort

```
public class List {
 private Node head, tail;
 public void sort() {
  if (head==null || head.next==null) return;
  Node firstUnsorted = head.next;
  while (firstUnsorted!=null) {
   Node target = firstUnsorted;
   Node nextUnsorted = firstUnsorted.next;
   while (target.prev!=null &&
          target.prev.value>target.value) {
    final Node candidate = target.prev;
    final Node pred = candidate.prev;
    final Node succ = target.next;
    if (pred!=null) {
     pred.next = target;
    } else {
     head = target;
    }
    target.prev = pred;
    if (succ!=null) {
     succ.prev = candidate;
    } else {
     tail = candidate;
    }
    candidate.next = succ;
    target.next = candidate;
    candidate.prev = target;
   }
   firstUnsorted = nextUnsorted;
  }
 }
 public void append(int value) {
  final Node node = new Node(value);
  if (tail==null) {
   tail = node;
   head = tail;
  } else {
   tail.next = node;
   node.prev = tail;
   tail = tail.next;
  }
 }
}
```

running times. Moreover, by combining an algorithmic profile with a traditional hotness profile, the developer can understand all three factors (problem size, algorithmic complexity, and implementation cost) affecting the overall performance of the program.

This paper makes the following contributions:

1. We introduce *Algorithmic Profiling*, an approach to automatically infer approximations of the *expected algorithmic cost functions* of algorithm implementations. An algorithmic profiler requires no human intervention or code annotations. It simply analyzes program executions on a set of representative inputs.
2. We discuss different approaches to automatically determine the *input* of an algorithm and its size (the domain of the cost function). Traditionally, a human analyst has to define the concrete meaning of the abstract notion of "input size" (e.g., the size of this array, the number of nodes in this tree) for a given algorithm implementation.
3. We discuss different approaches to automatically determine the *cost* of an algorithm (the range of the cost function). Traditionally, a human analyst has to specify how to measure the cost of

Listing 2. Example Program

```
public class Main {
 public static void main(String[] args) {
  measure();
 }
 private static void measure() {
  for (int size=0; size<1000; size++) {
   for (int i=0; i<10; i++) {
    final List list = new List();
    constructRandom(list, size);
    sort(list);
   }
  }
 }
 private static void constructRandom(
     List list, int size) {
  final Random r = new Random();
  for (int i=0; i<size; i++) {
   list.append(r.nextInt(size));
  }
 }
 private static void sort(List list) {
  list.sort();
 }
}

public class Node {
 public Node prev;
 public Node next;
 public final int value;
 public Node(int value) {
  this.value = value;
 }
}
```

an algorithm (e.g., the number of comparisons, or the number of swaps).

4. We present an approach to automatically *partition a program into multiple algorithms*. Any realistic program involves many different algorithms, and the boundaries between algorithms are difficult to define. We introduce an intuitive heuristic to identify boundaries between different algorithms in a program.
5. We describe AlgoProf, our prototype implementation of an Algorithmic Profiler for Java programs, and we demonstrate it in several small case studies.

The remainder of this paper is structured as follows. Section 2 presents our approach, Section 3 describes our implementation, Section 4 demonstrates the approach, Section 5 discusses limitations and future work, Section 6 presents related work, and Section 7 concludes.

2. Approach

This section describes the idea of Algorithmic Profiling on a conceptual level. The next section will describe our prototype, AlgoProf, which implements some, but not all, of the key aspects of this approach. We explain our approach based on a running example, which consists of the full program (**Listing 2**) containing the insertion sort algorithm shown in Listing 1. The program constructs an unsorted linked list of `Node`s in `Main.constructRandom()` and sorts that list in `Main.sort()`. This process is repeated for lists of length 0 to 999, ten times for each length, in `Main.measure()`.

Most traditional profilers attach execution cost to syntactic constructs (e.g., methods or statements) of the program. **Figure 2** illustrates an example of such a profile, showing the calling context

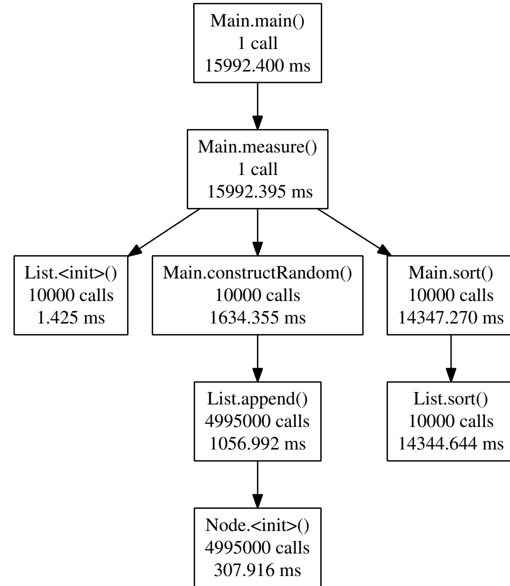

Figure 2. Traditional Profile: Calling Context Tree

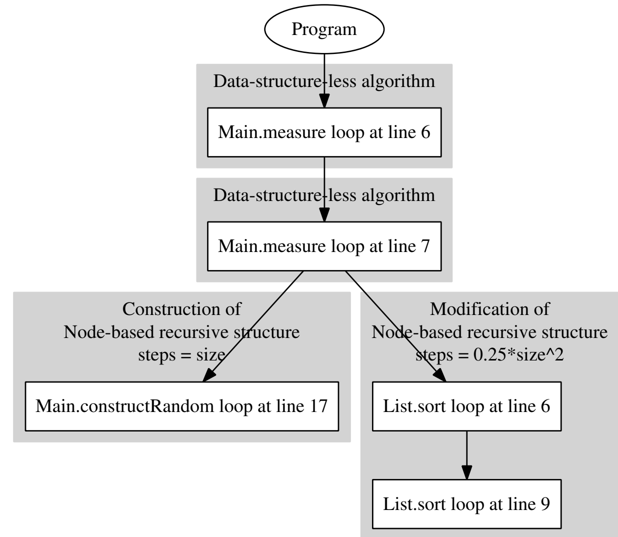

Figure 3. Algorithmic Profile: Repetition Tree

tree of our running example. Each method (or, more precisely, each calling context) is annotated with the number of times it was called and the total time spent in it (its hotness). The profile shows that `List.append` and the `Node` constructor are the most frequently called methods, and that `List.sort` is the hottest method (in terms of exclusive time). The profile does not provide any information for the cause of the hotness (why the method takes so much time), and it does not allow the developer to predict how different inputs would affect the hotness. These two aspects are important for understanding and improving program performance.

Our algorithmic profiles provide this missing information. They focus on *repetitions*, which are the essential ingredients of any algorithm [21]. We identify all *loops* in control-flow graphs and all *recursions* in the program's call graph. Instead of attributing execution cost to a calling context tree, we attribute execution cost

and inputs to a "repetition tree" (a dynamic loop and recursion nesting tree). **Figure 3** shows the repetition tree for our running example. It shows that our program contains five loops, and it shows their dynamic nesting. Instead of using cost metrics such as invocation counts, wall-clock time, or instruction counts, we use higher-level cost metrics such as repetition counts and data structure access counts. Moreover, unlike traditional profilers, we also determine the algorithm input and the size of that input. Given a set of program runs, each providing an input size and a cost, we then produce plots mapping input size to cost (like those in Figure 1) and we infer cost functions (like `steps = 0.25*size^2` in Figure 3).

For our running example, the annotation in the bottom right gray box in Figure 3 shows that our profiler automatically identified an algorithm that *modifies* a `Node`-based recursive structure (a data structure consisting of objects of class `Node`). This tells us that the algorithm does not just traverse the structure, but that it modifies it, without creating new `Nodes`. It also shows that, given a structure of *size* `Nodes`, this algorithm takes 0.25*size^2 algorithmic steps. This information represents a high-level summary of the algorithm's performance. A developer can retrieve the data from which that cost function was inferred (the plot in Figure 1 (a)), and get summary statistics (such as the number of times the algorithm was called, or the range of input sizes it encountered). Given this information – that the algorithm's expected complexity is quadratic, and that the sizes of the inputs are not negligible – the developer can decide to constrain the size of the input, or to replace the algorithm with a more efficient one.

2.1 Construct Repetition Tree

The repetition tree is our basic structure for representing a set of program executions. Figure 3 shows a repetition tree for a program without recursion. To construct repetition trees of recursive programs, we collapse recursive call chains, and we represent the header method of the recursive chain as a node in the repetition tree. The repetition tree thus consists of loop nodes and recursion header nodes. Each node represents a repetitive computation. It keeps track of each invocation of that computation (entrance to the loop, and entry call of the recursion header method). For each invocation of a repetitive computation, we gather the number of repetitions (counting loop back edge traversals, and counting subsequent calls to the recursion header method).

An *algorithm* corresponds to a connected subgraph of the repetition tree. It has a root node, to which we attribute cost and input size. It may also contain descendants of the root node, if they are deemed to be part of the same algorithm (see Section 2.5 for how we group repetition nodes into algorithms). Figure 3 represents each algorithm with a gray rectangle, annotated with information about its input and with an empirical cost function.

2.2 Measure Cost

Computational complexity theory uses *models of computation* that define the set of primitive operations used in the computation and their respective costs. These models, and their primitive operations, are chosen by the person performing the analysis. A commonly used model is the *random access machine model*, where the primitive operations are memory reads and writes, and where these operations have unit cost.

Our algorithmic profiling approach also admits multiple possible cost models. Our models are defined by their primitive operations. We abstract away from the underlying platform and focus on operations that are meaningful to the developer at the source code level:

Algorithmic Steps. An algorithmic step corresponds to one iteration of a loop or recursion. An algorithmic step is a generic operation that abstracts away from the specific operations performed in each repetition. It thus allows the comparison of the cost of arbitrary repetitions using a single measure.

Structure Reads. A structure read corresponds to a read of a reference in a recursive data type or a load of an element in an array. It allows us to distinguish between read-only traversals of structures and modifications of structures.

Structure Writes. Analog to a structure read, a structure write corresponds to an update of a reference in a recursive data type or a store of an element in an array.

Structure Element Creations. A structure element creation corresponds to the allocation of a new object of a recursive data type. It allows us to distinguish between modifications and constructions of a data structure.

Input Reads. An input read corresponds to a read operation of data from outside the program (e.g., a read from a file or network socket). Some algorithms may not operate on in-memory structures, but they may consume input from outside the program.

Output Writes. An output write corresponds to a write operation of data to the outside world (e.g., a write to a file or network socket).

2.3 Determine Inputs

To determine the inputs of an algorithm, an algorithmic profiler determines which data structures or external files the algorithm accesses.

Recursive Data Structures. To find the recursive data structures related to an algorithm, the profiler tracks accesses to field references within all recursive data structures (`Node.next` and `Node.prev` in our running example), and whenever an access occurs at runtime, it determines the current node in the repetition tree (the innermost loop or recursion), and it associates that data structure with the corresponding repetition node.

Arrays. To find arrays related to an algorithm, the profiler tracks all array loads and stores and associates the corresponding arrays with the current repetition tree node.

Program Inputs/Outputs. To find program inputs and outputs related to an algorithm, the profiler tracks all reads and writes to the external world, and associates the corresponding streams or file handles to the current repetition tree node.

Given our approach, an algorithm can be related to multiple inputs. For example, an algorithm may traverse a graph-like data structure and serialize it into a file, it may traverse a given data structure and produce a "translation" in the form of a different structure (e.g., convert a linked list into an array), or it may process information from two independent data structures (e.g., compute the dot product of two vectors). An algorithmic profiler will keep track of all the inputs it encounters.

Some algorithms (for example mathematical algorithms) do not operate on data *structures* and do not read or write external data, but they encode data in primitive types (e.g., in variables of type `int`). For such algorithms, determining the "input" (and input size) is difficult. For example, the input size n for multiplication (with a complexity of $O(n^2)$ when using the schoolbook long multiplication algorithm) is defined by the *number of digits* in the factors, while the input size m for the factorial algorithm (with a complexity of $O(m^2 \log m)$ when using bottom-up multiplication) is the *value* of the argument. Note that for arbitrary precision arithmetic, the algorithms will have to use either recursive types or arrays to represent their numbers, and thus our approach will be able to compute some measure of the input size.

2.4 Measure Input Size

To measure the input size, we traverse the recursive data structure, we count the array's elements, or we measure the size of the external file. In an executing program, structures are dynamic: they shrink and grow. Many algorithms do not just traverse a structure, but they modify it throughout their execution. There thus is no single size of a structure, and we have to measure the size of a structure every time an algorithm accesses that structure. To find a single number representing the size (needed to compute a cost function relating structure size to execution cost) we use the *maximum* size of the structure throughout the algorithm's operation. Using the maximum is reasonable as it produces intuitive results for the many common algorithms that *create* a structure (size is 0 at the beginning and N at the end), or algorithms that *destroy* a structure.

If we repeatedly want to measure the size of a continuously *changing* structure, we need to address the issue of identity. How do we know whether two structures that are not identical should be considered the same? Is a structure with one less element still the same structure? We answer these questions as follows. Each time a structure is accessed, we take a snapshot of that structure by traversing it. A snapshot of a structure I corresponds to the set S of all elements in that structure at that time. We use one of the following *equivalence criteria* to determine whether two snapshots represent the *same* structure.

All Elements Equivalent. Two snapshots S_1 and S_2 are equivalent if all elements of the two snapshots are equivalent ($S_1 \equiv S_2$).

Some Elements Equivalent. Two snapshots S_1 and S_2 are equivalent if some of the elements of the two snapshots are equivalent ($S_1 \cap S_2 \neq \emptyset$). This less strict equivalence criterion is not only useful because data structures evolve, but also because the traversals capturing the two snapshots may have started from different elements, and thus the two snapshots may not see all the elements of the entire structure (if the structure instance is not strongly connected).

Same Array. This equivalence criterion only works for arrays. Array elements are contained in an array object, and thus two snapshots that correspond to identical arrays ($I_1 \equiv I_2$) can be considered equivalent.

Same Type. Two snapshots are equivalent if they have the same type ($t(S_1) \equiv t(S_2)$). This criterion considers two entirely disconnected structure instances equivalent.

2.5 Group Repetition Nodes into Algorithms

Given a realistic program, does that program implement one big algorithm, or does it implement a collection of multiple algorithms? We argue that both answers are valid, and that algorithms can invoke other algorithms just like methods can invoke other methods. While a program could be described as one large algorithm, the computational constructs programmers usually call "algorithm" are more limited in focus. In our motivating example, sorting the list represents a traditional algorithm, and constructing the list represents another algorithm. The loop nest in the `measure` method is more difficult to classify: it could be considered yet another algorithm, each of its two loops could be considered an independent algorithm, or the two loops could be considered too trivial for being called an algorithm at all. To approximate an intuitive notion of algorithm, we partition the repetition tree of an entire program run into connected subgraphs using one of several possible strategies. Moreover, given an interactive visualization tool for the repetition tree, developers could group nodes into algorithms according to their own intuition.

Listing 3. Combining Costs

```
for (int o=0; o<3; o++) {
 ...
 for (int i=0; i<o; i++) {
  ...
 }
 ...
}
```

Our profiling approach uses an automatic strategy for grouping repetition tree nodes into algorithms. The general idea is to group parent and child repetitions that access at least one common input into a single algorithm. We use the equivalence criteria defined in the previous section to determine whether the two structure snapshots from the two repetitions represent the same structure. In our running example, the two loops in the sort method intuitively form a single algorithm. For this example, all of the applicable equivalence criteria (*Same Array* is not applicable for a recursive data structure) would lead to this intuitive grouping of repetition nodes. One could envision other strategies, such as the grouping of loops located in the same method (which would work for the sort algorithm in the running example).

2.6 Combine Costs

When two repetition tree nodes are grouped into an algorithm, the child's cost is added to the parent's cost. In particular, for a given invocation of the parent, the parent's overall cost is equal to the parent's cost (e.g., algorithmic steps) plus the sum of the child's costs across all invocations of the child inside that parent invocation.

In the example of **Listing 3**, the cost of an invocation of the outer loop would be 3 outer loop iterations plus (0+1+2) inner loop iterations, leading to a total of 6 algorithmic steps. By also counting the outer loop's iterations, this approach accounts for outer loop iterations even when the inner loop does not execute.

2.7 Infer Cost Function

Given a set of <input size, cost> tuples, an algorithmic profiler should produce plots like those in Figure 1. Those plots show the raw data and a cost function. The step of automatically inferring a cost function or estimating an upper bound asymptotic complexity from the raw data is the subject of study in the area of *empirical algorithmics* [8, 9, 14], which investigates the use of regression approaches and heuristics for this purpose. In this paper we do not discuss this step, and in our algorithmic profiling prototype we currently fit cost functions by hand.

2.8 Classify Algorithms

Besides inferring a cost function for an algorithm, an algorithmic profiler can provide additional information that helps in understanding the algorithm. In particular, given the information about the algorithm's input, the profiler can distinguish between several different kinds of algorithms:

Traversal. A traversal algorithm performs a read-only traversal of a recursive data structure or an array. It performs *Structure Reads*, but does not perform any *Structure Writes* or *Structure Element Creations*.

Modification. A modification algorithm modifies the links of a recursive data structure or the contents of an array. It performs *Structure Writes*, but does not perform any *Structure Element Creations*. As Figure 3 shows, the loop nest in `List.sort` is considered a "Modification of a `Node`-based recursive structure".

Construction. A construction algorithm performs *Structure Element Creations*, that is, it allocates objects of recursive types. As Figure 3 shows, the loop in `List.constructRandom` is considered a "Construction of a `Node`-based recursive structure".

Input. An input algorithm performs *Input Reads*, that is, it consumes external input.

Output. An output algorithm performs *Output Writes*, that is, it produces external output.

Data-structure-less algorithm. Any algorithm that does not fall into any of the above categories is considered data-structure-less. The absence of a measurable input also means that we are unable to infer the input size and cost function. As Figure 3 shows, the two loops in `Main.measure` are considered "Data structure-less algorithms".

Traversal, modification, and construction are mutually exclusive with respect to a given data structure. That is, if an algorithm constructs a specific data structure (e.g., the `constructRandom` loop in the running example creates a `Node`-based structure), it is classified as a construction of that data structure, but not as a modification or traversal of that data structure. This mutual exclusion is limited to operations on the same data structure, e.g., the same algorithm may traverse one data structure and construct a different data structure.

3. Implementation

To evaluate our algorithmic profiling approach we built AlgoProf, an algorithmic profiler for Java. AlgoProf analyzes and instruments Java bytecode.

3.1 Instrumentation

AlgoProf uses dynamic binary instrumentation to instrument the following constructs in the application code:

Loop entry and loop exit. These drive the construction of the loop nodes in the repetition tree.

Loop back edges. These support counting algorithmic steps.

Method entries and exits. These enable the construction of the recursion nodes in the repetition tree. Moreover, they enable the counting of algorithmic steps in recursive algorithms. By using a static analysis to determine headers in recursive method cycles [21], AlgoProf can limit this instrumentation to only those methods that are recursion headers.

Array and reference instance field accesses. These *ALOAD and *ASTORE as well as PUTFIELD and GETFIELD bytecode instructions enable the detection and measurement of input data and the counting of read or write-based costs. By using a static analysis to determine recursive data structures [22], AlgoProf can limit field access instrumentation to accesses of fields participating in a recursive cycle (e.g., `Node.next` and `Node.prev`, but not `Node.payload`).

Object allocations. These NEW bytecode instructions enable the counting of allocation-based costs. Static analysis can limit this instrumentation to allocations of instances of classes that are part of a recursive type (e.g., `Node`).

AlgoProf currently does not track external program input and output operations. Support for this could be added by instrumenting the Java I/O classes.

3.2 Dynamic Analysis

At runtime, AlgoProf is called from the instrumented code. It collects all the information necessary for producing an algorithmic profile. Most importantly, it incrementally builds a repetition tree for each thread executing in the program. The tree already folds recursive calls, so on any given path to the root, a given method can only occur once. To support the unwinding of recursive calls, AlgoProf also maintains an unfolded shadow stack (`stack`). Each stack element represents a loop or recursion and points to the corresponding repetition tree node. At any given time, AlgoProf maintains a reference to the current repetition tree node (`tn`). The element at the top of the shadow stack (`stack.top()`) usually points to the same repetition tree node pointed to by `tn`. AlgoProf handles calls from the instrumented code in the following ways:

Loop entry.
```
tn = tn.getOrCreateChild(loop)
stack.push(tn)
```

Loop exit.
```
remeasureInputs()
finalizeRepetition(tn)
tn = stack.pop()
```

Loop back edge.
```
tn.cost{STEP}++
```

Method entry.
```
header = tree.findOnPathToRoot(tn, method)
if (header!=null) {
  tn = header
  tn.cost{STEP}++
} else {
  tn = tn.getOrCreateChild(method)
}
tn.recursionDepth++
stack.push(tn)
```

Method exits.
```
tn.recursionDepth--
if (tn.recursionDepth==0) {
  remeasureInputs()
  finalizeRepetition(tn)
}
tn = stack.pop()
```

Array access.
```
tn.cost{type(array), LOAD/STORE}++
identifyAndMeasureArray(array, tn)
```

Reference instance field access.
```
if (partOfRecursiveType(type(object))) {
  tn.cost{id(object), GET/PUT}++
  tn.cost{id(object), type(object), GET/PUT}++
  identifyAndMeasureStructure(object, tn)
}
```

Object allocation.
```
if (partOfRecursiveType(type(object))) {
  tn.cost{type(object), NEW}++
}
```

Note that AlgoProf correctly handles exceptional control flow, i.e., when exceptions cause control to exit a loop or a method, AlgoProf performs the corresponding *Loop exit* or *Method exit* operation.

3.3 Measuring Cost

The `cost` field in the repetition tree nodes represents a map from specific primitive operations (on specific inputs) to their execution counts.

Algorithmic step. cost{STEP} → 15 means that the repetition performed 15 algorithmic steps.

Array access. cost{input#1, LOAD} → 10 means that the repetition performed 10 LOAD operations on an array known as input with ID 1.

Recursive structure access. cost{input#3, PUT} → 99 means that the repetition performed 99 PUT operations on the recursive structure known as input with ID 3.

Recursive structure access (by element type). cost{input#3, Vertex, PUT} → 33 means that the repetition performed 33 PUT operations on fields of type `Vertex` in the recursive structure known as input with ID 3 (e.g., a recursive structure modeling a graph as `Edge` and `Vertex` objects).

Recursive structure element creation. cost{ListNode, NEW} → 9 means that the repetition allocated 9 `ListNode` instances (a class that is part of a recursive structure).

The analysis calls `finalizeRepetition` whenever a repetition terminates (at a loop exit or at a return from the outermost call of a recursive method). This finalization of the repetition adds the contents of the `cost` field to the history of invocations of this repetition tree node. This means that each node in the repetition tree contains historical information (data structure sizes and operation counts) for each and every invocation of that repetition. While this can lead to large memory requirements for our profiler, keeping historic input size and cost information is necessary to infer cost functions. To reduce this overhead, an optimized version of a profiler could try to infer the cost function online, and discard the individual data points, or it could try to sample a subset of invocations for frequently invoked repetitions.

3.4 Measuring Input Size

To identify an input and to determine its size, we traverse the corresponding array or recursive structure. We measure input sizes in three possible situations:

`IdentifyAndMeasureArray` is called due to an array access. For array identification, out of the strategies described in Section 2.5, AlgoProf implements the *Some Elements Identical* strategy, because it is effective for algorithms that "resize" arrays by reallocation[1].

AlgoProf supports two strategies for measuring the size of an array: the *capacity* strategy uses the array's capacity (the number of elements the array can store), while the *unique element count* strategy traverses all elements of the array (for reference arrays, all non-null elements), and computes the size of the set of unique elements. This second strategy is useful to approximate the amount of space used in an array by algorithms that allocate an array but only use a small fraction of its capacity. However, it has the drawback that it does not count duplicate elements. This is particularly problematic for arrays of small primitive types (e.g., booleans or bytes).

AlgoProf treats multi-dimensional arrays analog to algorithmic steps. That is, it counts all elements of the top-level array and all elements of the lower-level arrays. For example, the 2-dimensional triangular array `new int[][] {new int[0], new int[1], new int[2]}` has a size of $3 + (0 + 1 + 2)$ elements, which is exactly the same as the number of algorithmic steps in the analog loop nest in Listing 3.

`IdentifyAndMeasureStructure` is called due to a recursive reference field access. For recursive data structure identification, out of the strategies described in Section 2.5, AlgoProf implements the *Some Elements Identical* strategy.

A recursive data structure can consist of multiple Java classes. For example, a graph can be modeled as a recursive structure involving a `Vertex` and an `Edge` class. AlgoProf provides a total object count (i.e., the total number of objects involved in a graph), and it provides separate counts of objects for each specific type (i.e., the number of `Vertex` objects involved in the graph, and the number of `Edge` objects involved in that graph). Many recursive structures also involve arrays. For example, a tree may consist of a `Node` class that has a `Node[]` field containing references to all its children. When traversing a recursive structure, we also traverse these arrays. In addition to object counts, AlgoProf also provides the counts of non-null array elements traversed in this way. Analog to object counts, AlgoProf provides a total reference count as well as separate reference counts by types. For example, for a graph modeled with a `Vertex` class that has a `Vertex[]` field containing references to all its incident vertices, the graph's edges are implicit and correspond to the references in the `Vertex[]`.

`IdentifyAndMeasureStructure` receives an object reference. It starts traversing the structure at that reference. If the structure is not strongly connected, then it will not reach all objects that might be deemed part of that structure (had the traversal started from a different object reference). This is one reason for why AlgoProf uses the *Some Elements Identical* strategy to determine the equivalence of two recursive structure snapshots.

`RemeasureInputs` is called when control exits the repetition (exit from a loop or recursive call chain). It enables an optimization: instead of taking a structure snapshot at every access to a structure, we only take two snapshots: first at the repetition's first accesses of the structure (starting from the first reference accessed), and a second time when the repetition exits (starting from the last reference accessed in that repetition). This way, if a repetition is a *Construction*, such as in one of the examples in **Listing 4**, we can still traverse and measure the completely constructed input in the end, but we do not need to traverse the structure at every access. We only need to memorize the one accessed reference at every access, so we know where to start our traversal from at the exit of the repetition.

3.5 Using AlgoProf

AlgoProf produces an algorithmic profile consisting of a repetition tree similar to the one shown in Figure 3. For each algorithm, it produces multiple plots of its complexity, based on the combinations of their inputs (sizes of all their accessed structures) and cost measures (algorithmic steps, the various structure access counts, and element allocation counts). Figure 1 (a) shows an example of such a plot for the sort algorithm in Figure 3, with the number of `Node` objects in the `Node`-based recursive structure as input size and algorithmic steps as the cost measure. AlgoProf currently does not automatically fit a cost function onto the measured data points, but we manually fit those functions using a statistics package. Given that we have several cost measures, and given that some algorithms access multiple inputs, the number of cost functions for each algorithm can become large. We use simple heuristics to automatically

[1] Implementations of resizable arrays allocate a new, larger, array when the current backing-array gets full. Treating the original (small) and the newly allocated (grown) array as the same input is essential when reasoning about an algorithm operating on such a dynamically resizable array.

Listing 4. First access in Constructions cannot see whole structure

```
Node constructListWithLoop(int size) {
  Node list;
  for (int i=0; i<size; i++) {
    Node head = new Node();
    // first PUTFIELD: reachable structure size 1
    head.next = list;
    list = head;
  }
  return head;
}
Node constructListWithRecursion(int size) {
  if (size==0) return null;
  else {
    Node list = constructList(size-1);
    Node head = new Node();
    // first PUTFIELD: reachable structure size 1
    head.next = list;
    return head;
  }
}
void constructPartiallyUsedArray() {
  int[] values = new int[1000];
  for (int i=0; i<10; i++) {
    // first IASTORE: array "size" 1
    values[i] = i*2;
  }
}
```

highlight useful ones: we focus on algorithmic steps as a cost, and we exclude those cost functions of inputs that never change in size or that cause constant cost.

Moreover, for realistic applications, we use traditional CCT hotness profiles (gathered with Java's built-in hprof profiler) to focus algorithmic profiling on hot regions of code.

4. Demonstration

We now summarize the behavior of AlgoProf on several examples representing creations and traversals of various data structure implementations, and we present two small, illustrative examples.

4.1 Handling Different Kinds of Data Structures

Table 1 provides an overview over 18 illustrative example programs we profiled with AlgoProf. Those programs essentially implement traversals of various data structure implementations. Each example focuses on one kind of data structure but implements several algorithms (e.g., building the structure, traversing the structure iteratively, and traversing the structure recursively).

The examples involve naked arrays, lists, trees, and graphs (column "Struct"). Column "Impl." shows how this structure is implemented (as an "array", or as a "linked" (recursive) data structure). Column "Linkage" shows how the elements of the structure are linked. Column "T" describes how one can define the structure's payload type (the type of the information stored inside the elements): the type of the payload is hard-coded (B), the structure uses inheritance to allow subclasses to define the payload type (I), or the structure uses generics (G) to allow defining the payload type. Column "Rem." contains further remarks: For arrays it specifies whether they are one or multi-dimensional, for dynamically resized array-based lists it describes how they grow (grow by one, or double in size), and for trees it describes their arity (binary or N-ary).

The remaining three columns summarize the results. Column "I" states whether all the inputs we as programmers considered inputs were detected (x) or not (-). For all these examples, AlgoProf

Listing 5. Repetition nest not grouped by AlgoProf

```
int[][] array = ...;

for (int i=0; i<array.length; i++)
  // no access to array[i] here
  for (int j=0; j<array[i].length; j++)
    array[i][j] = ...;
```

detected the inputs as expected. Column "S" shows that AlgoProf correctly measured the size of each input. Column "G" describes whether the loops we consider to be part of one algorithm were indeed grouped together (x) or not (-). In this column, a (*) means that we ended up with a correct grouping, but that a slight change in the algorithm's implementation could lead to an incorrect grouping. The way AlgoProf groups repetitions into algorithms does not work well for array-based systems. The reason is that in repetition nests that implement an algorithm, such as the one in **Listing 5**, sometimes only the innermost repetition actually accesses the array. That is, outside the innermost loop, the outer loop contains no *ALOAD or *ASTORE instructions that would access the array. Thus, AlgoProf does not merge the two loops into one algorithm, but it creates two separate algorithms instead: the innermost loop which is accessing the array, and the outer loop as a data-structure-less algorithm.

We believe that this limitation could be overcome with a data-flow analysis that determines which loops increment the indices used in the array accesses. For Listing 5, the outer loop increments variable i, which is then used in the array access in the inner loop. Such an approach could be considered the dual to the way we handle recursive structures: we look at the accesses of the recursive links (e.g., the `next` field in a linked list node), not at the accesses of the payload (e.g., the `value` field in a linked list node; or the element in an array).

Struct	Impl.	Linkage	T	Rem.	I	S	G
array	array	NA	B	1d	x	x	*
array	array	NA	B	2d	x	x	-
list	array	NA	B	double	x	x	*
list	array	NA	B	grow by 1	x	x	*
list	array	NA	G	grow by 1	x	x	*
list	array	NA	I	grow by 1	x	x	*
list	linked	directed	B		x	x	x
list	linked	directed	G		x	x	x
list	linked	directed	I		x	x	x
tree	array	NA	B	binary	x	x	*
tree	linked	directed	B	binary	x	x	x
tree	linked	bidi	B	binary	x	x	x
tree	linked	directed	B	n-ary	x	x	x
tree	linked	bidi	B	n-ary	x	x	x
graph	array	directed	B	2d	x	x	-
graph	linked	directed	B		x	x	x
graph	linked	bidi	B		x	x	x
graph	linked	unidirected	B		x	x	x

Table 1. Data Structure Examples

4.2 Uncovering Algorithmic Inefficiencies

Listing 6 shows an implementation of a dynamically-growing array-backed list. Given that Java's arrays cannot grow, such a list needs to allocate a new, larger array when it runs out of space in the current array. A naive developer will grow the array by one element (or by a constant number of elements) which will lead to quadratic cost. By changing one line in the code (grow the array by doubling its size), the cost can be made linear.

Listing 6. Growing an array-backed list
```
final ArrayList list = new ArrayList();
for (int i=0; i<size; i++) {
  list.append("n"+i);
}
...
public void append(final String value) {
  growIfFull();
  array[size++] = value;
}
private void grow() {
  if (size==array.length) {
    final String[] newArray =
      new String[array.length+1]; // naive
      new String[array.length*2]; // ideal
    for (int i=0; i<array.length; i++)
      newArray[i] = array[i];
    array = newArray;
  }
}
```

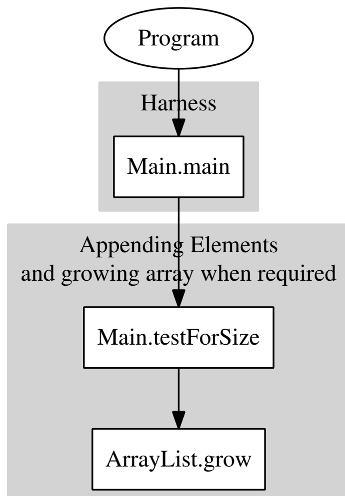

Figure 4. Repetition tree for growing an array-backed list

Figure 4 shows the program's repetition tree, indicating three repetition nodes grouped into two algorithms. The top algorithm consists of the harness running `testForSize` with various sizes. The lower algorithm consists of the loop calling `list.append()` and the inner loop that grows the list. Given that the two loops are grouped together, we see the total cost of appending *size* elements. **Figure 5** shows the corresponding cost function. For the naive developer's implementation, the cost grows quadratically in the size of the list. For the ideal implementation, the cost grows linearly.

4.3 Being Agnostic to Programming Paradigm

The insertion sort implementation in Listing 1 is imperative, iterative, and uses a mutable data structure. Does AlgoProf produce the same profile for an insertion sort implementation that is functional, recursive, and uses an immutable data structure? We found that the profile was almost identical[2]. The repetition tree contains the same repetitions as the tree in Figure 1, and the repetitions are grouped into the same algorithms with the same complexities. This demonstrates one of the key points about algorithmic profiling: the implementations may look entirely different, but their automatically generated algorithmic profiles agree.

[2] `http://sape.inf.usi.ch/algorithmic-profiling`

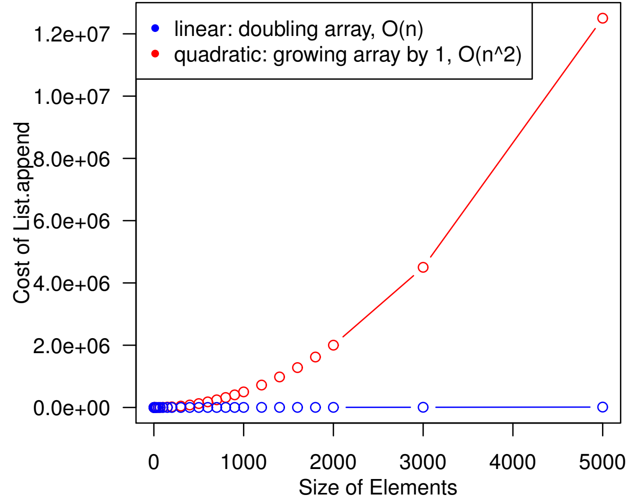

Figure 5. Cost functions for growing array by 1 and doubling array

5. Limitations and Future Work

AlgoProf is based on heuristics. Its grouping of repetitions into algorithms and its notion of "input" (the way it identifies the data an algorithm is tied to) may not always match the developer's intuition. Moreover, AlgoProf can only infer a cost function for algorithms that operate on data structures, because it cannot infer any input size for algorithms that operate on primitive data types. Finally, its cost functions only represent approximations. However, we believe that the repetition tree, the (partial) grouping of repetition tree nodes into algorithms based on the data they access, the classification of these algorithms (into construction, modification, traversal, input, and output), and the approximations of their cost functions, provide useful information to a developer. We are not aware of any other profiling approach that can produce such information automatically.

The probably most severe limitation of AlgoProf is its overhead, both in terms of space and time. Realistic benchmarks execute several orders of magnitude slower under AlgoProf. However, AlgoProf only represents an initial prototype, without any significant optimizations. There is a clear need – and we believe a great potential – for optimizations. For example, taking complete snapshots of structures at ever access, or even at every start and end of a repetition, and storing the complete snapshots in memory, is wasteful, and incremental approaches, together with more powerful static analyses, could probably drastically reduce the overhead of our current prototype. Similar gains might be possible by piggybacking data structure size measurements on the heap traversals performed by the garbage collector

While AlgoProf correctly deals with multiple threads, our approach specifically targets sequential algorithms. AlgoProf produces a repetition tree for each thread, and it completely ignores any communication between threads.

Besides investigating possible optimizations for our approach, we also would like to perform a more thorough evaluation. AlgoProf cannot possibly be "correct" in what it deems an algorithm, that algorithm's input, and a useful measure of cost. All it can do is match a human developer's intuition. Thus, it would be interesting to explore how algorithmic profiles help developers to detect and fix performance bugs, and which of its strategies and heuristics are most effective for this purpose.

6. Related Work

Goldsmith et al. [4] had a similar goal to ours: they ran a program given several inputs, measured cost and fitted a curve. However, except for the cost, which was automatically determined by counting basic block invocations, the other aspects (e.g., algorithm identification and input size determination) had to be performed manually. Later work [3] proposed to switch the cost metric from basic block counts to loop iterations, but it ignored recursions, and it still required all the former manual interventions. Related work in empirical algorithmics [8, 9, 14] discusses the difficulties in inferring algorithmic cost functions and asymptotic complexity from experimental data. And further research analyzes program executions to extract performance models other than cost functions, for example models in the form of layered queueing networks [6].

Jump [7] introduces dynamic shape analysis, focusing on the shapes and sizes of recursive structures in the heap. Prior research [13, 15] performed shape analysis statically. We traverse recursive heap structures at runtime, when identifying inputs, and we measure their sizes. We do not, however, try to infer any shape properties. We possibly might benefit from understanding their shape properties, for example to incrementalize our approach to measuring structure sizes.

Further work on analyzing data structures at runtime includes approaches to find costly data structures [10], to measure object lifetimes [2], to track data structures for post-mortem analysis [16], to profile the use of recursive data structures [11, 12], to perform "copy profiling" [19], to find "low-utility data structures" [20], to perform "container profiling" [17] and to find "inefficiently used containers" [18]. It would be interesting to try to combine those approaches with algorithmic profiling.

Recently, Bergel et al. presented domain-specific profiling [1], an approach that also produces higher-level profiles. That work takes almost the opposite direction as ours: algorithmic profiling is a domain-*independent* profiling approach that strives to achieve a level of abstraction without requiring any developer involvement.

7. Conclusions

Algorithmic profiling is an automatic approach to infer the computational cost of the algorithms embodied in a program. Given a program and a set of representative program executions, an algorithmic profiler identifies algorithms and inputs to those algorithms, measures meaningful notions of cost, and infers a cost function for each algorithm that relates cost to input size. The profiler presents the programmer with a repetition tree, in which it highlights the algorithms. It annotates each algorithm with the kind of operations it performs (e.g., construct, modify, traverse), the kind of data structures it processes, and an estimated cost function.

Developers can use algorithmic profiles to identify algorithms with high computational complexity, or to better understand program regions with high measured execution times. Given an algorithmic profile, programmers can further estimate how a program scales, that is, how a program's running time would be affected by further increases in input size.

We believe that *algorithmic profiling* relates to *algorithmic complexity analysis* like *software testing* relates to *formal verification*: it does not provide any guarantees, but can be effective in finding and fixing real-world problems.

References

[1] A. Bergel, O. Nierstrasz, L. Renggli, and J. Ressia. Domain-specific profiling. In J. Bishop and A. Vallecillo, editors, *Objects, Models, Components, Patterns*, volume 6705 of *Lecture Notes in Computer Science*, pages 68–82. Springer, 2011.

[2] B. Dufour, B. G. Ryder, and G. Sevitsky. Blended analysis for performance understanding of framework-based applications. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 118–128. ACM, 2007.

[3] S. F. Goldsmith. *Measuring Empirical Computational Complexity*. PhD thesis, University of California, Berkeley, 2009.

[4] S. F. Goldsmith, A. S. Aiken, and D. S. Wilkerson. Measuring empirical computational complexity. In *Proceedings of the the 6th joint meeting of the FSE*, ESEC-FSE '07, pages 395–404. ACM, 2007.

[5] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler. *SIGPLAN Not.*, 39(4):49–57, 2004.

[6] T. Israr, M. Woodside, and G. Franks. Interaction tree algorithms to extract effective architecture and layered performance models from traces. *J. Syst. Softw.*, 80:474–492, April 2007.

[7] M. Jump and K. S. McKinley. Dynamic shape analysis via degree metrics. In *Proceedings of the 2009 international symposium on Memory management*, ISMM '09, pages 119–128. ACM, 2009.

[8] C. McGeoch, P. Sanders, R. Fleischer, P. R. Cohen, and D. Precup. *Experimental algorithmics*, chapter Using finite experiments to study asymptotic performance. Springer, 2002.

[9] C. C. Mcgeoch. *Experimental analysis of algorithms*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1986.

[10] N. Mitchell, E. Schonberg, and G. Sevitsky. Making sense of large heaps. In *Proceedings of the 23rd European Conference on ECOOP 2009*, Genoa, pages 77–97. Springer, 2009.

[11] S. Pheng and C. Verbrugge. Dynamic data structure analysis for java programs. In *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, pages 191 –201, 2006.

[12] E. Raman and D. I. August. Recursive data structure profiling. In *Proceedings of the 2005 workshop on Memory system performance*, MSP '05, pages 5–14. ACM, 2005.

[13] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Program. Lang. Syst.*, 20:1–50, January 1998.

[14] P. Sanders and R. Fleischer. Asymptotic complexity from experiments? a case study for randomized algorithms. In *WAE '00: Proceedings of the 4th International Workshop on Algorithm Engineering*, pages 135–146. Springer, 2001.

[15] R. Wilhelm, S. Sagiv, and T. W. Reps. Shape analysis. In *Proceedings of the 9th International Conference on Compiler Construction*, CC '00, pages 1–17. Springer, 2000.

[16] X. Xiao, J. Zhou, and C. Zhang. Tracking data structures for post-mortem analysis. In *ICSE 2011 NIER Track*, 2011.

[17] G. Xu and A. Rountev. Precise memory leak detection for java software using container profiling. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 151–160. ACM, 2008.

[18] G. Xu and A. Rountev. Detecting inefficiently-used containers to avoid bloat. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 160–173, New York, NY, USA, 2010. ACM.

[19] G. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky. Go with the flow: profiling copies to find runtime bloat. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 419–430. ACM, 2009.

[20] G. Xu, N. Mitchell, M. Arnold, A. Rountev, E. Schonberg, and G. Sevitsky. Finding low-utility data structures. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 174–186. ACM, 2010.

[21] D. Zaparanuks and M. Hauswirth. The beauty and the beast: Separating design from algorithm. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'11)*, 2011.

[22] D. Zaparanuks and M. Hauswirth. Vision paper: The essence of structural models. In J. Whittle, T. Clark, and T. Kühne, editors, *Model Driven Engineering Languages and Systems*, volume 6981 of *Lecture Notes in Computer Science*, pages 470–479. Springer, 2011.