Efficient Flow Profiling for Detecting Performance Bugs

Rashmi Mudduluru
Indian Institute of Science, Bangalore, India
mudduluru.rashmi@csa.iisc.ernet.in

Murali Krishna Ramanathan
Indian Institute of Science, Bangalore, India
muralikrishna@csa.iisc.ernet.in

ABSTRACT

Performance issues in large applications arise only in particular scenarios under heavy load conditions. It is therefore difficult to catch them during testing and they easily escape into production. This necessitates the design of a common and efficient instrumentation strategy that profiles the flow of objects during an execution. Designing such a strategy which enables profile generation precisely with *low* overhead is non-trivial due to the number of objects created, accessed and paths traversed by them in an execution.

In this paper, we design and implement an efficient instrumentation technique that efficiently generates object flow profiles for `Java` programs, without requiring any modifications to the underlying virtual machine. We achieve this by applying BALL-LARUS numbering on a specialized hybrid flow graph (HFG). The HFG path profiles that are collected during runtime are post-processed offline to derive the object flow profiles.

We implemented the profiler and validated its efficacy by applying it on `Java` programs. The results demonstrate the scalability of our approach, which handles 0.2M to 0.55B object accesses with an average runtime overhead of 8x. We also demonstrate the effectiveness of the generated profiles by implementing a client analysis that consumes the profiles to detect performance bugs. The analysis detects 38 performance bugs which when refactored result in significant performance gains (up to 30%) in running times.

CCS Concepts

•**Software and its engineering → Software performance;** *Garbage collection; Object oriented languages;*

Keywords

Profiling; Dynamic Analysis; Memory Management; Performance Analysis

1. INTRODUCTION

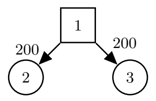
Object oriented programs often suffer from performance issues such as *memory bloat* [4, 5, 29, 39] caused due to unnecessary creation of objects. These issues arise only in particular scenarios under heavy load conditions. As a result, performance bugs caused by memory issues easily escape testing and manifest during production. It is therefore necessary to have efficient profiling tools that can help identify execution paths with performance issues.

Path profiling, introduced by Ball and Larus [2], is a powerful technique that identifies the frequency of control flow paths taken during runtime. This approach has been advanced in several ways [7, 20, 34, 31, 42]. The popularity of path profiling stems from the fact that it provides accurate information while keeping the runtime overhead minimal. Path profiling provides insights on the most frequently executed regions of code which helps identify performance bottlenecks. While this technique is useful in catching performance bugs due to repeated computations, the same is not true for performance bugs arising due to memory issues. This is because there is no direct correlation between the most frequently executed paths and the paths that exert memory pressure by creating objects inefficiently. As a result, profiling the control flow graph is not sufficient for tracking paths leading to memory issues.

To identify regions of code that create memory problems, we need to track the creation of objects and the data flow path that they take. Existing performance bug detection tools [38, 41, 24, 22] typically focus on specific bug patterns and often suffer from imprecision and high runtime overheads. We propose a generic profiling technique called *object flow profiling* that identifies those data flow paths that could lead to runtime performance degradation due to memory pressure. This information could be leveraged by various performance analysis tools that require precise information regarding the creation and use of objects.

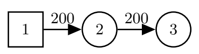
1.1 Challenges of Object Flow Profiling

```
1. x = new ...;
2. y = x;
3. x.deref();
```



(a)

```
1. x = new ...;
2. y = x;
3. y.deref();
```



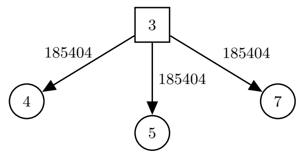
(b)

Figure 1: Object flow graphs.

In order to efficiently identify memory issues, we find that an object flow profile needs to provide the following information – (a) the *flows* (or def-use paths) taken by various object instances from each allocation site in the execution,

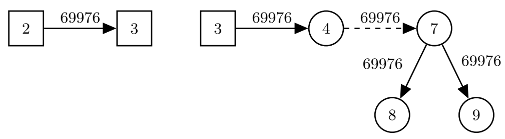
ISSTA'16, July 18–20, 2016, Saarbrücken, Germany
© 2016 ACM. 978-1-4503-4390-9/16/07...$15.00
http://dx.doi.org/10.1145/2931037.2931066

```
1 public Color getIrradiance(...){
2  for(Light vpl:vLights[set]){
3   Ray r1 = new Ray(p, vpl.p);
4   float dotNlD = -(r1.dx*...);
5   float dotND = r1.dx *...;
6   if(dotNlD > 0 && dotND > 0){
7       float r2 = r1.getMax()...;
8   }
9 }
```



(a) sunflow.

```
 1 public Integer getRotation(){
 2  Integer v = p.getObj(...);
 3  r = new Integer(v.intValue());
 4  return r;
 5 }
 6 public int findRotation(){
 7  Integer r = getRotation();
 8  if(r != null)
 9      rval = r.intValue();
10  return rval;
11 }
```



(b) pdfbox.

Figure 2: Simplified code and object graphs from benchmarks. Dashed arrows represent inter-procedural edges.

and (b) the number of object instances traversing a specific flow. The former helps understand the cause of performance issues and the latter helps expose the most likely candidate for optimizations. We propose to provide the corresponding data as a set of *object flow graphs*, where each graph corresponds to one allocation site. We define the overall process of obtaining this information as *object flow profiling*.

We explain object flow profiles using simple examples as shown in Figure 1, where the code fragments are executed 200 times in some execution. In the corresponding object flow graphs, square nodes represent allocation sites and all other nodes represent object accesses via references. In Figure 1a, node 1 corresponds to the allocation site. It has two edges ($1 \rightarrow 2$) and ($1 \rightarrow 3$). This is because the object reference x is *defined* at line 1 and *used* at lines 2 and 3. The edges represent the flow of memory references of the associated fragment. The weights on the edges indicate the number of objects that have taken the corresponding object flow. The graph in Figure 1b has two edges ($1 \rightarrow 2$) and ($2 \rightarrow 3$). The edge ($1 \rightarrow 2$) exists because the *use* of the object reference x at line 2 is based on its *definition* at line 1. The edge ($2 \rightarrow 3$) corresponds to the *definition* of reference y at line 2 and its *use* at line 3. There is a unique flow ($1 \rightarrow 2 \rightarrow 3$) unlike the two flows in Figure 1a.

One of the desired goals for obtaining object flow profiles is *less* runtime overhead. Achieving it has multiple hurdles:

- At any point in the execution, multiple objects originating from various allocation sites are usually alive and their corresponding flows are distinct.
- An object originating from a specific allocation site can also have multiple flows (e.g., Figure 1a).
- The number of objects traversing each flow needs to be maintained (the edge weights in the object flow graph).

1.2 Our Proposed Approach

Efficient control flow profiling in the literature is usually based on Ball-Larus (BL) numbering [2]. The efficiency is due to the use of integer operations to track the control flow paths resulting in reduced instrumentation (and runtime) overhead. Due to the hurdles with generating object flow profiles as described above, BL numbering cannot be directly applied to control (or data) flow graphs. Therefore, we propose the construction of a novel *hybrid flow graph* (HFG) on which we apply BL numbering [2]. The construction of HFG and the corresponding numbering is such that it helps track multiple flows for the same object instance using a *unique* integer, thereby reducing the instrumentation overhead.

We propose a novel and scalable approach to generate object flow profiles in sequential programs. The application that needs to be profiled forms the input to our approach. The output is a set of object flow graphs (and profiles) corresponding to different allocation sites. Our approach consists of three phases – (a) construction of the HFG and the static encoding of its edges, (b) instrumentation and execution of the program to collect the HFG path profiles, and (c) offline transformation of the HFG profiles to object flow profiles.

1.3 Technical Contributions

The paper makes the following technical contributions:

1. We propose a profiling technique that outputs object flow profiles which can be leveraged by client analyses to identify performance bottlenecks.
2. We design and implement an approach that constructs a novel HFG, generate the HFG path profiles with less instrumentation overhead and obtain the object flow profiles by performing offline post-processing.
3. We demonstrate the scalability of our approach by evaluating our implementation on a number of real world benchmarks and track upto 0.55 billion accesses with an overhead of 8x.
4. We verify the usefulness of our approach by implementing a client analysis that uses the profiles to identify optimization avenues. Refactoring the code appropriately shows a gain of upto 30% in running time.

2. MOTIVATION

We use examples from open source codebases and illustrate how their object flow profiles can point to performance issues that cannot be identified by path profiling.

Example 1.

Figure 2a shows a code fragment from `sunflow` and its associated object flow profile. The object flow profile indicates that node 3 is a source of creation of a large number of objects. Further, the edges in the graph show the flow taken by these objects. This object flow profile with a large number of objects is indicative of data flow paths that could potentially create memory pressure. On inspecting these *hot paths*, we understand that the objects created at node 3 are only used within the loop and their references do not escape to the heap. Therefore, we can optimize this code by reusing objects across iterations.

In contrast, a path profile for this example would contain a single acyclic path with its corresponding frequency count. This is not sufficient as it does not give information regarding the data flow path taken by objects created at a particular object creation site.

Example 2.

Figure 2b shows code fragment from `pdfbox` and its corresponding object flow profile. In this code, the method *findRotation* invokes *getRotation* at line `7` which returns a reference to a newly allocated object. There is a dereference at line `3` which is eventually used for an allocation at the same line. This dereference and allocation is unnecessary because *findRotation* can directly invoke *p.getObj*(...) eliminating the need for the allocation at line `3`. The object flow profile for this example shows that the objects created at node 2 are used at node 3 which is an allocation site itself. This is the only path taken by all objects originating at node 2. The second part of the graph shows how the objects created at node 3 are subsequently used. This profile clearly shows the flow of objects and the number of objects taking each flow. A client analysis looking for memory issues can identify that all objects created at node 2 are only used for creating other objects at node 3 and therefore could indicate a performance bug. Observe that a path profile cannot help in this example. The control flow path is $7 \to 2 \to 3 \to 4 \to 7 \to 8 \to 9 \to 10$ which is quite different from the object flow profile.

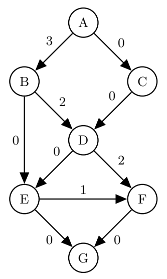
These examples set the basis that profiling object flows as described above can help identify performance bugs due to memory issues precisely and efficiently.

3. DESIGN

3.1 Background

We now present the BL numbering [2] technique[1] to encode CFGs using an illustrative example. Consider the CFG shown in Figure 3a with eight distinct paths. In the BL technique, for a graph with n paths, each path is given a unique integer ID ranging from 0 to $n-1$. The edges of the graph are assigned integers such that adding the edge IDs on any path from the start to the end node produces the path ID. Figure 3b shows the path IDs for the CFG in Figure 3a.

To profile the paths traversed during a program execution, instrumentation is placed on edges such that on reaching the end of a path, the count of its corresponding ID is incremented. Since the instrumentation mainly involves integer increments, the runtime overhead of the instrumented



(a) Control flow graph.

Path	ID
ACDEG	0
ACDEFG	1
ACDFG	2
ABEG	3
ABEFG	4
ABDEG	5
ABDEFG	6
ABDFG	7

(b) Path identifiers.

Figure 3: Illustrative example for BL numbering.

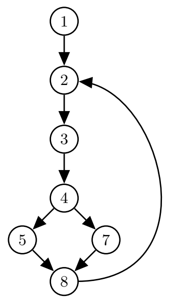
program is low. Moreover, the path taken at runtime can be obtained by decoding the path identifiers.

We elide the details pertaining to the encoding and decoding algorithms of BL numbering [2]. For the purposes of this paper, it is sufficient to have an encoder and a decoder that perform BL numbering. The encoder takes as input a directed acyclic graph (G) and outputs a graph (G') annotated with edge weights. The decoder takes the integer and G' as inputs and outputs the corresponding directed path.

3.2 Design Challenges

```
1 B b = null;
2 for i = 1 to n {
3   X a = new X(i);
4   if(i <= 10)
5     b = new X();
6   else
7     b = a;
8   a.foo(b);
9 }
```

(a) Code snippet.



(b) CFG.

Figure 4: Example illustrating design challenges.

Our approach for object flow profiling is inspired by the design of BL numbering. A similar encoding to track object flows can provide data for several analyses to obtain precise data with less overhead. However, handling object flows involves several challenges as described below:

Firstly, unlike control flow, objects do not necessarily flow between the statements that are executed consecutively. For example, consider the example code in Figure 4a and its corresponding CFG in Figure 4b. For simplicity, we omit the exit node and therefore the edge from node 2 to exit in the CFG. Suppose the *else* branch of the condition is taken. Even though control is transferred from line `4` to line `7` as seen in Figure 4b, the *flow* of the object referenced by a is from line `3` to line `7`. Because the control flow profiles are not equivalent to the object flow profiles, the former cannot be a substitute for the latter. Furthermore, naively tracking the objects on each statement to identify their source can significantly slow the execution down.

Secondly, a major challenge is the need to track parallel object flows. For example, the object created at line `3` in Figure 4a is assigned to the reference variable a which is

[1]The technique described here is preliminary without all the optimizations. We refer the reader to [2] for more details.

used at lines 7 and 8. For both the lines, the object flows from line 3. Hence, both the flows need to be recorded for the execution. In order to track such parallel flows, a single identifier per object allocation site can be insufficient. This necessitates tracking the entire object access history.

Finally, the flow needs to be tracked for each object created dynamically. The memory requirements to maintain the data on all objects can be significant even for moderately large programs. For example, `mst` with 317 LoC creates `67.1 million` objects after executing for about 6 seconds.

3.3 Addressing the Design Challenges

Based on the above challenges, it becomes clear that in order to efficiently track object flow while following a control flow path at runtime, we need to construct a specialized graph. We observe that an *intelligent* overlay of the object flow on the CFG can suffice. The CFG already has the data flow overlayed (by definition) but there are a number of irrelevant nodes corresponding to the data flow (e.g., primitives). Even if we identify the relevant nodes in the graph, as specified by the second challenge in Section 3.2, obtaining object flow profiles requires tracking the entire history of object reference definitions and the accesses for all objects.

A history of object accesses $a_1 \ldots a_n$ is a control flow path consisting of nodes along which an object instance has been accessed. If we can track this history of object accesses online, we can analyze it to obtain the object flow profiles as an offline process. However, in order to be able to precisely encode the history of object accesses (*a.k.a* object access paths) as an integer using BL numbering, the graph structure should be such that there is an edge between any two consecutive accesses of the object instance. More specifically, in order to precisely encode the object access paths, we need to construct a graph H that satisfies Property 1.

PROPERTY 1. *For any given object instance o, if there exists an execution where two consecutive accesses (def-use or use-use) of o are at statements given by vertices u and v in the corresponding* CFG*, then the vertices u and v along with the edge $u \rightarrow v$ must exist in H.*

In Property 1, consecutive access refers to both def-use and use-use pairs of an object reference. We can trivially use the CFG as H to satisfy the above property by adding edges between *any* two accesses of o. However, this can unnecessarily increase the potential number of object access paths by introducing infeasible paths. The increased number of paths impacts memory overhead because we need to assign memory to store information pertaining to all the paths efficiently. We overcome this challenge by constructing a novel *hybrid flow graph* which satisfies Property 1 while ensuring the number of infeasible access paths is minimum.

3.4 Design Overview

The overall design of our approach is shown in Figure 5. From the input program binary, a PDG is constructed and slices are obtained based on the allocation site targets. These slices are used to construct the HFG to which additional bypass edges are added. The HFG is given as to the instrumentation phase. The instrumented bytecode (IB) upon execution outputs a set of HFG IDs corresponding to its paths. These paths are then decoded to obtain the *object flow profiles.*

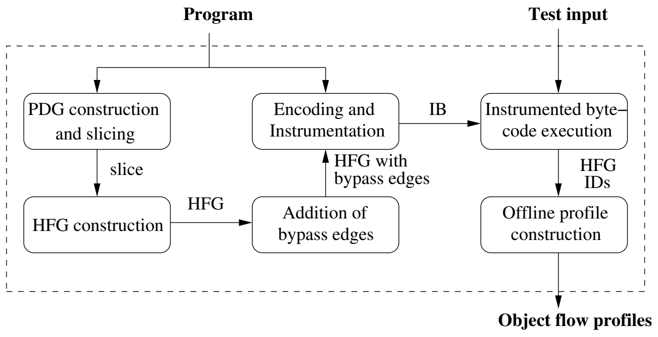

Program
Test input
PDG construction and slicing
Encoding and Instrumentation
IB
Instrumented byte-code execution
slice
HFG with bypass edges
HFG IDs
HFG construction
HFG
Addition of bypass edges
Offline profile construction
Object flow profiles

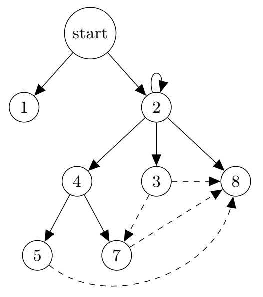
Figure 5: Overall Design. IB - instrumented bytecode.

3.5 Hybrid Flow Graph

In the first phase, we statically analyze the program to build a custom HFG that encodes the object access paths.

3.5.1 Identifying Relevant Nodes

As part of the graph construction, we need to identify the nodes of the control flow graph (CFG) that require instrumentation. This is because of the absence of flow on many nodes of the CFG (e.g., nodes that represent operations on primitive types). We identify all the *relevant* nodes by taking a forward slice [35] from the nodes representing allocation sites. We achieve slicing with the help of program dependence graphs (PDG) [9] and therefore initially construct a PDG as part of our approach.



(a) PDG.

```
3  X a = new X(i);
7  b = a;
8  a.foo(b);
```

(b) Node 3 slice.

```
5  b = new X();
8  a.foo(b);
```

(c) Node 5 slice.

Figure 6: PDG (solid arrows: control flow, dashed arrows: data flow) and slices for the allocation sites from the example code in Figure 4a.

Figure 6a shows the PDG for the example code given in Figure 6afig:code. The solid arrows represent the control dependence edges and the dotted arrows correspond to the data dependence edges. *start* represents a pseudo start node. There are control dependence edges from *start* to the nodes labelled 1 and 2 as they are executed unconditionally. The statements at 3, 4 and 8 are executed based on the condition specified in 2, resulting in the outgoing control dependence edges from 2. There are data dependence edges from node 3 to nodes 7 and 8 as *a* is *defined* in 3 and *used* at 7 and 8.

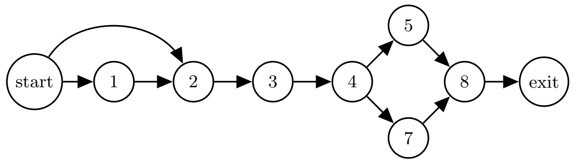
The problem of finding a slice for a given target is reduced to that of forward reachability from the corresponding node [9] in the PDG. We mark three types of object references as slicing targets: (a) references to newly created objects, (b) references that are incoming parameters to the method under consideration, and (c) references to objects that are returned from method calls. Figures 6b and 6c show the respective slices for targets a (at line `3`) and b (at line `5`). Aliasing in these slices is handled by the construction of the PDG. The *def - use* edges in the PDG ensure that there exists an edge between any two nodes that might alias to the same memory location. Even though this is conservative, it does not affect the precision of our approach as we profile the actual path taken in our object flow profiles.

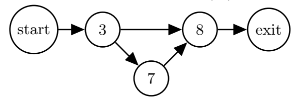
3.5.2 Overlaying Object Flow on the Control Flow Graph

We construct the HFG for each slice thus obtained. Algorithm 1 describes the construction of the *hybrid flow graph* (H) that preserves the control dependences among the statements in the slice. The CFG of the method under consideration and the slices derived for the allocation sites in the method form the inputs to the algorithm.

Algorithm 1 Construct_HFG

```
Input: slice $S$, Control flow graph ($C$)
Output: Hybrid flow graph ($H$) for $S$
 1: $H \leftarrow$ remove_cycles($C$)
 2: for each node $u$ in $H$ do
 3:     if $u \in S$ then goto 2
 4:     for each predecessor $p$ of $u$ in $H$ do
 5:         for each successor $s$ of $u$ in $H$ do
 6:             add edge from $p$ to $s$
 7:             remove $u$ from $H$
```


(a) Acyclic CFG.


(b) HFG for allocation site 3.

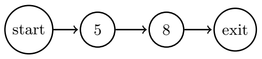

(c) HFG for allocation site 5.

Figure 7: Various graphs for the example code in Figure 4a.

The algorithm converts the CFG into an acyclic CFG by invoking the auxiliary function *remove_cycles*. We handle loops in the CFG, while preserving the ability to maintain profiles as in [2]. The auxiliary function deletes every back edge ($b_s \rightarrow b_d$) in the CFG and inserts two additional edges in its place. The first edge is from the source node of the CFG to b_d and the second edge is from b_s to the exit node of the CFG. Figure 7a shows the graph obtained after invoking *remove_cycles* on the CFG considered in Figure 4b. The back edge $8 \rightarrow 2$ is replaced with the edge $start \rightarrow 2$.

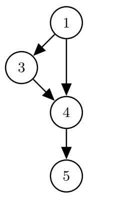
We process the acyclic CFG (labelled H) with the associated slice S as follows. If a node u in H is not present in S, we delete u from H and add a directed edge from every predecessor of u to all its successors in H (lines `4 - 7`). This ensures that H does not contain any node that is absent in S and also encodes the control flow among the statements in S. We apply Algorithm 1 for all the slices derived in the previous phase. Figures 7b and 7c show the HFGs for the slices presented in Figures 6b and 6c respectively. For example, in Figure 7b, edges $3 \rightarrow 7$ and $3 \rightarrow 8$ are present. This is because the statements represented by nodes `4` and `5` are not part of the slice and when all the absent nodes are removed, the edges under consideration are added.

3.5.3 Adding Bypass Edges

The HFG captures all the control flow paths possible during execution with respect to the nodes in the slice. It also has an edge for *some* of the consecutive accesses of an object (e.g., `3` $\rightarrow$ `8` in Figure 7b) that is not present in the original CFG. However, we need to satisfy Property 1 which requires an edge between consecutive accesses of the object on *any* execution. Applying Algorithm 1 alone is insufficient to generate graphs that satisfy this property.

```
1  A a = new A();
2  if(cond)
3    c = a;
4  c.foo();
5  a.bar();
```

(a) Source.


(b) HFG.

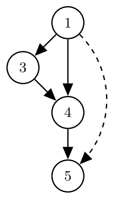

(c) HFG + bypass edge.

Figure 8: Motivating example for bypass edges (`c` is global).

Algorithm 2 Add_bypass_edges

```
Input: Set of def-use sets ($D$), HFG ($H$)
Output: HFG with bypass edges ($H'$)
 1: $H' \leftarrow H$
 2: for each $D_i$ in $D$ do
 3:     $d \leftarrow def(D_i)$; $U_i \leftarrow use(D_i)$
 4:     for each $u$ in ($U_i$) do
 5:         find $v \in U_i$ with following properties
 6:             $dominator(v,u,H')$ is true
 7:             For any other node $w$,
 8:                 if $dominator(w,u,H')$ is true,
 9:                 then $path(v,w,H')$ is false
10:         if ($v$ is found) then $H \leftarrow H' \cup (v \rightarrow u)$
11:         else $H' \leftarrow H' \cup (d \rightarrow u)$
```

While the graphs shown in Figure 7b and 7c satisfy Property 1, it may not hold always. Consider the code shown in Figure 8a and its HFG in Figure 8b for the slice corresponding to line `1`. Suppose the execution takes the control flow path $1 \rightarrow 4 \rightarrow 5$. Although the most recent reference to object accessed at node `5` is at node `1`, there is no edge from `1` to `5` in the graph. This is because even though control flows from `4` to `5` in the HFG, it does not necessarily mean that the object instances accessed at the nodes are *always* the same since the HFG is an overlay of object flow on control flow.

To address this such that Property 1 is satisfied, we add *bypass* edges to the HFG. Intuitively, for any two nodes u and v in a HFG, a bypass edge $u \rightarrow v$ is added, if u and v access the same object instance o, u is the most recent access of o across *any* control flow path, and the edge does not already exist in the graph.

Algorithm 2 describes our approach to add the bypass edges. It takes as input a set of def-use sets (D), which is computed statically for the object references and the hybrid flow graph (H). We define auxiliary functions - (a) *dominator*(x,y,H) which is true *iff* x is a dominator of y in H (i.e every path from entry to y in H contains x), and (b) *path*(x,y,H) which is true if there is a path from x to y in H. For every def-use set (D_i), we find the definition d and the set of uses of the definition U_i. For each use u in U_i, we attempt to find another use v in the same set such that v is a dominator of u and for all other dominators w of u, there is no path from v to w. In other words, v is the last node that is *always* accessed before the access at u and hence the associated bypass edge is added (lines `3 - 8`). If we are unable to find such a node, we add an edge from the definition to the use (line `9`). For the HFG of Figure 8b and its corresponding def-use sets, Algorithm 2 adds the edge from `1` to `5` as shown in Figure 8c.

The HFG of Figure 8b, has the following set of def-use sets – D = `{{1,{3,5}},{3,{4}}}`. When we consider D_0 = `{1,{3,5}}`, we have d is `1` and U_i = `{3,5}`. Considering use `5`, it does not have any dominator from `1` that belongs to U_i. Therefore an edge is added from `1` to `5` as shown in Figure 8c. The presence of this bypass edge enables the encoding of all the object access paths.

In summary, we observe that given an acyclic CFG C, a slice target t_i, and the HFG H' (output of Algorithm 2), for any object o allocated at t_i and for *any* two consecutive accesses of o having a feasible control flow path between the accesses in C, there exists an edge between the nodes corresponding to those accesses in H', thereby satisfying Property 1

3.5.4 Merged HFG for a Method

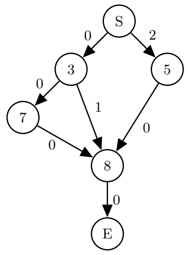
In the previous section, we build an HFG per slice target. If the paths are encoded in separate graphs, for every object access path, the identity of the graph needs to be known apart from maintaining the path in it. To reduce the lookup associated with it, we merge all the HFGs for all the slice targets in a method. Merging of the graphs is a simple union of the nodes and edges in the graphs. Figure 9a without the edge labels shows the merged HFG for the example in Figure 4a. Because there is no deletion of nodes or edges, if two consecutive accesses find an edge in one of the HFG for a slice, there will also be an edge in the merged HFG.

3.6 Encoding Object Access Paths

We encode the HFG paths using the technique described in [2] for which a brief background was provided in Section 3.1. Figure 9a shows the labels obtained for the edges of the merged HFG by applying this technique. Figure 9b shows the path IDs for all the three possible paths in the HFG.

3.7 Instrumentation

Algorithm 3 describes the procedure for instrumenting the bytecode and accepts the labelled HFG (H_L) and the program bytecode (B) as inputs. It maintains two maps per method, `P` and `M`, where `P` persists across multiple invocations of the method and `M` is valid only during the lifetime of a single invocation. `M` maintains a mapping from the ob-



$$\begin{tikzcd}
 & S \arrow[dl, "0"'] \arrow[dr, "2"] & \\
3 \arrow[d, "0"'] \arrow[dr, "1"] & & 5 \arrow[dl, "0"] \\
7 \arrow[r, "0"'] & 8 \arrow[d, "0"] & \\
 & E &
\end{tikzcd}$$

(a) encoded HFG.

Path	ID
S,3,7,8,E	0
S,3,8,E	1
S,5,8,E	2

(b) Path identifiers.

Figure 9: Illustration of encoding (S: start node, E: exit node).

Algorithm 3 Instrument

```
Input: Labelled HFG ($H_L$), Program bytecode ($B$)
Output: Instrumented bytecode
 1: // P: map from path identifier to number of objects
 2: // M: map from object id to (path id, node id of last access)
 3:
 4: for each instruction $i$ in $B$ do
 5:     if $i$ defines a new object then
 6:         $oid \leftarrow getRuntimeID(def(i))$
 7:         $p \leftarrow getLabel(H_L,start,i)$; M[$oid$] $\leftarrow (p, i)$
 8:     if $i$ has an object access then
 9:         $oid \leftarrow getRuntimeID(use(i))$
10:         $(p, l) \leftarrow$ M[$oid$]; $p \leftarrow p + getLabel(H_L,l,i)$
11:         M[$oid$] $\leftarrow (p, i)$
12:     if $i$ is a method exit instruction then
13:         for every $key$ in M do
14:             $(p, l) \leftarrow$ M[$key$]
15:             $p \leftarrow p + getLabel(H_L,l,end)$
16:             if P[$p$] is not found then P[$p$] $\leftarrow 1$
17:             else Increment P[$p$] by 1
```

ject identifier to a tuple, (p, l), where p is the path traversed by the object in the HFG, and l is the last node that was accessed in the HFG. At method exit, this map is discarded after updating the count of number of objects per path in `P`. The algorithm places instrumentation for object allocations, accesses and exit of a method. When a new object is created, its current path ID is the label from the *start* to the current node in the HFG and `M` is updated accordingly (lines `5-7`). When an object is accessed, its path ID is incremented by the label on the edge from its last access to the current access (lines `8-11`). On method exit, the path identifier is updated as before and subsequently, `P` is either initialized to one or incremented by one to show that an object has traversed the corresponding path in the method (lines `12-17`). Finally, the data in `M` is discarded at the end of the method body. This ensures that the memory requirements are just proportional to the various object access paths and not to the total number of objects observed during an execution.

3.8 Offline Profile Construction

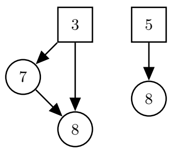
When the instrumented bytecode is executed, the map `P` gives the number of times each object access path is executed in a method. For example, executing the instrumented code for the example given in Figure 4a for $n = 100$ will generate the HFG profile as given in Figure 10a.

Algorithm 4 Construct_profiles

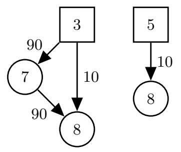
```
Input: Def-use graph ($OG$), Map P
Output: Object flow profiles ($O$)
1: for each path identifier $p$ in P do
2:     $d \leftarrow decode(p)$
3:     for each edge $(i \rightarrow j) \in d$ do
4:         while $i \rightarrow j \notin OG$ do
5:             $i \leftarrow predecessor(i)$ in $d$
6:         $e \leftarrow (i \rightarrow j)$
7:         In $OG$, $count(e) \leftarrow count(e)$ + P[$p$]
```

ID	count
0	90
1	10
2	10

(a) HFG profile.



(b) OFG.



(c) OFP.

Figure 10: Offline profile construction (OFG: Object Flow Graph; OFP: Object Flow Profile).

Algorithm 4 processes this data (offline) to generate the object flow profiles. It takes as input, the object flow graph (OG) and the map P per method. An object flow graph can be constructed by statically analyzing the program code. An edge $a \rightarrow b$ is present in the object flow graph whenever node a defines an object reference that is used at node b. Figure 10b is the object flow graph for the code given in Figure 4a. Algorithm 4 annotates the edge $i \rightarrow j$ of the graph with the number of objects that are defined at node i and are used at node j.

Initially the path identifier is decoded using the BL technique [2] (line 2) to obtain the actual path d, which represents the set of nodes in the merged HFG. The edges in OG are def-use pairs whereas the edges in the HFG path (d) can also be use-use pairs. Therefore, to identify the definition for every use in d, we perform a backward traversal of d (`line 3 - 5`) to detect the definition for the use under consideration. We update the edge label corresponding to that in OG with the number of objects that correspond to the path p (line 7). Figure 10c shows the object flow profiles obtained in this manner for execution of the code in Figure 4a.

To illustrate how these object flow profiles are generated offline, consider the same execution as discussed in Section 2 for the code in Figure 4a. Consider the path ID 0 and its corresponding count of 90 as shown in Figure 10a. From Figure 9b, we can see that this path will be decoded as $S \rightarrow 3 \rightarrow 7 \rightarrow 8 \rightarrow E$ where S and E represent the *start* and the *end* nodes respectively. Consider this decoded path d and the object flow graph OG as shown in Figure 10b. The edges $3 \rightarrow 7$ and $7 \rightarrow 8$ that are present in d are also present in OG. Therefore, these edges in OG are updated with the count of this path which is 90. Similarly, path IDs 1 and 2 are decoded as $S \rightarrow 3 \rightarrow 8 \rightarrow E$ and $S \rightarrow 5 \rightarrow 8 \rightarrow E$ respectively and the corresponding edges annotated as shown in Figure 10c.

3.9 Object Flow Summaries

We now describe our approach to generate object flow profiles interprocedurally. Because tracking the flow of every object precisely in a context sensitive manner can be expensive, we propose a novel mechanism to derive object flow summaries per method. Subsequently, these summaries can be leveraged by client analyses to track the flow of objects interprocedurally. While these summaries provide an approximation of the number of objects flowing across methods, this suffices for most practical purposes. This is because client analyses searching for optimization avenues can use the approximate interprocedural numbers.

Two types of references can act as conduits for the objects to flow across methods – (a) parameters to the method (including the receiver) and (b) the reference to the returned objects. The flows due to returns do not require any special handling as these references are identified as slicing target (see Section 3.5.1). However, parameters to the method require special handling as described below.

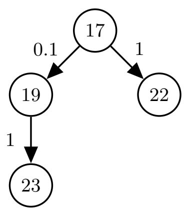
Algorithm 5 Handle_parameters

```
Input: HFG $H$, HFG profiles $H_P$, object flow graphs rooted
    at incoming parameters $O$
Output: Object flow profiles for incoming parameters
1: for each edge $u \rightarrow v$ in $O$ do
2:     $edge\_flow = \Sigma H_P[p]$ s.t path $u \xrightarrow{*} v \in$ path $p$
3:     $node\_flow = \Sigma H_P[p]$ s.t node $u \in$ path $p$
4:     $P[u \rightarrow v] = edge\_flow \ / \ node\_flow$
```

We derive object flow summaries by analyzing the execution across various contexts and assigning probabilities of object flows within a method which will act as a summary. These summaries are then inlined context sensitively by the client analyses to obtain an approximation on the flow of objects across methods. Algorithm 5 describes the procedure to derive these summaries. In Section 3.5.1, we marked each incoming reference parameter as a slicing target. This ensures that each incoming reference is treated as a producer and its flow in HFG is tracked at runtime. The associated HFG profiles (H_P), the HFG (H) and the object flow profiles (O) for the method rooted at incoming parameters form the inputs to the algorithm. Recall that an HFG profile is a mapping from HFG paths to the number of objects taking that HFG path. The algorithm calculates probability for every edge $u \rightarrow v$ in O as the ratio of sum of number of objects on the paths containing $u \xrightarrow{*} v$ ($edge\_flow$) to the sum of number of objects on the paths containing u ($node\_flow$). A probability of 1 on any edge indicates that every object arriving at u flows to v. Any other probability gives an approximation of the number of objects flowing along the edge $u \rightarrow v$. We thus obtain an approximation of the object flows which suffices in practice as shown in our experimental results.

```
17 void func(X x,int c){
18  if(c == 0)
19    y = x;
20  else
21    y = new X();
22  print x;
23  arr[k] = y;
24 }
```

(a) Example code.



(b) Object Flow Profile.

Figure 11: Inter procedural example.

We illustrate our approach using an example as shown in Figure 11a. In the implementation of $func$, the incoming reference is assigned to y based on the value of c (line 18). So, the calling context of $func$ decides which branch is taken. Assume that the method $func$ is called from different contexts for a total of 10 times. Also assume that one of these calls follows the if branch and the remaining nine calls take the $else$ branch. Figure 11b shows the object flow profile summary for the incoming reference x as computed by Algorithm 5. The edge value of 0.1 on the edge $17 \rightarrow 19$ indicates that the probability of an object flowing from node 17 to node 19 is 0.1 as only 1 out of the 10 objects reaching node 17 flows into node 19. For all the other edges, the objects flow with a probability of 1. Suppose a client analysis analyzes an allocation site in the calling context of $func$. Since it knows the number of objects that flow into $func$, it can obtain an approximation of object flows by multiplying the number of objects with the probabilities appropriately.

4. EXPERIMENTAL VALIDATION

Table 1: Benchmark characteristics.

Benchmark	LoC	classes	methods
euler	1101	8	55
moldyn	621	8	46
montecarlo	1378	18	197
pdfbox	38032	381	3561
mst	317	10	93
tsp	790	13	45
health	382	8	28
pmd	48930	742	5338
avrora	51396	551	2957
luindex	36075	246	2081
lusearch	36248	213	1795
sunflow	21985	204	4014

We perform elaborate experimentation on large JAVA programs to validate the efficacy of our approach. We run all the benchmarks in single-threaded mode. All our experiments are performed on a 64 bit, 8 core Ubuntu-13.11 desktop, equipped with an Intel core i7 processor and 16GB RAM. We use OpenJDK 64-Bit Server VM version 1.7. We have used `soot` [33] for performing the static analysis and bytecode instrumentation. Table 1 shows the characteristics of benchmarks used.

We evaluate our approach broadly on two dimensions:

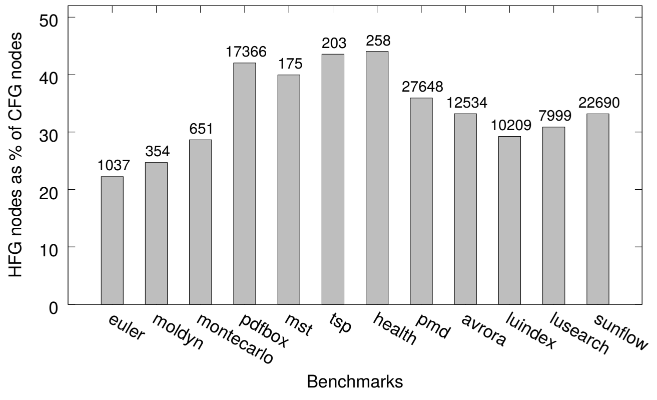
1. The efficiency of generating the object flow profiles.
2. The usefulness of the generated profiles in identifying performance issues.

4.1 Profiling

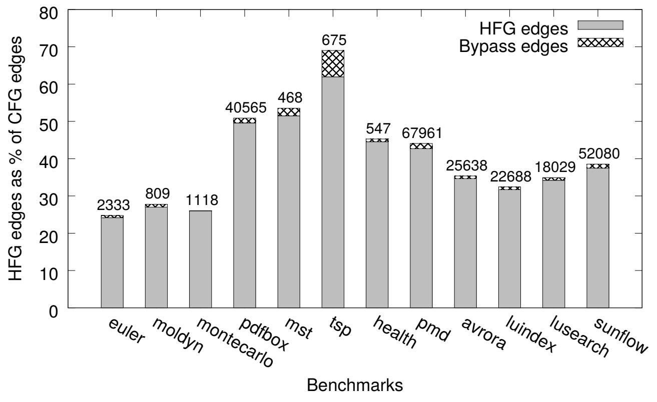
We now present the performance statistics of our approach across these three phases *viz.,* the static analysis phase, encoding and instrumentation phase and the offline analysis phase.

4.1.1 Static Analysis Phase

This phase involves constructing the HFGs, numbering them and instrumenting the bytecode accordingly. Figure 12 shows the characteristics of HFGs in comparison with their corresponding CFGs. Figure 12a shows the percentage of CFG nodes used for constructing the corresponding HFGs. The



(a) % of HFG nodes.



(b) % of HFG edges.

Figure 12: HFG characteristics

labels atop the bars show the actual number of HFG nodes. The number of HFG nodes is not more than 50% of the CFG nodes since we discard all the irrelevant nodes with the help of slicing. This ensures that we instrument fewer nodes thereby reducing the runtime overhead later. Figure 12b shows the number of HFG edges as a percentage of the CFG edges. The crossed parts of the bars represent the additional bypass edges as a percentage of CFG edges. The bar labels show the total number of edges present in the HFGs. Although the bypass edges account only for a small percent (0.6 - 7%) of the number of HFG edges, they add significant value by providing precision and reducing overhead. This is because Algorithm 2 adds only the most relevant edges which also helps in less runtime overhead. The reduced number of HFG nodes and edges ensures that the BL numbering is efficient. In our experiments, all the static path numbers could be represented by a 64 bit integer.

Table 2 shows the running times of the various stages in our static analysis phase. The overall analysis time is proportional to the number of nodes and edges in the HFG. A significant percentage of the time is spent in bytecode instrumentation as it has to go over every instruction that is part of the HFG and insert appropriate hooks. The HFG construction also consumes a reasonable proportion of the overall analysis time (e.g., 60% for `pmd`) since it involves graph operations such as construction and traversal. The time taken for encoding the path numbers is negligible.

Table 2: Static analysis characteristics. T_{HFG}: HFG construction time, T_{BL}: encoding time, T_I: instrumentation time, T: overall time. All times are given in milliseconds.

Benchmarks	T_{hfg}	T_{BL}	T_I	T
euler	23789	31	4180	28000
moldyn	2450	18	5278	7746
montecarlo	563	25	8896	9484
pdfbox	8269	167	27290	35726
mst	465	13	5852	6330
tsp	1214	9	5860	7083
health	595	13	6027	6635
pmd	73272	204	47272	120748
avrora	3640	106	24217	27963
luindex	5555	88	21297	26940
lusearch	4273	108	18487	22868
sunflow	25110	201	45010	70321

4.1.2 *Runtime Results*

Table 3: Runtime characteristics.

Benchmarks	#objects (in M)	#accesses (in M)	#paths
euler	14.6	533.3	218
moldyn	0.002	0.2	82
montecarlo	0.2	101	75
pdfbox	7.5	51.5	653
mst	67.1	553.7	50
tsp	6.8	538.9	75
health	15.7	94.17	60
pmd	0.2	1.3	2658
avrora	1.9	56.8	7103
luindex	0.5	2.4	1643
lusearch	8.2	98.4	4352
sunflow	29.4	67.3	1334

Table 3 shows the runtime characteristics obtained by executing the benchmarks. The significantly large number of objects (ranging from 2K to 67.1M) and the total number of object accesses (e.g., 553.7M for `mst`) during an execution indicate that less processing be done per access for scalability. The table also presents the number of distinct dynamic paths in the HFGs for different benchmarks. The relatively fewer number of paths is because several object flows are encoded in the same path. It is this property of HFGs that allows us to encode several object flows using BL numbering.

Figure 13 presents the execution times for the different benchmarks normalized with respect to the execution times without instrumentation and ranges from 1.2x to 15.2x. The labels on the bars show the execution times of the instrumented programs in seconds. The execution times are obtained by taking an average over five runs. This overhead is a function of the structure of the code in these benchmarks. While a few benchmarks with lot of accesses (`euler`, `mst` and `tsp`) have relatively higher overhead, even `sunflow` with fewer object accesses has higher overhead. This is because the runtime call stack is long causing more objects to be maintained in the map (as shown in Algorithm 3) and consequently increases the lookup time for each access. However, in the case of `health`, even with the high number of objects and accesses, the runtime overhead is 3x since the call stack is short. While the traditional BL numbering profiles control flow, we track every object in the execution resulting

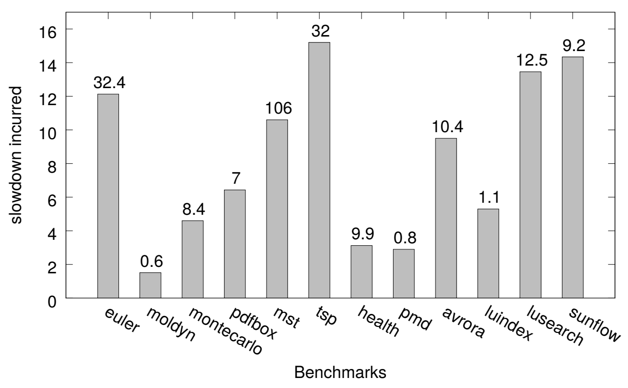


Figure 13: Execution time normalized w.r.t execution time without instrumentation.

in more overhead. For example, for one control path, there could be multiple objects that need to be profiled. However, this is in general better than naive approaches like [41] which reports overhead ranging from 30-50x. The overhead can be further reduced by using probabilistic sampling (e.g., [23]) while maintaining precision.

4.1.3 *Offline Post-Processing Phase*

Table 4: Offline analysis characteristics.

Benchmarks	#nodes	#edges	Outdegree (max)	Analysis time (ms)
euler	732	1159	120	339
moldyn	260	348	83	137
montecarlo	677	817	18	217
pdfbox	11899	14972	230	567
mst	117	154	19	96
tsp	265	320	37	97
health	158	223	9	95
pmd	10967	14637	63	1341
avrora	9202	12500	58	1298
luindex	9349	13210	47	970
lusearch	7175	9797	82	1533
sunflow	9545	13210	191	880

Table 4 presents the results of the offline analysis phase. The number of nodes and the edges in the object flow graph is given. The maximum out degree for various benchmarks shows the presence of parallel flows in real programs and the need for efficient encoding techniques. The overall post processing time is insignificant.

4.2 Effectiveness of Object Flow Profiles

Our profiling technique generates object flow profiles in the form of graphs. The analysis tools built on top of this consist of traversing these graphs and identifying performance issues. To validate the usefulness of the object flow profiles, we implement a client analysis that identifies common object reuse opportunities. This analysis traverses the object flow graphs and identifies the following bug patterns: (a) object flow paths on which the references do not escape to the heap, (b) paths on which every node is a producer indicating creation of unnecessary objects.

Table 5 shows the results of applying this client analysis on the benchmarks. We analyze the object flow graphs and

Table 5: Issues detected by the client analysis and associated performance gains. PC: Potential candidates, TP: True positives, PI: Performance improvement, * denotes benchmarks which generate less than 10K objects.

Benchmark	PC	TP	PI
euler	3	2	18.3
moldyn *	1	1	-
montecarlo	7	6	0.8
pdfbox	15	11	10.9
mst	-	-	-
tsp	2	2	7.5
health	4	2	30
pmd *	2	1	-
avrora *	1	1	-
luindex *	8	4	-
lusearch	4	4	-
sunflow	4	4	4.5

consider object flow paths as potential candidates for optimization only when the number of objects in these paths is greater than 10k. Reducing the threshold may show more candidates but need not translate into performance gains. Column 2 in the table gives the number of candidates for the analysis. For the benchmarks `moldyn`, `pmd`, `avrora` and `luindex`, none of the candidates are greater than the 10k threshold and we consider 1k as a threshold. We manually analyze the reported candidates and verify their validity. Column 3 presents the number of true-positives. Overall, out of 51 potential candidates reported, 38 are true candidates for optimization. The high true positive percentage is due to the precise profiles generated by our approach. In the absence of bypass edges which contribute to generating precise profiles, the client analysis misses 12 of the potential candidates of which four happen to be true positives.

The last column shows the percentage of runtime improvement obtained by refactoring the true-positives The performance gains are negligible for the benchmarks labelled * due to fewer objects that flow along the problematic regions because garbage collection is not affected. Moreover, for `montecarlo` and `lusearch`, we are unable to refactor for all candidates, though they belong to sub-optimal code, because many of them correspond to immutable (String, Integer) objects and hence see negligible improvement.

5. RELATED WORK

Profiling of control flow paths with minimum overhead has been studied extensively. The seminal work by Ball and Larus [2] proposes a numbering scheme to identify control flow paths with minimum overhead. A number of other approaches improve upon this technique [34, 7, 20, 31, 42]. To understand the dynamic behaviour of programs, approaches have been proposed to efficiently detect control dependence dynamically [37, 3]. In [32], the authors extend the path profiling algorithm to account for loops and paths that span procedure boundaries. While these approaches provide efficient mechanisms for tracking control flow, we leverage the original numbering scheme to build a scalable and precise approach for generating object flow profiles.

Control flow profiling has also been applied to understand the runtime behaviour of programs in distributed and concurrent settings. In [31], the authors propose algorithms to precisely encode calling contexts at runtime by giving a unique integer to each context optimally. Other profiling techniques [43, 30, 36] help a wide range of applications that are looking for parallelization opportunities, regression testing, bug reproduction, etc. We target applications that require monitoring objects flows to identify memory issues.

Recently, several approaches have been proposed for debugging performance issues in object oriented languages [27, 16, 10, 25, 26]. [16] describes techniques for identifying memory issues in JavaScript programs. JITProf [10] provides a profiling framework for JavaScript applications that finds code regions which prevent JIT optimizations. This shows that the problem we solve is non-trivial and that there is need for efficient tools for analyzing large programs and identifying memory issues in them. Profiling data related to dynamically created objects has been studied in the context of improving garbage collection [14, 6, 18, 21, 1, 15]. Our work is orthogonal to these approaches where we identify ways to ensure less garbage creation.

Several dynamic analysis techniques have been proposed to identify performance issues caused due to the inefficient creation and usage of objects. This encompasses techniques to detect object bloat [29, 39, 5, 4], leak detectors [19, 40, 28], etc. These analyses require the information pertaining to object flow profiles and can potentially be transformed to use the profiling technique designed in this paper.

Specific patterns of computations can cause performance degradation and many dynamic analyses exist to detect such patterns [22, 13, 17, 8, 11]. These approaches rely on control flow data to identify repetitions. Currently, we handle the flow of objects and do not track the corresponding values and other state associated with it.

Static analysis approaches to detect performance issues are also quite popular [5, 24]. Bhattacharya *et al* [5] use a combination of static and dynamic analysis to detect object bloat. The work in [12] statically identifies locations where dead objects can be reclaimed. One of the drawbacks of many of the static analyses is the underlying imprecision which can result in increased number of false positives. In contrast, the output of our approach provides runtime information thereby reducing the possibility of false positives.

6. CONCLUSIONS

In this paper, we designed and implemented a profiling technique for efficient generation of object flow profiles, without any modifications to the underlying virtual machine. We efficiently collected runtime path profiles on a novel *hybrid flow graph* by applying BL numbering and derived the object flow profiles offline. Experiments showed that our approach could handle upto 0.55B object accesses with an average overhead of 8x. We demonstrated the effectiveness of the profiles by building a client analysis that used the profiles to detect 38 different performance issues. Manually refactoring the code resulted in performance gains upto 30%.

7. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their constructive feedback. We are grateful to Monika Dhok, Malavika Samak and Sri Varun Poluri for their help in getting the benchmarks to run with soot and for reviewing the initial drafts of the paper. The second author is supported by a Google Faculty Research award.

8. REFERENCES

[1] O. Agesen and A. Garthwaite. Efficient object sampling via weak references. In *Proceedings of the 2Nd International Symposium on Memory Management*, ISMM '00, pages 121–126, New York, NY, USA, 2000.

[2] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, pages 46–57, Washington, DC, USA, 1996.

[3] T. Bao, Y. Zheng, Z. Lin, X. Zhang, and D. Xu. Strict control dependence and its effect on dynamic information flow analyses. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 13–24, New York, NY, USA, 2010.

[4] S. Bhattacharya, K. Gopinath, and M. G. Nanda. Combining concern input with program analysis for bloat detection. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 745–764, New York, NY, USA, 2013.

[5] S. Bhattacharya, M. G. Nanda, K. Gopinath, and M. Gupta. Reuse, recycle to de-bloat software. In *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP'11, pages 408–432, Berlin, Heidelberg, 2011.

[6] S. M. Blackburn, S. Singhai, M. Hertz, K. S. McKinely, and J. E. B. Moss. Pretenuring for java. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, pages 342–352, New York, NY, USA, 2001.

[7] D. C. D'Elia and C. Demetrescu. Ball-larus path profiling across multiple loop iterations. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 373–390, New York, NY, USA, 2013.

[8] L. Della Toffola, M. Pradel, and T. R. Gross. Performance problems you can fix: A dynamic analysis of memoization opportunities. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 607–622, New York, NY, USA, 2015.

[9] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[10] L. Gong, M. Pradel, and K. Sen. Jitprof: Pinpointing jit-unfriendly javascript code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 357–368, New York, NY, USA, 2015.

[11] L. Gong, M. Pradel, M. Sridharan, and K. Sen. DLint: Dynamically checking bad coding practices in javascript. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 94–105, New York, NY, USA, 2015.

[12] S. Z. Guyer, K. S. McKinley, and D. Frampton. Free-me: A static analysis for automatic individual object reclamation. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 364–375, New York, NY, USA, 2006.

[13] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie. Performance debugging in the large via mining millions of stack traces. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 145–155, Piscataway, NJ, USA, 2012.

[14] M. Hertz, S. M. Blackburn, J. E. B. Moss, K. S. McKinley, and D. Stefanović. Generating object lifetime traces with merlin. *ACM Trans. Program. Lang. Syst.*, 28(3):476–516, May 2006.

[15] W. huang, W. Srisa-an, and J. M. Chang. Dynamic pretenuring schemes for generational garbage collection. In *Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '04, pages 133–140, Washington, DC, USA, 2004.

[16] S. H. Jensen, M. Sridharan, K. Sen, and S. Chandra. Meminsight: Platform-independent memory debugging for javascript. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 345–356, New York, NY, USA, 2015.

[17] M. Jovic, A. Adamoli, and M. Hauswirth. Catch me if you can: Performance bug detection in the wild. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 155–170, New York, NY, USA, 2011.

[18] M. Jump, S. M. Blackburn, and K. S. McKinley. Dynamic object sampling for pretenuring. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, pages 152–162, New York, NY, USA, 2004.

[19] M. Jump and K. S. McKinley. Detecting memory leaks in managed languages with cork. *Softw. Pract. Exper.*, 40(1):1–22, Jan. 2010.

[20] J. R. Larus. Whole program paths. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 259–269, New York, NY, USA.

[21] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM*, 26(6):419–429, June 1983.

[22] A. Nistor, L. Song, D. Marinov, and S. Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 562–571, Piscataway, NJ, USA, 2013.

[23] R. Odaira and T. Nakatani. Continuous object access profiling and optimizations to overcome the memory wall and bloat. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 147–158, New York, NY, USA, 2012.

[24] O. Olivo, I. Dillig, and C. Lin. Static detection of asymptotic performance bugs in collection traversals.

In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 369–378, New York, NY, USA, 2015.

[25] M. Pradel and T. R. Gross. Automatic generation of object usage specifications from large method traces. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 371–382, Washington, DC, USA, 2009.

[26] M. Pradel, M. Huggler, and T. R. Gross. Performance regression testing of concurrent classes. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 13–25, New York, NY, USA, 2014.

[27] M. Selakovic and M. Pradel. Automatically fixing real-world javascript performance bugs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 811–812, Piscataway, NJ, USA, 2015.

[28] R. Shaham, E. K. Kolodner, and M. Sagiv. Heap profiling for space-efficient java. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 104–113, New York, NY, USA, 2001.

[29] A. Shankar, M. Arnold, and R. Bodik. Jolt: Lightweight dynamic analysis and removal of object churn. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 127–142, New York, NY, USA, 2008.

[30] W. N. Sumner and X. Zhang. Memory indexing: Canonicalizing addresses across executions. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 217–226, New York, NY, USA, 2010.

[31] W. N. Sumner, Y. Zheng, D. Weeratunge, and X. Zhang. Precise calling context encoding. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 525–534, New York, NY, USA, 2010.

[32] S. Tallam, X. Zhang, and R. Gupta. Extending path profiling across loop backedges and procedure boundaries. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 251–262, March 2004.

[33] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–, 1999.

[34] K. Vaswani, A. V. Nori, and T. M. Chilimbi. Preferential path profiling: Compactly numbering interesting paths. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07.

[35] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981.

[36] R. Wu, X. Xiao, S.-C. Cheung, H. Zhang, and C. Zhang. Casper: An efficient approach to call trace collection. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, pages 678–690, New York, NY, USA, 2016.

[37] B. Xin and X. Zhang. Efficient online detection of dynamic control dependence. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 185–195, New York, NY, USA, 2007.

[38] G. Xu. Resurrector: A tunable object lifetime profiling technique for optimizing real-world programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 111–130, New York, NY, USA, 2013.

[39] 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 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 419–430, New York, NY, USA, 2009.

[40] G. Xu, M. D. Bond, F. Qin, and A. Rountev. Leakchaser: Helping programmers narrow down causes of memory leaks. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 270–282, New York, NY, USA, 2011.

[41] D. Yan, G. Xu, and A. Rountev. Uncovering performance problems in java applications with reference propagation profiling. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 134–144, Piscataway, NJ, USA, 2012.

[42] X. Zhang and R. Gupta. Whole execution traces. In *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*, pages 105–116, 2004.

[43] X. Zhang, A. Navabi, and S. Jagannathan. Alchemist: A transparent dependence distance profiling infrastructure. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 47–58, Washington, DC, USA, 2009.