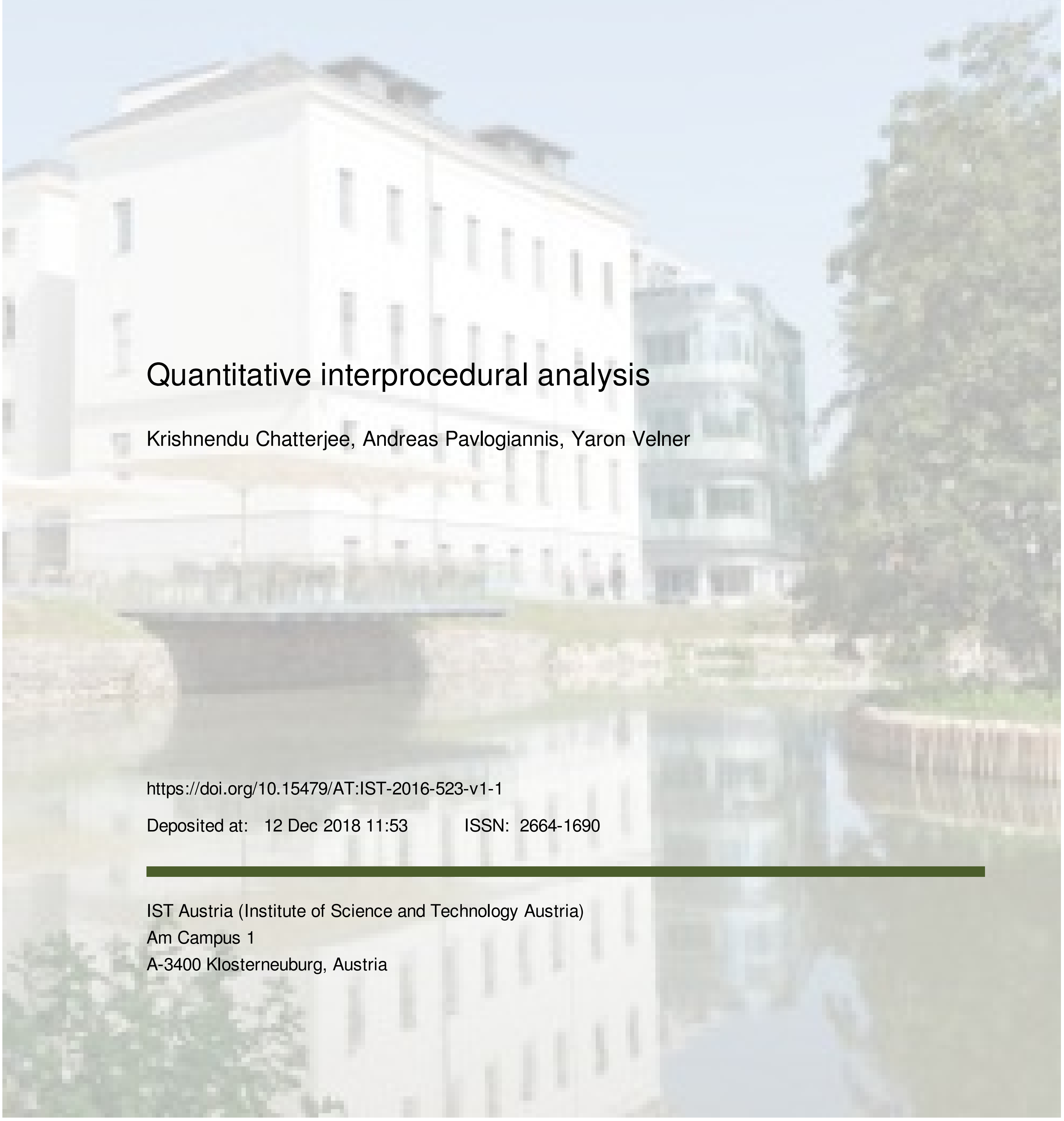

# Quantitative interprocedural analysis

Krishnendu Chatterjee, Andreas Pavlogiannis, Yaron Velner

https://doi.org/10.15479/AT:IST-2016-523-v1-1

Deposited at: 12 Dec 2018 11:53 ISSN: 2664-1690

# Quantitative Interprocedural Analysis

Krishnendu Chatterjee and Andreas Pavlogiannis and Yaron Velner

Technical Report No. IST-2016-523-v1+1
Deposited at 31 Mar 2016 09:11
https://repository.ist.ac.at/523/1/main.pdf

# Quantitative Interprocedural Analysis*

Krishnendu Chatterjee[1], Andreas Pavlogiannis[†1] and Yaron Velner[2]

[1]IST Austria (Institute of Science and Technology Austria) Klosterneuburg, Austria
[2]Tel Aviv University, Israel

March 31, 2016

**Abstract**

We consider the quantitative analysis problem for interprocedural control-flow graphs (*ICFG*s). The input consists of an *ICFG*, a positive weight function that assigns every transition a positive integer-valued number, and a labeling of the transitions (events) as good, bad, and neutral events. The weight function assigns to each transition a numerical value that represents a measure of how good or bad an event is. The quantitative analysis problem asks whether there is a run of the *ICFG* where the ratio of the sum of the numerical weights of good events versus the sum of weights of bad events in the long-run is at least a given threshold (or equivalently, to compute the maximal ratio among all valid paths in the *ICFG*). The quantitative analysis problem for *ICFG*s can be solved in polynomial time, and we present an efficient and practical algorithm for the problem. We show that several problems relevant for static program analysis, such as estimating the worst-case execution time of a program or the average energy consumption of a mobile application, can be modeled in our framework. We have implemented our algorithm as a tool in the Java Soot framework. We demonstrate the effectiveness of our approach with three case studies. First, we show that our framework provides a sound approach (no false positives) for the analysis of inefficiently-used containers. Second, we show that our approach can also be used for static profiling of programs which reasons about methods that are frequently invoked. Third, we show that the static profiling can be also lifted to collections of methods (e.g., libraries). Our experimental results show that our tool scales to relatively large benchmarks, and discovers relevant and useful information that can be used to optimize performance of the programs.

***Keywords***— Interprocedural analysis, Quantitative objectives, Mean-payoff and ratio objectives, Memory bloat, Static profiling.

## 1 Introduction

**Static and interprocedural analysis.** Static analysis techniques provide ways to obtain information about programs without actually running them on specific inputs. Static analysis explores the program behavior for *all* possible inputs and *all* possible executions. For non-trivial programs, it is impossible explore all the possibilities, and hence static analysis uses approximations (abstract interpretations) to account for all the possibilities [1]. Static analysis algorithms generally operate on the interprocedural control-flow graphs (for brevity *ICFG*s). An *ICFG* consists of a collection of control-flow graphs (CFGs), one for each procedure of the program. The CFG of each procedure has a *unique entry* node and a *unique exit* node, and other nodes represent statements of the program and conditions (in other words, basic blocks of the program).

*This work has been supported by the Austrian Science Foundation (FWF) under the NFN RiSE (S11405-07), FWF Grant P23499-N23, ERC Start grant (279307: Graph Games), ERC grant 321174-VSSC, Israel Science Foundation grant 652/11, and Microsoft faculty fellows award.

†pavlogiannis@ist.ac.at

In addition, there are *call* and *return* nodes for each procedure which represent invoking of procedures and return from procedures. Call-transitions connect call nodes to entry nodes; and return-transitions connect exit nodes to return nodes. Algorithmic analysis of *ICFG*s provides the mathematical framework for static analysis of programs. Interprocedural analysis with objectives such as reachability, set-based information, etc have been deeply studied in the literature [2, 3, 4, 5, 6, 7].

**Quantitative objectives.** A *qualitative* (or Boolean) objective assigns to every run of a program a Boolean value (accept or reject). A *quantitative* objective assigns to every run of a program a real value that represents a quality measure of the run. The analysis of programs with quantitative objectives is gaining huge prominence due to embedded systems with requirements on resource consumption, promptness of responses, performance analysis etc. Quantitative objectives have been proposed in several applications such as for worst-case execution time (see [8] for survey), power consumption [9], prediction of cache behavior for timing analysis [10], performance measures [11, 12, 13], to name a few. Another important feature of quantitative objectives is that they are very well-suited for *anytime algorithms* [14] (where anytime algorithms generate imprecise answers quickly, and proceed to construct progressively better approximate solutions with refinements) (for a more elaborate discussion see [15]).

**Mean-payoff and ratio objectives.** One of the most well-studied and mathematically elegant quantitative objectives is the *mean-payoff* objective, where a rational-valued weight is associated with every transition and the goal is to ensure that the long-run average of the weights along a run is at least a given threshold [16, 17, 18]. For example, consider a weight function that assigns to every transition the resource (such as power) consumption, then the mean-payoff objective measures the average resource consumption along a run. Along with *ICFG*s with mean-payoff objectives, we also consider *ratio* objectives. For ratio objectives, the transitions (events) of the *ICFG*s are labelled as good, bad, or neutral events, and a positive weight function assigns a positive integer-valued weight to every transition, and the weight function represents how good or bad an event is. The quantitative analysis problem asks if there is a run of the program such that the ratio of the sum of the weights of the good events versus the weights of the bad events in the long-run is at least a given threshold. For example, consider a weight function that assigns weight 1 to each transition, and a labeling of events as follows: whenever a request is made is a bad event, whenever a request is pending is a good event, and whenever no request is pending is a neutral event. The ratio objective assigns the long-run average time between requests and the corresponding grant per request for a run, and measures timeliness of responses to requests. Finite-state systems (or intraprocedural finite-state programs) with mean-payoff objectives have been studied in the literature in [16, 18, 19, 13] for performance modeling, and more recently applied in synthesis of reactive systems with quality guarantee [11] and robustness [20], reliability requirements and resource bounds of reactive systems [12, 21, 22].

**Interprocedural quantitative analysis.** Quantitative objectives such as mean-payoff and ratio objectives provide the appropriate framework to express several important system properties such as resource consumption and timeliness. While finite-state systems with mean-payoff objectives have been studied in the literature, the static analysis of *ICFG*s with mean-payoff and ratio objectives has largely been ignored. An interprocedural analysis is *precise* if it provides the meet-over-all-*valid*-paths solution (a path is valid if it respects the fact that when a procedure finishes it returns to the site of the most recent call). In the quantitative setting, the problem corresponds to finding the maximal value over all valid paths and to produce a witness (symbolic) path for that value. In this work, we consider precise interprocedural quantitative analysis for *ICFG*s with mean-payoff and ratio objectives.

**Our contributions.** In this work we present a flexible and general modelling framework for quantitative analysis and show how it can be used to reason about quantitative properties of programs and about potential optimizations in the program. We present an efficient polynomial-time algorithm for precise interprocedural quantitative analysis, which is implemented as a tool. We demonstrate the efficiency of the algorithm with two case studies, and show that our approach scales to programs with thousands of methods.

1. *(Theoretical modeling).* We show that *ICFG*s with mean-payoff and ratio objectives provide a robust framework that naturally captures a wide variety of static program analysis optimization and reasoning problems.

(a) *(Detecting container usage).* An exceedingly important problem for performance analysis is detection of runtime bloat that significantly degrades the performance and scalability of programs [23, 24]. A common source of bloat is inefficient use of containers [24]. We show that the problem of detecting usage of containers can be modeled as *ICFG*s with ratio objectives. A good use of a container corresponds to a good event and no use of the container is a bad event, and a misuse is represented as a low ratio of good vs bad events. Hence the container usage problem is naturally modeled as ratio analysis of *ICFG*s. While the problem of detecting container usage was already considered in [24], our different approach has the following benefits (see Section 5.2 for a comparison). First, our approach can handle recursion ([24] does not handle recursion). Second, our approach is sound, and does not yield false positives. Third, the approach of [24] ignored DELETE operations and we are able to take into consideration both ADD and DELETE operations (thus provide a more refined analysis). Moreover, our algorithmic approach for analysis of *ICFG*s is polynomial, whereas the algorithmic approach of [24] in the worst case can be exponential.

(b) *(Static profiling of programs).* We use our framework to model a conceptually new way for static profiling of programs for performance analysis. A line in the code (or a code segment) is referred as *hot* if there exists a run of the program where the line of code is frequently executed. For example, a function is referred as *hot* if there exists a run of the program where the function is frequently invoked, i.e., the frequency of calls to the function among all function calls is at least a given threshold. Similarly, a collection of functions is referred as hot if there exists a run of the program where the collection is frequently invoked (note that a collection of methods might be frequently invoked even if each individual method is not). Again this problem is naturally modeled as ratio problem for *ICFG*s, and our approach statically detects methods that are more frequently invoked. Optimization of frequently executed code would naturally lead to performance improvements and reasoning about hot spots in the code can assist the complier to apply optimization such as function inlining and loop unrolling (see Section 3.2 and Section 3.3 for more details).

(c) *(Other applications).* We show the generality of our framework by demonstrating that it provides an appropriate framework for theoretical modeling of diverse applications such as interprocedural worst-case execution time analysis, evaluating speedup in parallel computation, and interprocedural average energy consumption analysis.

2. *(Algorithmic analysis).* The quantitative analysis of *ICFG*s with mean-payoff objectives can be achieved in polynomial time by a reduction to pushdown systems with mean-payoff objectives (which can be solved in polynomial time [25]). However, the resulting algorithm in the worst case has time complexity that is a polynomial of degree thirteen and space complexity that is a polynomial of degree six (which is prohibitive in practice). We exploit the special theoretical properties of *ICFG*s in order to improve the theoretical upper bound and get an algorithm that in the worst case runs in cubic time and with quadratic space complexity. In addition, we exploit the properties of *real-world* programs and introduce optimizations that give a practical algorithm that is much faster than the theoretical upper bound when the relevant parameters (the total number of entry, exit, call, and returns nodes) are small, which is typical in most applications. Finally we present a linear reduction of the quantitative analysis problem with ratio objectives to mean-payoff objectives.

3. *(Tool and experimental results).* We have implemented our algorithm and developed a tool in the Java Soot framework [26]. We show through two case studies that our approach scales to relatively large programs from well-known benchmarks. The details of the case studies are as follows:

   (a) *(Detecting container usage).* Our experimental results show that our tool scales to relatively large benchmarks (DaCapo 2009 [27]), and discover relevant and useful information that can be used to optimize performance of the programs. Our tool could analyze all containers in several benchmarks (like muffin) whereas [24] could analyze them partially (in muffin only half of the containers were analyzed in [24] — before the predefined time bound was exceeded). Our sound approach allows us to avoid false reports (that were reported by [24]) and our simple mathematical modelling even allows us to report misuses that were not reported by [24].

   (b) *(Static profiling of programs).* We run an analysis to detect (i) hot methods and (ii) hot collections of methods for various thresholds. Our experimental results on the benchmarks report only a small fraction of the functions as hot for high threshold values, and thus give useful information about potential functions to be optimized for performance gain. In addition we perform a dynamic profiling and mark the top 5% of the most frequently invoked functions as hot. Our experiments show a significant correlation between the results of the static and dynamic analysis. In addition, we show that the sensitivity and specificity of the static classification can be controlled by considering different thresholds, where lower thresholds increase the sensitivity and higher thresholds increase the specificity. We investigate the trade-off curve (ROC curve) and demonstrate the prediction power of our approach.

Thus we show that several conceptually different problems related to program optimizations are naturally modeled in our framework, and demonstrate that we present a flexible and generic framework for quantitative analysis of programs. Moreover, our case studies show that our tool scales to benchmarks with real-world programs.

A preliminary version of this work has appeared in [28]. The current version expands upon [28] by including (i) full proofs of lemmas, (ii) a more detailed presentation with examples and illustrations, as well as (iii) an additional application case (Section 3.3) and experimental results (Section 5.4).

## 2 Definitions

In this section we present formal definitions of interprocedural control-flow graphs, and the quantitative analysis problems. We will use an example program, described in Figure 1 and Figure 2, to demonstrate each definition.

**Interprocedural control-flow graphs (*ICFG*s).** A program $P$ with $m$ methods is modeled by an interprocedural control-flow graph (*ICFG*) $\mathcal{A}$ which consists of a tuple $\langle A_1, \dots, A_m \rangle$ of $m$ modules, where each module $A_i = \langle N_i, \mathit{En}_i, \mathit{Ex}_i, \mathit{Calls}_i, \mathit{Retns}_i, \delta_i \rangle$ represents a method (or the control-flow graph of a method) in the program. A module $A_i$ contains the following components:

- A finite set of nodes $N_i$.
- An entry node $\mathit{En}_i$, which represents the first node of the method.
- An *exit* node $\mathit{Ex}_i$, which represents termination of the method.
- A finite set $\mathit{Calls}_i$ that denotes the set of *calls* of the method, and a finite set $\mathit{Retns}_i$ that denotes the set of *returns*.
- A transition relation $\delta_i$ defined as follows: A transition in $A_i$ is either (i) between two nodes in the same module (internal transitions) or between a return node and a node in the same module (i.e., $u, v$ such that $u \in N_i \cup \mathit{Retns}_i$ and $v \in N_i \cup \mathit{Calls}_i$); or (ii) between a call node of a module $A_i$ and the entry node of a module $A_j$ (which models the invocation of method $j$ from method $i$); or (iii) between the exit node of module $A_i$ and a return node of a module $A_j$ (which models the case that method $i$ terminated and the run of the program continues in method $j$ which invoked method $i$).

We denote by $N = \bigcup_{1 \le i \le m} N_i$ and similarly, $\mathit{En} = \bigcup_{1 \le i \le m} \{\mathit{En}_i\}$; $\mathit{Ex} = \bigcup_{1 \le i \le m} \{\mathit{Ex}_i\}$; $\mathit{Calls} = \bigcup_{1 \le i \le m} \mathit{Calls}_i$; $\mathit{Retns} = \bigcup_{1 \le i \le m} \mathit{Retns}_i$; and $\delta = \bigcup_{1 \le i \le m} \delta_i$. In the sequel we use $N$ (resp. $N_i$) to denote all nodes in $\mathcal{A}$ (resp. $A_i$), and refer to the nodes of $N \setminus (\mathit{Calls} \cup \mathit{Retns} \cup \mathit{En} \cup \mathit{Ex})$ as *internal nodes*.

**Quantitative *ICFG*s.** A *quantitative ICFG* (*QICFG* for brevity) consists of an *ICFG* $\mathcal{A}$ and a weight function $w$ that assigns a rational-valued weight $w(e) \in \mathbb{Q}$ to every transition $e$, where $\mathbb{Q}$ is the set of all rationals.

**Example 1.** *Consider an example program shown in Figure 1 and Figure 2. In this example, (i)* Modules: $A_1 = \mathit{main}, A_2 = \mathit{foo}$*; (ii)* Nodes: $N_2 = \{$*f:Entry, if(x>1), f:call foo, f:foo ret, x++, f:Exit*$\}$*; (iii)* Entry and exit: $\mathit{En}_2 =$ *f:Entry and* $\mathit{Ex}_2 =$ *f:Exit; (iv)* Calls and returns: $\mathit{Calls}_2 = \{$*f:call foo*$\}$ *and* $\mathit{Retns}_2 = \{$*f:foo ret*$\}$*; and (v)* Transition: *for example,* $($*x++,f:Exit*$) \in \delta_2$, $($*f:Exit,f:foo ret*$) \in \delta_2$ *and* $($*f:Exit,m:foo ret*$) \in \delta_2$.

**Configurations and paths.** A *configuration* consists of a sequence $c = (r_1, \dots, r_j, u)$, where each $r_i$ is a return node (i.e., $r_i \in \mathit{Retns}$) and $u \in N$ is a node in one of the modules. Intuitively, when the module

```
void main(){
  while( x ){
    if( y > 0 )
      foo( x );
    else
      z = 7;
  }
  z++;
  return;
}
```

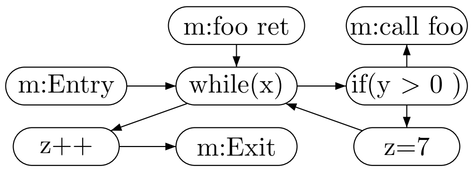

Figure 1: main

```
int foo( int x ){
  if( x > 1 )
    foo(x - 1);
  x++;
  return x;
}
```

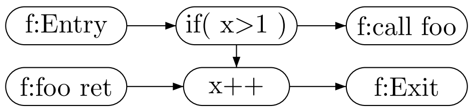

Figure 2: foo

that $u$ belongs to terminates, the program will continue in $r_j$. A sequence of configurations is *valid* if it does not violate the transition relation, and a *path* $\pi$ is a valid sequence of configurations. We note that a path can be equivalently represented by the first configuration and a sequence of transitions. For a path $\pi = \langle c_1, c_2, \dots, c_\ell \rangle$ we denote by (i) $n_i$: the node of configuration $c_i$ (i.e., $c_i = (r_1, \dots, r_j, n_i)$), and (ii) $\alpha_i$: the stack string of $c_i$ (i.e., $\alpha_i = r_1, \dots, r_j$). For a path $\pi$ let $\pi[1,i]$ denote the prefix of length $i$ of $\pi$. A run of the program is modeled by a path in $\mathcal{A}$.

**Example 2.** *Consider the program and the corresponding ICFG shown in Figure 1 and Figure 2. An example of a run in the program modeled as a path in the ICFG is as follows:* $\langle(\epsilon,$ *m:Entry*$), (\epsilon,$ *while(x)*$), (\epsilon,$ *if* $(y > 0)), (\epsilon,$ *m:call foo*$),$ (*m:foo ret, f:Entry*), (*m:foo ret, if* $(x > 1)$), (*m:foo ret, f:call foo*), (*m:foo ret, f:foo ret, f:Entry*), $\dots\rangle$*, where* $\epsilon$ *denotes the empty stack.*

**Ratio analysis problem.** In this work we consider the *ratio analysis* problem, where every transition of a *ICFG* has a label from the set $\{good, bad, neutral\}$. Intuitively, desirable events are labelled as *good*, undesirable events are labelled as *bad*, and other events are labelled as *neutral*. The ratio analysis problem, given a *ICFG*, a labeling of the events, and a threshold $\lambda > 0$, asks to determine whether there is a run where the ratio of sum of weights of *good* events vs the sum of weights of the *bad* events that is greater than the threshold $\lambda$. Formally, we consider a positive integer-weight function $w$, that assigns a positive integer-valued weight to every transition, and for good and bad events the weight denotes how good or how bad the respective event is. For a finite path $\pi$ we denote by $good(w(\pi))$ (resp., $bad(w(\pi))$) the sum of weights of the good (resp., bad) events in $\pi$. In particular, for the weight function $w$ that assigns weight 1 to every transition, we have that $good(w(\pi))$ (resp. $bad(w(\pi))$) represents the number of good (resp. bad) events. We denote $Rat(w(\pi)) = \frac{good(w(\pi))}{\max\{1, bad(w(\pi))\}}$ the ratio of the sum of weights of good and bad events in $\pi$ (note that in denominator we have $\max\{1, bad(w(\pi))\}$ to remove the pathological case of division by zero). For an infinite path $\pi$ we denote

$$LimRat(w(\pi)) = \begin{cases} \liminf_{i\to\infty} Rat(w(\pi[1,i])) & \text{if } \pi \text{ has infinitely many} \\ & \text{good or bad events;} \\ 0 & \text{otherwise;} \end{cases}$$

informally this represents the ratio as the number of relevant (good/bad) events goes to infinity. Our analysis focuses on paths with unbounded number of relevant events, and infinitely many events provide an elegant abstraction for unboundedness. Hence, we investigate the following problem:

> Given a *ICFG* with labeling of good, bad, and neutral events, a positive integer weight function $w$, and a threshold $\lambda \in \mathbb{Q}$ such that $\lambda > 0$, determine whether there exists an infinite path $\pi$ such that $\mathit{LimRat}(w(\pi)) > \lambda$.

**Remark 1.** *Our approach can be extended to reason about finite runs by a adding an auxiliary transition, labeled as a neutral event, from the final state of the program to its initial state (also see Section 3.4).*

**Mean-payoff analysis problem.** In the mean-payoff analysis problem we consider a *QICFG* with a rational-valued weight function $w$. For a finite path $\pi$ in a *QICFG* we denote by $w(\pi)$ the total weight of the path (i.e., the sum of the weights of the transitions in $\pi$), and by $\mathit{Avg}(w(\pi)) = \frac{w(\pi)}{|\pi|}$ the average of the weights, where $|\pi|$ denotes the length of $\pi$. For an infinite path $\pi$, we denote $\mathit{LimAvg}(w(\pi)) = \liminf_{i \to \infty} \mathit{Avg}(w(\pi[1,i]))$. The mean-payoff analysis problem asks whether there exists an infinite path $\pi$ such that $\mathit{LimAvg}(w(\pi)) > 0$. In Section 4 we show how the ratio analysis problem of *ICFG*s reduces to the mean-payoff analysis problem of *QICFG*s.

**Assigning context-dependent and path-dependent weights.** In our model the numerical weights are assigned to every transition of a *ICFG*. First, note that since we consider weight functions as an input and allow all weight functions, the weights could be assigned in a dependent way. Second, in general, we can have an *ICFG*, and a finite-state deterministic automaton (such as a deterministic mean-payoff automaton [12]) that assigns weights. The deterministic automaton can assign weights depending on different contexts (or call strings) of invocations, or even independent of the context but dependent on the past few transitions (i.e., path-dependent), i.e., the automaton has the stack alphabet and transition of the *ICFG* as input alphabet and assigns weights depending on the current state of the automaton and an input letter. We call such a weight function *regular weight function.* Given a regular weight function specified by an automaton $A$ and an *ICFG* we can obtain a *ICFG* (that represents the path-dependent weights) by taking their synchronous product, and hence we will focus on *ICFG*s for algorithmic analysis. The regular weight function can also be an *abstraction* of the real weight function, e.g., the regular weight function is an *over-approximation* if the weights that it assigns to the good (resp. bad) events are higher (resp. lower) than the real weights. If the original weight function is bounded, then an over-approximation with a regular weight function can be obtained (which can be refined to be more precise by allowing more states in the automaton of the regular weight function). Note that the new *ICFG* which is obtained from an *ICFG* and automaton $A$ has a blowup in the number of states of $A$, and thus there is a tradeoff between the precision of $A$ and the size of the new *ICFG* constructed.

## 3 Applications: Theoretical Modeling

In this section we show that many diverse problems for static analysis can be reduced to ratio analysis of *ICFG*s. We will present experimental results (in Section 5) for the problems described in Section 3.1 and Section 3.2.

### 3.1 Container analysis

The inefficient use of containers is the cause of many performance issues in Java. An excellent exposition of the problem with several practical motivations is presented in [24]. The importance of accurate identification of misuse of containers that minimizes (and ideally eliminates) the number of false warnings was emphasized in [24] and much effort was spent to avoid false warnings for real-world programs. We show that the ratio analysis for *ICFG*s provides a mathematically sound approach for the identification of inefficient use of containers.

**Two misuses.** We aim to capture two common misuses of containers following the definitions in [24]. The first inefficient use is an *underutilized container* that always holds very few number of elements. The cause of inefficiency is two-fold: (i) a container is typically created with a default number of slots, and much more memory is allocated than needed; and (ii) the functionality that is associated with the container is typically not specialized to the case that it has only very few elements. The second inefficiency is caused by *overpopulated containers* that are looked up rarely, though potentially they can have many elements. This causes a memory waste and performance penalty for every lookup. Thus we consider the following two cases of misuse:

1. A container is *underutilized* if there exists a constant bound on the number of elements that it holds for all runs of the program.
2. For a threshold $\lambda$, a container is *overpopulated* if for all runs of the program the ratio of Get vs Add operations is less than $\lambda$.

We note that our approach is demand-driven (where users can specify to check the misuse of a specific container).

**Modeling.** The modeling of programs as *ICFG*s is standard. We describe how the weight function and the ratio analysis problem can model the problem of detecting misuses. We abstract the different container operations into Get, Add, and Delete operations. For this purpose we require the user to annotate the relevant class methods by Get, Add, or Delete; and by a weight function that corresponds to the number of Get, Add, or Delete operations that the method does (typically this number is 1). For example, in the class **HashSet**, the **add** method is annotated by Add, the **contains** method is annotated by Get and the **remove** operation is annotated by Delete. The **clear** operation which removes all elements from the set is annotated by Delete but with a large weight (if **clear** appears in a loop, it dominates the add operations of the loop). We note that the annotation can be automated with the approach that is described in [24].

1. When detecting underutilized containers we define Add operations as good events and Delete operations as bad events, and check for threshold 1. Note that the relevant threshold is 1: if the (long-run) ratio of Add vs Delete is not greater than 1, then the total number of elements in the container is bounded by a constant.
2. When detecting overpopulated containers we define Get operations as good events and Add operations as bad events, and check for the given threshold $\lambda$.

In addition, since we wish to analyze heap objects, the allocation of the container is a bad event with a large weight (i.e., similar effect as of **clear**); see Example 3. The container is misused iff the answer to the ratio analysis problem is NO (note that in the problem description for container analysis we have quantification over all paths and for ratio analysis of *ICFG*s the quantification is existential). The detection is demand-driven and done for an allocated container **c**.

*Details of modeling.* Intuitively, a transition in the call graph is good if it invokes a functionality that is annotated by a good operation (i.e., Add operation for the underutilized analysis and Get operation for the overpopulated analysis) and the object that invokes the operation points to container **c**, and it is bad if the invoked operation is annotated as bad. Formally, for a given allocated container **c**: If at a certain line a variable **t** that *may point to* **c** invokes a good functionality, then we denote the transition as good. If **t** *must point to* **c** and invokes a bad functionality, then we denote it as a bad events. All other transitions are neutral. Note that our modeling is conservative. The misuse is detected for the container **c** if all runs of the program have a ratio of good vs bad events that is below the threshold (in other words, the container is not misused if there exists a path where the ratio of good vs bad events is above the threshold, and this exactly corresponds to the ratio analysis of *ICFG*s).

**Example 3.** *We illustrate some important aspects of the container analysis problem with an example. Consider the program shown in Figure 3. We consider the containers that are allocated in line 9 and in line 20 and analyze them for underutilization. There exist runs that go through line 14 and properly use the container that is allocated in line 9, since qux method can add unbounded number of elements to the hash table (due to its recursive call). However, the container in line 20 is underutilized, since in every run the number of elements is bounded by 2. However, note that if the* Delete *operation is not handled, then the container is reported as properly used. We note that since we assign large weights to the allocation of the container,*

```
 1 void qux( Hashtable h, int x ){
 2    h.put( x, x/2 );
 3    if( x > 0 ){
 4       qux( h, x/2 );
 5    }
 6 }
 7
 8 Hashtable bar( int x ){
 9    return new Hashtable(x*x);
10 }
```

```
12 void foo( int x ){
13    if( x % 2){
14       Hashtable h1 = bar(x);
15       qux(h1, x);
16    }
17    else
18    {
19       for( int y = 0 ; y < x ; y++ ){
20          Hashtable h2 = new Hashtable(y);
21          h2.put(y,x);
22          for( int z = 0 ; z < y ; z++ )
23          {
24             h2.put(z,y);
25             ...
26             h2.remove(z);
27          }
28       }
29    }
30 }
```

Figure 3: An example for underutilized container analysis.

*this prevents the analysis from reporting that h2 properly uses the container that is allocated in line 20. In summary, the example illustrates the following important features: (1) the proper usage of the container should be tested also outside of its allocation site*[1] *(as opposed to the approach of [24]); (2) sometimes the proper usage of a container is due to recursion; and (3) handling* DELETE *operations appropriately increases the precision of analysis. While these important features are illustrated with the toy example, such behaviors were also manifested in the programs of the benchmarks (see Subsection 5.2 for details).*

*Soundness.* Our ratio analysis approach for *ICFG*s is both sound and complete (with respect to the weighted abstracted *ICFG*). Since we use a conservative approach for assigning bad and good events, the *ICFG* we obtain for the misuse analysis of containers is sound and we get the following result.

**Theorem 1.** (SOUNDNESS). *The underutilized and overpopulated container analysis through the ratio analysis problem on ICFGs is sound (do not report false positives, i.e., any reported misused container is truly misused).*

**Remark 2.** *We remark about the significance of the soundness of our approach.*

- *The soundness criteria is a very important and desirable feature for container analysis (for details see [29, 24]), because a reported misused container needs to be analyzed manually and incurs a substantial effort for optimization. Hence as argued in [29, 24], spurious warnings (false positive) of misuse must be minimized. In our approach, a misuse is reported iff in every run a misuse is detected, and with a sound (over approximation) annotation of the weights our approach is sound.*
- *The soundness of our approach is with respect to a sound (over approximation) annotation of the* ADD *,* DELETE *and* GET *operations. In addition, for a given ICFG, our ratio analysis is* precise *(i.e.. both sound and complete), hence, our container analysis is sound.*

## 3.2 Static profiling of methods

Finding the most frequently executed lines in the code can help the programmer to identify the critical parts of the program and focus on the optimization of these parts. It can also assist the compiler (e.g., a C compiler) to decide whether it should apply certain optimizations such as *function inlining* (replacing a

[1]E.g., an HashTable is allocated in bar function but the proper usage is done outside the allocation site, namely, after the termination of bar.

function call by the body of the called function) and *loop unrolling* (re-write the loop as a repeated sequence of similar independent statements). These optimizations can reduce the running time of the program, but on the other hand, they increase the size of the (binary) code. Hence, knowing whether the function or loop is *hot* (frequently invoked) is important when considering the time vs. code size tradeoff. In this subsection we present the model for profiling the frequency of function calls (which allows finding hot functions), and we note that our profiling technique is generic and can be scaled to detect other hot spots in the code (e.g., hot loops).

**Problem description.** Given a program with several functions, a function $f$ is called $\lambda$-hot, if there exists an (interprocedural) run (of unbounded length) of the program where the frequency of calls to $f$ (among all function calls in the run) is at least $\lambda$. Formally, for a run, given a prefix of length $i$, let $\#f(i)$ denote the number of calls to $f$ and $\#c(i)$ denote the number of function calls in the prefix of length $i$. The function is $\lambda$-hot if there exists a run such that $\liminf_{i\to\infty} \frac{\#f(i)}{\#c(i)} > \lambda$.

**Modeling.** The modeling of programs as *ICFG*s is straightforward. We describe the labeling of events and weight function in *ICFG*s to determine if a function $f$ is $\lambda$-hot. First we label call-transitions to $f$ as good events and assign weight 1; then we label all other call-transitions as bad events and assign them weight 1. To ensure that the number of calls to $f$ also appear in the denominator (in the total number of calls) we label transitions from the entry node of $f$ as bad events with weight 1. The function $f$ is $\lambda$-hot iff the answer to the ratio analysis problem with threshold $\lambda$ is YES.

## 3.3 Static profiling of libraries

Modern software comprises thousands of methods and classes, which are usually grouped into libraries. In several programming languages (e.g. C++), the developer decides whether libraries will be statically or dynamically linked. While there are several factors that affect this choice (e.g., compatibility issues, licensing restrictions etc.), one consideration is that of performance, as statically linked libraries allow for speedups such as the following.

1. Interprocedural optimizations coming from the compiler by performing interprocedural analysis on the target program together with the statically linked libraries.
2. Avoiding runtime overheads imposed by invocations of methods that lie in dynamically linked libraries.
3. Cache-level optimizations, by allowing the compiler to arrange statically linked libraries in such order to try to minimize cache misses.

On the other hand, infrequently used libraries might better be dynamically linked, (i) to keep the interprocedural optimizations fast, (ii) keep the size of the executable small, and (iii) have fewer cache misses. Estimating statically the frequently used libraries can assist static vs dynamic linking.

**Problem description.** Similarly as in Section 3.2, given a program that links with several libraries, a library $\ell$ is called $\lambda$-hot, if there exists an (interprocedural) run (of unbounded length) of the program where the frequency of calls to methods of $\ell$ (among all method calls to libraries in the run) is at least $\lambda$. Formally, for a run, given a prefix of length $i$, let $\#\ell(i)$ denote the number of calls to methods of the library $\ell$ and $\#t(i)$ denote the number of function calls to library methods in the prefix of length $i$. The library $\ell$ is $\lambda$-hot if there exists a run such that $\liminf_{i\to\infty} \frac{\#\ell(i)}{\#t(i)} > \lambda$.

**Modeling.** The modeling is similar to that of Section 3.2. The program is modeled as an *ICFG*. Call transitions to methods of the library $\ell$ are labeled as good events, whereas call transitions to methods of other libraries are labeled as bad events. To ensure that the number of calls to methods of $\ell$ also appear in the denominator (in the total number of library calls) we label transitions from the entry node of every method of $\ell$ as bad events. All labeled events receive weight 1. The library $\ell$ is $\lambda$-hot iff the answer to the ratio analysis problem with threshold $\lambda$ is YES.

## 3.4 Estimating worst-case execution time

The approach of [15] for estimating worst-case execution time (WCET) is also naturally captured by ratio analysis. While the intraprocedural problem was considered in [15], our approach allows the more general interprocedural analysis. In this approach, we consider (as in [15]) that each program statement is assigned a cost that corresponds to its running time (e.g., number of CPU cycles).

**Modeling.** The modelling of WCET analysis of the program is as follows: We add to the *ICFG* of the program a transition from every terminal node to the initial node, and every such transition is a bad event with weight 1. All the other transitions are good events and their weight is their cost (running time). The WCET of the program is at most $N$ cycles if and only if the answer to the ratio analysis problem with threshold $N$ is NO.

## 3.5 Evaluating the speedup in a parallel computation

The speed of a parallel computing is limited by the time needed for the sequential fraction of the program. For example, if a program runs for 10 minutes on a single processor core, and a certain part of the program that takes 2 minutes to execute cannot be parallelized, then the minimum execution time cannot be less than two minutes (regardless of how many processors are devoted to a parallelized execution of this program). Hence, the speedup is at most 5. Amdahl's law [30] states that the theoretical speedup that can be obtained by executing a given algorithm on a system capable of executing $n$ threads of execution is at most $\frac{1}{B+\frac{1}{n}(1-B)}$, where $B$ is the fraction of the algorithm that is strictly serial. Our ratio analysis technique can be used to (conservatively) estimate the value of $B$ and thus to evaluate the outcome of Amdahl's law.

**Modeling.** As in Section 3.4, we consider that the cost of every program statement is given, and we add to the *ICFG* of the program a transition from every terminal node to the initial node, this time as a neutral event with weight 0. All the transitions of the code that cannot be parallelized are defined as good events, and the other transitions are defined as bad events. We denote by $P$ the fraction of the code that can be parallelized and by $S$ the fraction of the code that is strictly serial. The value of $\frac{S}{P}$ is at most $\lambda$ if and only if the answer to the ratio analysis problem with threshold $\lambda$ is NO. Hence $B$ is bounded by $\frac{1}{1+\frac{1}{\lambda}}$ for which the answer to the ratio analysis problem with threshold $\lambda$ is NO.

## 3.6 Average energy consumption

In the case of many consumer electronics devices, especially mobile phones, battery capacity is severely restricted due to constraints on size and weight of the device. This implies that managing energy well is paramount in such devices. Since most mobile applications are non-terminating (e.g., a web browser), the most important metric for measuring energy consumption is the average memory consumption per time unit [31], e.g., watts per second.

**Modeling.** We consider that the running time and energy consumption of each statement in the application code is given (or is approximated). In our modeling we split each transition in the *ICFG* into two consecutive transitions, the first is a good event and the next is a bad event. The good event is assigned with a weight that corresponds to the energy consumption of the program statement and the bad event is assigned with a weight that corresponds to the running time of the statement. The average energy consumption of the application is at most $\lambda$ if and only if the answer to the ratio analysis problem is NO.

# 4 Algorithm for Quantitative Analysis of *QICFG*s

In this section we present three results. The mean-payoff analysis problem for *QICFG*s can be solved in polynomial time, this can be derived from [25]. First, we present an algorithm that significantly improves the current theoretical bound for the problem for *QICFG*s. Second, we present an efficient algorithm that in most practical cases is much faster as compared to the theoretical upper bound. Finally, we present a linear reduction of the ratio analysis problem to the mean-payoff analysis problem for *QICFG*s.

## 4.1 Improved algorithm for mean-payoff analysis

In this section we first discuss the basic polynomial-time algorithm for mean-payoff analysis of *QICFG*s that can be obtained from the results on pushdown systems shown in [25]. Due to space constraints the technical proofs are relegated to the supplementary material.

**Results of [25] and reduction.** The results of [25] show that pushdown systems with mean-payoff objectives can be solved in polynomial time. Given a pushdown system with state space $Q$ and stack alphabet $\Gamma$, the polynomial-time algorithm of [25] can be described as follows. The algorithm is iterative, and in each iteration it constructs a finite graph of size $O(|Q| \cdot |\Gamma|^2)$ and runs a Bellman-Ford style algorithm on the finite graph from each vertex. The Bellman-Ford algorithm on the finite graph from all vertices in each iteration requires $O(|Q|^3 \cdot |\Gamma|^6)$ time and $O(|Q|^2 \cdot |\Gamma|^4)$ space. The number of iterations required is $O(|Q|^2 \cdot |\Gamma|^2)$. Thus the time and space requirement of the algorithm are $O(|Q|^5 \cdot |\Gamma|^8)$ and $O(|Q|^2 \cdot |\Gamma|^4)$, respectively. A *QICFG* can be interpreted as a pushdown system where $N$ corresponds to $Q$ and *Retns* corresponds to $\Gamma$.

**Theorem 2.** (Basic algorithm [25]). *The mean-payoff analysis problem for QICFGs can be solved in* $O(|N|^5 \cdot |Retns|^8)$ *time and* $O(|N|^2 \cdot |Retns|^4)$ *space, respectively.*

**Improved algorithm.** We will present an improved polynomial-time algorithm for the mean-payoff analysis of *QICFG*s. The improvement relies on the following properties of *QICFG*s:

1. The transitions of a module are independent of the stack of a configuration, while in pushdown systems the transitions can depend on the top symbol of the stack. This enables to reduce the size of the finite graphs to be considered in every iteration.
2. Every call node has only one corresponding return node. Therefore, if a module $A_1$ invokes a module $A_2$, then the behavior of $A_1$ after the termination of $A_2$ is independent of $A_2$. This enables us to reduce the number of iterations to $O(|Calls|)$.

To present the improved algorithm and its correctness formally, we need a refined analysis and extensions of the results of [25]. We first describe a key aspect and present an overview of the solution.

**Remark 3.** *(Infinite-height lattice). Our algorithm will be an iterative algorithm till some fixpoint is reached. However, for interprocedural analysis with finite-height lattices, fixpoints are guaranteed to exist. Unfortunately in our case for mean-payoff objectives, it is an infinite-height lattice. Thus a fixpoint is not guaranteed. For this reason the analysis for mean-payoff objectives is more involved, and this is even in the case of finite graphs. For example, for reachability objectives in finite graphs linear-time algorithms exist, whereas for finite graphs with mean-payoff objectives the best-known algorithms (for over three decades) are quadratic [32]. This difference is even more pronounced in our case of recursive graphs.*

**Solution overview.** In finite graphs the solution for the mean-payoff analysis is to check whether the graph has a cycle $C$ such that the sum of weights of $C$ is positive. If such a cycle exists, then a *lasso* path that leads to the cycle and then follows the cyclic path forever has positive mean-payoff value. For *QICFG*s we show that it is enough to find either a loop in the program such that the sum of weights of the loop is positive or a sequence of calls and returns with positive total weight such that the last invoked module is the same as the first invoked module. For this purpose we compute a *summary function* that finds the maximum weight (according to the sum of weights) path between every two statements of a method (i.e., between every two nodes of a module). The computation is an extension of the Bellman-Ford algorithm to *QICFG*s. We show that it is enough to compute a summary function for *QICFG*s with a *stack height* that is bounded by some constant, and then all that is left is to mark pairs of nodes such that the weight of a maximal weight path between them is unbounded. In finite graphs the maximum weight between two vertices is unbounded only if the graph has a cycle with positive sum of weights (i.e., a path with positive total weight that can be pumped). For *QICFG*s it is also possible to *pump* special types of acyclic paths. We first characterize these pumpable paths (up to Lemma 2). We then show how to compute a bounded summary function (Lemma 3 and the paragraph that follows it and Example 6). Finally we show how to use the summary function to solve the mean-payoff analysis problem. We start with the basic notions related to stack heights and pumpable paths, and their properties which are crucial for the algorithm.

*Stack heights.* The *configuration stack height* of $c = (r_1, \ldots, r_j, u)$, denoted as $\mathsf{SH}(c)$, is $j$. For a finite path $\pi = \langle (\alpha_1, n_1), \ldots, (\alpha_\ell, n_\ell) \rangle$, the stack height of the path (denoted by $\mathsf{SH}(\pi)$) is the maximal stack height of all the configurations in the path. Formally $\mathsf{SH}(\pi) = \max\{|\alpha_1|, \ldots, |\alpha_\ell|\}$. The *additional stack height* of $\pi$ is the additional height of the stack in the segment of the path, i.e., the additional stack height $\mathsf{ASH}(\pi)$ is $\mathsf{SH}(\pi) - \max(|\alpha_1|, |\alpha_\ell|)$.

*Pumpable pair of paths.* Let $\pi = \langle c_1 t_1 t_2 \ldots \rangle$ be a finite or infinite path (where each $t_i$ is a transition in the *QICFG*). A *pumpable pair of paths* for $\pi$ is a pair of non-empty sequences of transitions: $(p_1, p_2) = (t_{i_1} t_{i_1+1} \ldots t_{i_1+\ell_1}, t_{i_2} t_{i_2+1} \ldots t_{i_2+\ell_2})$, for $\ell_1, \ell_2 \geq 0$, $i_1 \geq 0$ and $i_2 > i_1 + \ell_1$ such that for every $j \geq 0$ the path $\pi^j_{(p_1,p_2)}$ obtained by pumping the pair $p_1$ and $p_2$ of paths $j$ times each is a valid path, i.e., for every $j \geq 0$ we have

$$\pi^j_{(p_1,p_2)} = \langle c_1 t_1 \ldots t_{i_1-1} (p_1)^j t_{i_1+\ell_1+1} \ldots t_{i_2-1} (p_2)^j t_{i_2+\ell_2+1} \ldots \rangle$$

is a valid path. We illustrate the above definitions with the next example.

**Example 4.** *Consider the program from Figure 1 and Figure 2 and the corresponding ICFG. A possible path in the program is*

*m:Entry $\to$ while(x) $\to$ if(y>0) $\to$ m:call foo $\to$ f:Entry $\to$ if(x>1) $\to$ f:call foo $\to$ f:Entry $\to$ if(x>1) $\to$ x++ $\to$ f:Exit $\to$ f:foo ret $\to$ x++ $\to$ f:Exit $\to$ m:foo ret $\to$ while(x)*

*and we denote this path with $\pi$. Then $\mathsf{ASH}(\pi) = 2$, and the pair of paths f:Entry $\to$ if(x>1) $\to$ f:call foo and f:foo ret $\to$ x++ $\to$ f:Exit is a pumpable pair of paths.*

In the next lemmas we first show that every path with large additional stack has a pumpable pair of paths, and then establish the connection of additional stack height and the existence of pumpable pair of paths with positive weights in Lemma 2. The key intuition for the proof of the next lemma is that a path with $\mathsf{ASH}(\pi) > |\mathit{Calls}| + 1$ must contain a recursive call that can be pumped.

**Lemma 1.** *Let $\pi$ be a finite path with $\mathsf{ASH}(\pi) = d > |\mathit{Calls}| + 1$. Then $\pi$ has a pumpable pair of paths.*

*Proof.* Intuitively a path with $\mathsf{ASH}(\pi) > |\mathit{Calls}| + 1$ must contain a recursive call that can be pumped. We now present the detailed argument. Let $c_0$ and $c_k$ be the starting and the end configurations of the finite path $\pi$, respectively. Let $\ell = \max\{\mathsf{SH}(c_0), \mathsf{SH}(c_k)\}$. Given $\pi$, let $c_1$ be the first configuration in $\pi$ of stack height strictly greater than $\ell$ and with a call node $n_1 \in \mathit{Calls}_i$ (for some module $A_i$) such that there exists a configuration $c_2$ in $\pi$ with a call node $n_2 \in \mathit{Calls}_i$ satisfying the following conditions: (i) $n_1 = n_2$ and (ii) in the path segment in $\pi$ between $c_1$ and $c_2$ the stack height is always at least $\mathsf{SH}(c_1)$. Moreover, let $c_3$ be the first configuration after $c_2$ of stack height $\mathsf{SH}(c_1)$ and with a return node $n_3 \in \mathit{Retns}_i$. We first justify the existence of these configurations: (i) the existence of $c_1$ and $c_2$ follows by the pigeonhole principle and the fact that $\mathsf{ASH}(\pi) > |\mathit{Calls}| + 1$; and (ii) the existence of $c_3$ follows because $\mathsf{SH}(c_1) > \mathsf{SH}(c_k)$ and hence the call corresponding to $c_1$ must return in the path $\pi$. Note that existence of $c_3$ (i.e., the return of the call of $c_1$) implies the existence of a configuration $c_4$ with a return node $n_4 \in \mathit{Retns}_i$ in the path such that $\mathsf{SH}(c_4) = \mathsf{SH}(c_2)$, (this corresponds to the return of the call of $c_2$). Note that since $n_1 = n_2$, it follows that $n_3 = n_4$ (as they corresponds to the return of the same call node). The path segment $p_1$ of $\pi$ between $c_1$ and $c_2$, and the path segment $p_2$ of $\pi$ between $c_4$ and $c_3$, constitutes a pumpable pair. The result follows. $\square$

**Lemma 2.** *Let $c_1, c_2$ be two configurations and $j \in \mathbb{Z}$. Let $d \in \mathbb{N}$ be the minimal additional stack height of all paths between $c_1$ and $c_2$ with total weight at least $j$. If $d > |\mathit{Calls}| + 1$, then there exists a path $\pi^*$ from $c_1$ to $c_2$ with additional stack height $d$ that has a pumpable pair $(p_1, p_2)$ with $w(p_1) + w(p_2) > 0$.*

*Proof.* Let us consider the set of paths $\Pi$ between $c_1$ and $c_2$ with total weight at least $j$, and let $\Pi_{\min}$ be the subset of $\Pi$ that has minimal additional stack height. The proof is by induction on the length of paths in $\Pi_{\min}$. Consider a path $\pi$ from $\Pi_{\min}$ that has the shortest length among all paths in $\Pi_{\min}$. Since $\mathsf{ASH}(\pi) = d > |\mathit{Calls}| + 1$, then by Lemma 1 it contains a pumpable pair. Let us consider the path $\pi_1$ obtained from $\pi$ by pumping the pumpable pair zero times (i.e., the pumpable pair is removed). Since we remove a part of the path we have that $\mathsf{ASH}(\pi_1) \leq \mathsf{ASH}(\pi)$. If $w(\pi_1) \geq w(\pi)$, then we obtain a path $\pi_1$

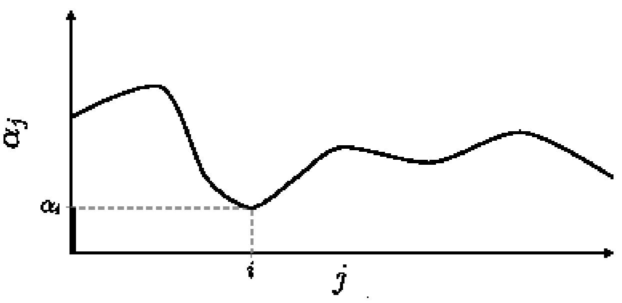

Figure 4: Example of a path $\pi = \langle c_1, \ldots, c_\ell \rangle$, where $j$ indexes the $j$-th configuration of $\pi$, and $\alpha_j$ is the stack of $c_j$. The configuration $c_i$ is a local minimum of $\pi$. The suffix $\pi_i$ of $\pi$ starting at the $i$-th configuration is a non-decreasing path, as the top symbol of $\alpha_i$ is never popped.

with weight at least $j$, with either smaller additional stack height than $\pi$, or of shorter length, contradicting that $\pi$ is the shortest length minimal additional stack height path with weight at least $j$. Hence we must have $w(\pi_1) < w(\pi)$, and hence the pumpable pair has positive weight. Now for an arbitrary path $\pi$ in $\Pi_{\min}$ we obtain that it has a pumpable pair. Either the pumpable pair has positive weight and we are done, else removing the pumpable pair we obtain a shorter length path of the same stack height, and the result follows by inductive hypothesis on the length of paths. □

**Example 5.** *We illustrate Lemma 2 on our running example. Consider again the program from Figure 1 and Figure 2 and the corresponding ICFG. Additionally, consider a weight function that assigns -1 to the transition that $foo$ calls recursively itself, +2 to the transitions to the exit node of $foo$ (i.e., when $foo$ returns), and 0 to every other transition. Examine two configurations $c_1 = (\epsilon, f : if(x > 1))$, $c_2 = (\epsilon, f : Exit)$, and note that for $j = 4$, the minimal additional stack height $d$ of all paths from $c_1$ to $c_2$ with total weight at least $j$ is $d = 4$, as they all have to make at least 4 recursive calls to $foo$ to witness a weight of at least 4 (in particular, $i$ invocations and returns to and from $foo$ contribute a weight of $i \cdot (-1+2) = i$). Since there are only 2 calls in total, Lemma 2 identifies a pumpable pair of paths f:Entry $\rightarrow$ if(x>1) $\rightarrow$ f:call foo and f:foo ret $\rightarrow$ x++ $\rightarrow$ f:Exit with positive total weight. Indeed, the weight is $-1 + 2 > 0$, and the pair of paths is pumbable due to recursion, as pointed in Example 4. Observe that this conclusion cannot be made using Lemma 2 with e.g. $j = 1$, as in that case there is a run from $c_1$ to $c_2$ that witnesses weight at least $j$ and has additional stack height only 1 (i.e., the run that only calls $foo$ recursively once), which is less than the number of calls in the program.*

Our algorithm for the mean-payoff analysis problem is based on detecting the existence of certain *non-decreasing* paths with positive weight. The maximal weights of such non-decreasing paths between node pairs are captured with the notion of a *summary function* and *bounded summary functions* (with bounded additional stack height). We now define them, and establish the lemma related to the number of bounded summary functions to be computed.

*Local minimum and non-decreasing paths.* A configuration $c_i$ in a path $\pi = \langle c_1, \ldots, c_\ell \rangle$ is a *local minimum* if the stack height of $c_i$ is minimal in $\pi$, i.e., $|\alpha_i| = \min(|\alpha_1|, \ldots, |\alpha_\ell|)$. A path from configuration $(\alpha, n_1)$ to $(\alpha\beta, n_2)$ is a *non-decreasing $\alpha$-path* if $(\alpha, n_1)$ is a local minimum. Note that if a sequence of transitions is a non-decreasing $\alpha$-path for some $\alpha \in Retns^*$, then the same sequence of transitions is a non-decreasing $\gamma$-path for every $\gamma \in Retns^*$. Hence, we say that $\pi$ is a non-decreasing path if there exists $\alpha \in Retns^*$ such that $\pi$ is a non-decreasing $\alpha$-path. Figure 4 illustrates the concepts of local minimum and non-decreasing paths.

*Summary function.* Given the *QICFG* $\mathcal{A}$ and $\alpha \in Retns^*$, we define a summary function $s_\alpha : \bigcup_{1 \leq \ell \leq m}(N_\ell \times N_\ell) \to \{-\infty\} \cup \mathbb{Z} \cup \{\omega\}$ as:

1. $s_\alpha(n_1, n_2) = z \in \mathbb{Z}$ iff the weight of the maximum weight non-decreasing path from configuration $(\alpha, n_1)$ to configuration $(\alpha, n_2)$ is $z$.

2. $s_\alpha(n_1, n_2) = \omega$ iff for all $j \in \mathbb{N}$ there exists a non-decreasing path from $(\alpha, n_1)$ to $(\alpha, n_2)$ with weight at least $j$.
3. $s_\alpha(n_1, n_2) = -\infty$ iff there is no non-decreasing path from $(\alpha, n_1)$ to $(\alpha, n_2)$.

We note that for every $\alpha, \beta \in \mathit{Retns}^*$ it holds that $s_\alpha \equiv s_\beta$. Hence, we consider only $s \equiv s_\epsilon$ (where $\epsilon$ is the empty string and corresponds to empty stack). The computation of the summary function is done by considering stack height bounded summary functions defined below.

*Stack height bounded summary function.* For every $d \in \mathbb{N}$, the *stack height bounded summary function* $s_d : \bigcup_{1 \le \ell \le m}(N_\ell \times N_\ell) \to \{-\infty\} \cup \mathbb{Z} \cup \{\omega\}$ is defined as follows: (i) $s_d(n_1, n_2) = z \in \mathbb{Z}$ iff the weight of the maximum weight non-decreasing path from $(\epsilon, n_1)$ to $(\epsilon, n_2)$ with additional stack height at most $d$ is $z$; (ii) $s_d(n_1, n_2) = \omega$ iff for all $j \in \mathbb{N}$ there exists a non-decreasing path from $(\epsilon, n_1)$ to $(\epsilon, n_2)$ with weight at least $j$ and additional stack height at most $d$; and (iii) $s_d(n_1, n_2) = -\infty$ iff there is no non-decreasing path with additional stack height at most $d$ from $(\epsilon, n_1)$ to $(\epsilon, n_2)$.

*Facts of summary functions.* We have the following facts: (i) for every $d \in \mathbb{N}$, we have $s_{d+1} \ge s_d$ (monotonicity); and (ii) $s_{d+1}$ is computable from $s_d$ and the *QICFG*. By the above facts we get that if $s_d \equiv s_{d+1}$, i.e., if a fix point is reached, then $s \equiv s_d$. For interprocedural analysis with finite-height lattices, fix points are guaranteed to exist. Unfortunately in our case, the image of $s_i$ is infinite and moreover, it is an infinite-height lattice. Thus a fix point is not guaranteed. The next lemma shows that we can compute all the non-$\omega$ values of $s$ with the bounded summary function.

**Lemma 3.** *Let $d = |\mathit{Calls}| + 1$. For all $n_1, n_2 \in N$, if $s(n_1, n_2) \in \mathbb{Z} \cup \{-\infty\}$, then $s(n_1, n_2) = s_d(n_1, n_2)$.*

*Proof.* Obviously $s(n_1, n_2) \ge s_d(n_1, n_2)$. If $s_d(n_1, n_2) < s(n_1, n_2)$ it follows that there exists a non-decreasing path $\pi$ from $n_1$ to $n_2$ with $w(\pi) > s_d(n_1, n_2)$. By the definition of the bounded height summary function it follows that $\mathsf{ASH}(\pi) > d$, and w.l.o.g we assume that $\pi$ has the minimal additional stack height among all non-decreasing paths from $n_1$ to $n_2$ with weight $w(\pi)$. Then by Lemma 2 it follows that $\pi$ has a pumpable pair $(p_1, p_2)$ with $w(p_1) + w(p_2) = w_p > 0$. Hence, for every $j \ge 0$ the path $\pi^j$ that is obtained from $\pi$ by pumping the pair $(p_1, p_2)$ exactly $j$ times has weight $w(\pi^j) = w(\pi) + (j-1) \cdot w_p$, and it is a valid non-decreasing path from $n_1$ to $n_2$. Hence, for every $\ell \in \mathbb{N}$ the path $\pi^j$ for $j = \lceil \frac{\ell - w(\pi)}{w_p} + 1 \rceil$ satisfies $w(\pi^j) \ge \ell$ (if $\ell \le w(\pi)$, then we set $j = 1$). By definition we get that $s(n_1, n_2) = \omega$, and this completes the proof. $\square$

By Lemma 3 we get that if $s_{d+1}(n_1, n_2) > s_d(n_1, n_2)$ (for $d = |\mathit{Calls}| + 1$), then $s(n_1, n_2) = \omega$. Hence, the summary function $s$ is obtained by the fix point of the following computation: (i) Compute $s_{i+1}$ from $s_i$ up to $s_d$ for $d = |\mathit{Calls}| + 1$; (ii) for $i \ge |\mathit{Calls}| + 1$, if $s_{i+1}(n_1, n_2) > s_i(n_1, n_2)$, then set $s_{i+1}(n_1, n_2) = \omega$; (iii) a fix point is reached after at most $O(|\mathit{Calls}|)$ iterations (say $j$ iterations), and then we set $s \equiv s_j$. This establishes that we require only $O(|\mathit{Calls}|)$ iterations as compared to $O(|N|^2 \cdot |\mathit{Retns}|^2)$ iterations. The number of returns and calls are the same and thus we significantly improve the number of iterations required from the quartic worst-case bound to linear bound. We now describe the computation of every iteration to obtain $s_{i+1}$ from $s_i$.

*Computation of $s_{i+1}$ from $s_i$.* We first compute a partial function, namely, $s'_{i+1} : \mathit{En} \times \mathit{Ex} \to \{-\infty, \omega\} \cup \mathbb{Z}$ that satisfies $s'_{i+1}(n_1, n_2) = s_{i+1}(n_1, n_2)$ for every $n_1 \in \mathit{En}$ and $n_2 \in \mathit{Ex}$. We initialize $s'_0(n_1, n_2) = s_0(n_1, n_2)$. For every module $A_\ell$ we construct a finite graph $G^i_\ell$ by taking all the nodes and transitions of $A_\ell$ and by adding a transition between every call node and its corresponding return node. For every transition between a pair of nodes $n_1, n_2 \in N_\ell \setminus (\mathit{Calls}_\ell \cup \mathit{Retns}_\ell)$ we assign the weight according to the original weight in $\mathcal{A}$. For every transition between a call node that invokes module $A_p$ and a corresponding return node we assign the weight $s'_i(\mathit{En}_p, \mathit{Ex}_p)$. To compute $s'_{i+1}$ for module $A_\ell$ we run one Bellman-Ford iteration over $G^i_\ell$ for source node $\mathit{En}_\ell$ and target node $\mathit{Ex}_\ell$. We observe the next two key properties of $s'_i$:

- For every iteration $i$, a module $A_\ell$, and pair of nodes $n_1, n_2 \in N_\ell$ we have that the weight of the maximum weight path between $n_1$ and $n_2$ in $G^i_\ell$ is exactly $s_{i+1}(n_1, n_2)$ (the proof is by a simple induction over $i$).
- If $s'_{i+1} \equiv s'_i$, then $s_{i+1} \equiv s_i$ (follows from the first key property).

Hence, to compute $s$ we compute $s'_{i+1}$ from $s'_i$ until we get $s'_{i+1} \equiv s'_i$, and then we compute all pairs maximum weight paths (e.g., by the Floyd-Warshall algorithm) over every $G^i_\ell$ and get $s_{i+1}$ (and $s_{i+1} \equiv s$). The Floyd-Warshall algorithm has a cubic time complexity and quadratic space complexity [33]. Therefore, the time complexity for computing every iteration of $s_i$ is $O(\sum |N_\ell|^2)$ and the complexity of the last step is $O(\sum |N_\ell|^3)$. The space complexity of the last step is $O(\max\{|N_1|, \dots, |N_m|\}^2)$, but to store $s_i$ we require $O(\sum |N_\ell|^2)$ space.

*Summary graph.* Given *QICFG* $\mathcal{A}$ with a summary function $s$, we construct the *summary graph* $\mathsf{Gr}(\mathcal{A}) = (V, E)$ of $\mathcal{A}$ with a weight function $w : E \to \mathbb{Z} \cup \{\omega\}$ as follows: (i) $V = N \setminus (\mathit{Ex} \cup \mathit{Retns})$; and (ii) $E = E_{internal} \cup E_{calls}$ where $E_{internal} = \{(n_1, n_2) \mid n_1, n_2 \in N_\ell$ for some $\ell$, and $s(n_1, n_2) > -\infty\}$ contains the transitions in the same module and $E_{calls} = \{(n_1, n_2) \mid n_1 \in \mathit{Calls}$ and $n_2 \in \mathit{En}$ and $n_1$ is a call to a module with entry node $n_2\}$ contains the call transitions. The weights of $E_{internal}$ are according to the summary function $s$ and the weights of $E_{calls}$ are according to the weights of these transitions in $\mathcal{A}$ (i.e., according to $w$). A simple cycle in $\mathsf{Gr}(A)$ is a *positive simple cycle* iff one of the following conditions hold: (i) the cycle contains an $\omega$ edge; or (ii) the sum of the weights of the cycles according to the weights of the summary graph is positive. Lemma 4 shows the equivalence of the mean-payoff analysis problem and positive cycles in the summary graph.

**Lemma 4.** *A QICFG $\mathcal{A}$ has a path $\pi$ with $\mathit{LimAvg}(w(\pi)) > 0$ iff the summary graph $\mathsf{Gr}(\mathcal{A})$ has a (reachable) positive cycle.*

*Proof.* If $\mathsf{Gr}(\mathcal{A})$ does not contain a positive cycle, then it follows that the weight of every non-decreasing path in $\mathcal{A}$ is bounded by the weight of the maximum weight path in $\mathsf{Gr}(\mathcal{A})$. Hence, for every infinite path $\pi$ we get that every prefix of $\pi$ is a non-decreasing path from the initial configuration with bounded weight (sum of weights bounded from above), and therefore $\mathit{LimAvg}(w(\pi)) \le 0$. Conversely, if $\mathsf{Gr}(\mathcal{A})$ has a positive cycle, then it follows that there is a path $\pi_0\pi_1$ in $\mathsf{Gr}(\mathcal{A})$ such that $\pi_0$ and $\pi_1$ are non-decreasing paths, $\pi_1$ begins and ends in the same node (possibly at higher stack height) and $w(\pi_1) > 0$. Hence, the path $\pi_0\pi_1^\omega$ is a valid path and satisfies $\mathit{LimAvg}(w(\pi_0\pi_1^\omega)) = \frac{w(\pi_1)}{|\pi_1|} > 0$, where $\pi_1^\omega = \pi_1 \cdot \pi_1 \cdot \pi_1 \dots$ is the infinite concatenation of the finite path $\pi_1$. The desired result follows. □

*Algorithm and analysis.* Algorithm 1 solves the mean-payoff analysis problem for *QICFG*s. The computation of the summary function requires $O(|\mathit{Calls}|)$ computations of the partial summary function $s'_i$, which requires $m$ runs of Bellman-Ford algorithm, each run over a graph of size $|N_\ell|$ (hence, each run takes $O(|N_\ell|^2)$ time). In addition the computation requires $m$ runs of all pairs maximum weight path (Floyd-Warshall) algorithm. Each run is over a graph of size $O(|N_\ell|)$ (hence, each run takes $O(|N_\ell|^3)$ time and $O(|N_\ell|^2)$ space). Finally we detect positive cycles by running Bellman-Ford algorithm once over the summary graph, which takes $O(|N|^2)$ time and $O(|N|)$ space. Thus we obtain the following result.

**Theorem 3.** (Improved algorithm). *Algorithm 1 solves the mean-payoff analysis problem for QICFGs in $O\left((|\mathit{Calls}| \cdot (\sum |N_\ell|^2)) + (\sum |N_\ell|^3) + |N|^2\right)$ time and $O(\sum |N_\ell|^2)$ space.*

**Remark 4.** *Note that in the worst case the running time of Algorithm 1 is cubic and the space requirement is quadratic.*

The next example is an illustration of a run of Algorithm 1.

**Example 6.** *Consider the QICFG that consists of modules $f$ and $g$ (Figures 5 and 6) and the entry of $f$ is the initial entry of the program. We now describe the run of Algorithm 1 over the QICFG. For simplicity, we denote the graph of $f$ by $F$ and the graph of $g$ by $G$ (and not by $G_1$ and $G_2$). Note that the number of call nodes is* 3.

*We first compute the summary function $s'$ and the first step is to compute $s'_0$. We have $s'_0$(f:entry,f:exit) $= -35$, and $s'_0$(g:entry,g:exit) $= -25$.*

*In order to compute $s'_1$(f:entry,f:exit) we construct a graph $F^0$ from $F$ by adding a transition from the node f:call g to the node f:ret g with weight $s'_0$(g:entry,g:exit) and find the maximum weight path from f:entry to f:exit in $F^0$. We get $s'_1$(f:entry,f:exit) $= -35$. In order to compute $s'_1$(g:entry,g:exit) we construct a graph*

**Algorithm 1** Mean-payoff *QICFG* Analysis

```
for $\ell \leftarrow 1$ to $m$ do
    $s'_0(En_\ell, Ex_\ell) \leftarrow$ Bellman-Ford$(A_\ell)$ {Compute $s'_0$ by running Bellman-Ford algorithm on $A_\ell$}
end for
$i \leftarrow 1$
loop
    for $\ell \leftarrow 1$ to $m$ do
        Construct $G_\ell^{i-1}$ according to $s'_{i-1}$
        $s'_i(En_\ell, Ex_\ell) \leftarrow$ Bellman-Ford$(G_\ell^{i-1})$ {Compute $s'_i$ by running Bellman-Ford algorithm over $G_\ell^{i-1}$}
    end for
    if $s'_i \equiv s'_{i-1}$ then
        break
    end if
    if $i > |Calls| + 1$ then
        for $\ell \leftarrow 1$ to $m$ do
            if $s'_i(En_\ell, Ex_\ell) > s'_{i-1}(En_\ell, Ex_\ell)$ then
                $s'_i(En_\ell, Ex_\ell) = \omega$
            end if
        end for
    end if
    $i \leftarrow i + 1$
end loop
$s \leftarrow$Floyd-Warshall$(s'_i)$
Construct $\mathsf{Gr}(\mathcal{A})$ from $s$
Bellman-Ford$(\mathsf{Gr}(\mathcal{A}))$ {Run Bellman-Ford over $\mathsf{Gr}(\mathcal{A})$}
if $\mathsf{Gr}(\mathcal{A})$ has a positive cycle then
    return YES
else
    return NO
end if
```

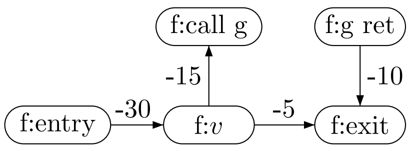

Figure 5: Module f

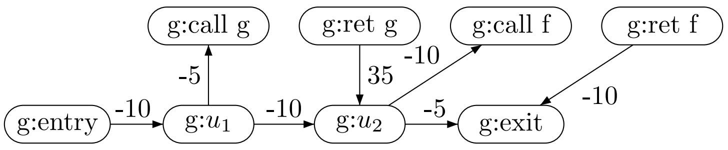

Figure 6: Module g

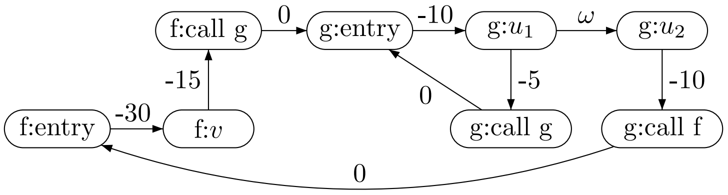

Figure 7: Summary graph of f and g

*$G^0$ from $G$ by adding a transition from g:call g to g:ret g with weight $s'_0$(g:entry,g:exit) and a transition from g:call f to g:ret f with weight $s'_0$(f:entry,f:exit) and find the maximum weight path from g:entry to g:exit in $G^0$. We get $s'_1$(g:entry,g:exit) $= -10$.*

*Since $s'_1 \neq s'_0$, we continue to compute $s'_2$. We construct $F^1$ and $G^1$ in the same manner as we constructed $F^0$ and $G^0$ (but take the values of $s'_1$ instead of $s'_0$) and get $s'_2$(f:entry,f:exit) $= -35$, $s'_2$(g:entry,g:exit) $= 5$. For $i = 3$ we get $s'_3$(f:entry,f:exit) $= -35$, $s'_3$(g:entry,g:exit) $= 20$. For $i = 4$, $s'_4$(f:entry,f:exit) $= -35$, $s'_4$(g:entry,g:exit) $= 35$.*

*For $i = 5$ we get $s'_5$(f:entry,f:exit) $= -20$ and $s'_5$(g:entry,g:exit) $= 50$. Since $i > |Calls| + 1$ and $s'_5$(f:entry,f:exit) $>$ $s'_4$(f:entry,f:exit) and $s'_5$(g:entry,g:exit) $>$ $s'_4$(g:entry,g:exit) we assign assign $s'_5$(f:entry,f:exit) $= \omega$ and $s'_5$(g:entry,g:exit) $= \omega$. In the sixth iteration we get a fix point (that is, $s'_6 \equiv s'_5$) and exit the loop block.*

*From $F^5$ and $G^5$ we compute the summary function $s$. For example $s$(g:entry,g:$u_1$) $= \omega$ and $s$(f:entry,f:v) $= -30$. Finally, we construct the summary graph (see Figure 7) and check whether a positive cycle exists. The cycle f:entry$\to$f:v$\to$f:call g$\to$g:entry$\to$g:$u_1$$\to$g:$u_2$$\to$g:call f$\to$f:entry contains an $\omega$-edge and thus, it is a positive cycle. Hence algorithm returns YES.*

## 4.2 Efficient algorithm for mean-payoff analysis

In this section we further improve the algorithm for the mean-payoff analysis problem for *QICFG*s, and the improvement depends on the fact that typically the number of entry, exit, call, and returns nodes is much smaller than the size of the *QICFG*s. Formally, in most typical cases we have $|Ex \cup Retns \cup Calls \cup En| << |N|$. Let $X_\ell = \{Ex_\ell, En_\ell\} \cup Retns_\ell \cup Calls_\ell$ and $X = \bigcup_\ell X_\ell$. We present an improvement that enables us to construct the summary function over graphs of size $O(|X_\ell|)$ (instead of graphs of size $O(|N_\ell|)$ of Section 4.1), and with at most $O(|Calls|)$ iterations. Hence, the algorithm in most typical cases will be much faster and require much smaller space.

**Compact representation.** The key idea for the improvement is to represent the modules in *compact form.* The compact form of a module $A_\ell$, denoted by $\mathsf{Comp}(A_\ell)$, consists of the entry, exit, call, and returns node of $A_\ell$. There is transition between every node in $\mathsf{Comp}(A_\ell)$, and the weight of each transition is the maximum weight path between the nodes with additional stack height 0 (and if there is no such path, then the weight is $-\infty$). Formally, $\mathsf{Comp}(A_\ell) = (V, E)$; where $V = X_\ell$; $E = V \times V$, and $w(v_1, v_2) = s_0(v_1, v_2)$ (where $s_0$ is the bounded height summary function of height 0). If in $\mathsf{Comp}(A_\ell)$ there is a cycle with positive weight that is reachable from the entry node, then we say that $A_\ell$ is a *positive mean-payoff witness.* The computation of the compact form for a module $A_\ell$ requires $O(|X_\ell| \cdot |N_\ell|^2)$ time and $O(|N_\ell|)$ space (running Bellman-Ford on each $A_\ell$), and thus the compact form for all modules can be computed in $O(\sum |X_\ell| \cdot |N_\ell|^2)$ time and $O(\max |N_\ell|)$ space (note that the space can be reused).

*Witness in summary graph of compact forms.* After constructing the compact forms, we compute a summary function for $\mathsf{Comp}(A_1), \ldots, \mathsf{Comp}(A_m)$, and a corresponding summary graph. We say that there is a path with positive mean-payoff iff there exists a positive cycle in the summary graph or there exists a path to the entry node of a positive mean-payoff witness. The correctness of the algorithm relies on the next lemma.

**Lemma 5.** *Let $\mathcal{A} = \langle A_1, \ldots, A_m \rangle$ be a QICFG, let $\mathsf{Gr}(\mathcal{A})$ be its summary graph and let $\mathsf{Comp}(\mathsf{Gr}(\mathcal{A}))$ be the summary graph that is formed by $\mathsf{Comp}(A_1), \ldots, \mathsf{Comp}(A_m)$. The following assertions are equivalent:*

1. *$\mathsf{Gr}(\mathcal{A})$ has a (reachable) positive cycle.*
2. *$\mathsf{Comp}(\mathsf{Gr}(\mathcal{A}))$ has a (reachable) positive cycle or a positive mean-payoff witness.*

*Proof. (of Lemma 5).* We first observe that every node in $\mathsf{Comp}(\mathsf{Gr}(\mathcal{A}))$ exists also in $\mathsf{Gr}(\mathcal{A})$ and that the weight of every path in $\mathsf{Comp}(\mathsf{Gr}(\mathcal{A}))$ has the same weight for the corresponding path in $\mathsf{Gr}(\mathcal{A})$ (this can be easily shown by induction over the number of iterations that are needed to obtain a fix point in the bounded height summary function). Hence, if $\mathsf{Gr}(\mathcal{A})$ has a positive simple cycle that contains a call node $c$ for $c \in \mathit{Calls}$, then $c$ is a node also in $\mathsf{Comp}(\mathsf{Gr}(\mathcal{A}))$ and by the observation above, $c$ is part of a positive cycle in $\mathsf{Comp}(\mathsf{Gr}(\mathcal{A}))$. Therefore, $\mathsf{Comp}(\mathsf{Gr}(\mathcal{A}))$ has a positive cycle. Otherwise, $\mathsf{Gr}(\mathcal{A})$ has a positive simple cycle that does not contain a call node. Hence, there is a module $A_\ell$ with a reachable positive simple cycle that has additional stack height 0. Therefore $\mathsf{Comp}(\mathsf{Gr}(\mathcal{A}))$ has a positive cycle or a positive mean-payoff witness. This concludes the proof of one direction and the proof for the converse direction is trivial. □

The above lemma establishes the correctness of the computation on compact form graphs, and gives us the following result. The following result is obtained from Theorem 3 by replacing $|N_\ell|$ with $|X_\ell|$ and $|N|$ by $|X|$, and the additional $\sum |X_\ell| \cdot |N_\ell|^2$ time and $\max |N_\ell|$ space are required for the compact form computation.

**Theorem 4.** (Efficient algorithm). *The mean-payoff analysis problem for QICFGs can be solved in $O\left(\left(|\mathit{Calls}| \cdot (\sum |X_\ell|^2)\right) + (\sum |X_\ell|^3) + |X|^2 + \sum |X_\ell| \cdot |N_\ell|^2\right)$ time and $O(\sum |X_\ell|^2 + \max |N_\ell|)$ space, where $X_\ell = \{\mathit{Ex}_\ell, \mathit{En}_\ell\} \cup \mathit{Retns}_\ell \cup \mathit{Calls}_\ell$ and $X = \bigcup_\ell X_\ell$.*

## 4.3 Reduction: Ratio analysis to mean-payoff analysis

We now establish a linear reduction of the ratio analysis problem to the mean-payoff analysis problem. Given a *ICFG* $\mathcal{A}$ with labeling of good, bad, and neutral events, a positive integer weight function $w$, and rational threshold $\lambda > 0$, the reduction of the ratio analysis problem to the mean-payoff analysis problem is as follows. We consider a *QICFG* $\mathcal{A}'$ with weight function $w_\lambda$ for the mean-payoff objective defined as follows: for a transition $e$ we have

$$w_\lambda(e) = \begin{cases} w(e) & \text{if } e \text{ is labelled with } \textit{good} \\ -\lambda \cdot w(e) & \text{if } e \text{ is labelled with } \textit{bad} \\ 0 & \text{otherwise (if } e \text{ is labelled with } \textit{neutral}\text{)} \end{cases}$$

The next lemma establishes the correctness of the reduction.

**Lemma 6.** *Given a ICFG $\mathcal{A}$ with labeling of good, bad, and neutral events, a positive integer weight function $w$, and rational threshold $\lambda > 0$, let $\mathcal{A}'$ be the QICFG with weight function $w_\lambda$.*

*There exists a path $\pi$ in $\mathcal{A}$ with $LimRat(w(\pi)) > \lambda$ iff there exists a path $\pi$ in $\mathcal{A}'$ with $LimAvg(w_\lambda(\pi)) > 0$.*

*Proof. (of Lemma 6).* Observe that by the definition of $w_\lambda$ we have that for every $\epsilon > 0$ and a finite path $\pi$:

$$Rat(w(\pi)) \geq \lambda + \epsilon \text{ iff } Avg(w_\lambda(\pi)) \geq \epsilon.$$

*LimAvg implies LimRat.* Consider an infinite path $\pi$. If $LimAvg(w_\lambda(\pi)) > 0$, then by definition there is an $\epsilon > 0$ and $n_0 \in \mathbb{N}$ such that for every $n \geq n_0$ we have $Avg(w_\lambda(\pi[1,n])) \geq \epsilon$. Hence by the above observation there exist $\epsilon > 0$ and $n_0 \in \mathbb{N}$ such that for every $n \geq n_0$ we have $Rat(w(\pi[1,n])) \geq \lambda + \epsilon$. Moreover, it follows that in $\pi$ there are infinitely many edges with positive weights (according to $w_\lambda$) and hence $\pi$ has infinitely many good events. Hence, we get that $LimRat(w(\pi)) > \lambda$.

*LimRat implies LimAvg.* The proof for the converse direction is less trivial and relies on properties of *QICFG*s that we established. Suppose that there is an infinite path $\pi$ with $LimRat(w(\pi)) > \lambda$. We note that every infinite path has infinitely many local minima and let $c_1, c_2, \ldots$ be an infinite sequence of local minima in $\pi$. We have the following facts:

1. The segment between every two such configurations $c_i$ and $c_j$ for $i < j$ is a non-decreasing path (since each $c_i$ is a local minimum).
2. There is a configuration $c_p$ with a node $n_p$ such that for every $\ell \in \mathbb{N}$ there exists a configuration $c_j$ (for $p < j$) such that the segment between $c_p$ and $c_j$ is of length greater than $\ell$ and $n_p = n_j$, i.e., $c_p$ and $c_j$ have the same node (follows from the pigeonhole principle, since the number of local minima is infinite and we have finitely many nodes).

We claim that there exists a non-decreasing finite path $\pi^*$ that is a segment of $\pi$, which begins at $c_p$ and ends at a configuration that has the same node (possibly at different stack height), and we have $Rat(w(\pi^*)) > \lambda$. Assume towards the contradiction of the claim that for every configuration $c_j$ with $n_p = n_j$, with $p < j$, we have $Rat(w(\pi^*)) \leq \lambda$. If $\pi$ has only finitely many good or bad events, then $LimRat(w(\pi)) = 0 < \lambda$. Else we consider the following sequence of prefixes of $\pi$: $\pi_0$ is the prefix of $\pi$ that ends in $c_p$; and $\pi_i$ is the segment that starts in $c_p$ and ends in the $i$-th local minimum after $c_p$ that has the same node $n_p$. Then we have

$$Rat(\pi_0 \cdot \pi_i) \leq \lambda + \frac{w(\pi_0)}{i};$$

since the length of $\pi_i$ is at least $i$. Hence, by definition $LimRat(w(\pi)) \leq \lambda$, which establishes the desired contradiction. Thus we have that $Rat(w(\pi^*)) > \lambda$ and therefore $Avg(w_\lambda(\pi^*)) > 0$ and the path $\pi' = \pi_0(\pi^*)^\omega$ is a valid path (since $\pi^*$ is a non-decreasing path that begins and ends in the same node) with $Avg(w_\lambda(\pi')) > 0$. The desired result follows. □

**Remark 5.** *Note that in our reduction from ratio analysis to mean-payoff analysis we do not change the QICFG, but only change the weight function. Thus our algorithms from Theorem 3 and Theorem 4 can also solve the ratio analysis problem for QICFGs. Moreover, our proof of lemma 6 shows that for all paths $\pi$, if we have $LimAvg(w_\lambda(\pi)) > 0$, then we also have $LimRat(w(\pi)) > \lambda$, i.e., any witness for the mean-payoff analysis is also a witness for the ratio analysis.*

## 5 Experimental Results: Three Case Studies

In this section we present our experimental results on two case studies described in Section 3.1 and Section 3.2. We run our case studies on several benchmarks in Java, including DaCapo 2009 benchmarks [27], and we use [34, 35] to assist Soot for the construction of the control-flow graphs. First we present some optimizations that proved useful for speed-up in the benchmarks.

## 5.1 Optimization for case studies

We present four optimizations for the case studies: the first two are general, and the last two are specific to our case studies.

**Faster computation of stack height bounded summary function.** We note that if module $A_\ell$ invokes only modules $A_{j_1}, \dots, A_{j_k}$, and $s'_i(En_{j_h}, Ex_{j_h}) = s'_{i-1}(En_{j_h}, Ex_{j_h})$ for all $h \in \{1, \dots, k\}$, then $s'_i(En_\ell, Ex_\ell) = s'_{i-1}(En_\ell, Ex_\ell)$. Hence, when computing $s'_i$, we maintain a set $\mathcal{L}^i = \{\ell \mid s'_i(En_\ell, Ex_\ell) > s'_{i-1}(En_\ell, Ex_\ell)\}$, and in the next iteration we run Bellman-Ford algorithm only for the modules that invoke modules from $\mathcal{L}^i$.

**Reducing the number of iterations for fix point.** We now present an optimization that allows us to reduce the number of bounded height summary functions from $O(|\mathit{Calls}|)$ to a practically constant number. We note that the $O(|\mathit{Calls}|)$ theoretical bound is tight. However, only pathological cases can reach even a fraction of this bound. We note that in typical programs the average nesting of function calls is practically constant (say 10). So if we do not get a fix point after 10 iterations (i.e., $s'_{11} > s'_{10}$), then it is probably because there is a recursive call with positive weight. If this is the case, then if we build the summary graph according to $s_{11}$, we will get a positive cycle in the summary graph, that is, we will get a witness for a path with a positive mean-payoff, and we can stop the computation (since by definition $s \geq s_{11}$, we get that this witness is valid). Hence, our optimized algorithm is to compute the bounded height summary function $s'_i$ and if $s'_i > s'_{i-1}$ and $i = 10, 20, 30, \dots$, then we construct the summary graph and look for a witness path. If a path is found, then we are done. Otherwise we continue and compute $s'_{i+1}$.

**Removing redundant modules.** Consider an *ICFG* $\mathcal{A} = \langle A_1, \dots, A_m \rangle$ in which every node is reachable from the program entry (the entry node of the main method). We say that module $A_i$ is *non-redundant* if (i) the module has non-zero weight transitions (good or bad events); or (ii) it invokes a non-redundant module, and is called *redundant* otherwise. Let $A_i$ be a redundant module. For every path $\pi$ that contains a transition to $En_i$ (an invocation of $A_i$), the segment of $\pi$ between that transition and the first transition to $Ex_i$ contains only neutral transitions. Because all nodes of $\mathcal{A}$ are reachable, we can safely replace each call node that invokes $A_i$ by an internal node that leads to the corresponding return node, and label it as a neutral event. Our optimization then consists of removing redundant modules, as follows:

1. First, we perform a single-source interprocedural reachability from the program entry, which requires linear time ([2]), and discard all non-reachable nodes in all modules.
2. Then, we perform a backwards reachability computation on the call graph of $\mathcal{A}$, starting from the set of all modules that contain non-zero weight transitions. All detected redundant modules are discarded, and calls to them are replaced according to the above description.

Hence, when computing the bounded height summary function, the size of the graph is smaller and the Bellman-Ford algorithm takes less time. Additionally, the number of calls $|\mathit{Calls}|$ decreases, which reduces the number of iterations required in the main loop of Algorithm 1. In the first case study, typically more than half of the methods are eliminated in this process.

**Incremental computation of summary functions.** We present the final optimization which is relevant for our second and third case studies. Let $\mathcal{A}^1$ be a *QICFG* and let $\mathcal{A}^2$ be a *QICFG* that is obtained from $\mathcal{A}^1$ only by increasing some of the transitions weights. Let $s^1$ be the summary function of $\mathcal{A}^1$. Then we can compute the summary function of $\mathcal{A}^2$ by setting $s'^2_0 \equiv s^1$ and by computing $s'^2_i$ from $s'^2_{i+1}$ in the usual way. The correctness is almost trivial. Since the weights of $\mathcal{A}^2$ are at least as the weights of $\mathcal{A}^1$, we get that if we conceptually add a transition $(n_1, n_2)$ with weight $s^1(n_1, n_2)$ for every two nodes (in the same module) in $\mathcal{A}^2$, then the weights of the paths with the maximal weight in $\mathcal{A}^2$ remain the same. By assigning $s'^2_0 = s^1$ we only add such conceptual transitions. Hence, the correctness follows. We now describe how this optimization speed up the analysis of the second case study. In the static profiling for function frequencies, we need to build a summary graph for every function $f$, and then run the mean-payoff analysis for every such graph. Given this optimization, we can first compute (only once) a summary graph for the case that all method invocations are bad events. We denote this *QICFG* by $\mathcal{A}^*$ and the corresponding summary function by $s^*$. Note that in $\mathcal{A}^*$ all weights are negative, and the mean-payoff analysis answer is trivially NO. But still the summary function computation, which computes the quantitative information about the maximum weight context-free paths, provides useful information and saves recomputation. To determine the frequency of $f$ we assign weights to $\mathcal{A}$ and get $\mathcal{A}^f$, and the difference between $\mathcal{A}^f$ and $\mathcal{A}^*$ is only in the weight that is assigned to the invocation of $f$. We then compute the summary function $s^f$ for $\mathcal{A}^f$ by first assigning $s_0'^f = s^*$. In practical cases, programs can have thousands of methods, but only small portion of them will have a path to $f$. So along with the previous optimizations we get that only few Bellman-Ford runs are required to compute $s^f$. Overall, the computation of $s^*$ is expensive, and may take several minutes for a large program, but it is done only once, and then the computation of each $s^f$ is much faster.

## 5.2 Container analysis

*Technical details about experimental results.* We discuss a few relevant details about our experiments and results.

- We use the points-to analysis tool of [36]. This tool provides interprocedural on demand analysis for a may-alias relationship of two variables. We say that a variable may point to an allocated container if it may-alias the container, and a variable must point to an allocated container if it may-alias only one allocated container.
- For the underutilized containers the threshold is 1, and for the analysis of overpopulated containers we set a threshold of 0.1 for our experimental results. That is, if the ratio between the number of added elements to the number of lookup operations is more than 10, then the container is overpopulated.

**Experimental results.** Our experimental results on the benchmarks are reported in Table 1. In the table, **# M** and **# CO** represent the number of methods and containers that are reachable from the main entry of the program, respectively; **# OP** and **# UC** represent the number of overpopulated and underutilized containers discovered by our tool, respectively; and **TA(s)** and **TQ(s)** represent the time required for alias analysis and the time required for the quantitative analysis of *QICFG*s (in seconds), respectively; and the entries of the respective columns represent the time for overpopulated/underutilized container analysis. We now highlight some interesting aspects of our experimental results. First, our approach for container analysis discovers containers that are overpopulated or underutilized, while maintaining soundness. Second, the cases that we identify reveal useful information for optimization, for example, in the first (batik-rasterizer) and the second (batik-svgpp) benchmarks we identify containers that always have a small bounded number of elements.

| **Benchmark** | **# M** | **# CO** | **# OP** | **# UC** | **TA(s)** | **TQ(s)** |
|---|---|---|---|---|---|---|
| batik-rasterizer | 21433 | 9 | 1 | 2 | 124/125 | 144/143 |
| batik-svgpp | 7859 | 3 | 0 | 3 | 20/20 | 14/13 |
| mrt | 9798 | 10 | 1 | 0 | 70/13 | 41/59 |
| java_cup | 8173 | 10 | 0 | 0 | 19/19 | 25/22 |
| xalan | 8729 | 6 | 3 | 2 | 5/5 | 41/43 |
| polyglot | 8068 | 8 | 2 | 2 | 0/0 | 17/17 |
| antlr | 8607 | 15 | 5 | 2 | 11/12 | 25/24 |
| jflex | 21852 | 43 | 3 | 6 | 2473/2614 | 178/210 |
| avrora | 13331 | 75 | 9 | 9 | 145/141 | 111/113 |
| muffin | 22503 | 50 | 3 | 5 | 2500/157 | 352/173 |
| bloat06 | 10675 | 211 | 32 | 14 | 399/250 | 2241/2165 |
| eclipse06 | 9335 | 74 | 8 | 4 | 37/22 | 222/164 |
| jython06 | 12210 | 66 | 9 | 5 | 154/68 | 13593/8376 |

Table 1: Experimental results for container usage analysis

**Key charasteristics and comparison with existing work.** Our formulation for the misused container analysis problem has several key characteristics that distinguish it from existing literature:

1. Our framework can handle recursion, which has been a challenging task in the past [24].

| Benchmark | # M | # I | T |
|---|---|---|---|
| antlr | 768 | 326 | 1.2 |
| bloat | 2576 | 676 | 30.8 |
| eclipse | 1056 | 215 | 2.3 |
| fop | 429 | 47 | 0.4 |
| luindex | 567 | 239 | 0.7 |
| lusearch | 842 | 237 | 2.5 |
| pmd | 2547 | 589 | 11.5 |

Table 2: Experimental results for frequency of functions

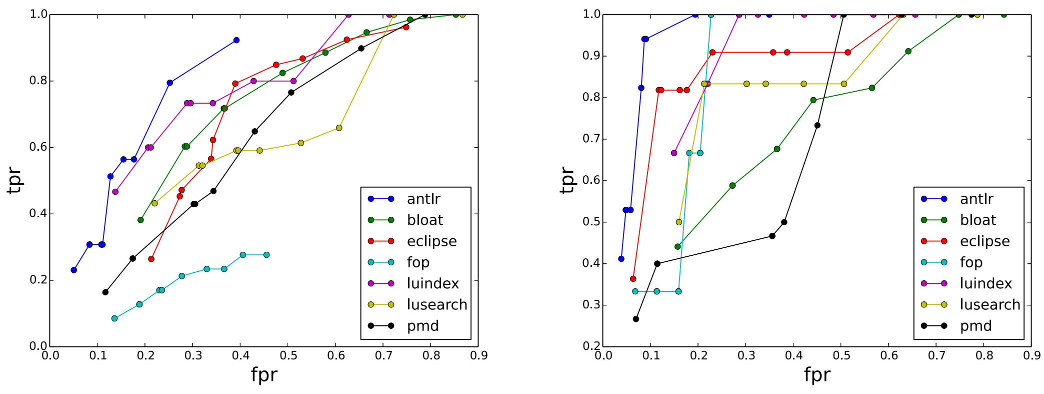

Figure 8: The ROC curves for the analysis of frequently invoked methods. The left plot shows the results when all methods are analyzed. The right plot shows the results when only the active methods are analyzed.

2. We present a sound and complete approach for ratio analysis of *QICFG*s. Since we also follow a conservative modeling, we result in a sound analysis approach for detecting containers misuse.
3. Our algorithm is polynomial time (once the points-to relation is computed)
4. Finally, our approach also allows us to handle DELETE operations, which have been difficult to handle in the past [24].

Example 3 illustrates the advantages of our approach. One drawback of is that it is conservative: while for underutilized containers analysis (ADD vs DELETE) our approach captures all cases reported in [24], our approach for overpopulated containers analysis is more conservative (to obtain soundness).

With our approach we were able to fully analyze all the containers in all benchmarks. Below we present example snippets of code from the benchmarks where our analysis gives different results from the analysis of the existing tool of [24] with which we have compared our results. The example in Figure 9 shows that handling DELETE operations leads to more refined analysis: in the example, if DELETE operations are not handled, then the misuse is not detected. The example in Figure 10 shows that the proper utilization of containers might depend on the recursive calls. Finally, the example in Figure 11 illustrates that the proper use of containers can be outside its allocation site, and thus detecting proper use requires a quantitative interprocedural framework, such as the one proposed in this work.

## 5.3 Static profiling: frequency of function calls

**Experimental results.** We examined ten thresholds, namely $1/30, 2/30, 3/30, ..., 10/30$, and for each threshold $\lambda$ we say that a method is *statically hot* if it is $\lambda$-hot (according to the definition in Subsection 3.2). We compared the results to dynamic profiling from the DaCapo benchmarks [27]. In the dynamic profiling we define the top 5% of the most frequently invoked functions as *dynamically hot*. For example, if

```
public void cleared() {
   ...
   if (list != null){
      ...
   }
   else{
      Object o = elementsById.remove(id);
      if (o != this) // oops not us!
         elementsById.put(id, o);
   }
}
```

```
public void run() {
   while(true) {
      ...
      if (...) {
         ...
         rc.cleared();
      }
      ...
   }
   ...
}
```

Figure 9: An example from benchmark batik. The method **run** invokes **cleared** in a loop, and in every invocation, one element of elementsById is removed and one element is added. Thus in this loop the total number of elements in elementsById is bounded.

```
void addCNAME(CNAMERecord cname) {
   if (backtrace == null)
      backtrace = new Vector();
   backtrace.insertElementAt(cname, 0);
}

public SetResponse findRecords(Name name, short type) {
   ...
   if (type != Type.CNAME && type != Type.ANY && rrset.getType() == Type.CNAME)
   {
      zr = findRecords(cname.getTarget(), type);
      zr.addCNAME(cname);
      ...
   }
   ...
   return zr;
}
```

Figure 10: An example from benchmark muffin. The method **findRecords** has a recursive call, and method **addCNAME** adds an element to vector backtrace. A path with recursion depth $n$ adds $n$ elements to backtrace. Hence, backtrace may have unbounded number of elements and it is not underutilized.

```java
Hashtable cgi(Request request)
{
   Hashtable attrs = new Hashtable(13);
   ...
   if (query != null)
   {
      StringTokenizer st = new StringTokenizer(decode(query), "&");
      while (st.hasMoreTokens())
      {
         ...
         attrs.put(key, value);
      }
   }
   ...
   return attrs;
}

public Reply recvReply(Request request)
{
   ...
   else if (request.getPath().equals("/admin/set"))
   {
      Hashtable attrs = cgi(request);
      ...
      for (int i = 0; i < enabled.size(); i++)
      {
         ...
         prefs.put(key, (String) attrs.get(key));
      }
      ...
   }
   ...
   return reply;
}
```

Figure 11: An example from benchmark muffin. The method **cgi** allocates the container attrs and potentially adds it many elements. The method **recvReply** performs a **get** operation over attrs in a loop. Since we analyze not only the operations that are nested in the allocation site, we detect that attrs is not overpopulated (the analysis of [24] reports it as overpopulated).

a program has 1000 functions, and in the benchmark 500 functions were invoked at least once, then the 25 most frequently invoked functions are dynamically hot. We note that theoretically speaking, the definitions of dynamic and static hotness are incomparable (basically for any $\lambda$), but our experimental results show a good correlation between the two notions. To illustrate the correlation we treat our static analysis as a *classifier* of hot methods, and the *specificity* and *sensitivity* of the classifier are controlled by the threshold $\lambda$. The sensitivity of a classifier is measured by the *true positive rate* (tpr), which is

$$\frac{\#\text{dynamically hot methods that are reported as statically hot}}{\#\text{dynamically hot methods}}$$

The specificity is determined by the *false positive rate* (fpr):

$$\frac{\#\text{non-dynamically hot methods that are reported as statically hot}}{\#\text{methods}}$$

For high values of $\lambda$, the classifier is expected to capture only dynamically hot methods (but it will miss most of the dynamically hot methods), and thus it will have very high fpr but very low tpr. For very low values of $\lambda$ the classifier will report most of the methods as hot, so most of the hot methods will be reported as hot, and we will have very high tpr but very low fpr. A fundamental metric for classifier evaluation is a *receiver operating characteristic (ROC) graph.* A ROC graph is a plot with the false positive rate on the X axis and the true positive rate on the Y axis. The point (0,1) is the perfect classifier, and the area beneath an ROC curve can be used as a measure of accuracy of the classifier.

In our experimental evaluation we only considered application functions (and not library functions), and the results are presented in Table 2. In the table, **# M** represents the number of application methods (that are reachable from the main entry of the program), **# I** represents the number of application methods that were actually invoked in the execution of the benchmark, **T** represents the average running time for the static analysis of a single method (i.e., to check whether a single method is $\lambda$-hot for a fixed $\lambda$) (in seconds). For each $\lambda$ we present the tpr and fpr values of the classifications. We present an evaluation for two cases. In the first case we statically analyze all the methods and calculate the tpr and fpr accordingly. In the second case we consider only the *active methods*, namely, the methods that were invoked at least once in the program, and we remove all the other methods from the program control flow graph. This simulates a typical case where the programmer has prior knowledge on methods that are definitely not hot and can instruct the static analysis to ignore them. The ROC curves are presented in Figure 8, where the most left points on the graph are for $\lambda = 10/30$ and the fpr and tpr increases as $\lambda$ decreases (until it finally reaches 1/30). In general, for most of the programs the static analysis gives useful and quite accurate information. Specifically, the threshold $\lambda = 7/30$ captures more than half of the hot methods for most benchmarks (i.e., except fop and antlr) and with a fpr less than 0.3 which means that if a method was statically reported as not hot, then with probability 0.7 it is really not hot. We note that the analysis over fop gives quite poor results because only 10% of the methods were active. However, when we analyzed only the active methods we get better results for fop, see the right hand graph in Figure 8. When we only consider the active methods, the threshold $\lambda = 9/30$ captures most of the dynamically hot methods while maintaining a fpr less than 0.1 (for most programs).

## 5.4 Static profiling: frequency of class method calls

We have experimented with the potential of our framework to statically predict collections of methods, where the set will be frequently invoked. As discussed in Section 3.3, when such sets of methods are software libraries, good predictions can assist static vs dynamic library linking. As our prototype implementation has focused on Java programs, where static linking does not apply (the JVM dynamically loads classes as they are referenced), we have grouped methods based on the class they belong to. Then we applied the modeling of Section 3.3, where a Java class is used as a substitute for a library.

We examined the thresholds $1/30, 2/30, 3/30, ..., 16/30$. Similarly to the case of hot methods, for every threshold $\lambda$ we say that a class is *statically hot* if the collection of its methods is $\lambda$-hot (according to

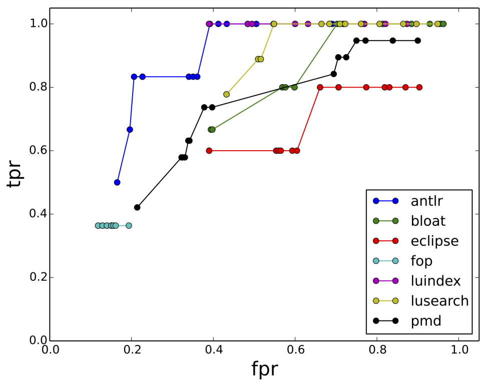

Figure 12: The ROC curves for the analysis of frequently invoked classes.

the definition in Section 3.3). Again, we compared the results to dynamic profiling from the DaCapo benchmarks [27]. In the dynamic profiling we define the top 5% of the most frequently invoked classes as *dynamically hot*. Table 3 presents a summary of the analysis, where **# C** represents the number of classes (that are reachable from the main entry of the program), **# I** represents the number of classes that were actually invoked in the execution of the benchmark, **T** represents the average running time for the static analysis of a single class (i.e., to check whether the collection of methods of a whole class is $\lambda$-hot for a fixed $\lambda$) (in seconds).

| **Benchmark** | **# C** | **# I** | **T** |
|---|---|---|---|
| antlr | 103 | 79 | 8.3 |
| bloat | 284 | 168 | 29.7 |
| eclipse | 187 | 69 | 4.2 |
| fop | 197 | 24 | 0.2 |
| luindex | 101 | 64 | 3.1 |
| lusearch | 164 | 69 | 9.0 |
| pmd | 379 | 130 | 10.5 |

Table 3: Experimental results for frequency of classes

Figure 12 shows the corresponding ROC curves. The leftmost point of each curve corresponds to $\lambda = 16/30$, and the fpr and tpr increase as we move to the right. We see that the static analysis provides useful information, as for most benchmarks it achieves relatively large tpr while keeping the tpr reasonably low. The static analysis performs poorly in the case of fop, for which we have seen in Section 5.3 that the overlap between methods discovered in the static and dynamic analysis is very small.

*Hot methods vs hot collections of methods.* We clarify some differences between the concepts of hot methods and hot collections of methods. First, we note that frequently invoked collections of methods cannot be detected simply by detecting frequently invoked methods, and a separate analysis is required. Figure 13 illustrates the difference on a small example. We have used a larger threshold range in our analysis of hot classes, and thus groups of functions are in general expected to meet larger thresholds than individual functions. In our experimental results, the information computed is not of direct use, as all classes are loaded on runtime by the JVM. However, it is demonstrated that our framework can statically analyze programs and provide meaningful suggestions as to which groups of methods (e.g., classes, libraries) will be frequently used.

*Remarks.* We run the experiments on a single thread Intel Pentium 3.80GHz. For Table 1 results, the alias analysis did not complete for some benchmarks (e.g. fop, pmd). In Table 2 we only show benchmarks for

```
class A:
{
    public void f1()
    {
        ...
    }
    public void f2()
    {
        ...
    }
    public void f3()
    {
        ...
    }
}

void main()
{
    A a = new A();
    while(1)
    {
        a.f1();
        a.f2();
        a.f3();
    }
}
```

Figure 13: Illustration of the difference between hot methods and hot collections of methods. Here, the collection of the methods of class $A$ is $\lambda$-hot for any threshold $\lambda \leq 1$. However, each of the methods $f1()$, $f2()$ and $f3()$ of $A$ are at most $1/3$-hot. Hence determining frequently invoked collections of methods requires separate analysis, and cannot rely on simply detecting frequently invoked methods.

which we managed to obtain dynamic profiles. For a few benchmarks (e.g. jython) the quantitative analysis took too long for the entire benchmark. In such cases, our tool could be used to focus on specific functions.

## 6 Related Work

**Interprocedural analysis.** Algorithms that operate on the interprocedural control-flow graphs provide the framework for static analysis of programs, and have numerous applications. Precise interprocedural analysis is crucial for dataflow analysis and has been studied in several works [2, 3]. The study of interprocedural analysis has also been extended to weighted pushdown systems, where the weight domain is a bounded idempotent semiring [4, 5]. Analysis of such weighted pushdown systems has been used in many applications of program analysis [5, 6, 7]. Our work is different because the objectives (mean-payoff and ratio analysis) we consider are very different from reachability and bounded domains. The mean-payoff objective is a function that assigns a real-valued number to every path. In contrast to bounded domain functions, the range of a mean-payoff function is potentially uncountable. We develop novel techniques to extend the summary graph approach for finite-height lattices to solve mean-payoff analysis of *QICFG*s (which requires computing fix points for infinite-height lattices).

**Mean-payoff analysis.** Mean-payoff objectives are quantitative metrics for performance modeling in many applications and very well-studied in the context of finite-state graphs and games. Finite-state graphs and games with mean-payoff objectives have been studied in [32, 13, 16, 18] for performance modeling, and robust synthesis of reactive systems [20, 11]. Quantitative abstraction-refinement frameworks for finite-state systems with mean-payoff objectives have also been studied in [15]. While the mean-payoff objectives have been considered in depth for finite-state systems, they have not been considered in depth for interprocedural analysis. Pushdown systems with mean-payoff objectives were considered in [25]. We significantly improve the complexity of the polynomial-time algorithm for interprocedural mean-payoff analysis that can be obtained by a reduction to the results of [25].

**Detecting inefficiently-used containers.** Bloat detection and detecting inefficiently used containers have been identified in many previous works as a major reason for program inefficiency. Dynamic approaches for the problem were studied in many works such as [37, 23, 38, 39, 40, 41, 42]. A static approach to analyze the problem was first considered in [24], which is the most closely related work to our case study. The work of [24] provides an excellent exposition of the problem with several practical motivations. It also describes the clear advantages of the static analysis tools, and identifies that soundness in detecting inefficiently used containers (with no or low false positive rates) is a very important feature. Our approach for the problem is significantly different from the approach of [24]. A big part of the contribution of [24] is an automated annotation for the functionality of the containers operation. The main algorithmic approach of [24] is to use CFL-reachability (context-free reachability) to identify nesting loop depths and then use this information for detecting misuse of containers. Our algorithmic approach is very different: we use a quantitative analysis approach, i.e., ratio analysis of *QICFG*s to model the problem.

**Static profiling of programs.** Static and dynamic profiling of programs is in the heart of program optimization. Static profiling are typically used in branch predictions where the goal is to assign probabilities to branches, and typically require some prior knowledge on the probability of inputs. Static profiling of programs for branch predictions has been considered in [43, 44, 45, 46]. Dynamic profiling has also been used in many applications related to performance optimizations, see [47] for a collection of dynamic profiling tools. Two main drawbacks of dynamic profiling are that they require inputs, and they cannot be used for compiler optimizations. We use static profiling to determine if a function is invoked frequently along some run of the program, and do not require any prior knowledge on inputs. The techniques used in [43, 44, 46] involves solving linear equations with sparse matrix solvers, whereas our solution method is different (by quantitative analysis of *QICFG*s).

## 7 Conclusion

In this work we considered the quantitative (ratio and mean-payoff) analysis for interprocedural programs. We demonstrated how interprocedural quantitative analysis can aid to automatically reason about properties of programs and potential program optimizations. We significantly improved the theoretical known upper-bound for the polynomial-time solution, and presented several practical optimizations that proved to be useful in real programs. We have implemented the algorithm in Java, and showed that it scales to DaCapo benchmarks of real-world programs. This shows that interprocedural quantitative analysis is feasible and useful. Some possible directions of future works are as follows: (1) extend our framework with multiple quantitative objectives and study their applications; and (2) extend [15] to have an abstraction-refinement framework for quantitative interprocedural analysis.

**Acknowledgments.** We thank Mooly Sagiv for pointing us to reference [24], Nimrod Zibil for discussions on C optimizations, and anonymous reviewers for their helpful comments.

## References

[1] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[2] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.

[3] Markus Müller-Olm and Helmut Seidl. Precise interprocedural analysis through linear algebra. In *POPL*, pages 330–341, 2004.

[4] Ahmed Bouajjani, Javier Esparza, and Tayssir Touili. A generic approach to the static analysis of concurrent programs with procedures. In *POPL*, 2003.

[5] Thomas W. Reps, Stefan Schwoon, Somesh Jha, and David Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. Comput. Program.*, 58(1-2):206–263, 2005.

[6] Akash Lal, Thomas W. Reps, and Gogul Balakrishnan. Extended weighted pushdown systems. In *CAV*, pages 434–448, 2005.

[7] Thomas W. Reps, Akash Lal, and Nicholas Kidd. Program analysis using weighted pushdown systems. In *FSTTCS*, pages 23–51, 2007.

[8] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.

[9] Vivek Tiwari, Sharad Malik, and Andrew Wolfe. Power analysis of embedded software: a first step towards software power minimization. *IEEE Trans. VLSI Syst.*, 2(4):437–445, 1994.

[10] Christian Ferdinand, Florian Martin, Reinhard Wilhelm, and Martin Alt. Cache behavior prediction by abstract interpretation. *Sci. Comput. Program.*, 99.

[11] R. Bloem, K. Chatterjee, T. A. Henzinger, and B. Jobstmann. Better quality in synthesis through quantitative objectives. In *CAV*, pages 140–156, 2009.

[12] K. Chatterjee, L. Doyen, and T. A. Henzinger. Quantitative languages. *ACM Trans. Comput. Log.*, 11(4), 2010.

[13] J. Filar and K. Vrieze. *Competitive Markov Decision Processes.* Springer-Verlag, 1997.

[14] Mark S. Boddy. Anytime problem solving using dynamic programming. In *AAAI*, pages 738–743, 1991.

[15] Pavol Cerný, Thomas A. Henzinger, and Arjun Radhakrishna. Quantitative abstraction refinement. In *POPL*, pages 115–128, 2013.

[16] A. Ehrenfeucht and J. Mycielski. Positional strategies for mean payoff games. *Int. Journal of Game Theory*, 8(2):109–113, 1979.

[17] H. Bjorklund, S. Sandberg, and S. Vorobyov. A combinatorial strongly subexponential strategy improvement algorithm for mean payoff games. In *MFCS'04*, pages 673–685, 2004.

[18] U. Zwick and M. Paterson. The complexity of mean payoff games on graphs. *Theoretical Computer Science*, 158:343–359, 1996.

[19] T. A. Liggett and S. A. Lippman. Stochastic games with perfect information and time average payoff. *Siam Review*, 11:604–607, 1969.

[20] R. Bloem, K. Greimel, T. A. Henzinger, and B. Jobstmann. Synthesizing robust systems. In *FMCAD*, pages 85–92, 2009.

[21] U. Boker, K. Chatterjee, T. A. Henzinger, and O. Kupferman. Temporal specifications with accumulative values. In *LICS*, 2011.

[22] M. Droste and I. Meinecke. Describing average- and longtime-behavior by weighted MSO logics. In *MFCS*, pages 537–548, 2010.

[23] Nick Mitchell and Gary Sevitsky. The causes of bloat, the limits of health. In *OOPSLA*, pages 245–260, 2007.

[24] Guoqing (Harry) Xu and Atanas Rountev. Detecting inefficiently-used containers to avoid bloat. In *PLDI*, pages 160–173, 2010.

[25] Krishnendu Chatterjee and Yaron Velner. Mean-payoff pushdown games. In *LICS*, pages 195–204, 2012.

[26] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *CASCON'99*.

[27] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: java benchmarking development and analysis. *SIGPLAN Not.*, 41(10):169–190, 2006.

[28] Krishnendu Chatterjee, Andreas Pavlogiannis, and Yaron Velner. Quantitative Interprocedural Analysis. In *POPL*, 2015.

[29] Yichen Xie, Mayur Naik, Brian Hackett, and Alex Aiken. Soundness and its role in bug detection systems. In *Proc. of the Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[30] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[31] Aaron Carroll and Gernot Heiser. An analysis of power consumption in a smartphone. In *USENIX*, 2010.

[32] R.M. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Mathematics*, 23:309–311, 1978.

[33] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* The MIT Press, 3rd edition, 2009.

[34] Eric Bodden, Andreas Sewe, Jan Sinschek, Mira Mezini, and Hela Oueslati. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *ICSE '11*, pages 241–250. ACM, 2011.

[35] Ondrej Lhoták and Laurie J. Hendren. Scaling java points-to analysis using spark. In *CC*, pages 153–169, 2003.

[36] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for java. In *PLDI*, pages 387–400, 2006.

[37] Nick Mitchell, Gary Sevitsky, and Harini Srinivasan. Modeling runtime behavior in framework-based applications. In *ECOOP*, 2006.

[38] Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. A scalable technique for characterizing the usage of temporaries in framework-intensive java applications. In *SIGSOFT FSE*, pages 59–70, 2008.

[39] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Efficiently and precisely locating memory leaks and bloat. In *PLDI*, pages 397–407, 2009.

[40] Ohad Shacham, Martin T. Vechev, and Eran Yahav. Chameleon: adaptive selection of collections. In *PLDI*, pages 408–418, 2009.

[41] Ajeet Shankar, Matthew Arnold, and Rastislav Bodík. Jolt: lightweight dynamic analysis and removal of object churn. In *OOPSLA*, 2008.

[42] Guoqing (Harry) Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. Finding low-utility data structures. In *PLDI*, 2010.

[43] Thomas Ball and James R. Larus. Branch prediction for free. In *PLDI*, pages 300–313, 1993.

[44] Youfeng Wu and James R. Larus. Static branch frequency and program profile analysis. In *MICRO 27*, pages 1–11. ACM, 1994.

[45] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach.* Morgan Kaufmann, 2006.

[46] Tim A. Wagner, Vance Maverick, Susan L. Graham, and Michael A. Harrison. Accurate static estimators for program optimization. In *PLDI*, 1994.

[47] Wikipedia. List of performance analysis tools, 2015.