TouchCost: Cost Analysis of TouchDevelop Scripts

Pietro Ferrara[1,2], Daniel Schweizer[2], and Lucas Brutschy[2]

[1] IBM Thomas J. Watson Research Center, U.S.A.
[2] ETH Zurich, Switzerland
pietro.ferrara@us.ibm.com
{daschwei@student,lucas.brutschy@inf}.ethz.ch

Abstract. TouchDevelop is a novel programming environment and language for mobile devices. TouchDevelop applications are typically developed by non-expert users, rather small, and published on the cloud.
In this paper, we introduce TouchCost, a new static analysis that infers the cost of loops in TouchDevelop programs. TouchCost (i) applies an existing generic analyzer to infer numerical invariants, (ii) extracts cost relation systems from these invariants, and (iii) solves them using an existing upper bound solver.
TouchCost has been implemented and applied to all TouchDevelop scripts that are currently published on the cloud. Experimental results show that TouchCost is both scalable and precise. Studying the outputs of TouchCost, we glimpse two major applications: (i) establishing at runtime the cost of a loop, and move its execution to the cloud if it is too expensive w.r.t. the available resources, and (ii) helping a non-expert developer to debug his program.

1 Introduction

In 2012 more mobile devices (e.g., smartphones and tablets) than personal computers and laptops have been sold [1, 23]. The main characteristics of modern mobile devices are (i) an almost continuous connection to the cloud, (ii) relatively limited resources (e.g., computational power and battery), and (iii) various sensors and capabilities (e.g., GPS and camera). This technology shift has important consequences on programming languages and execution environments. In particular, they should take into account (i) novel input devices (e.g., touchscreens) when developing programs, and (ii) a runtime environment with limited local resources, but with (almost) continuous access to an extremely resourceful cloud infrastructure.

Microsoft TouchDevelop[3] [24] is a novel development environment and programming language for mobile applications. The main design principle of TouchDevelop is to allow one to develop mobile applications directly on mobile devices. In addition, TouchDevelop applications can be shared through the cloud infrastructure. Since its release in August 2011, more than 40.000 TouchDevelop scripts

[3] http://www.touchdevelop.com

have been shared. Some of them became quite popular, and they have been downloaded and ran by thousands of users. Usually, TouchDevelop users are not expert developers, and the most part of the scripts are small [20].

Static cost analysis [3] has been deeply studied, and it achieved significant results. The main goal of cost analysis is to compute statically (i.e., at compile time) and automatically (i.e., without any user annotation or interaction) the cost of a program. Its applications are extremely diverse.

Given this scenario, the main contribution of this work is TouchCost, the application of static cost analysis to all existing TouchDevelop scripts to infer the cost of loops. As far as we know, TouchCost is the first cost analysis that has been applied cost to a huge set (several thousands) of real programs. Given a TouchDevelop script, we apply an existing numerical domain [19] to infer numerical invariants. We then build up cost relation systems and pass them to PUBS [2], an up-to-date upper bound solver, obtaining loops' bounds.

TouchDevelop represents an ideal target for cost analysis since (i) TouchDevelop scripts are usually written by non-professional developers , and therefore debugging and optimizing them may improve significantly the quality and the efficiency of these programs, and (ii) these scripts run on mobile devices with limited resources and continuous access to an extremely resourceful cloud infrastructure, and therefore the information inferred by cost analysis may be adopted at runtime to reduce the amount of local resources consumed by the execution.

The analysis has been implemented and applied to all TouchDevelop scripts on the cloud containing loops. The experimental results show that the overall analysis is both scalable and precise. TouchCost proves that existing engines for cost analyses are mature enough to be applied to real programs on a large scale. We have also investigated the results obtained by TouchCost to propose possible applications of the inferred information. First of all, since mobile devices have limited local resources and often access to a resourceful cloud, the costs inferred by TouchCost could be used to decide at runtime to move the execution to the cloud if there is a shortage of some local resources. In addition, TouchDevelop scripts are developed by novices, and particularly high or low costs expose bugs or possible misunderstandings of the developer. Therefore, we found out several published programs in which TouchCost results can be useful for debugging.

The rest of the paper is structured as follows. In the rest of this Section, we will discuss some related work. Sections 2, 3, and 4 will recall the main components of TouchCost, that is, TouchDevelop, Sample, and PUBS, respectively. Section 5 will presents the technical core of TouchCost, while Section 6 will discuss the experimental results.

1.1 Related Work

Various tools performing cost and termination analyses have been formalized and developed. As far as we know, the COSTA system [3] represents the most advanced tool in the field of automatic cost analysis for object-oriented programming languages, and it includes the implementation of some recent research results on finding linear ranking functions [5]. This tool analyzes Java bytecode, and it relies

on PUBS [2] to solve cost relation systems produced by extracting some numerical constraints from a Java bytecode program. TouchCost relies on PUBS as well, but we deal with TouchDevelop code, and we apply a sound and relatively precise heap abstraction of the program. Instead, COSTA approximates the heap with the maximal length of the paths reachable from local variables. This approach is not precise enough for TouchDevelop programs since these heavily rely on the mobile environment approximated by the heap analysis. In particular, COSTA's heap abstraction would approximate the length of all the collections with their *maximal* length. For instance, it would not distinguish between the number of songs and the number of pictures in the mobile device.

Worst case execution time analyses (WCET) have been widely studied, implemented, and applied to industrial software [25]. WCET is focused on deriving realistic, platform-dependent timing information, and usually loop bounds are manually provided by the user [12]. Therefore, various analyses targeted the inference of loop bounds [17], but they target a specific platform, or type of loops, and in general they cannot straightforwardly applied to TouchDevelop scripts that heavily interact with the mobile environment.

Other work has been focused on the analysis of memory consumption [8], and on functional [6] and logic [11] programming. Instead, TouchCost is aimed at targeting various types of costs, and it deals with TouchDevelop code, that is, with a language that mixes imperative and object-oriented constructs.

As far as we know, TouchCost is the first automatic static cost analysis that has been applied to a wide set of mobile programs. Therefore, it represents the first extensive study on the application of cost analysis, and the experimental results (i) show that existing engines for these analyses can be applied to real programs on a large scale, and (ii) open new insights about possible applications of static cost analysis for mobile programs.

2 TouchDevelop

The core of TouchDevelop is a structured programming language designed to develop mobile applications directly on a mobile device. This language mainly mixes imperative and object-oriented features. A TouchDevelop program consists of a set of actions. Intuitively these correspond to methods in object-oriented programming languages. One of the most important design principles is to allow the developer to access all the main components of the mobile device (e.g., GPS sensors) through some standard libraries. Therefore, the API offers various predefined classes to access these components. The target audience of TouchDevelop is "everyone who might traditionally have been able to write a BASIC program on a regular keyboard and ordinary PC. This includes students and hobbyist programmers"[22]. In addition, TouchDevelop scripts can be shared through the cloud infrastructure. Currently, more than 20.000 scripts developed by more than 2.000 users have already been published.

Loops: The TouchDevelop programming language defines three distinct types of loops: while expr do block, for $0 \leq$ index $<$ expr do block, and foreach l in coll

do block. The first type is a standard while loop. The for loop defines an index variable, and it increments this variable from 0 to expr. expr is evaluated only once at the beginning of the execution of the loop. The index variable is modified only by the implicit increment, and it cannot be changed in any other way. Finally, the foreach loop iterates over all elements which are part of a collection *before* starting the execution of the loop. Semantically, this is equivalent to taking a snapshot of the collection just before the execution of the loop, and to iterate over the elements of this snapshot.

Running Example: The program in Figure 1 is the running example we will adopt to explain how TouchCost works. It contains a simple foreach loop, that iterates over all the pictures in the mobile device, prints them on the screen, and waits 1 second before showing the next picture. The loop is iterated n times, where n is the number of pictures contained in the mobile device. Note that we explicitly kept this example simple and minimal, since it will be used to show how TouchCost works step by step in details.

```
action showPics() {
  foreach pic in media→pictures do {
    pic→post_to_wall;
    time→sleep(1);
  }
}
```

Fig. 1. The running example

3 Sample

Sample (Static Analyzer of Multiple Programming LanguagEs) [13, 14] is a generic static analyzer based on the abstract interpretation theory [10]. Relying on compositional analyses, Sample combines various heap abstractions and value (e.g., numerical) domains. It has already been applied to various value analyses (e.g., strings [9], types [13], access permissions [15], and data leaking [26]). It supports some common numerical analyses through Apron[4] [18], which is a library dedicated to the static analysis of the numerical variables. Additionally, some heap analyses are already part of Sample. In particular, [15] adopts a standard abstraction that binds each abstract reference to its allocation site, while [14] plugs a TVLA-based shape analysis.

First of all, Sample compiles source code to *Simple*, the internal language based on control flow graphs (cfg). Sample contains compilers for Java, Scala, and TouchDevelop. The *Simple* program is then passed to the fixpoint engine together with a heap and a value analysis. This produces an abstract result over the cfg, that is, an entry and exit state for each statement of the program. This result is passed to a property checker that produces some alarms if the given property is not statically proved, or to an inference engine that produces some invariants (e.g., the access permissions required or guaranteed by a method [15]).

Simple: *Simple* contains a minimal set of statements (mainly, variable's and field's assignments and accesses, object instantiations, and method calls), while conditional statements and loops are represented directly on the cfg. Each node in a cfg contains a (possible empty) list of statements (representing their

[4] http://apron.cri.ensmp.fr/library/

concatenation), while edges may be weighted with a Boolean value or not. In particular, there is an edge from n_1 to n_2 if the first statement in n_2 may be executed directly after the last statement in n_1. We call an edge from n to some other node an out-edge of n. Weighted edges represents a conditional jump: the edge is traversed only if the expression evaluated by the last statement of the block is true or false depending on the edge's weight. Therefore, the out-edges of a block can be (i) one edge without any weight, (ii) two weighted edges (one with weight true and the other one false), or (iii) none to represent an exit point of the current action.

Loops: In *Simple*, the different TouchDevelop loops (namely, while, for, and foreach) are translated into specific cfg structures.

A loop while expr do block is translated to (i) an initial block in which expr is evaluated, and that has two out-edges, (ii) the out-edge with weight true points to a node containing block, and this points back to the block evaluating expr, (iii) the out-edge with weight false points to the block representing what is *after* the while loop.

for loops are translated in a similar way, initializing the counter to 0 before entering the loop, evaluating *once* the bound of the for loop before entering it, and incrementing the counter by one inside the loop body.

The foreach loop is equivalent to (i) taking a snapshot of the collection just before the execution of the loop, and (ii) iterating over the elements of this snapshot. The iterations are performed by incrementing a counter and accessing the elements contained in the snapshot of the collection.

Running example: The code introduced in Figure 1 is compiled to the cfg in Figure 2. In particular, we can see that (i) the first block initializes i to zero and copies the collection, (ii) the block in the middle contains the loop guard, (iii) the block representing the body of the loop extracts the i-th element from the copy of the collection, execute the body of the foreach loop (that is, it prints the current picture), and increments i by one, and (iv) the false evaluation of the Boolean condition of the guard leads to exit of the action.

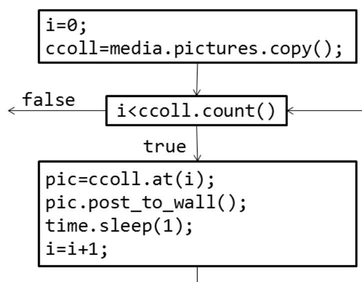

Fig. 2. The cfg of our running example

4 PUBS

TouchCost adopts PUBS[5] [2] to infer upper bounds on loops. In this Section, we briefly recall the main ingredients of PUBS. PUBS takes as input a cost relation system, and it returns an upper bound of the cost of this system.

Fist of all, we define the basic ingredients of cost relations. A *linear expression* has the form $\sum_{i=1}^{n} a_i * x_i + b$. A *linear constraint* is defined as $l_1 \leq l_2$ where l_1 and l_2 are both linear expressions. A *guard* is defined as a set of linear constraints,

[5] `http://costa.ls.fi.upm.es/~costa/pubs/pubs.php`

and it represents their conjunction. *Basic cost expression* are then defined as follows:

```
exp ::= r | nat(l) | exp1 + exp2 | exp1 * exp2 | log_n(exp) |
        n^exp | max(S) | exp - r
```

where `l` is a linear expression, `r` is a real positive number, `S` is a set of basic cost expressions, `nat(l)` returns $\max(0, \mathtt{l})$.

These expressions are the basic blocks to define cost relations. A cost relation is a pair $\langle C(\overline{x}) = \mathtt{exp} + \sum_{i=1}^{k} D_i(\overline{y}_i), \phi \rangle$ where C and D_i are cost relation symbols (that is, symbols representing the costs of an action or a loop), `exp` is a basic cost expression, $\overline{x}$ and $\overline{y}_i$ are distinct variables, and ϕ is a guard. $C(\overline{x}) = \mathtt{exp} + \sum_{i=1}^{k} D_i(\overline{y}_i)$ will be called the *cost body* in the rest of this paper. Finally, a cost relation system is a set of cost relations.

Running example: Consider now the running example we introduced in Figure 1, and in particular its cfg in Figure 2. For the sake of simplicity, let us suppose that the cost of one iteration of the loop body is 1. In addition, the cost relation system about this block should represent the fact that i is incremented by one, and in order to execute this block $\mathsf{i} < \mathsf{ccoll.count}()$ must hold. All these facts are represented by the following cost relation:

$$\left\langle \begin{array}{l} C(\mathsf{old_i}, \mathsf{pics_count}) = 1 + C(\mathsf{i}, \mathsf{pics_count}), \\ \qquad\qquad \{\mathsf{i} == \mathsf{old_i} + 1, \mathsf{old_i} < \mathsf{pics_count}\} \end{array} \right\rangle$$

where C represents the cost of the foreach loop in terms of the initial value of i, pics_count represents the number of pictures in the mobile device at the beginning of the execution, and old_i represents the value of i at the beginning of the loop and i its value at the end.

Instead, if $\mathsf{i} < \mathsf{ccoll.count}()$ does not hold, the cost of the execution of the loop is zero: $\langle C(\mathsf{old_i}, \mathsf{pics_count}) = 0, \{\mathsf{old_i} \geq \mathsf{pics_count}\} \rangle$

Finally, we have to represent the fact that, before entering the loop, i is equal to zero: $\langle C_l(\mathsf{pics_count}) = C(0, \mathsf{pics_count}), \{\mathsf{pics_count} \geq 0\} \rangle$ where C_l represents the cost of the whole loop.

The goal of TouchCost is to apply Sample to infer automatically these cost relations starting from the program in Figure 2. When we pass these cost relations to PUBS, we obtain the cost $nat(\mathsf{pics_count})$, that is exactly the cost of our loop.

5 TouchCost

Given a TouchDevelop action, TouchCost (1) compiles it and augments its cfg, (2) applies Sample to this augmented cfg, (3) extracts from the abstract results a cost relation system for each loop, and (4) pass these cost relation systems to PUBS, obtaining their upper bounds.

5.1 Augmented Control Flow Graph (1)

Identifying Loops: First of all, given a control flow graph we have to identify the structures that represent loops. We traverse the control flow graph and we

consider the edges that have two *deterministic* (that is, non-weighted) in-edges and two weighted out-edges (one true and one false). Such node is potentially the *initial* node of the loop. Then, starting from the out-edge labeled true, we check if there is a cyclic path coming back to this node. If this is the case, we have found a loop.

Augmenting the Cfg: We need to infer relations between the entry and the exit values of variables. Unfortunately, a numerical domain usually does not infer such information, since the old value of a variable once this has been assigned. For instance, in the running example of Figure 1, once we increment i by one, we do not know that the value at the end of the loop is equal to the value at the beginning incremented by one. We have then to make a copy of all the variables modified inside a loop at the beginning of the loop body. For instance, the running example requires a cost relation that tells us that i at the end of the loop is equal to its initial value plus 1. So we introduce a variable old_i to represent the value of i at the beginning of the loop's body.

Therefore, for each loop in the cfg, we find all variables V that are assigned inside the loop, and we add a new assignment $\mathsf{old_v} := \mathsf{v}$ at the beginning of the loop for each variable $\mathsf{v} \in \mathsf{V}$.

Running example: Figure 3 depicts the augmented cfg we obtain for the cfg our running example of Figure 2.

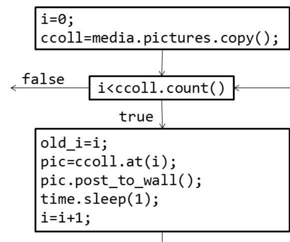

Fig. 3. The augmented cfg

5.2 Sample's Analysis (2)

In order to run Sample on the augmented control flow graph, we have to instantiate the analysis with a heap and a numerical analysis.

Heap Analysis: Since TouchDevelop programs do not usually perform significant computation over the heap, and this rarely influences how many times loops are iterated, we apply a standard and efficient allocation site-based heap abstraction [4]. In addition, we build a precise model of the collections and the mobile environment (e.g., to distinguish the number of elements in the songs' collection from the pictures' collection).

Numerical Analysis: Sample has already been applied to various value analyses. Apron [18] is a library that provides a standard interface to various numerical domains, and it is plugged in Sample. In our analysis, we apply Linear equalities [19] to infer input-output relations that will be used to build up the cost relation system that is passed to PUBS.

Running example: Consider the running example introduced in Figure 1. The heap analysis translates the method call ccoll.count() to the symbolic identifier we use to represent the number of elements in ccoll (represented by ccoll_count). This allows us to infer that the loop guard is $\mathsf{i} < \mathsf{ccoll_count}$. In addition, the heap semantics infers that $\mathsf{ccoll_count} == \mathsf{pics_count}$ (where pics_count represents the number of pictures in the mobile device at the beginning of the execution), since ccoll is a copy of media.pictures, and the linear equalities domain infers that, inside the loop body after the increment of i, the constraint $\mathsf{i} == \mathsf{old_i} + 1$ holds.

Note that, even on this simple running example, we need a sound static analysis that defines the semantics of TouchDevelop APIs, and in particular of its data structures. For instance, we need to semantically track that when we copy media.pictures and we assign it to ccoll we have that ccoll_count == pics_count.

5.3 Extracting Cost Relation Systems (3)

At the end of the analysis, Sample returns an abstract entry and exit state for each statement in the program. Using this information, we extract the cost relation systems that will be passed to PUBS.

First of all, we identify all the loops in the cfg as described in Section 5.1. Then for each loop we build up a cost relation system that is aimed at computing how many times the loop is iterated.

Cost Relations of Loops: We introduce two cost relation symbols for each loop: C_l represents the whole loop, while C represents the loop's iterations starting from a given state (e.g., after i iterations). The parameters of these two symbols are the variables involved in the Boolean guard of the loop, or in its body.

For the cost symbol C, the cost body adds the cost of one iteration of the loop body to the cost of the following iterations. The body of the loop could contain four different patterns of cfg structures, and they will be discussed in the following of this Section. For now, we suppose that the cost symbol representing block is provided by $cost(\mathsf{block})$. Formally, the cost body is then defined by $C(\overline{\mathsf{old_x}}) = cost(\mathsf{block}) + C(\overline{\mathsf{x}})$, where $\overline{\mathsf{old_x}}$ and $\overline{\mathsf{x}}$ represent a sequence of variables old_x and x, respectively. For all the variables that are not assigned in the loop body, we have that old_x = x. For all the other variables, we try to extract an update rule from the information inferred by the linear equality domain after the last statement in the loop body.

Through the analysis introduced in Section 5.2 on the augmented cfg described in Section 5.1, the state inferred by Sample after the last statement in the loop body contains all the relations between the values of the variables at the beginning and at the end of the execution of one iteration of the loop. Since we have applied the linear equalities domain, we represent this state as a set of linear equalities. We want to extract from this set only the constraints that involve variables relevant to compute the cost of the loop. We also want to extract information that is (i) strong enough to infer the cost of the loop, and (ii) as little as possible to preserve the efficiency of the analysis. This means that we have to consider only the variables that influence how many times a loop is iterated.

We start from all the variables appearing in the loop guard. Then, for each of these variables, we try to find an update rule involving this variable. By update rule of variable x we mean a constraint that contains both x and old_x. Then, given a variable v, we consider all the equalities involving v and old_v. As a first try, we consider *only* the linear constraints that are fully described by v and old_v, that is, constraints of the form $\mathsf{v} == a * \mathsf{old_v} + b$. In practice, such constraints are often strong enough to infer the cost of the loop. If PUBS fails to compute the cost of the loop using these constraints, we consider all linear

constraints of the form $\mathsf{v} == \mathsf{expr}$, where expr is a linear expression containing $\mathsf{old_v}$.

These update rules are plugged in the cost relation of C together with the loop guard b. Formally, the cost relation for C is defined by $\langle C(\overline{\mathsf{old_x}}) = \mathit{cost}(\mathsf{block}) + C(\overline{\mathsf{x}}), \mathit{UR} \cup \{b\}\rangle$ where UR represents the set of update rules we extracted, and *cost* represents the cost symbol or variable of the loop body. This cost relation represents one iteration of the while loop. We add then a cost relation to represent when we exit from the loop. Formally, $\langle C(\overline{\mathsf{x}}) = 0, \{!\mathsf{b}\}\rangle$.

Finally, we have to introduce a cost relation to represent the whole execution of the loop. This is done by a cost relation that represents that the cost of the whole loop C_l is equal to the cost of the loop C when the parameters are the values of the variable before entering the loop for the first time. Therefore, for each variable v involved in the loop, we look at all linear constraints IV containing v that the numerical analysis inferred at the program point just before the loop. We then build up a cost relation $\langle C_l(\overline{\mathsf{x}}) = C(\overline{\mathsf{x}}), \{IV\}\rangle$.

Cost Relations of Other Cfg Structures: Inside the loop body, we could have (i) one block, (ii) a sequence of blocks, (iii) a nested loop, or (iv) a conditional.

In the first case, TouchCost represents symbolically the cost of a single block with a *cost variable* (that is, the block starting at program point p is represented by a symbol c_p). If we have a sequence of blocks, we obtain the summation of the cost symbols of each block.

If we have a nested loop, we extract the cost relations representing the loop recursively, and we use the cost symbol C_l representing the whole loop in the cost body of the current structure.

Finally, if we have a conditional, we introduce C_t to represent the true branch, and C_f for the false branch. Then we add (i) $\langle C_t(\overline{x}) = \mathit{cost}(\mathsf{block1}), \{b\}\rangle$ to represent the true branch, (ii) $\langle C_f(\overline{x}) = \mathit{cost}(\mathsf{block2}), \{!b\}\rangle$ to represent the false branch, and (iii) $\langle C_i(\overline{x}) = C_t(\overline{x}) + C_f(\overline{x}), \{\}\rangle$ to represent the whole if statement.

Boolean Conditions: Up to now, we have simplified the presentation by using b and $!b$ in the guards when dealing with loops and conditionals. Indeed, PUBS allows only linear relations in these guards[6]. Therefore, we consider only the Boolean conditions of the following form:

```
c ::= true | false | expr1 <op> expr2 | ! c | c1 AND c2 | c1 OR c2
```

where `expr1` and `expr2` are linear expressions, and $<\mathtt{op}> \in \{!=, ==, \geq, \leq\}$. All these conditions have to be translated into linear integer expressions to fulfill PUBS' syntax. `true` is translated to $1 == 1$, and `false` to $0 == 1$. `expr1 <op> expr2` is already a linear expression, while `! c` is translated to a positive form by using the De Morgan's laws if `c` is an `AND` or `OR` expression, by negating `true` or `false`, or by modifying `<op>` if the condition is a comparison of linear expressions. Linear integer conditions with $<$ or $>$ as comparison operators are translated to equivalent conditions with $\geq$ or $\leq$ as operators, and by adding 1 to the left or right part of the comparison.

[6] See `http://costa.ls.fi.upm.es/~costa/pubs/help.php` for the complete syntax of cost relations and expressions

AND and OR conditions lead to several cost relations, since we cannot represent a conjunction or disjunction directly in PUBS guard. These are semantically equivalent to translate the conditions into equivalent cfg structures, and the cost relations for these structures are obtained as described in this Section previously.

Note that we may not be able to translate the original Boolean condition to integer linear relations that are supported by PUBS (e.g., if the original condition was not linear). In these cases, we simply omit the Boolean condition and its negation from the guards. In this way, we introduce some approximation (that is, the fact that a particular part of the code is executed only if the condition holds), but we preserve the soundness of the analysis.

Running example: Consider now the running example of Figure 1. First of all, we have $\langle C(\mathsf{old_i}, \mathsf{ccoll_count}) = 0, \{\mathsf{old_i} \geq \mathsf{ccoll_count}\}\rangle$. This represents the situation in which the execution exit the loop. Instead, one iteration of the loop is represented by

$$\left\langle \begin{array}{l} C(\mathsf{old_i}, \mathsf{ccoll_count}) = c_b + C(\mathsf{i}, \mathsf{ccoll_count}), \\ \qquad\qquad \{\mathsf{i} = \mathsf{old_i} + 1, \mathsf{old_i} \leq \mathsf{ccoll_count} - 1\} \end{array} \right\rangle$$

In fact, the linear equalities domain infers that $\mathsf{i} = \mathsf{old_i}+1$, and c_b represents symbolically the cost of the loop body. In addition, the loop guard $\mathsf{old_i} < \mathsf{ccoll_count}$ is translated to $\mathsf{old_i} \leq \mathsf{ccoll_count} - 1$. Finally, the cost of the whole loop is

$$\left\langle \begin{array}{l} C_l(\mathsf{i}, \mathsf{ccoll_count}) = C(\mathsf{i}, \mathsf{ccoll_count}), \\ \qquad \{\mathsf{i} == 0, \mathsf{ccoll_count} == \mathsf{pics_count}\} \end{array} \right\rangle$$

The numerical domain infers that initially $\mathsf{i} == 0$ and $\mathsf{ccoll_count} == \mathsf{pics_count}$.

5.4 Using PUBS (4)

The last step of TouchCost is to pass the cost relation systems we inferred for each loop to PUBS. The output of PUBS could be: (i) a sound upper bound of the given cost relation system, or (ii) a failure. In the second case, we do not know if this failure was due to some information that is not precisely tracked by PUBS, or if the analyzed loop may not terminate.

Running example: PUBS returns $nat(\mathsf{pics_count})$ when we apply it to the cost relation system we inferred for our running example in Section 5.3. This is the exact cost of the loop, since it is iterated a number of times equal to the number of pictures we have in our mobile device, and this value is represented by pics_count.

6 Experimental Results

Table 1 reports the experimental results. We ran the experiments on an Intel Core 2 2.83Ghz QUAD CPU with 4GB RAM running Ubuntu 12.04. Column **Type** reports the script category we target. In particular, `all` denotes all the scripts published on the cloud before May, 16th 2013 containing loops, while the other categories refer to the scripts (always containing loops) that are tagged with the given name. Tags are used to categorize different types of scripts, and we used them to investigate how TouchCost behaves when dealing with

Type	#scr.	LOC	#loops	Comp.	Prec.	S t.	TC t.	Avg. LOC	Avg. St	Avg. TCt
all	5 405	1 222 250	19 035	13 403	70.4%	10 153	4 293	226	1.88	0.79
entertainment	164	42 485	553	419	75.8%	459	145	259	2.80	0.88
games	161	54 754	973	764	78.5%	749	274	340	4.65	1.70
libraries	129	23 548	405	236	58.3%	314	94	183	2.43	0.73
tools	82	22 460	374	132	35.3%	200	48	274	2.44	0.59
lifestyle	78	23 903	179	126	70.4%	135	55	306	1.73	0.70
music	59	9 163	149	100	67.1%	44	29	155	0.74	0.49
education	56	9 710	158	131	82.9%	71	25	173	1.27	0.45
gamelibraries	47	8 002	189	101	53.4%	237	47	170	5.04	1.00
Sample root	73	8 964	233	167	71.7%	94.4	56.9	123	1.29	0.78

Table 1. Experimental results

different types of scripts. Column **#scr.** reports the number of scripts, **LOC** the number of lines of code, **#loops** the number of loops, **Comp.** the number of loops for which TouchCost computed the cost, **Prec.** the precision rate (that is, **Comp./#loops**), **S t.** the time spent by Sample to perform the heap and numerical analysis, **TC t.** the time spent by TouchCost to build up the cost relation systems and use PUBS to solve them, **Avg. LOC** the average number of lines of code per scripts, and **Avg. St** and **Avg. TCt** the average time of Sample and TouchCost analysis per script, respectively. All the times are in seconds.

6.1 Global Performances and Precision

We first perform a quantitative analysis considering row `all` in Table 1. This benchmark consists of 5 405 scripts, and by more than 1 million of LOC. In terms of performances, the overall analysis (that is, **S t.** + **TC t.**) took 4h00'06" to analyze 1.222 KLOC (about 2.5 seconds per script). In terms of precision, TouchCost inferred the cost of about 70% of the existing loops. On the one hand, this result underlines that, on average, TouchCost automatically infers the cost of the most part of existing loops. On the other hand, different categories of scripts expose different levels of precision. For instance, `games` scripts are usually relatively big, and TouchCost compute the cost of almost the 80% of the loops in these scripts. Instead, `tools` scripts seem to be more challenging, since the precision rate for this category is around 35%. In addition, one could expect that bigger scripts contain more complex code, and therefore TouchCost is less precise on such scripts. Indeed, our experimental results show that there is no correlation between the length of the script and the precision of TouchCost. For instance, `gamelibraries` scripts are smaller than the average, but the precision of TouchCost is around 50%. This category of scripts seems to be particularly critical and complex to analyze: the analysis of Sample takes almost three times the average, and this underlines the presence of rather complex code.

6.2 Precision on TouchDevelop Sample Scripts

We now inspect manually the precision of TouchCost when dealing with TouchDevelop sample root scripts containing loops. TouchDevelop samples[7]

[7] `https://www.touchdevelop.com/pboj`

```
action main() {
  // Initialize the board
  while true do {
    // Update the board
    time→sleep(0.01);
  }
}
```

(a) Script iuks

```
data _board : Board
data _bubbles : Sprite Set
data _touch : Sprite
action pop bubbles () {
  var p := _board→touch current
  _touch→set pos(p→x, p→y)
  foreach bubble in _touch→overlap
                with(_bubbles) do
    ...
}
```

(b) Script uigca

```
action TouchList (...) {
  var count := 0
  var b := true
  while b do {
    //Perform some computation
    count := count + 1
    if count = 5 then
      b := false
  }
}
```

(c) Script vrgt

```
action show(board: Board) {
  duration := 5;
  dt := time→now;
  while time→now→subtract(dt) < duration do {
    //Show something to the user
  }
}
```

(d) Script jhyg

```
action getConfCalls (s : DateTime, e : DateTime) {
  app := social→
        search_appointments(s, e);
  foreach a1 in app
  do {
    //Extracts information
  }
  //Build up the conference call
  //Call the people
}
```

(e) Script mpuj

```
action main() {
  b := true;
  while b do {
    code→play_round;
    b := wall→
    ask_boolean("Try␣again?", " ... ");
  }
}
```

(f) Script avpm

```
action b () {
  for 0 ≤ i < 999999 do {
    if i / 2 = math → floor(i / 2) then {
      wall →
        set background(colors → white)
      time → sleep(1) }
    else {
      wall →
        set background(colors → black)
      time → sleep(1) }
  }
}
```

(g) Script lypy

```
var len := 0
while len ≥ 0 do {
  ...
  len := len − 1
}
```

(h) Script hyax

Fig. 4. Case studies

contain a significant set of scripts developed by the TouchDevelop team to show the main features of this language. Row `sample root` in Table 1 reports the experimental results we obtained on these scripts. We analyzed 73 scripts (about 9.000 LOC). All together, these scripts contain 233 loops, and we failed to compute the cost of 66 loops.

Non-Terminating Loops: First of all, we noticed that some of the loops are not necessarily terminating, and therefore the results of TouchCost are precise. We have identified three main reasons of non-terminating loops.

User inputs: Some loops are iterated until the user provides a "good" input. Consider for instance script `avpm`[8] in Figure 4f. Action main iterates a while loop until the user says that he wants to stop. Statically, this loop may be non-terminating, since we do not know when the user will decide to stop.

while true *loops:* Another type of loops for which TouchCost cannot compute their cost is represented by script `iuks` in Figure 4a. TouchDevelop provides a specific gameloop action that "is triggered by a timer approximately every 50

[8] Given a TouchDevelop script name `<script>`, the code of the script is available at `https://www.touchdevelop.com/<script>`

milliseconds"[9]. Nevertheless, several users prefer to implement their own game loop iterators. In this way, they can establish exactly the triggering rate by adding a time→sleep statement inside the while loop.
Time constraints: Another recurrent pattern is exposed by script `jhyg` in Figure 4d. In this case, a loop is iterated during a given amount of time (e.g., 5 seconds). This is obtained by (i) recording the time just before entering the loop (variable dt), (ii) checking how much time is passed each time the loop is iterated, and (iii) exiting the loop if this subtraction exposes that that enough time (that is, at least duration seconds) has passed. Even if in this case we know that the loop will eventually terminate (since the time is always strictly increasing), we cannot know statically how many times the loop is iterated.

Approximation: There are then some cases in which we fail to infer an upper bound on the number of iterations because of a too rough approximation. In particular, we identified two main sources of imprecision.
Collections: Figure 4b reports an excerpt of action popbubble of script `uigca`. This action is aimed at popping a bubble that is touched by the user. Therefore, it contains a foreach loop that iterates over the bubbles that overlaps with the existing bubbles. Unfortunately, the abstract semantics of Sprite.overlap in Sample is too imprecise, and we fail to infer any upper bound on this loop. Our experience shows that this situation is common to various scripts, and it is the main source of imprecision of TouchCost. Therefore, we are currently working on more refined analyses for TouchDevelop analyses [7], and we expect it will fix this issue. Nevertheless, it will slow down the analysis, and we will have to study in which cases it is worth to apply more refined analysis.
Disjunctive information: We found quite few cases in which TouchCost failed to compute the cost of a loop because of complicated disjunctive invariants. One of these cases is the action TouchList of script `vrgt` sketched in Figure 4c. The loop is iterated 5 times, but this is obtained by (i) a Boolean flag b as loop guard, (ii) counting the number of iterations through a variable count, and (iii) setting b to true when $\mathsf{count} = 5$. In order to compute that this loop is iterated 5 times, we would need to track disjunctive information through trace partitioning [21] that is already supported by Sample[16], and translate this information to a cost relation system.

6.3 Applications of TouchCost

Finally, we inspect the results of the analysis investigating some particular cases to study possible applications of TouchCost. In particular, since TouchDevelop scripts are executed on mobile devices usually connected to the cloud, this information can be used at runtime to decide to move the execution to the cloud if the application is too expensive w.r.t. the available resources. In addition, since TouchDevelop users are usually novices, TouchCost provides useful information to debug their programs.
Moving the Execution: We start by considering script `mpuj` in Figure 4e. This is the most popular script on the cloud: on May, 2013 it counted more than 2300

[9] `https://www.touchdevelop.com/help/events`

users and 40.000 runs. In addition, it is the evolution of `slji`, that counted more than 450 users and 10.000 runs. This script extracts the conference calls from the calendar of the mobile device, and on request it dials the numbers of the people involved in the conference calls.

It consists of 15 actions, and only getConfCalls contains a loop. Figure 4e sketches the main components of this action. The body of the loop extracts the subject and the location of the conference calls, and it builds up some strings to represent this information. TouchCost infers that the loop is iterated a number of times equal to the number of elements contained in the collection returned by social→search_appointments(start, end). At compile time, the number of elements depends on start and end, two parameters of the action whose value is unknown. Instead, at runtime, when the action is called, we know the actual values of these two parameters. Then we can use this information to establish exactly how many times the loop is iterated when the action is called. So the runtime environment could decide to move the computation to the cloud if the loop requires (e.g., computational) resources that are not locally available.

Wrong Implementations and Bugs: We then investigate some *extreme* costs. Since many TouchDevelop users are novices, they sometimes implement a functionality in the wrong way. A quite common example is a loop that is iterated an enormous number of times. Consider for instance the snippet of code in Figure 4g of script `lypy`. This script is relatively popular, with more than 1000 runs and 100 users. Action b iterates 999.999 times a loop that changes the color of the background, and put the device to sleep for 1 second. The user intended to write an endless loop, and the cost analysis infers that the loop is iterated 999 999 times. Finally, the cost information can expose some bugs as well. Script `hyax` contains three actions (Generate, Generate2, Generate3) that follow the pattern sketched in Figure 4h.

TouchCost correctly infers that the loop is iterated only once. This is definitely a bug, and even the various versions of this script (see scripts `fosieeps`, `nlqo`, and `lwnlb`) are all bugged. Generally, when TouchCost infers that the loop is iterated only once, we may issue an alarm and ask the developer to check if this behavior is intended, or if it is a bug.

References

1. The future of mobile, 2012. Available at `http://www.businessinsider.com/the-future-of-mobile-slide-deck-2013-3`.
2. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-form upper bounds in static cost analysis. *Journal of Automated Reasoning*, 46(2):161–203, 2011.
3. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of object-oriented bytecode programs. *Theoretical Computer Science*, 413(1):142–159, 2012.
4. L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.
5. A. M. Ben-Amram and S. Genaim. On the linear ranking problem for integer linear-constraint loops. In *Proceedings of POPL '13*. ACM, 2013.

6. R. Benzinger. Automated higher-order complexity analysis. *Theoretical Computer Science*, 318(1-2):79–103, 2004.
7. Y. Bonjour. Must analysis of collection elements. Master's thesis, ETH Zurich, September 2013.
8. V. Braberman, F. Fernández, D. Garbervetsky, and S. Yovine. Parametric prediction of heap memory requirements. In *Proceedings of ISMM '08*. ACM, 2008.
9. G. Costantini, P. Ferrara, and A. Cortesi. Static analysis of string values. In *Proceedings of ICFEM '11*, LNCS. Springer, 2011.
10. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of POPL '77*. ACM, 1977.
11. S. K. Debray and N.-W. Lin. Cost analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, 1993.
12. C. Ferdinand, R. Heckmann, H. Theiling, and R. Wilhelm. Convenient user annotations for a wcet tool. In *Proceedings of WCET '03*, 2003.
13. P. Ferrara. Static type analysis of pattern matching by abstract interpretation. In *Proceedings of FORTE/FMOODS '10*, LNCS. Springer, 2010.
14. P. Ferrara, R. Fuchs, and U. Juhasz. TVAL+ : TVLA and value analyses together. In *Proceedings of SEFM '12*, LNCS. Springer, 2012.
15. P. Ferrara and P. Müller. Automatic inference of access permissions. In *Proceedings of VMCAI '12*, LNCS. Springer, January 2012.
16. D. Gabi. Disjunction on demand. Master's thesis, ETH Zurich, 2011.
17. J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution. In *Proceedings of RTSS '06*. IEEE Computer Society, 2006.
18. B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *Proceedings of CAV '09*, LNCS. Springer, 2009.
19. M. Karr. On affine relationships among variables of a program. *Acta Informatica*, 6(2):133–151, 1976.
20. S. Li, T. Xie, N. Tillmann, M. Moskal, J. de Halleux, M. Fahndrich, and S. Burckhardt. A comprehensive field study of end-user programming on mobile devices. Technical Report TR-2013-3, Microsoft Research, 2013.
21. L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In *Proceedings of ESOP '05*, LNCS. Springer, 2005.
22. N. Tillmann, M. Moskal, J. de Halleux, and M. Fahndrich. Touchdevelop - programming cloud-connected mobile devices via touchscreen. Technical Report TR-2011-49, Microsoft Research, 2011.
23. N. Tillmann, M. Moskal, J. de Halleux, M. Fahndrich, J. Bishop, A. Samuel, and T. Xie. Touchdevelop - app development on mobile devices. In *Proceedings of ITiCSE 2012*. ACM, 2012.
24. N. Tillmann, M. Moskal, J. de Halleux, M. Fahndrich, and S. Burckhardt. Touchdevelop - app development on mobile devices. In *Proceedings of FSE '12, Demonstration*. ACM, 2012.
25. R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem. overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3), 2008.
26. M. Zanioli, P. Ferrara, and A. Cortesi. SAILS: static analysis of information leakage with Sample. In *Proceedings of SAC '12*. ACM, 2012.