ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ

Πτυχιακή Εργασία

Στατική sharing ανάλυση για βελτιστοποίηση δυναμικής ανίχνευσης data race

https://github.com/Bela-Kamilo/static-sharing-analysis-for-optimizing-dynamic-race-detection

Κυπαρίσσης Αλέξανδρος CSD 4210

Επόπτης καθηγητής: Πολύβιος Πρατικάκης
...

Ηράκλειο Κρήτης 2025

ΠΙΝΑΚΑΣ ΠΕΡΙΕΧΟΜΕΝΩΝ

1. Περίληψη

Δυναμικές, on-the-fly, αναλύσες προγραμμάτων παρέχουν σημαντικά περισσότερη πληροφορια απο ο,τι στατικές προσεγγίσεις με σημαντικά μεγαλύτερο κόστος απόδοσης. Σκοπός αυτής της εργασίας ειναι η εύρεση προσβάσεων στο heap του JVM, οι οποίες είναι αδύνατο να προκαλέσουν data race, πριν την εκτέλεση ενός προγράμματος. Βάση αυτής της πληροφορίας, μια δυναμική ανίχνευση data race μπορεί να αγνοήσει με ασφάλεια συγκέκριμενες εντολές μειώνοντας τον χρόνο εκτέλεσής της. Παρουσιάζεται μια στατική ανάλυση, ανεξάρτητη της ροής του προγράμματος, της οποίας η βάση είναι μια points-to ανάλυση.

Abstract

Dynamic, on-the-fly, program analyses yield significantly more information than static approaches at the cost of significant performance cost. The aim of this work is finding thread safe JVM memory accesses, which are sure not to cause data races, before execution time. Using this information, a dynamic race detector can safely ignore certain instructions reducing its execution time. A static, context-insensitive, points-to based analysis is presented

2. Εισαγωγή

<κειμενο>

2.1 Data race

Το specification της Java οριζει το data race ως εξης :

<in quote format :>
When a program contains two conflicting accesses that are not ordered by a happens-before relationship, it is said to contain a *data race*.

Two accesses to (reads of or writes to) the same variable are said to be *conflicting* if at least one of the accesses is a write.[1]

2.1 Points-to ανάλυση

Η points-to ανάλυση έχει σκοπο να ανακαλύψει ποιές μεταβλητές της στοίβας του προγράμματος ενδέχεται να αναφέρονται σε ποιά objects.
Κάθε τελεστής `new` σηματοδοτεί την δημιουργία ενός object.
Εντολές εκχώρησης και κλήσεις μεθόδων μεταβιβάζουν τα objects μεταξύ των μεταβλητών υποδεικνύοντας σε ποιά objects ενδέχεται να αναφέρονται.
Το αποτέλεσμα είναι μία αντιστοίχιση των μεταβλητών και συνόλων από objects.

3. Σχετική δουλειά

<thread saitizer>

<Andersen’s analysis>

4. Αναλυτική περιγραφή

Η είσοδος της ανάλυσης είναι Java bytecode, η επεξεργασία του οποίου γίνεται μέσω του SootUp framework.[2]
Η είσοδος μετατρέπεται σε μορφή ενδιάμεσης αναπαράστασης (Jimple IR).

<entry method καθε run() >

Εξετάζεται κάθε εντολη που εν δυνάμει εκτελείται απο αυτή την μέθοδο και όποια άλλη μέθοδο καλείται αναδρομικά.
Παράγεται μια αντισοιχία μεταβλητών και συνόλων απο objects στα οποία εν δυνάμει αναφέρονται και μια αντιστοίχιση μεθόδων και προσβάσεων μνήμης.
Οι προσβάσεις μνήμης είναι της μορφής : READ/WRITE object#.field .

<read/write effect sets/set sizes>

4.1 Jimple IR

Η Jimple ειναι η ενδιάμεση αναπαράσταση του SootUp.

- stackless
- no nested structures

παράδειγμα hello world :

```
public class HelloWorld extends java.lang.Object
{
  public void <init>()
  {
    HelloWorld r0;
    r0 := @this: HelloWorld;
    specialinvoke r0.<java.lang.Object: void <init>()>();
    return;
  }

  public static void main(java.lang.String[])
  {
    java.lang.String[] r0;
    java.io.PrintStream r1;
    r0 := @parameter0: java.lang.String[];
    r1 = <java.lang.System: java.io.PrintStream out>;
    virtualinvoke r1.<java.io.PrintStream:
    void println(java.lang.String)>("Hello world!");
    return;
  }
}
```

Αναλυτικά η γραμμάτική της [3]:

stmt ⟶ *assignStmt* | *identityStmt* | *gotoStmt* | *ifStmt* | *invokeStmt* | *switchStmt* | *monitorStmt* | *returnStmt* | *throwStmt* | *breakpointStmt* | *nopStmt*;
assignStmt ⟶ local = *rvalue*; | field = *imm*; | local . field = *imm*; | local [*imm*] = *imm*;
identityStmt ⟶ local := @this: type; | local := @parameter*n*: type; | local := @exception;
gotoStmt ⟶ `goto` label;
ifStmt ⟶ `if` *conditionExpr* `goto` label;
invokeStmt ⟶ `invoke` *invokeExpr*;
switchStmt ⟶ `lookupswitch` *imm* {`case` value$_1$: `goto` label$_1$; ... `case` value$_n$: `goto` label$_n$; `default:` `goto` defaultLabel;}; | `tableswitch` *imm* {`case` low: `goto` lowLabel; ... `case` high: `goto` highLabel; `default:` `goto` defaultLabel;}
monitorStmt ⟶ `entermonitor` *imm*; | `exitmonitor` *imm*;
returnStmt ⟶ `return` *imm*; | `return`;
throwStmt ⟶ `throw` *imm*;
breakpointStmt ⟶ `breakpoint`; *nopStmt* ⟶ `nop`;

imm $\longrightarrow$ local | constant
conditionExpr $\longrightarrow$ imm_1 *condop* imm_2 *condop* $\longrightarrow$ $>$ | $<$ | = | $\neq$ | $\leq$ | $\geq$
rvalue $\longrightarrow$ *concreteRef* | *imm* | *expr* *concreteRef* $\longrightarrow$ field | local . field | local [*imm*]
invokeExpr $\longrightarrow$ `specialinvoke` local.m(imm_1, ..., imm_n) | `interfaceinvoke` local.m(imm_1, ..., imm_n) | `virtualinvoke` local.m(imm_1, ..., imm_n) | `staticinvoke` m(imm_1, ..., imm_n)
expr $\longrightarrow$ imm_1 *binop* imm_2 | (type) *imm* | *imm* `instanceof` type | *invokeExpr* | `new` refType | `newarray` (type) [*imm*] | `newmultiarray` (type) [imm_1] ... [imm_n] []* | `length` *imm* | `neg` *imm*
binop $\longrightarrow$ + | - | $>$ | $<$ | = | $\neq$ | $\leq$ | $\geq$ | $*$ | / | $<<$ | $>>$ | $<<<$ | % | `rem` | & | | | `cmp` | `cmpg` | `cmpl`

4.2 Αφαίρεση θέσεων μνήμης

Κάθε new σηματοδοτείται μοναδικά με έναν ακέραιο αριθμό αναπαριστόντας μια θέση μνήμης ενός object.

```
x= new¹ A();     //object 1 created here
y= new² A();     //object 2 created here
```

Ένα new ενδεχομένως να δημιουργίσει περισσότερα από ένα objects.
Τα objects που δημιουργούνται από την ίδια εντολή αντιμετοπίζονται σαν ένα

```
while(...){
       x= new³ A(); //object(s) 3 created here
      ...
}
```

Οι θέσεις μνήμης μας, αναπαριστόντας objects, εχούν πεδία.
Αναφερόμαστε σε αυτά ως 1.someField

4.3 Points-to Sets

Η πληροφορία “η μεταβλητή x αναφέρεται στο object 1” αναπαριστάται με την αντιστοιχίση $x \rightarrow \{1\}$.

Μια αντισοίχιση του παραπάνω παραδείγματος

```
x= new¹ A();     //object 1 created here
y= new² A();     //object 2 created here
```

ειναι η εξής :

$x \rightarrow \{1\}$
$y \rightarrow \{2\}$

4.4 May

Μία pointer ανάλυση χαρακτηρίζεται may ή must.
Σκοπός της παρούσας ανάλυσης είναι οι αντιστοιχίσεις

```
someVariable → {1,3}
WRITES(someMethod) → {2.someField,4.someField}
```

να εκφράζουν οτί η μεταβλητή someVariable ενδέχεται να αναφέρεται στο object 1 ή στο object 3 και η μέθοδος someMethod ενδέχεται να γράφει το πεδίο someField στην μνήμη όπου είναι δεσμευμένα τα objects 2 ή 4. Επομένως η ανάλυσή μας είναι may ανάλυση.

4.5 Ασφαλής αφαίρεση προγράμματος

Σκοπός της ανάλυσης είναι οι αντιστιχοίσεις μεταβλητών-θέσεων μνήμης και μεθόδων-παρενεργειών (side effects) να αναπαριστούν κάθε πιθανή εκτέλεση του προγράμματος όντας οι μικρότερες δυνατές.

Οί παρακάτω αντιστοιχίσεις είναι και οι δύο ασφαλείς :

```
x= new¹ A();     //object 1 created here
y= new² A();     //object 2 created here
```

```
x → {1}          x → {1,2}
y → {2}          y → {1,2}
```

4.6 Intra-procedral

Κάθε μέθοδος της οποίας η κλήση βρίσκεται σε κάποια διαδρομή εκτέλεσης από την αρχή του προγράμματος εξετάζεται μία μοναδική φορά, χωρίς πληροφορία της ροή του προγράμματος

4.7 Πρόβλημα ικανοποίησης περιορισμών

<κειμενο>

4.8 Κανόνες Παραγωγής

<κειμενο>

4.9 Side Effects

<κειμενο>

4.10 Choco-solver

<κειμενο>

4.11 DaCapo Benchmarks

<κειμενο>

ΑΝΑΦΟΡΕΣ

[1] Java Language Specification, Java SE 7 Edition
https://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.4.5

[2] SootUp
https://soot-oss.github.io/SootUp/latest/

[3] Soot : a java bytecode optimization framework, Raja Vallée-Rai master thesis