ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Doop-Soot: Parallel Fact Generation

Μούρης Δημήτριος

Επιβλέπων: **Σμαραγδάκης Γιάννης,** Αναπληρωτής Καθηγητής ΕΚΠΑ

ΑΘΗΝΑ

ΣΕΠΤΕΜΒΡΗΣ 2016

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Doop-Soot: Parallel Fact Generation

Μούρης Δημήτριος
Α.Μ.: 1115201200114

ΕΠΙΒΛΕΠΩΝ: **Σμαραγδάκης Γιάννης,** Αναπληρωτής Καθηγητής ΕΚΠΑ

ΠΕΡΙΛΗΨΗ

Παραλληλοποίηση του Fact Generation του Doop. Το Doop χρησιμοποιείται για μπλαμα-πλπαλαμπλ

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Τεκμηρίωση

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: static program analysis, doop: fact generation, soot, πτυχιακές εργασίες, τμήμα πληροφορικής και τηλεπικοινωνιών

Πανεπιστήμιο Αθηνών

ABSTRACT

In this paper, we provide documentation for the LaTeX document class dithesis, which can be used for preparing undergraduate theses at the Department of Informatics and Telecommunications, University of Athens. The class conforms to all requirements imposed by the Library, as of September 2011. My thesis, which was based on the dithesis class, was accepted by the Library sometime during the summer semester of 2011.

SUBJECT AREA: Documentation

KEYWORDS: static program analysis, doop: fact generation, soot, undergraduate thesis, dept. of informatics

University of Athens

Αφιέρωση σε κάποιους.

ACKNOWLEDGEMENTS

Θα ήθελα να ευχαριστήσω τον επιβλέποντα κ. Αλέξη Δελή για τη συνεργασία και τη βοήθεια κατά την εκπόνηση αυτής της πτυχιακής.

Θα ήθελα επίσης να ευχαριστήσω το φίλο μου Μένιο για τις πολύτιμες παρατηρήσεις του σε προκαταρκτικές εκδόσεις του κειμένου.

ΠΕΡΙΕΧΟΜΕΝΑ

ΚΑΤΑΛΟΓΟΣ ΣΧΗΜΑΤΩΝ

ΚΑΤΑΛΟΓΟΣ ΠΙΝΑΚΩΝ

PROLOGUE

Το παρόν έγγραφο δημιουργήθηκε στην Αθήνα, το 2016, στα πλαίσια της τεκμηρίωσης της κλάσσης LaTeX dithesis. Η κλάσση αυτή διανέμεται με την ελπίδα ότι θα αποδειχθεί χρήσιμη, παρόλα αυτά *χωρίς καμιά εγγύηση*: χωρίς ούτε και την σιωπηρή εγγύηση εμπορευσιμότητας ή καταλληλότητας για συγκεκριμένη χρήση. Για περισσότερες λεπτομέρειες, ανατρέξτε στην άδεια LaTeX Project Public License.

1. INTRODUCTION

eisagwgh gia doop kai soot

2. DOOP

Doop is a framework for pointer, or points-to, analysis of Java programs. Doop implements a range of algorithms, including context insensitive, call-site sensitive, and object-sensitive analyses, all specified modularly as variations on a common code base.

2.1 Fact Generation

Doop before running a pointer or points-to analysis, intergrates with Soot to generate the facts. Facts are in Jimple (**J**ava s**imple**), a typed 3-address IR suitable for performing optimizations, it only has 15 statements.

2.2 Doop-Nexgen Time Examples

Soot 2.5.0	**antlr.jar**	**hsqldb.jar**	**batik.jar**
Fact Generation	1.16 min.	1.23 min.	2.26 min.
Total time	3.18 min.	3.21 min.	4.34 min.

Πίνακας 1: Soot 2.5.0 times

3. SOOT

Originally, Soot started off as a Java optimization framework. By now, researchers and practitioners from around the world use Soot to analyze, instrument, optimize and visualize Java and Android applications.

3.1 Bytecode To Jimple

Soot is able to translate Java bytecode to a typed 3-address IR, Jimple. Jimple (Java Simple) is a very convinient IR for performing optimizations, it only has 15 statements.

First step is a naive translation to untyped Jimple with new local variables. Then Types are inferred to the untyped jimple. An important step is the linearization of expressions to statements only reference at most 3 local variables or constants. The local variables which start with a $ sign represent stack positions and not local variables in the original program whereas those without $ represent real local variables.

We now describe how Soot handles Java bytecode classes. In a typical case, Soot is launched by specifying the target directory as a parameter. This directory contains the code of the program to analyze, called Application Code (only Java bytecode in this example). First, the main() method of the Main class is executed and calls Scene.loadNecessaryClass
This method loads basic Java classes and then loads specific Application classes by calling loadClass(). Then, SootResolver.resolveClass() is called. The resolver calls SourceLocator.ge
to fetch a reference to a ClassSource, an interface between the file containing the Java bytecode and Soot. In our case the class source is a CoffiClassSource because it is the coffi module which handles the conversion from Java bytecode to Jimple. When the resolver has a reference to a class source, it calls resolve() on it. This methods in turn calls soot.coffi.Util.resolveFromClassFile() which creates a SootClass from the corresponding Java bytecode class. All source fields of Soot class' methods are set to refer to a CoffiMethodSource object. This object is used later to get the Jimple representation of the method. For instance, if during an analysis with Soot the analysis code calls SootMethod.getActiveBody() and the Jimple code of the method was not already generated, getActiveBody() will call CofficMethodSource.g
to compute Jimple code from the Java bytecode. The Jimple code representation of the method can then be analyzed and/or transformed.

4. FOUR APPROACHES

Abstract: Linear Fact Generation

```
public class FactGenerator {
    /*  ...  */

    public void generate(sootClass) {
        if (c.hasSuperclass() && !c.isInterface () )
            _writer .writeDirectSuperclass(c,  c.getSuperclass());
        for(SootField f  :  c.getFields () )
            generate(f);
        for(SootMethod m : c.getMethods()) {
            Session session = new Session();
            generate(m, session);
        }
    }

    public void generate(SootMethod m, Session session) {
        /*  ...  */

        /*  This  instruction  spends more than 80% of FG time */
        m.retrieveActiveBody()

        /*  ...  */
    }

    /*  ...  */
}
```

Σχήμα 1: Linear Fact Generation

4.1 One Thread Per Method

Our first approach to parallelize Fact Generation. Similar as the linear one, but instead of having a loop over all Soot Methods, we create a runnable for each one of them.

```
public class FactGenerator {
    private ExecutorService MgExecutor = new ThreadPoolExecutor(8, 16, 0L,
        TimeUnit.MILLISECONDS, new LinkedBlockingQueue<Runnable>());
    /* ... */

    public void generate(sootClass) {
        if (c.hasSuperclass() && !c.isInterface () )
            _writer .writeDirectSuperclass(c, c.getSuperclass());
        for(SootField f : c.getFields () )
            generate(f);
        for(SootMethod m : c.getMethods()) {
            Session session = new Session();
            Runnable mg = new MethodGenerator();
            MgExecutor.execute(mg);
        }
    }
}

public class MethodGenerator {
    public void run() {
        generate(this.m, this.s)
    }

    /* ... */
}
```

Σχήμα 2: One Thread Per Method

4.2 One Thread Per Class

We observed that some threads did not have much work to do, and finishing instantly. All those new allocations and assignments were an overhead. So, in this approach we tried to feed the threads more and we create a new thread for each class, not for each method.

```
public class FactGenerator {
    private ExecutorService CgExecutor = new ThreadPoolExecutor(8, 16, 0L,
        TimeUnit.MILLISECONDS, new LinkedBlockingQueue<Runnable>());
    /*  ...  */

    public void generate(sootClass) {
        Runnable cg = new ClassGenerator();
        CgExecutor.execute(cg);
    }
}

public class ClassGenerator {
    public void run()  {
         if (c.hasSuperclass() && !c.isInterface () )
             _writer .writeDirectSuperclass(c,  c.getSuperclass());
        for(SootField f  :  c.getFields () )
            generate(f);
        for(SootMethod m : c.getMethods()) {
            Session session = new Session();
            Runnable mg = new MethodGenerator();
            MgExecutor.execute(mg);
            generate(m, session);
        }
    }

    /*  ...  */
}
```

Σχήμα 3: One Thread Per Class

4.3 Fork-Join Framework

The fork/join framework is an implementation of the ExecutorService interface that helps you take advantage of multiple processors. It is designed for work that can be broken into smaller pieces recursively. The goal is to use all the available processing power to enhance the performance of your application.

The center of the fork/join framework is the ForkJoinPool class, an extension of the AbstractExecutorS class. ForkJoinPool implements the core work-stealing algorithm and can execute ForkJoinTask processes.

The idea of using the fork/join framework is to write code that performs a segment of the work. The basic structure should be like the following pseudocode.

```
if (my portion of the work is small enough) {
    do the work directly
} else {
    split my work into two pieces
    invoke the two pieces and wait for the results
}
```

Σχήμα 4: Fork-Join Basic-Use

```
public class FactGenerator {
    private ForkJoinPool classGeneratorPool = new ForkJoinPool();
    /* ... */
    public void generate(sootClass) {
        if (c.hasSuperclass() && !c.isInterface())
            _writer.writeDirectSuperclass(c, c.getSuperclass());
        for(SootClass i : c.getInterfaces())
            _writer.writeDirectSuperinterface(c, i);
        for(SootField f : c.getFields())
            generate(f);
        if (c.getMethods().size() > 0) {
            ClassGenerator classGenerator = new ClassGenerator(_writer,
                _ssa, c, 0, c.getMethods().size());
            classGeneratorPool.invoke(classGenerator);
        }
    }
}

public class ClassGenerator {
    /* ... */
    public void compute() {
        List<SootMethod> sootMethods = _sootClass.getMethods();
        /* if (my portion of the work is small enough) */
        if (_to - _from < threshold) {
            for (int i = _from ; i < _to ; i++) {
                SootMethod m = sootMethods.get(i);
                Session session = new Session();
                generate(m, session);
            }
        } else { /* split work*/
            int half = (_to - _from)/2;
            ClassGenerator c1 = new ClassGenerator(_writer, _ssa,
                _sootClass, _from, _from + half);
            ClassGenerator c2 = new ClassGenerator(_writer, _ssa,
                _sootClass, _from + half, _to);
            invokeAll(c1, c2);
        }
    }
    /* ... */
}
```

Σχήμα 5: Fork-Join Framework

4.4 Multiple Classes Per Thread

Similar as the second approach, but insted of having one thread per class, now we have one thread per multiple classes. Some threads in the previous approaches did not have much work to do.

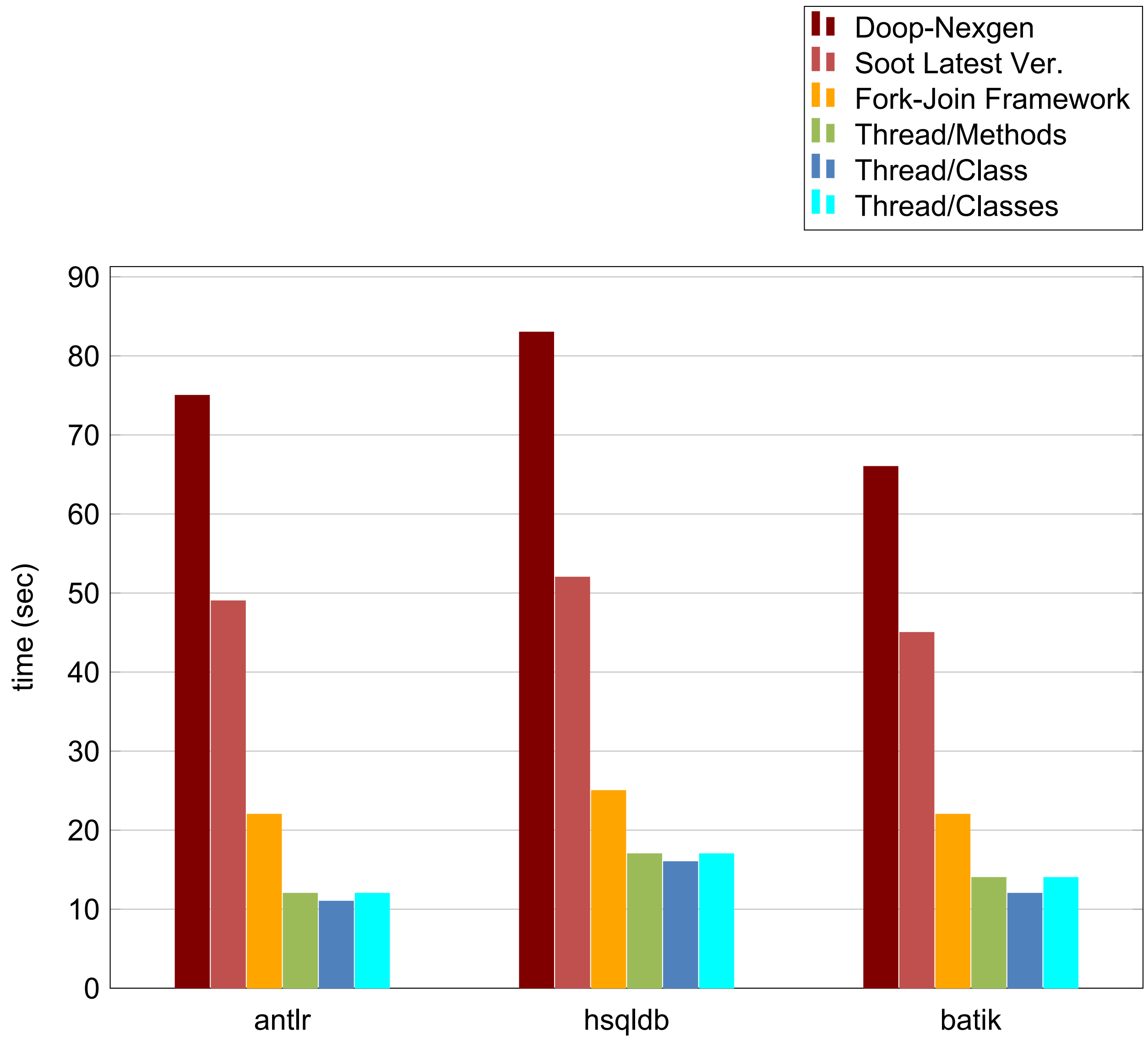
```
public class Driver {
    public Driver(ThreadFactory factory, boolean ssa, int totalClasses) {
        _factory = factory;
        _ssa = ssa;
        _classCounter = 0;
        _sootClasses = new ArrayList<>();
        _totalClasses = totalClasses;
        _cores = Runtime.getRuntime().availableProcessors();
        _executor = new ThreadPoolExecutor(_cores/2, _cores, 0L,
            TimeUnit.MILLISECONDS, new
            LinkedBlockingQueue<Runnable>());
    }

    public void doInParallel( List<SootClass> sootClasses) {
        for(SootClass c : sootClasses)
            generate(c);
        _executor.shutdown();
        _executor.awaitTermination(Long.MAX_VALUE,
            TimeUnit.NANOSECONDS);
    }

    void generate(SootClass _sootClass) {
        _classCounter++;
        _sootClasses.add(_sootClass);
        if ((_classCounter % _classSplit == 0) || (_classCounter +
            _classSplit-1 >= _totalClasses)) {
            Runnable runnable = _factory.newRunnable(_sootClasses);
            _executor.execute(runnable);
            _sootClasses = new ArrayList<>();
        }
    }
}

public class ThreadFactory {
    /* ... */
    public Runnable newRunnable(List<SootClass> sootClasses) {
        if (_makeClassGenerator)
            return new FactGenerator(_factWriter, _ssa, sootClasses);
        else
            return new FactPrinter(_ssa, _toStdout, _outputDir, _printWriter ,
                sootClasses);
    }
}
```

Σχήμα 6: Multiple Classes Per Thread

5. LOCKING

Threads and locks blah blah blah

6. TIME RESULTS



Σχήμα 7: Fact Generation Time Results