NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

UNDERGRADUATE THESIS

Doop-Soot: Parallel Fact Generation

Mouris Dimitris

Supervisors: **Smaragdakis Yannis,** Associate Professor NKUA
Antoniadis Anastasis, M.Sc. Student NKUA

ATHENS

JULY 2016

UNDERGRADUATE THESIS

Doop-Soot: Parallel Fact Generation

Mouris Dimitris
R.N.: 1115201200114

SUPERVISORS: **Smaragdakis Yannis,** Associate Professor NKUA
Antoniadis Anastasis, M.Sc. Student NKUA

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Doop-Soot: Παραλληλοποίηση της Δημιουργίας Γεγονότων

Μούρης Δημήτρης

Supervisors: **Smaragdakis Yannis,** Associate Professor NKUA
Antoniadis Anastasis, M.Sc. Student NKUA

ΑΘΗΝΑ

ΙΟΥΛΙΟΣ 2016

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Doop-Soot: Παραλληλοποίηση της Δημιουργίας Γεγονότων

Μούρης Δημήτρης
Α.Μ.: 1115201200114

SUPERVISORS: **Smaragdakis Yannis,** Associate Professor NKUA
Antoniadis Anastasis, M.Sc. Student NKUA

ΠΕΡΙΛΗΨΗ

Παραλληλοποίηση του Fact Generation του Doop. Το Doop χρησιμοποιείται για μπλαμαπλπαλαμπλ

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Τεκμηρίωση

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

Πανεπιστήμιο Αθηνών

ABSTRACT

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

SUBJECT AREA: Documentation

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

University of Athens

Αφιέρωση σε κάποιους.

ACKNOWLEDGEMENTS

blahblahblahblah

blahblahblahblah

blahblahblahblah

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PROLOGUE

Το παρόν έγγραφο δημιουργήθηκε στην Αθήνα, τον Φεβρουάριο του 2016.

blahblahblahblah

blahblahblahblah

1. INTRODUCTION

eisagwgh gia doop kai soot

2. DOOP

From Doop Website:

"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."

"Doop builds on the idea of specifying pointer analysis algorithms declaratively, using Datalog: a logic-based language for defining (recursive) relations. Doop carries the declarative approach further than past work by describing the full end-to-end analysis in Datalog and optimizing aggressively through exposition of the representation of relations (for example indexing) to the Datalog language level. Doop uses the Datalog dialect and engine of LogicBlox."

Compared to alternative context-sensitive pointer analysis implementations (such as Paddle) Doop is much faster, and scales better. Also, with comparable context-sensitivity features, Doop is more precise in handling some Java features (for example exceptions) than alternatives. //kati prepei na alla3ei edw..

Doop is launched by specifying the type of analysis to run and the target directory that contains java bytecode (.jar file). First doop generates the facts and imports them into a database -or more precisely a knowlegdebase- and then it runs the analysis that was asked to. [1]

2.1 Fact Generation

Doop before running a pointer or points-to analysis, intergrates with Soot to generate either Jimple (**J**ava s**imple**) or Shimple (a **S**sa version of J**imple**) intermidiate representations. Jimple is a typed 3-address IR suitable for performing optimizations; it only has 15 statements. Then from jimple the facts are generated and imported into a database with multiple tables. Shimple is a SSA-version of jimple; obviously first jimple is generated and then Soot applies a transformation pack to jimple body to create shimple.

The main problem is that fact generation takes more than 50% of total execution time (either jimple or shimple). Below are presented a few time examples of the sequential Doop Fact Generation and total execution time (both fact generation and analysis times).

2.2 Doop 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.

Table 1: Soot 2.5.0 times

2.3 Fact Table Example

Here are presented some of the facts that created for the *helloWorld* example.

```
facts/ActualParam.facts:0  helloWorld.main/java.io.PrintStream.println/0
      helloWorld.main/$stringconstant0
facts/ApplicationClass.facts:helloWorld
facts/AssignHeapAllocation.facts:helloWorld.main/invoke/instruction3 3   helloWorld
      helloWorld.main/$stringconstant0 <helloWorld: void main(java.lang.String[])>
facts/AssignLocal.facts:helloWorld.<init>/definition/instruction1  1   helloWorld.<init>/@this
      helloWorld.<init>/r0  <helloWorld: void <init>()>
facts/AssignLocal.facts:helloWorld.main/definition/instruction1     1
      helloWorld.main/@param0 helloWorld.main/r0 <helloWorld: void main(java.lang.String[])>
facts/ClassObject.facts:<class helloWorld>   java.lang.Class    helloWorld
facts/ClassType.facts:helloWorld
/*  ...  */
```

Figure 1: Facts Example

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. Soot provides four intermediate representations for analyzing and transforming Java bytecode:

1. Baf: a streamlined representation of bytecode which is simple to manipulate.
2. Jimple: a typed 3-address intermediate representation suitable for optimization.
3. Shimple: an SSA variation of Jimple.
4. Grimp: an aggregated version of Jimple suitable for decompilation and code inspection.
5. Jimple is Soot's primary IR and most analyses are implemented on the Jimple level. Custom IRs may be added when desired.

In our case, we use Soot to generate Jimple (or Shimple) facts from Java bytecode to run a pointer or points-to analysis. [2]

3.1 Applying the latest Soot version

To minimize the time consumed in fact generation phase, before we tried to parallelize it, we applied the latest (Sept. 2015) Soot version. We faced a lot of compatibility problems in order to use it properly but it gave a speed up of 40%. Below are presented a few time examples with Soot-2.5.0 and the latest one.

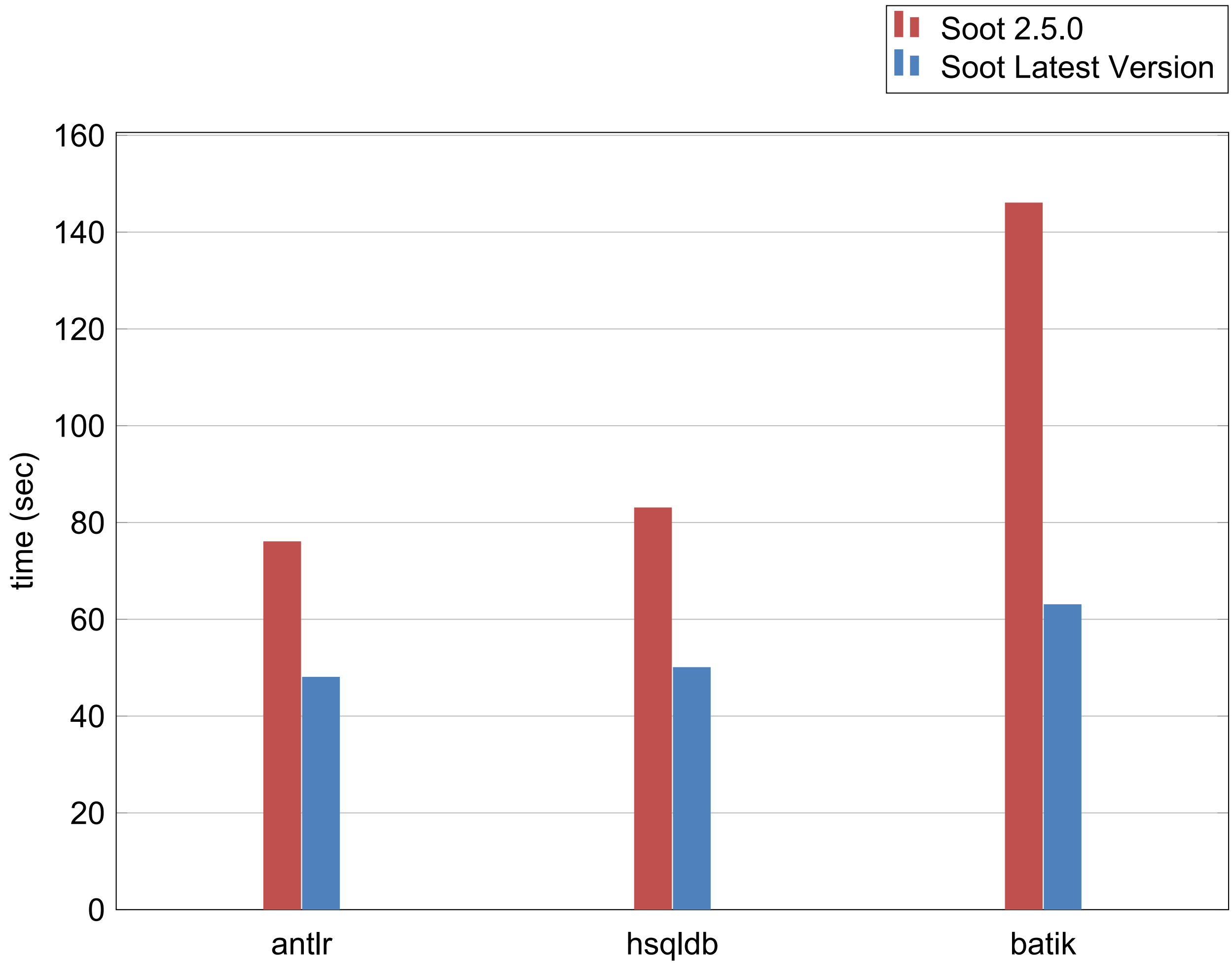

Figure 2: Fact Generation with different Soot versions

Then, to gain more speedup we tried to parallelize the fact generation part, which spends a similar amount of a simple analysis time. In order to do that, we had to understand the way bytecode is translated to jimple. Below we explain in more detail this procedure.

3.2 Bytecode To Jimple

As we mentioned before, Soot is able to translate Java bytecode to a typed 3-address IR, Jimple. Jimple (**J**ava s**imple**) is a very convinient IR for performing optimizations, it only has 15 statements.

Soot has various phases and a lot of different options for transformations given. The one that is responsible for bytecode to jimple translation is jb phase. In this phase, first Soot translates bytecode to untyped Jimple and introduces new local variables; Jimple is stackless, Soot is using variables for stack locations. Then it inferres Types to the untyped jimple. The next step is to linearize all the expressions to statements that only reference at most 3 local variables or constants.

Getting a little deeper, in a general case the way Soot handles Java bytecode classes is the following:
Soot is launched by specifying a directory with the Application code as a parameter. In

this example only Java bytecode, either a class file or a jar. First, the *main()* method of the Main class is executed and calls *Scene.loadNecessaryClasses()* (In our case Doop calls it). This method loads basic Java classes and then loads specific Application classes by calling *loadClass()*. Then, *SootResolver.resolveClass()* is called. The resolver calls *SourceLocator.getClassSource()* 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. Then, the resolver having a reference to a class source, 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 example, if during an analysis with Soot the analysis code calls *SootMethod.getActive* and the Jimple code of the method was not already generated, *getActiveBody()* will call *CofficMethodSource.getBody()* to compute Jimple code from the Java bytecode. The Jimple code representation of the method can then be analyzed and/or transformed.

3.3 Compiling & Running Soot

Soot uses ant [4] for compiling and building the project. So with minor modifications in the *ant.settings* file, is very easy to compile and run Soot.

```
ant              /* To compile */
ant classesjar   /* To generate the sootclasses jar file */
ant fulljar      /* To generate the complete soot jar file */
```

Figure 3: Compiling Soot

Running Soot and generating Jimple from java bytecode or from a jar file is presented below. (For Shimple IR, -ssa flag is needed).

```
(create a test.java)
javac test.java
java -cp ./lib/soot-trunk.jar soot.Main -f J -cp
    .:/usr/lib/jvm/java-7-openjdk-amd64/jre/lib/rt.jar test
```

Figure 4: Generating Jimple from .class

```
java -cp ./lib/soot-trunk.jar soot.Main -f J -cp
    .:/usr/lib/jvm/java-7-openjdk-amd64/jre/lib/rt.jar -process-dir pathtotest.jar
```

Figure 5: Generating Jimple from .jar

3.4 Jimple Examples

Below are two simple java programs along with their jimple translation. The first one is the classic HelloWorld, and the second is a simple inheritance test that depends on the user's input. 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.

3.4.1 Hello World

```
public class helloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World");
    }
}
```

Figure 6: HelloWorld.java

```
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;
    }

}
```

Figure 7: HelloWorld.jimple

3.4.2 Inheritance Test

```
public class inheritanceTest {
    public static void main(String[] args) {
        testA a;
        if (args.length < 1) {
            a = new testA(5);
        } else {
            a = new testB(5);
        }
        int result = a.getA();
        System.out.println("the value of a is " + result);
    }

    public static class testA {
        int a;

        public testA(int a) {
            this.a = a;
        }

        public int getA() {
            return this.a;
        }
    }

    public static class testB extends testA {
        public testB(int a) {
            super(a+100);
        }
    }
}
```

Figure 8: inheritanceTest.java

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

    public static void main(java.lang.String []) {
        java.lang.String [] r0;
        int $i0, i1;
        inheritanceTest$testA $r1, r2;
        inheritanceTest$testB $r3;
        java.io.PrintStream $r4;
        java.lang.StringBuilder $r5, $r6, $r7;
        java.lang.String $r8;
        r0 := @parameter0: java.lang.String[];
        $i0 = lengthof r0;
        if $i0 >= 1 goto label1;
        $r1 = new inheritanceTest$testA;
        specialinvoke $r1.<inheritanceTest$testA: void <init>(int)>(5);
        r2 = $r1;
        goto label2;
     label1:
        $r3 = new inheritanceTest$testB;
        specialinvoke $r3.<inheritanceTest$testB: void <init>(int)>(5);
        r2 = $r3;
     label2:
        i1 = virtualinvoke r2.<inheritanceTest$testA: int getA()>();
        $r4 = <java.lang.System: java.io.PrintStream out>;
        $r5 = new java.lang.StringBuilder;
        specialinvoke $r5.<java.lang.StringBuilder: void <init>()>();
        $r6 = virtualinvoke $r5.<java.lang.StringBuilder: java.lang.StringBuilder
            append(java.lang.String)>("the value of a is ");
        $r7 = virtualinvoke $r6.<java.lang.StringBuilder: java.lang.StringBuilder
            append(int)>(i1);
        $r8 = virtualinvoke $r7.<java.lang.StringBuilder: java.lang.String toString()>();
        virtualinvoke $r4.<java.io.PrintStream: void println(java.lang.String)>($r8);
        return;
    }
}
```

Figure 9: inheritanceTest.jimple

4. PARALLELIZING FACT GENERATION

We now describe the basic idea of Fact Generation from Doop side. Given all the classes (sootClasses) to generate, Doop iterates each one of them; writes all the superClasses, if exist, and then generate all fields (sootFields) and methods (sootMethods). Below is presented the *FactGenerator.java* which implements the work described above and then calls Soot.

```
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()

        /*  ...  */
    }

    /*  ...  */
}
```

Figure 10: Sequential Fact Generation

Having the previous basic structure in mind, and considering that *m.retrieveActiveBody()* spends more than 80% of total fact generation time, we tried to parallelize the method which calls *m.retrieveActiveBody()*. In order to do that, we approached the problem in four ways. The three of them are pretty much similar while the other one is based on a recursive Java Framework (Fork/Join Framework). Below are presented some Fact Generation time examples with the sequential FG.

Jars	Time (sec.)
antlr	48
eclipse	27
jython	32
hsqldb	50
batik	63

Table 2: Sequential Fact Generation Time Examples

4.1 One Thread Per Method

Our first approach to parallelize Fact Generation is similar as the sequential one, but instead of having a loop over all Soot Methods and call *generate(m, session)*, we assign the task to a thread for each one of them. We created a new Java class, MethodGenerator, which is identical to FactGenerator and in addition has a *run()* method to generate sootMethods.

```
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)
    }

    /*  ...  */
}
```

Figure 11: One Thread Per Method

Below are presented some Fact Generation time examples with the *One Thread Per Method FG* approach for various thread-pool sizes (such as 4, 16 and 32).

Jars	Time (sec.)		
Pool Size	**4**	**16**	**32**
antlr	21	14	13
eclipse	13	7	8
jython	14	9	9
hsqldb	23	15	16
batik	26	23	18

Table 3: One Thread Per Method Time Examples, with pool size: 4, 16, 32

4.2 One Thread Per Class

In our second approach, we tried to find out ways to gain more speedup. So, we observed that some threads did not have much work to do and finishing their task instantly. Allocating a new object and assigning to it a task just to finish instantly was an overhead. As a result, we tried to feed the threads more than just a method, so we created 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);
        }
    }

    /* ... */
}
```

Figure 12: One Thread Per Class

The results were slightly better than the previous but without achiving a remarkable speedup.

Below are presented some Fact Generation time examples with the *One Thread Per Class FG* approach for various thread-pool sizes (such as 4, 16 and 32).

Jars	**Time (sec.)**		
Pool Size	**4**	**16**	**32**
antlr	21	14	13
eclipse	13	7	8
jython	14	9	9
hsqldb	23	15	16
batik	26	23	18

Table 4: One Thread Per Class Time Examples, with pool size: 4, 16, 32

4.3 Fork-Join Framework

So in order to achive more speedup we tried a completely different approach than the two previous, a recursive Java FrameWork (Fork/Join framework). As Oracle describes in Java Documentation, 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 AbstractExecutorService 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.” [3]

```
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
}
```

Figure 13: 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) { /* How many classes can I process? */
            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);
        }
    }
    /* ... */
}
```

Figure 14: Fork-Join Framework

The results were worse than the two previous approaches (still better than then sequential approach). Below are presented some Fact Generation time examples with the *Fork/Join Framework FG* approach for threshold values (such as 2, 3 and 4). Threshold actually is the number of classes for a thread to process.

Jars	**Time (sec.)**		
Threshold (classes to generate)	**2**	**3**	**4**
antlr	23	25	25
eclipse	13	15	18
jython	16	17	19
hsqldb	25	28	31
batik	34	37	38

Table 5: Fork/Join Time Examples, with threshold 2, 3, 4 and pool size 16

4.4 Multiple Classes Per Thread

Our last and final approach is similar as the second one, but instead of having one thread per class, we now have one thread per multiple classes. Even in the second approach some threads did not have much work to do. Below is presented in an abstract way the final stage of the code implementing the *Multiple Classes Per Thread* approach.

```
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);
    }
}
```

Figure 15: Multiple Classes Per Thread

Getting away from the overhead produced by assignments and allocations of the first and second approach, this one had so far the best time results. Below are presented some Fact Generation time examples with the *Multiple Classes Per Thread FG* approach for various thread-pool sizes (such as 4, 16 and 32) and various number of classes per thread.

Jars	Classes Per Thread Pool Size	Time (sec.) 2	3	4
antlr	**4**	22	18	19
	16	13	12	14
	32	14	13	13
eclipse	**4**	12	10	11
	16	7	8	6
	32	8	8	8
jython	**4**	13	14	13
	16	11	7	9
	32	9	8	8
hsqldb	**4**	25	22	20
	16	17	14	16
	32	16	18	14
batik	**4**	23	24	25
	16	22	20	17
	32	21	17	17

Table 6: Multiple Classes Per Thread Time Examples, with pool size: 4, 16, 32 and classes per thread: 2, 3, 4

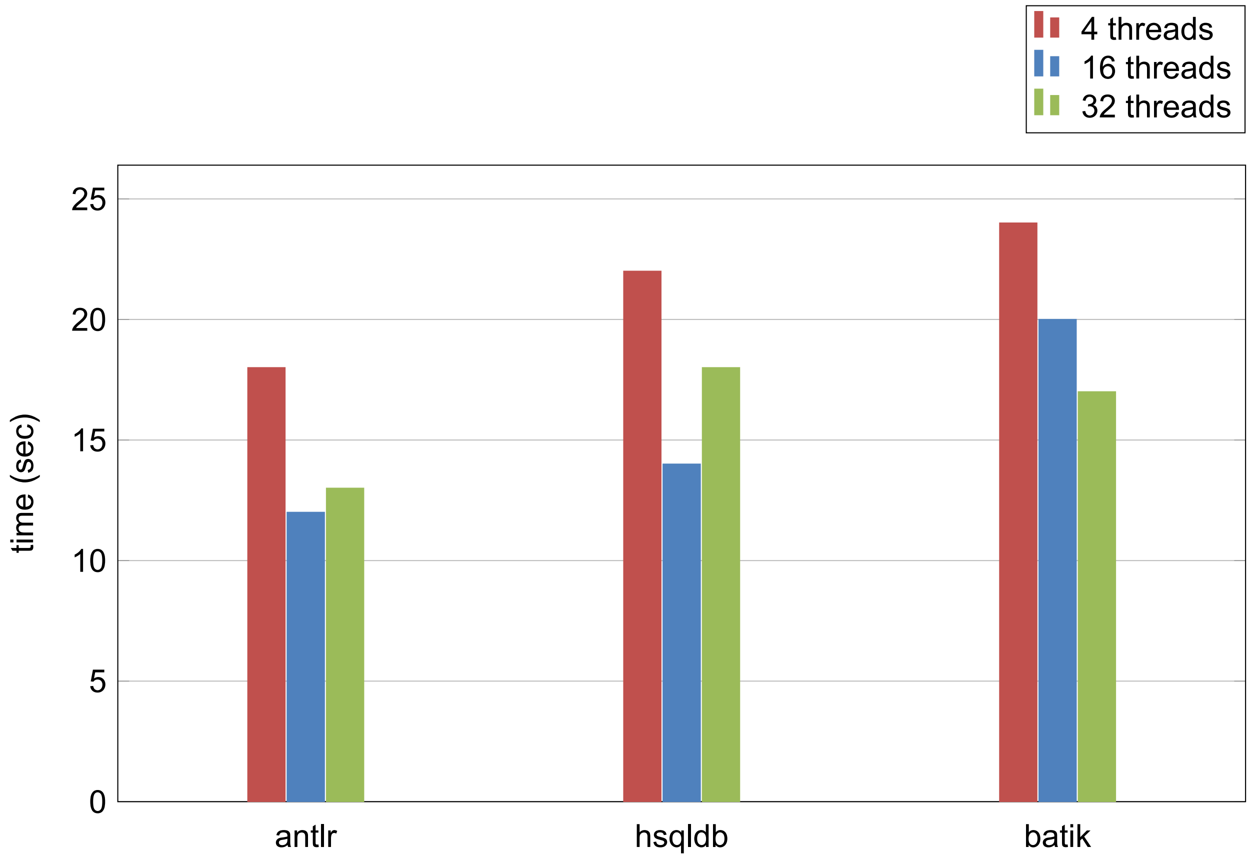

Figure 16: Increasing the number of threads with 3 classes per thread

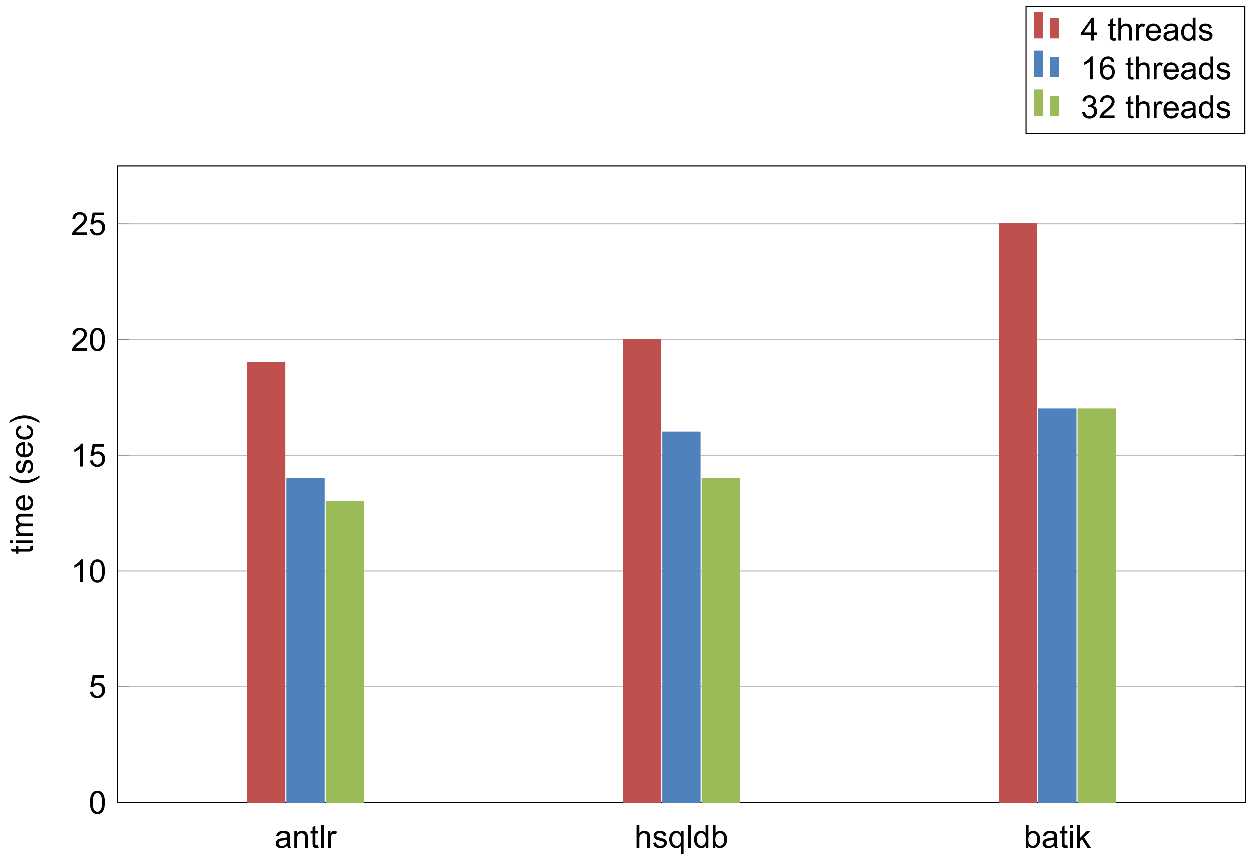

Figure 17: Increasing the number of threads with 4 classes per thread

5. LOCKING

Along with threads come locks. An incorrect use of locking could have very negative(??) effects on the results. Having more locks than we actually needed would had lead to a time result as slow as the sequential one. Having less, would had lead to races and deadlocks. So, locks should be used very consciously.

Luckily, in our case we used very few locks in both Doop and Soot. Below we explain in more detail.

5.1 Doop Side

The only two things we had to lock to prevent races and deadlocks were an access to the output file and three methods that accessing a sootMethod.

5.1.1 CSVDatabase

Lock file to prevent more than one thread to write at the same time.

```
synchronized(predicateFile) {
    Writer writer = getWriter(predicateFile);
    addColumn(writer, arg, shouldTruncate);
    for (Column col : args)
        addColumn(writer.append(SEP), col, shouldTruncate);
    writer.write(EOL);
}
```

Figure 18: CSVDatabase.java

5.1.2 Represantation

The other synchronization we had to provide was in three methods that were accessing SootMethods while threads were active.

```
public synchronized String signature(SootMethod) { /*...*/ }
public synchronized String handler(SootMethod, Trap, Session) { /*...*/ }
public synchronized String compactMethod(SootMethod) { /*...*/ }
```

Figure 19: Represantation.java

5.2 Soot Side

The way Soot is implemented, it has a class that encloses all global objects (*G.java*). As we mentioned before, Soot does much more than just translating bytecode to jimple, so it has various phases and a lot of different options for transformations given. The phase that translates bytecode to jimple is jb phase.

5.2.1 Type Assigner

The class *JimpleBodyPack* applies the transformations corresponding to the given options. In our case, it applies a *"jb.tr"* which means *"jimple body transformation"*. From bytecode to jimple translation this is the only pack needed, so we lock before applying Type Assigner and unlock afterwards.

```
lock.lock();
PackManager.v().getTransform("jb.tr").apply(b);
lock.unlock();
```

Figure 20: JimpleBodyPack.java

5.2.2 Pack Manager

The class that manages the Packs containing the various phases and their options is *PackManager*. Lock before retrieving Class Hierarchy Analysis, unlock afterwards.

```
lock.lock();
p.add(new Transform("cg.cha", CHATransformer.v()));
p.add(new Transform("cg.spark", SparkTransformer.v()));
p.add(new Transform("cg.paddle", PaddleHook.v()));
lock.unlock();
```

Figure 21: PackManager.java

5.2.3 Shimple -ssa

We also use Soot to provide **S**h**imple** which is an **SSA** variation of Jimple. Shimple generation is very similar to jimple but with a few differences. blahblabh prepei na to dw kiallo auto The class that handles the translation from jimple to shimple is *Shimple.java*. So far we have synchronized all Shimple-Body-creation methods.

6. TIME RESULTS

Below are presented fact generation times for Soot 2.5.0, latest Soot version and all our approaches.

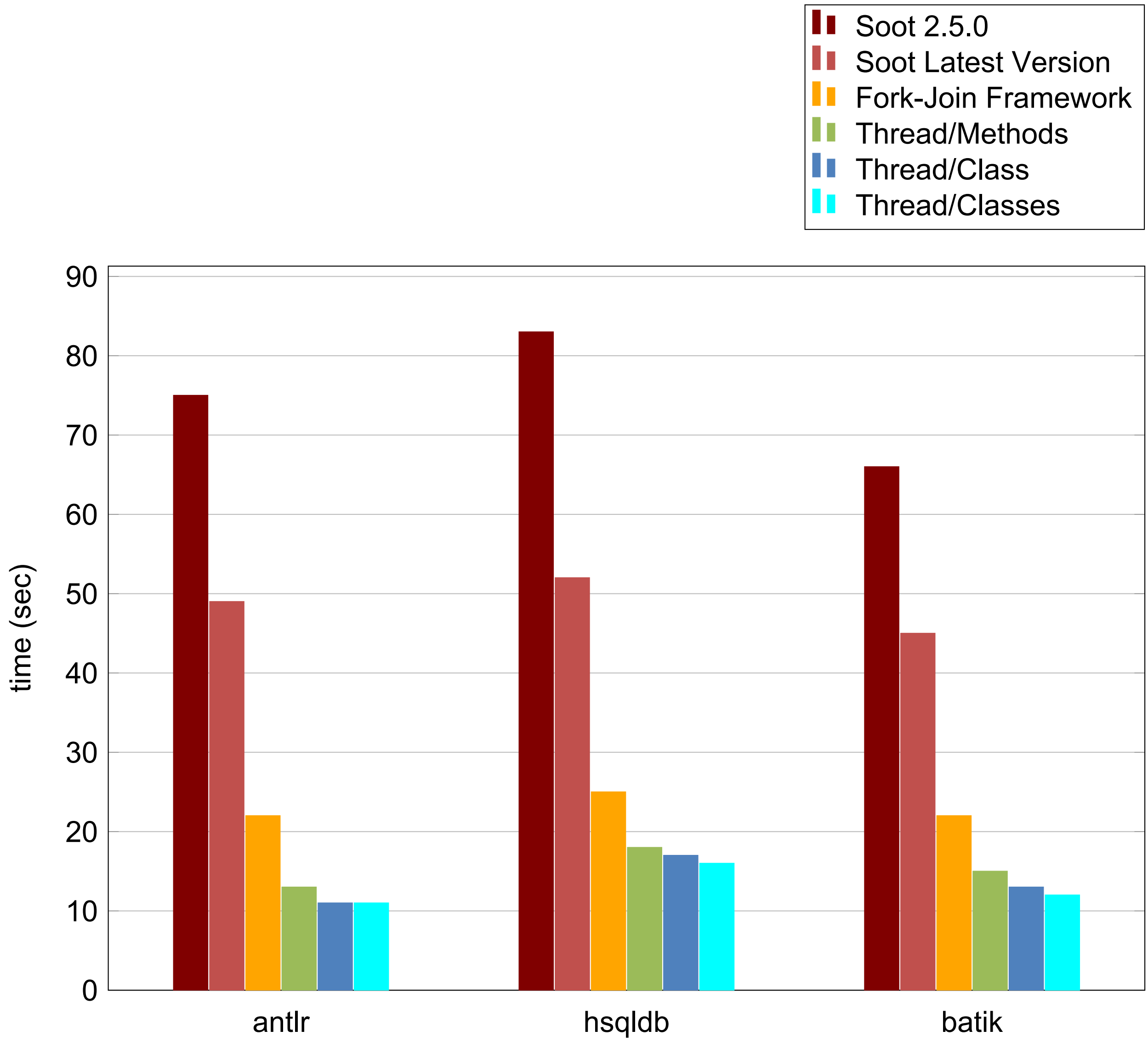

Figure 22: Fact Generation Time Results

In conclusion, just changing the soot version and applying the latest one gave a speedup up to 30-40%. With the latest soot-version and our best approach we achived a speedup of 60-80%. Our third attempt (Fork/Join Framework) was not a successful one, it was better than the sequential fact generation (obviously) but not as good as the other three.

Jars	Time (sec.)				
Approach	**Sequential**	**Fork/Join**	**Thread/Method**	**Thread/Class**	**Thread/Classes**
antlr	48	23	13	13	12
eclipse	27	13	7	7	6
jython	32	16	8	8	7
hsqldb	50	25	15	14	14
batik	63	34	18	17	17

Table 7: Summarizing best times of all approaches with pool size 16-32

ACRONYMS AND ABBREVIATIONS

IR	Intermediate Representation
SSA	Static Single Assignment
Jimple	Soot typed 3-address IR
Shimple	An SSA-version of Jimple
FG	Fact Generation
jb	Jimple Body

REFERENCES

[1] ”Doop: Framework for Java Pointer Analysis” [Online] Available: http://doop.program-analysis.org/
[2] ”Sable: Soot” [Online] Available: https://sable.github.io/soot/
[3] ”Oracle Java Fork/Join Framework” [Online] Available: https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html
[4] ”Apache Ant” [Online] Available: http://ant.apache.org/