NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

UNDERGRADUATE THESIS

Doop-Soot: Parallel Fact Generation

Dimitris I. Mouris

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

ATHENS

MAY 2016

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Doop-Soot: Παραλληλοποίηση της Δημιουργίας Γεγονότων

Δημήτρης Η. Μούρης

Επιβλέποντες: **Γιάννης Σμαραγδάκης,** Αναπληρωτής Καθηγητής ΕΚΠΑ
Αναστάσης Αντωνιάδης, Μεταπτυχιακός Φοιτητής ΕΚΠΑ

ΑΘΗΝΑ

ΜΑΙΟΣ 2016

UNDERGRADUATE THESIS

Doop-Soot: Parallel Fact Generation

Dimitris I. Mouris
R.N.: 1115201200114

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

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Doop-Soot: Παραλληλοποίηση της Δημιουργίας Γεγονότων

Δημήτρης Η. Μούρης
Α.Μ.: 1115201200114

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Γιάννης Σμαραγδάκης,** Αναπληρωτής Καθηγητής ΕΚΠΑ
Αναστάσης Αντωνιάδης, Μεταπτυχιακός Φοιτητής ΕΚΠΑ

ABSTRACT

The use of deductive databases for declarative program analysis has become increasingly popular in recent years. A typical example is Datalog databases. Generating the initial data that is kept in the database is often as expensive as the program analysis itself and does not scale well as the program size increases.

In this thesis we present a parallel approach of the fact-generation process from Java bytecode in order to perform pointer, or points-to analysis in Java programs using the Doop framework. Our goal is to evaluate the benefits of a parallel implementation compared to the non-parallel one.

SUBJECT AREA: Static program analysis

KEYWORDS: static program analysis, doop framework, soot framework, fact generation, java multi-threading

University of Athens

ΠΕΡΙΛΗΨΗ

Τα τελευταία χρόνια γίνεται όλο και πιο δημοφιλής η χρήση (συν)επαγωγικών βάσεων δεδομένων στη δηλωτική ανάλυση προγραμμάτων. Μια από τις πιο χαρακτηριστικές περιπτώσεις είναι βάσεις δεδομένων που χρησιμοποιούν Datalog. Η παραγωγή της αρχικής πληροφορίας που εισάγεται στη βάση δεδομένων για το αναλυόμενο πρόγραμμα σε πολλές περιπτώσεις είναι εξίσου δαπανηρή σε χρόνο με την ανάλυση του προγράμματος, ειδικά όσο το μέγεθος αυτού αυξάνεται.

Σε αυτή την πτυχιακή παρουσιάζουμε την παραλληλοποίηση της διαδικασίας παραγωγής της αρχικής πληροφορίας από Java bytecode με σκοπό να χρησιμοποιηθεί για ανάλυση δεικτών σε προγράμματα Java από το Doop framework. Ο στόχος μας είναι να αξιολογήσουμε τα οφέλη που προσφέρει μια παράλληλη υλοποίηση σε σχέση με την ακολουθιακή.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Στατική ανάλυση προγραμμάτων

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: στατική ανάλυση προγραμμάτων, doop framework, soot framework, δημιουργία γεγονότων, πολυνηματισμός σε java

Πανεπιστήμιο Αθηνών

To my parents, Ilias & Eirini.

ACKNOWLEDGMENTS

I would like to thank my supervisor, Prof. Yannis Smaragdakis for the chance he gave me with this project and also for his support and help all these months.

My sincere thanks also goes to M.Sc. candidate and my friend Anastasis Antoniadis for his patience and guidance which had been very helpful to finish this work.

May 2016

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

This project was developed in Athens, Greece between September 2015 and March 2016. At the very beginning of this work, it was essential to understand how the Doop framework works for pointer or points-to analysis. Equally important was the task of understanding the codebase of the Soot framework and growing familiar with its integration with Doop in order to translate Java bytecode to an IR (Jimple) and produce the facts for the analysis. The core of this work was focused on understanding the way Java bytecode to Jimple translation is implemented and attempting to parallelize it without disrupting the Doop work-flow.

1. INTRODUCTION

Soot [1] is a Java bytecode optimization framework which my colleagues use for fact generation in order to perform points-to analysis [2] of Java programs, in Datalog, using the Doop framework [3].

This thesis aims to minimize the time consumed by the fact-generation process and also guarantee the integrity and correctness of the generated facts. For this task, we had to parallelize the fact-generation process and proceed to the appropriate modifications in both Soot and Doop.

The rest of the thesis is organized as follows:

1. In Chapter 2 we present the Doop framework.
2. In Chapter 3 we describe the Soot framework which is invoked by Doop to generate the facts.
3. In Chapter 4 we present the four implemented approaches to parallelize fact generation.
4. In Chapter 5 we explain the locks needed to ensure thread safety.
5. In Chapter 6 we present our timing results.
6. In Chapter 7 we summarize our conclusions.

2. DOOP

Doop [3] is a framework for pointer, or points-to, analysis of Java programs. It implements a range of different algorithms such as context insensitive, call-site sensitive, object-sensitive analyses and a lot of other variations of these algorithms.

From the Doop website:
"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."

The advantage of Doop compared to alternative context-sensitive pointer analysis implementations, is that 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.

Doop is launched by specifying the type of analysis to run and the directory that contains java bytecode (.jar file) to analyze. At first, the facts are generated by soot and imported to a database —or more precisely a knowlegdebase— and then the specified analysis is run.

2.1 Fact Generation

Doop, before running a pointer or points-to analysis, invokes Soot to generate either Jimple (**J**ava s**imple**) or Shimple (an **S**sa version of J**imple**) intermediate representations. Jimple is a typed 3-address IR suitable for performing optimizations; it only has 15 statements. Afterward, the facts are generated from Jimple and imported into a database with multiple tables, so the analysis rules can process them. Shimple is an SSA-version of Jimple; first Jimple is generated and then Soot applies a group of transformations to the Jimple body to create Shimple.

The main motivation behind this thesis was the fact that the sequential fact-generation time amounts to more than 50% of the total execution time (both fact generation and analysis), either for Jimple or Shimple.

2.2 Doop Time Examples

Below we present a few timing[1] examples of the sequential Doop fact generation and total execution time (both fact generation and analysis times) with Soot version 2.5.0.

Soot 2.5.0	antlr.jar	hsqldb.jar	batik.jar
Fact generation	1.16 min.	1.23 min.	1.06 min.
Total time	3.18 min.	3.21 min.	3.13 min.

Table 1: Soot 2.5.0 FG timings compared to total execution times.

2.3 Fact Table Example

After the completion of the fact-generation process, Doop executes a set of Datalog rules, which are the specification of the selected analysis. Those Datalog rules are applied on the EDB and keep producing new facts until fix point is reached. At the end of an analysis, a symbolic link is created for the resulting database workspace, and another one at top level, each time pointing to the latest successfully completed analysis.

The fact files consist of tab-separated values, where every column corresponds to an argument of the Datalog predicate. A subset of the facts imported into the database for a simple `helloWorld` example is presented below.

The actual parameters of a method invocation.

`ActualParam(?index, ?param2, ?param3)`

(Assign actual parameter `?param3` to formal parameter `?param2` with index `?index` in a method invocation)

```
1 0
     java.io.PrintStream.requireNonNull/java.lang.NullPointerException.<init>/0
     java.io.PrintStream.requireNonNull/r1
2 0  helloWorld.main/java.io.PrintStream.println/0
     helloWorld.main/$stringconstant0
3 0  java.io.PrintStream.toCharset/java.io.PrintStream.requireNonNull/0
     java.io.PrintStream.toCharset/r0
4 1  java.io.PrintStream.toCharset/java.io.PrintStream.requireNonNull/0
     java.io.PrintStream.toCharset/$stringconstant0
5 /* ... */
```

Figure 1: ActualParam.facts

[1] All the time measurements were performed on a 64-bit machine with two octa-core Intel Xeon E5-2667 (v2) CPUs at 3.30GHz (for a total of 32 logical cores) and 256GB of RAM.

Assign local instruction.

`AssignLocal(?instruction, ?index, ?from, ?to, ?inmethod)`
(Assignment `?to = ?from` in instruction `?instruction` with index `?index` in method `?inmethod`)

```
1 helloWorld.<init>/definition/instruction1 1 helloWorld.<init>/@this
     helloWorld.<init>/r0 <helloWorld: void <init>()>
2 helloWorld.main/definition/instruction1 1 helloWorld.main/@param0
     helloWorld.main/r0 <helloWorld: void main(java.lang.String[])>
3 /* ... */
```

Figure 2: AssignLocal.facts

Class Object.

`ClassObject(?repr, ?type, ?actualtype)`
(Representation `?repr` of type `?type` corresponds to actual type `?actualtype`)

```
1 <class helloWorld>      java.lang.Class helloWorld
2 <class java.io.PrintStream> java.lang.Class java.io.PrintStream
3 <class java.lang.Object> java.lang.Class java.lang.Object
4 <class java.lang.String> java.lang.Class java.lang.String
5 <class java.lang.System> java.lang.Class java.lang.System
6 /* ... */
```

Figure 3: ClassObject.facts

Assign Heap Allocation.

`AssignHeapAllocation(?instruction, ?index, ?heap, ?to, ?inmethod)`
(Assignment `?to = ?heap` in instruction `?instruction` with index `?index` in method `?inmethod`)

```
1 java.io.PrintStream.requireNonNull/assign/instruction4 4
     java.io.PrintStream.requireNonNull/new
     java.lang.NullPointerException/0
     java.io.PrintStream.requireNonNull/$r2 <java.io.PrintStream:
     java.lang.Object requireNonNull(java.lang.Object,java.lang.String)>
2 helloWorld.main/invoke/instruction3 3 helloWorld
     helloWorld.main/$stringconstant0 <helloWorld: void
     main(java.lang.String[])>
3 java.io.PrintStream.toCharset/invoke/instruction2 2 charsetName
     java.io.PrintStream.toCharset/$stringconstant0
     <java.io.PrintStream: java.nio.charset.Charset
     toCharset(java.lang.String)>
4 /* ... */
```

Figure 4: AssignHeapAllocation.facts

Definition of a class type.

`ClassType(?type)`

(Type `?type` is a ClassType)

```
1 helloWorld
2 java.lang.Class
3 java.io.PrintStream
4 java.lang.Class
5 /* ... */
```

Figure 5: ClassType.facts

3. SOOT

Soot [1] is the framework used by Doop to generate a relational representation of the analyzed program. Soot originally started off as a Java optimization framework, but by now it performs a lot of different tasks such as 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, Soot is utilized to generate Jimple (or Shimple) facts from Java bytecode to run a pointer or points-to analysis.

3.1 Applying the latest Soot version

Doop used Soot version 2.5.0 for fact generation. In order to minimize the time consumed by Soot, before we tried to parallelize the fact generation procedure, we applied the latest (*Sept. 2015 develop branch*) Soot version. We faced some minor compatibility issues which were handled in order to have minimal impact on the generated facts, but this change gave a speed up of 145-160%. Below we present a few example timings with Soot-2.5.0 and the latest.

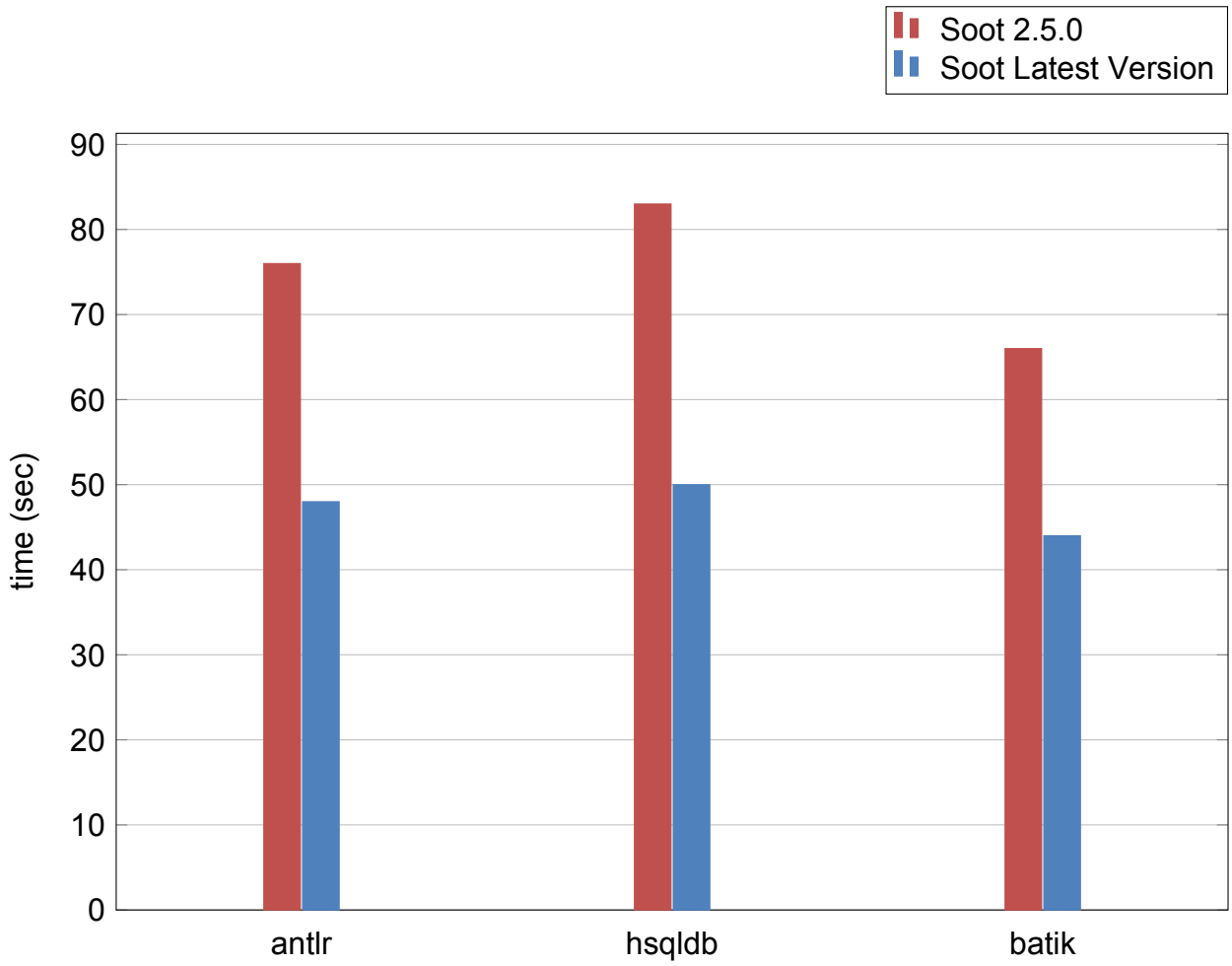

Figure 6: Fact Generation with different Soot versions

Then, to gain more speedup we tried to parallelize the fact generation part, which takes a similar amount of time as an entire simple analysis. 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 mentioned already, Soot is able to translate Java bytecode to a typed 3-address IR, Jimple. Jimple (**J**ava s**imple**) is a very convenient 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 the jb phase. In this phase, Soot first translates bytecode to untyped Jimple and introduces new local variables; Jimple is stackless, Soot is using variables for stack locations. Then it infers types for 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 (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 `Scene.loadNecessaryClasses()` directly and not the Main class). 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 Class-Source, an interface between the file containing the Java bytecode and Soot. For Java bytecode to Jimple translation the class source is a `CoffiClassSource` because it is the coffi module that handles this conversion. Then, the resolver having a reference to a class source, calls `resolve()` on it. This method 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. getActiveBody()` 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. Actually, this method (`getActiveBody()`) occupies the most of the Java bytecode to Jimple conversion time.

The above method, `getActiveBody`, as well as all methods it calls, are the ones we intend to parallelize and make thread-safe. We describe our locking policy in more detail for both Doop and Soot in Chapter 5.

3.3 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 that start with a $ sign represent stack positions and not local variables in the original program whereas those without $ represent real local variables.

3.3.1 Hello World

```
1 public class helloWorld {
2   public static void main(String[] args) {
3       System.out.println("Hello, World");
4   }
5 }
```

Figure 7: HelloWorld.java

```
1  public class helloWorld extends java.lang.Object {
2
3     public void <init>() {
4        helloWorld r0;
5        r0 := @this: helloWorld;
6        specialinvoke r0.<java.lang.Object: void <init>()>();
7        return;
8     }
9
10    public static void main(java.lang.String[]) {
11       java.lang.String[] r0;
12       java.io.PrintStream $r1;
13       r0 := @parameter0: java.lang.String[];
14       $r1 = <java.lang.System: java.io.PrintStream out>;
15       virtualinvoke $r1.<java.io.PrintStream: void
            println(java.lang.String)>("Hello, World");
16       return;
17    }
18
19 }
```

Figure 8: HelloWorld.Jimple

3.3.2 Inheritance Test

```
1  public class inheritanceTest {
2     public static void main(String[] args) {
3        testA a;
4        if (args.length < 1) {
5           a = new testA(5);
6        } else {
7           a = new testB(5);
8        }
9        int result = a.getA();
10       System.out.println("the value of a is " + result);
11    }
12
13    public static class testA {
14       int a;
15
16       public testA(int a) {
17          this.a = a;
18       }
19
20       public int getA() {
21          return this.a;
22       }
23    }
24
25    public static class testB extends testA {
26       public testB(int a) {
27          super(a+100);
28       }
29    }
30 }
```

Figure 9: inheritanceTest.java

```
1  public class inheritanceTest extends java.lang.Object {
2     public void <init>() {
3        inheritanceTest r0;
4        r0 := @this: inheritanceTest;
5        specialinvoke r0.<java.lang.Object: void <init>()>();
6        return;
7     }
8
9     public static void main(java.lang.String[]) {
10       java.lang.String[] r0;
11       int $i0, i1;
12       inheritanceTest$testA $r1, r2;
13       inheritanceTest$testB $r3;
14       java.io.PrintStream $r4;
15       java.lang.StringBuilder $r5, $r6, $r7;
16       java.lang.String $r8;
17       r0 := @parameter0: java.lang.String[];
18       $i0 = lengthof r0;
19       if $i0 >= 1 goto label1;
20       $r1 = new inheritanceTest$testA;
21       specialinvoke $r1.<inheritanceTest$testA: void <init>(int)>(5);
22       r2 = $r1;
23       goto label2;
24    label1:
25       $r3 = new inheritanceTest$testB;
26       specialinvoke $r3.<inheritanceTest$testB: void <init>(int)>(5);
27       r2 = $r3;
28    label2:
29       i1 = virtualinvoke r2.<inheritanceTest$testA: int getA()>();
30       $r4 = <java.lang.System: java.io.PrintStream out>;
31       $r5 = new java.lang.StringBuilder;
32       specialinvoke $r5.<java.lang.StringBuilder: void <init>()>();
33       $r6 = virtualinvoke $r5.<java.lang.StringBuilder:
            java.lang.StringBuilder append(java.lang.String)>("the value
            of a is ");
34       $r7 = virtualinvoke $r6.<java.lang.StringBuilder:
            java.lang.StringBuilder append(int)>(i1);
35       $r8 = virtualinvoke $r7.<java.lang.StringBuilder:
            java.lang.String toString()>();
36       virtualinvoke $r4.<java.io.PrintStream: void
            println(java.lang.String)>($r8);
37       return;
38    }
39 }
```

Figure 10: inheritanceTest.Jimple

4. PARALLELIZING FACT GENERATION

In order to parallelize the fact generation part, we did not change the way facts are generated from Soot. We actually called Soot concurrently from Doop and made Soot thread safe.

We now describe the basic idea of fact generation from the Doop side. Given all the classes (sootClasses) to generate, Doop iterates over each one of them and then generates all fields (sootFields) and methods (sootMethods). Below we show the `FactGenerator.java` which implements the work described above and then calls Soot `retrieveActiveBody`.

```
1  public class FactGenerator {
2     /* ... */
3
4     public void generate(sootClass) {
5        if(c.hasSuperclass() && !c.isInterface())
6           _writer.writeDirectSuperclass(c, c.getSuperclass());
7        for(SootField f : c.getFields())
8           generate(f);
9        for(SootMethod m : c.getMethods()) {
10          Session session = new Session();
11          generate(m, session);
12       }
13    }
14
15    public void generate(SootMethod m, Session session) {
16       /* ... */
17
18       /* This instruction consumes more than 80% of FG time */
19       m.retrieveActiveBody()
20
21       /* ... */
22    }
23
24    /* ... */
25 }
```

Figure 11: Sequential Fact Generation

Having the previous basic structure in mind, and considering that `m.retrieveActiveBody()` occupies more than 80% of total fact-generation time, we tried to parallelize the method that calls `m.retrieveActiveBody()`. In order to do that, we approached the problem in four ways. Three of them are pretty similar while the fourth one is based on a recursive Java Framework (Fork/Join Framework [4]). Below we present some Fact-Generation timings with the sequential FG and the latest Soot version.

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

Table 2: Sequential fact-generation timing examples

4.1 One Thread Per Method

Our first approach to parallelize fact-generation is similar to 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 every method.

```
1  public class FactGenerator {
2     private ExecutorService MgExecutor = new ThreadPoolExecutor(8, 16,
          0L, TimeUnit.MILLISECONDS, new LinkedBlockingQueue<Runnable>());
3     /* ... */
4
5     public void generate(sootClass) {
6        if(c.hasSuperclass() && !c.isInterface())
7           _writer.writeDirectSuperclass(c, c.getSuperclass());
8        for(SootField f : c.getFields())
9           generate(f);
10       for(SootMethod m : c.getMethods()) {
11          Session session = new Session();
12          Runnable mg = new MethodGenerator();
13          MgExecutor.execute(mg);
14       }
15    }
16 }
17
18 public class MethodGenerator {
19    public void run() {
20       generate(this.m, this.s)
21    }
22
23    /* ... */
24 }
```

Figure 12: One Thread Per Method

The results were very encouraging as we achieved a 325-400% speedup compared to the sequential FG (latest Soot version). Below we show some fact-generation timing 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 timing examples, with pool size: 4, 16, 32

4.2 One Thread Per Class

In our second approach, we tried to find ways to gain more speedup. So, we observed that some threads did not have much work to do, they were 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 with more than just one method, so we created a new thread for each class, not for each method.

```
1  public class FactGenerator {
2     private ExecutorService CgExecutor = new ThreadPoolExecutor(8, 16,
          0L, TimeUnit.MILLISECONDS, new LinkedBlockingQueue<Runnable>());
3     /* ... */
4
5     public void generate(sootClass) {
6        Runnable cg = new ClassGenerator();
7        CgExecutor.execute(cg);
8     }
9  }
10
11 public class ClassGenerator {
12    public void run() {
13       if(c.hasSuperclass() && !c.isInterface())
14          _writer.writeDirectSuperclass(c, c.getSuperclass());
15       for(SootField f : c.getFields())
16          generate(f);
17       for(SootMethod m : c.getMethods()) {
18          Session session = new Session();
19          Runnable mg = new MethodGenerator();
20          MgExecutor.execute(mg);
21          generate(m, session);
22       }
23    }
24
25    /* ... */
26 }
```

Figure 13: One Thread Per Class

The results were slightly better than our previous effort, but without achieving a remarkable speedup. Below we present some fact-generation timings 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	20	14	13
eclipse	11	7	7
jython	13	8	8
hsqldb	22	15	18
batik	26	18	19

Table 4: One thread per class timing examples, with pool size: 4, 16, 32

4.3 Fork/Join Framework

In order to achieve more speedup we tried a completely different approach than the two previous ones: a recursive Java Framework (Fork/Join framework [4]). As Oracle describes in Java Documentation, 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 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.”

```
1 if (my portion of the work is small enough) {
2    do the work directly
3 } else {
4    split my work into two pieces
5    invoke the two pieces and wait for the results
6 }
```

Figure 14: Fork/Join Basic-Use

```
1  public class FactGenerator {
2     private ForkJoinPool classGeneratorPool = new ForkJoinPool();
3     /* ... */
4     public void generate(sootClass) {
5        if(c.hasSuperclass() && !c.isInterface())
6           _writer.writeDirectSuperclass(c, c.getSuperclass());
7        for(SootClass i : c.getInterfaces())
8           _writer.writeDirectSuperinterface(c, i);
9        for(SootField f : c.getFields())
10          generate(f);
11       if (c.getMethods().size() > 0) {
12          ClassGenerator classGenerator = new ClassGenerator(_writer,
               _ssa, c, 0, c.getMethods().size());
13          classGeneratorPool.invoke(classGenerator);
14       }
15    }
16 }
17
18 public class ClassGenerator {
19    /* ... */
20    public void compute() {
21       List<SootMethod> sootMethods = _sootClass.getMethods();
22       /* if (my portion of the work is small enough) */
23       if (_to - _from < threshold) { /* How many classes can I
            process? */
24          for (int i = _from ; i < _to ; i++) {
25             SootMethod m = sootMethods.get(i);
26             Session session = new Session();
27             generate(m, session);
28          }
29       } else { /* split work*/
30          int half = (_to - _from)/2;
31          ClassGenerator c1 = new ClassGenerator(_writer, _ssa,
               _sootClass, _from, _from + half);
32          ClassGenerator c2 = new ClassGenerator(_writer, _ssa,
               _sootClass, _from + half, _to);
33          invokeAll(c1, c2);
34       }
35    }
36    /* ... */
37 }
```

Figure 15: Fork/Join Framework

As we already mentioned, Fork/Join framework is designed for work that can be broken into smaller pieces recursively, in contrast with the fact-generation process which is not designed to run recursively. Thus, the results were worse than the two previous approaches (still better than then sequential approach). Below are some fact-generation timing examples with the *Fork/Join Framework FG* approach for various threshold values (such as 2, 3 and 4). The threshold value 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 timing examples, with threshold 2, 3, 4 and pool size 16

4.4 Multiple Classes Per Thread

Our last approach is similar to 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 had minimal work to do, so we decided to grow the amount of work assigned to each thread. Below we show an abstract version of the code implementing the *Multiple Classes Per Thread* approach.

```
1  public class FactGenerator {
2     public FactGenerator(FactWriter writer, boolean ssa, int
          totalClasses) {
3        _writer = writer;
4        _ssa = ssa;
5        _classCounter = 0;
6        _sootClassArray = new ArrayList<>();
7        _totalClasses = totalClasses;
8        _cores = Runtime.getRuntime().availableProcessors();
9        _executor = new ThreadPoolExecutor(_cores/2, _cores, 0L,
             TimeUnit.MILLISECONDS, new LinkedBlockingQueue<Runnable>());
10    }
11
12    public void generate(sootClass) {
13       _classCounter++;
14       _sootClassArray.add(_sootClass);
15       if ((_classCounter % _classSplit == 0) || (_classCounter + 1 ==
             _totalClasses)) {
16          Runnable classGenerator = new ClassGenerator(_writer, _ssa,
                _sootClassArray);
17          _classGeneratorExecutor.execute(classGenerator);
18          _sootClassArray = new ArrayList<>();
19       }
20    }
21 }
```

Figure 16: Multiple Classes Per Thread: FactGenerator.java

```
1  public class ClassGenerator {
2     public void run() {
3        if(c.hasSuperclass() && !c.isInterface())
4           _writer.writeDirectSuperclass(c, c.getSuperclass());
5        for(SootField f : c.getFields())
6           generate(f);
7        for(SootMethod m : c.getMethods()) {
8           Session session = new Session();
9           Runnable mg = new MethodGenerator();
10          MgExecutor.execute(mg);
11          generate(m, session);
12       }
13    }
14
15    /* ... */
16 }
```

Figure 17: Multiple Classes Per Thread: ClassGenerator.java

This approach minimizes the overhead produced by assignments and allocations of our first and second effort. Therefore the *Multiple Classes Per Thread FG* approach produced the best timing results so far. Below we show some fact-generation timings for various thread-pool sizes (such as 4, 16 and 32) and various numbers 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 timing examples, with pool size: 4, 16, 32 and classes per thread: 2, 3, 4

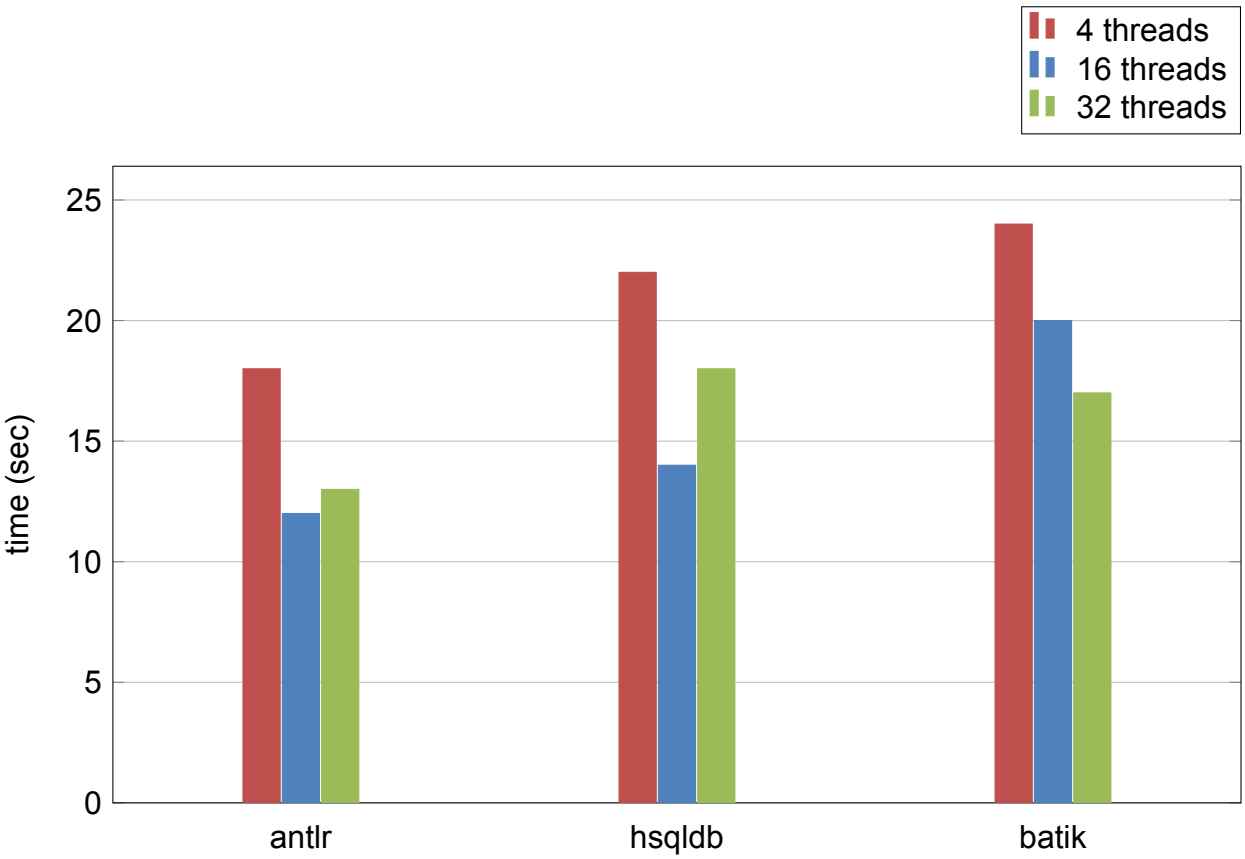

Figure 18: Increasing the number of threads with 3 classes per thread

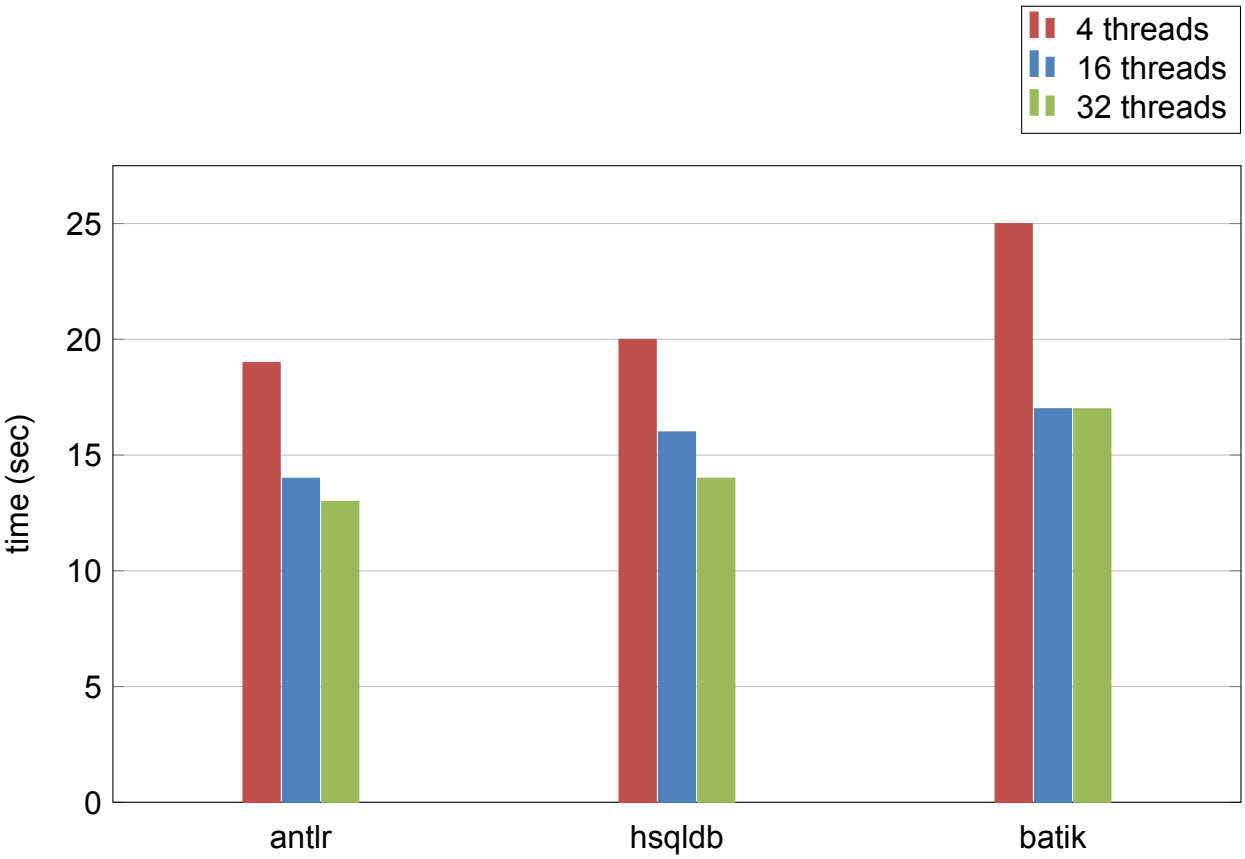

Figure 19: Increasing the number of threads with 4 classes per thread

Based on the `Multiple Classes per thread` approach, our final version is presented below:

```
1  public class Driver {
2     public Driver(ThreadFactory factory, boolean ssa, int totalClasses)
          {
3        _factory = factory;
4        _ssa = ssa;
5        _classCounter = 0;
6        _sootClasses = new ArrayList<>();
7        _totalClasses = totalClasses;
8        _cores = Runtime.getRuntime().availableProcessors();
9        _executor = new ThreadPoolExecutor(_cores/2, _cores, 0L,
            TimeUnit.MILLISECONDS, new LinkedBlockingQueue<Runnable>());
10    }
11
12    public void doInParallel(List<SootClass> sootClasses) {
13       for(SootClass c : sootClasses) {
14          generate(c);
15       }
16       _executor.shutdown();
17       _executor.awaitTermination(Long.MAX_VALUE, TimeUnit.NANOSECONDS);
18    }
19
20    void generate(SootClass _sootClass) {
21       _classCounter++;
22       _sootClasses.add(_sootClass);
23       if ((_classCounter % _classSplit == 0) || (_classCounter + 1 ==
            _totalClasses)) {
24          Runnable runnable = _factory.newRunnable(_sootClasses);
25          _executor.execute(runnable);
26          _sootClasses = new ArrayList<>();
27       }
28    }
29 }
```

Figure 20: Final approach: Driver.java

```
1  public class ThreadFactory {
2     public Runnable newRunnable(List<SootClass> sootClasses) {
3        if (_makeClassGenerator) {
4           return new FactGenerator(_factWriter, _ssa, sootClasses);
5        } else {
6           return new FactPrinter(_ssa, _toStdout, _outputDir,
               _printWriter, sootClasses);
7        }
8     }
9  }
10
11 public class FactGenerator implements Runnable {
12    public void run() {
13       for (SootClass _sootClass : _sootClasses) {
14          /* for all soot classes generate like the sequential
               FactGenerator */
15          /* ... */
16       }
17    }
18 }
```

Figure 21: Final approach: ThreadFactory.java, FactGenerator.java

At first, the Main method calls `driver.doInParallel(classes)` which takes as an argument all the `SootClasses` to be generated. `driver` calculates the available threads to run the fact-generation process and then calls the `Driver.generate(c)` method for each SootClass. Then `driver.generate` creates a new `ThreadFactory` thread for every `_classSplit` classes (e.g `_classSplit = 3` classes). `ThreadFactory` in turn calls the original `FactGenerator` (from the sequential approach) but instead of having a `generate` method, it has a `run` method with the same body.

5. LOCKING

Along with threads come locks. An incorrect use of locking could have severe impact on performance, produce incorrect results or even lead to non-termination of the fact-generation process. A conservative approach could lead to having more locks than necessary which would be a major source of bottleneck. Having fewer could lead to more races. So, locks had to be used with extreme care.

At the very beginning of this project, we tried a lot of different locking approaches such as lock everything. Of course the results were pretty much similar to the sequential ones. As a result we decided to implement more fine-grained locking and also change some global objects to thread-local in order to achieve thread safety. We first tried to understand which part of Soot is used by Doop to generate the facts, and then tried to make that specific part thread-safe.

5.1 Doop Side

In Doop the only tasks that required synchronization to avoid any race conditions, which would lead to the corruption of the fact-generation results, were the writing to the output files and three methods that were accessing the same field of each SootMethod object leading to a race condition. The code modifications are explained below.

5.1.1 CSVDatabase

In the file `CSVDatabase.java` we had to lock each output `.facts` file to prevent more than one threads from writing simultaneously. By synchronizing the predicate file, we ensure that output files are written by one thread at a time and as a result the integrity and correctness of the generated facts are guaranteed.

```
1 synchronized(predicateFile) {
2    Writer writer = getWriter(predicateFile);
3    addColumn(writer, arg, shouldTruncate);
4    for (Column col : args)
5       addColumn(writer.append(SEP), col, shouldTruncate);
6    writer.write(EOL);
7 }
```

Figure 22: CSVDatabase.java

5.1.2 Representation

The other synchronization we had to provide was in three methods in the `Representation.java` file. Those methods were accessing and trying to retrieve some fields from a SootMethod object passed as an argument while threads were still active.

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

Figure 23: Representation.java

5.2 Soot Side

In the implementation of Soot, all the global objects are enclosed in the class G (`G.java`) and are initialized and accessed using the Singleton design pattern. As we mentioned before, Soot does much more than just translate bytecode to Jimple, so it has various phases and a lot of different options which apply a variety of code transformations. The phase responsible for bytecode to Jimple translation is the jb phase. In this phase we identified all global objects, the methods that access and write to them and the methods called by these global objects and we synchronized them and/or made them thread-local.

5.2.1 Type Assigner

The class `JimpleBodyPack` applies the transformations corresponding to the given options. In our case, it applies `"jb.tr"` which means *"Jimple body transformation"*. From bytecode to Jimple translation this is the only group of transformations needed. So we used one lock before applying `Type Assigner` and unlocked afterwards. All the transformations that take place to perform the bytecode to Jimple translation are first inserted to and then retrieved from `packManager`, a global object. In order to prevent race conditions, we changed the `packManager` from global to thread-local. The new `packManager` that we use is `PackManager`.

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

Figure 24: JimpleBodyPack.java

5.2.2 Pack Manager

As mentioned before, the groups of transformations (Soot names them "packs") containing the various phases and their options are managed by the class PackManager. To ensure thread-safety, we used another lock which protects a group of transformations which access Class Hierarchy Analysis.

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

Figure 25: PackManager.java

This functionality is not currently necessary for Doop, but may be useful in the future and does not impact performance.

5.2.3 Shimple -ssa

We also use Soot to produce **Shimple** which is an **SSA** variation of Jimple. The way Shimple is generated is similar to Jimple generation with the exception of a different final step, during which Shimple.java is called, which is the class handling the translation from Jimple to Shimple. In this case, we synchronized all Shimple-body-creation methods that access global objects, or can be accessed from more than one threads simultaneously. To generate Shimple instead of Jimple, Soot must be given the -ssa flag.

6. EXPERIMENTAL RESULTS

Summarizing, below we gather all fact-generation times for the previous version of Soot (2.5.0), the latest and all our four approaches.

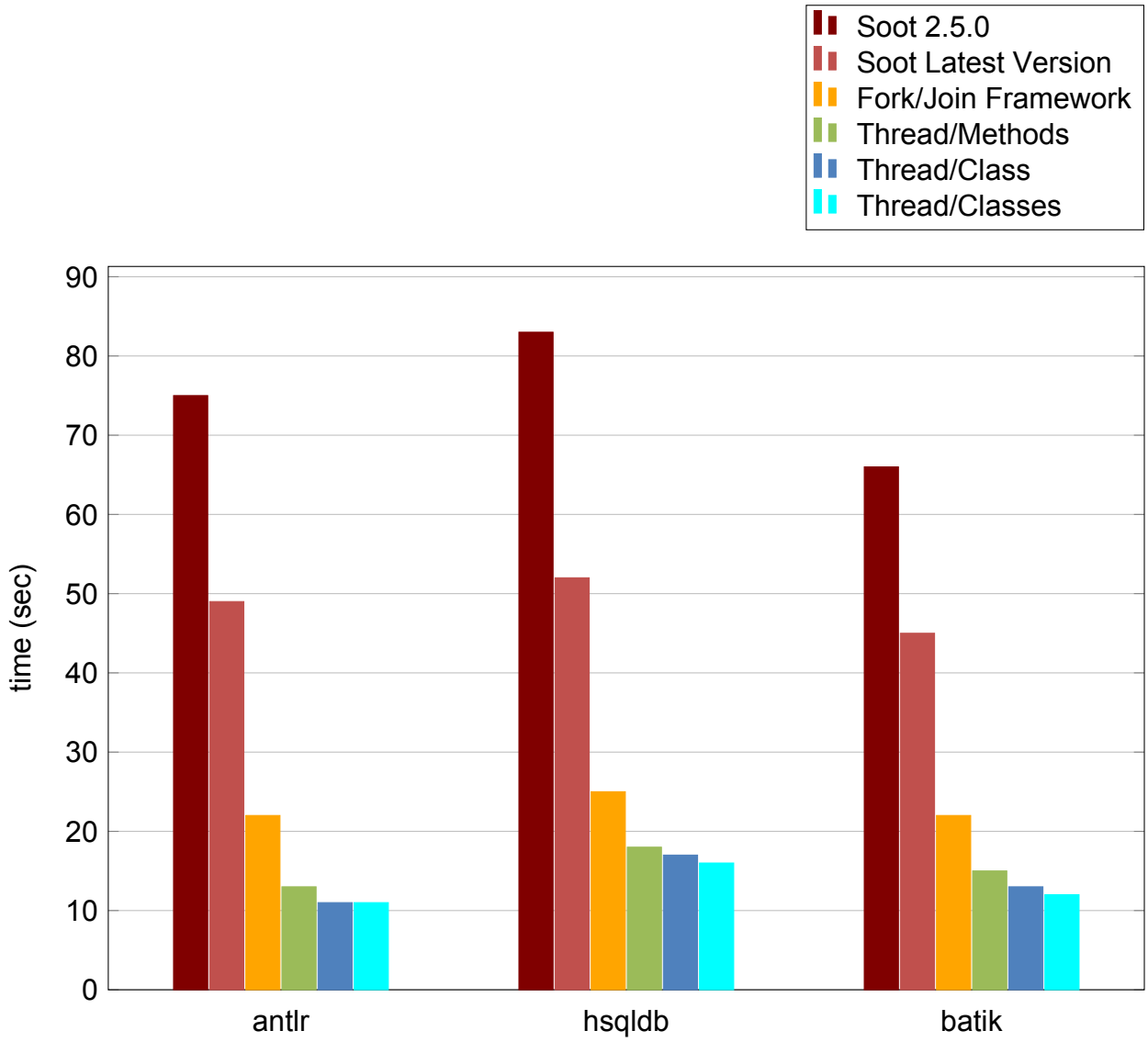

Figure 26: Fact-generation timing results

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 timings of all approaches with pool size 16-32

7. CONCLUSIONS

In conclusion, just by updating the Soot version and applying the latest, we achieved a speedup of 145-160%. With the latest soot-version and our best approach we achieved a speedup of 450-625% compared to the sequential fact-generation with Soot version 2.5.0. 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 approaches.

In most cases we used as few locks as possible in both Doop and Soot, so the locking did not significantly hinder the performance of the fact-generation process. We also have to mention that with those locks Soot is thread-safe for the purpose it is used by Doop (Java bytecode to Jimple or Shimple translation), but thread-safety is not guaranteed for all the other functionality it provides.

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
EDB	Extensional Database
jb	Jimple Body

REFERENCES

[1] ”Sable: Soot”
[Online]
Available: https://sable.github.io/soot/

[2] ”Points-to Analysis” [Online]
Available: http://yanniss.github.io/points-to-tutorial15.pdf

[3] ”Doop: Framework for Java Pointer Analysis”
[Online]
Available: http://doop.program-analysis.org/

[4] ”Oracle Java Fork/Join Framework”
[Online]
Available: https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html