McGill University
School of Computer Science
Sable Research Group

Improving the Compiling Speed of the AspectBench Compiler

Sable Technical Report No. 2006-3

Jingwu Li and Laurie Hendren
{jli98, hendren}@cs.mcgill.ca

August 14, 2006

Contents

List of Tables

List of Figures

Acknowledgment

I would like to express my deep sense of gratitude to my supervisor **Prof. Laurie Hendren**, for her invaluable help and guidance during the project. I am highly indebted to her for constantly encouraging me by giving her suggestions on my work. I am grateful to her for having given me the support and confidence.

I also greatefully thank my wife, Mingjian Wang, for her support, encouragement and understanding over the past years.

Jingwu Li
August 2006

Abstract

The AspectBench Compiler (abc) is an extensible AspectJ compiler and built based on Polyglot [13] and Soot [2]. It generates optimized Java bytecode which can run faster in many cases when compared with ajc. However, the compilation speed of abc can be substantially slower than ajc. Typically it is roughly 4 times slower than ajc [1]. By using various profiling tools, we observed some bottlenecks during the process of compiling AspectJ source code. We modified some data structures related to forward and backward flow analysis and changed some algorithms such as the variable name generator and around inlining to remove the bottlenecks or relieve the severity of those bottlenecks. The experimental results on selected benchmarks shows that the combined modifications reduce overall by 8% compilation time as compared with original abc. The speed up is especially notable for the benchmarks with around advice applied many places. At the same time, we also reduce the class file size for the benchmarks with around advices applied many places greatly, around 28%.

1 Introduction

Aspect Oriented Programming (AOP) is new programming technique to address crosscutting concerns. It is not like Object Oriented Programming which addresses common concerns within certain related objects and abstract common attributes and methods to capsulate in object class. AOP deals with unrelated objects and modularizes the common concerns across the whole program system and not limited in certain classes [15]. These crosscutting concerns, such as logging events, caching results, debugging support, error checking and security control, can be implemented by using AOP in a very easy and clean way.

AspectJ is a seamless aspect-oriented extension to the Java programming language create at Xerox PARC [16]. The first AspectJ compiler is called ajc which is now developed and supported by the AspectJ Eclipse project [7].

The AspectBench Compiler (abc) [1] is an alternative AspectJ compiler. It is a complete implementation of AspectJ. It aims to make it easy to implement both extensions and optimizations of the core language [3]. Generally, abc can generate optimized Jave bytecode which can run faster in many cases compared with the ones generated by ajc [5]. However, on the other hand, the compilation speed of abc can be substantially slower than ajc. Typically abc is roughly 4 times slower than ajc [1]. As we show in Section 2.4, our experiments show the compilation slowdown of many benchmarks is at least 8 times, especially for eigenv, nullcheck and lod-sim benchmarks, around 22 times slower, 15 times slower and 11 times slower respectively.

The abc compiler's slow compilation speed arises from the design and implementation strategy of abc. Abc leverages existing compiler technology by combining Polyglot which is an extensible compiler framework for Java in the frontend and Soot which is a framework for analysis, optimization and transformation of Java in the backend. During weaving phase, abc pays less attention to generate compacted and optimized code in the first place and lets Soot in the later pass to optimize those weaved codes. From the compiler developer's view, this strategy reduces the complexity to write the weaver. However, from the optimization view, it may slow down the compilation process. If we could do the optimization in the first place, we could save a lot of time in later phases to achieve the same aim. For example, when weaving advice, if we could test the extracted shadow method exists or not, we can avoid creating duplicate methods and save time in the later pass to remove the duplicated methods. This requires us to mix some tasks in several phases, not like the original abc which cleanly separates tasks among phases.

In this project, we investigate the hotspots in compiling with abc by using various profiling tools. To improve abc's compilation speed, we use various ways to modify abc such as use other implementation of HashSet, reuse objects during forward and backward flow analysis and change some algorithms such as variable name generator and around inlining and so on. We did the experiments on the selected benchmarks and yield results that those combined modifications reduced the compilation speed overall 8% compared with the original abc.

The organization of the report is as follows. In section 2 we describe the benchmarks used in this project, the experimental environment and the initial timing result. We describe the general strategies we used for optimization in section 3. We introduce various profiling tools and techniques used in optimization in section 4. Section 5 describes the observations of the bottlenecks we found and the attempts we took to attack those bottlenecks. Finally, we give the conclusion of the project in section 6 and discusses the future work in section 7.

2 Benchmarks and Initial Timing

2.1 Benchmarks

In this project, the most benchmarks we used are coming from two sources, the graduate optimizing compilers course (COMP 621) and the abc benchmarks [1]. To let our test suite have good representation of AspectJ, we chose the benchmarks with various aspect usages, including pointcut definitions, percflow, pertarget, inter-type declarations, before advice, after advice, around advice and trace matching. We also choose two tiny benchmarks hellono and helloworld to test the initialization speed of abc, the compilation speed of programs without aspects.

We used ten benchmarks in this project and we listed the size of the benchmarks and their descriptions in Table 1.

2.2 Platforms

Table 2 shows the environment and abc versions we used for measuring the performance of abc compiler.

2.3 Measurement Methodology

To measure performance, we compiled each benchmark 5 times and collected the compiling time of each run, removing the one farthest to the average time, and then computing the average time for the remaining 4 runs as the compiling time result for the benchmark.

2.4 Initial timing result

We list our initial timing result for abc 1.2.0 version and ajc 1.2.1 version in Table 3. It shows that abc is substantially slower than ajc when compiling nullcheck, eigenv, wig11, dcm-sim, lod-sim and weka benchmarks. The experiment shows that the compilation slowdown of those benchmarks is at least 8 times, especially for eigenv (around 22 times slower) , nullcheck (around 15 times slower) and lod-sim benchmarks (around 11 times slower).

To illustrate the slowdown of abc compilation speed comparing with ajc, we generated a graph as shown in Figure 1. From it we can see the compilation speed of abc is really slower than ajc.

Benchmark	Normal Java Codes		Aspect Codes		Description
	Classes	Methods	Classes	Methods	
hellono	1	1	0	0	simple hello world program without any aspects applied.
helloworld	1	1	1	6	simple hello world program with around advice aspects applied.
asac	7	15	1	6	This benchmark starts three threads and each thread uses a sorting algorithm (Bubble Sort, Selection Sort and Quick Sort) to sort arrays. It contains before, after and around advices. The around advices are matched in a many places in the benchmark.
nullcheck	23	116	2	7	A simulator program simulates the performance of certificate revocation schemes. These schemes look at reducing the risk of cryptographic certificates from becoming invalid. The aspects in program detect methods returning null on an error to force certain code standard. It contains before, after and around advices. The around advices are matched in a lot of places in the benchmark.
eigenv	1	9	2	11	This benchmark program is designed to compute a matrix's Eigen value and eigenvector. The value for each of the computed element in the matrix is the Fibonacci's number for its original element in the original matrix. It contains before, after and around advices. The around advices are matched in a lot of places in the benchmark.
wig11	24	849	7	14	AspectJ WIG Compiler is a compiler front-end which translates WIG programs to python programs. It contains some intertype declarations, before and after advices.
dcm-sim	25	126	4	8	Dynamic Coupling Metrics implemented using AspectJ. It uses a light weight data collection mechanism and can include the possibility of accounting for objects being freed by the GC. The aspects in this benchmark contain around, before and after advices.
lod-sim	24	120	4	6	This benchmark checks the Law of Demetere. It uses relatively complex join points, percflow, pertarget, and cflow.
weka	39	571	1	3	The stripped down weka that contains only those classes for the benchmark. It uses before and after advices.
weka-tm	38	571	1	3	Similar to weka benchmark except using trace matching.

Table 1: Benchmarks

abc version	1.0.2, 1.2.0
Soot version	2.2.3
os.arch	x86-64
os.name	Linux
os.version	2.6.15-23-amd64-generic
java.vendor	Sun Microsystems Inc.
java.version	1.4.2
Hardware	AMD Athlon(tm) 64 X2 Dual Core Processor 3800+ (CPU: 2010.317GHz Mem: 4GB)

Table 2: Software and Hardware Platforms

benchmark	wall time (s)		cpu time (s)	
	abc	ajc	abc	ajc
hellono	2.6426	1.34	2.6301	1.26
helloworld	3.48	1.75	3.4475	1.6751
asac	11.9676	2.7876	11.8826	2.585
nullcheck	78.945	5.2975	78.7625	3.8525
eigenv	65.9425	3.2301	65.83	3.0526
wig11	60.0176	7.7701	59.205	6.9376
dcm-sim	25.0401	3.685	24.9526	3.5101
lod-sim	45.8626	4.1651	45.7201	3.975
weka	32.9875	4.9876	32.795	4.7426

Table 3: Compilation time comparison between original abc-1.2.0 version and ajc 1.2.1 version

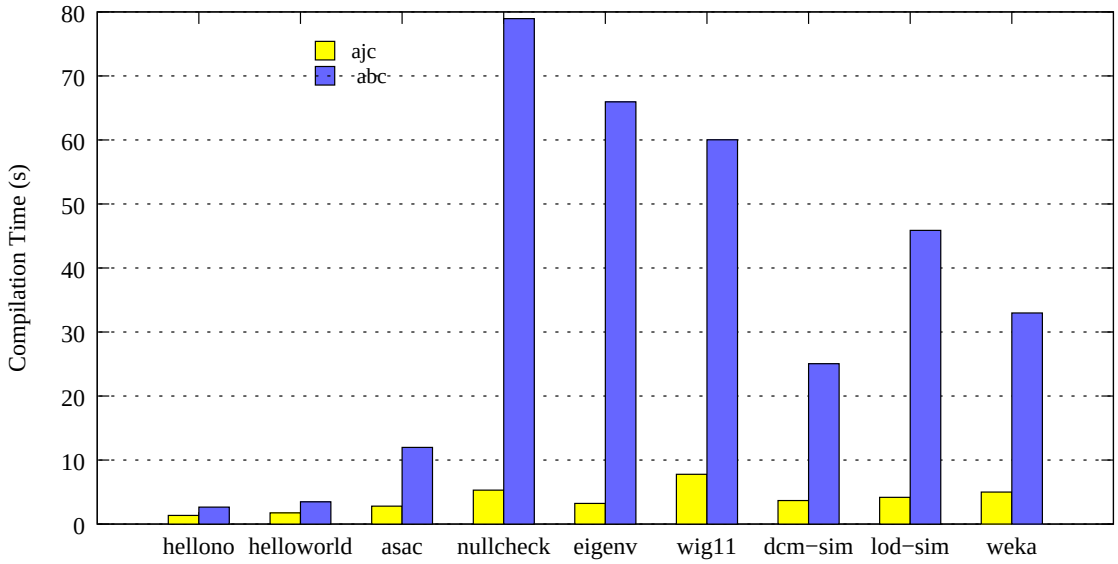

Figure 1: Compilation time (in second) comparison between abc 1.2.0 version and ajc 1.2.1 version

3 Strategies

The general tuning strategy introduced by Jack Shirazi [8] is iterating doing the following two steps:

1. Identify the main bottlenecks
2. From the top few bottlenecks, choose the quickest and easiest one to fix and address it.

Shiraza's tuning strategy suggests that addressing the cheapest single bottleneck rather than the absolute topmost one because the elimination of one bottleneck often changes the characteristics of the application and thus the bottlenecks of the application are consequently often changed.

Generally, I followed this strategy to attack the performance problems in abc.

1. Establish a set of benchmarks and build the initial timing base line.
2. Choose the profiling tools.
3. Measure the performance on the benchmarks by using the profiling tools.
4. Identify the top few bottlenecks.
5. Hypothesize the causes of the bottlenecks.
6. Create tests to verify the hypotheses.
7. Choose the quickest and easiest bottleneck or several closely related ones to fix.
8. Alter the application to reduce the bottleneck(s).
9. Test the alteration and make sure the modification is correct.
10. Measure the performance improvement after the modification.
11. Repeat from Step 3.

In the above performance tuning process, the only thing that violates Shiraza's tuning strategy is that we are not limiting our focus on the single cheapest bottleneck at a time, we may select several closely related bottlenecks to fix at a time. This is because that in some cases, if we only consider one bottleneck at a time we may narrow our view and limit our solution strategy. So we may get some temporary solutions and in the later on we have to change those solutions to fix other related bottlenecks. In the worst case, we may even need to use a solution totally different from the one we chose before. This is especially true when solve the bottlenecks caused by inefficient algorithms.

4 Profiling Technologies

In order to speed up compiling speed of abc, we need to have a holistic picture of abc during running time and identify the bottlenecks in the program. Once we know where the time goes we know where to focus our efforts. There are many profiling tools to help us to fulfill profiling task. In this project, we used several profiling tools (approaches): abc internal timer, JProfiler (a commercial profiler) [8], Sun's `hprof` [17] profiling agent and weaving advices into abc.

4.1 abc internal timer

Abc provide several flag to enable internal timer AbcTimer to track the time spend in each compiling phase in abc.main.Debug class. We listed the flags to enable abc internal timer in Listing 1.

Listing 1 Flags to enable abc internal timer

```
    public boolean abcTimer=true;
    public boolean polyglotTimer=true;
    public boolean sootResolverTimer=true;
    public boolean timerTrace=true;
```

The abc internal timer generates output is in the format as shown in Listing 2.

Listing 2 Output of abc internal timer

```
[ 18.835% ] Init. of Soot:  4638
[ 00.370% ] Loading Jars:  91
[ 01.937% ] Create polyglot compiler:  477
[ 11.858% ] Polyglot phases:  2920
[ 02.912% ] Initial Soot resolving:  717
[ 00.000% ] Soot resolving:  0
[ 00.016% ] Aspect inheritance:  4
[ 00.008% ] Declare Parents:  2
[ 00.097% ] Intertype Adjuster:  24
[ 04.382% ] Jimplification:  1079
[ 00.008% ] Fix up constructor calls:  2
[ 01.344% ] Update pattern matcher:  331
[ 00.061% ] Weave Initializers:  15
[ 00.000% ] Load shadow types:  0
[ 02.497% ] Compute advice lists:  615
[ 00.097% ] Add aspect code:  24
[ 12.435% ] Weaving advice:  3062
[ 00.126% ] Exceptions check:  31
[ 10.246% ] Advice inlining:  2523
[ 01.072% ] Interproc. constant propagator:  264
[ 03.387% ] Boxing remover:  834
[ 04.788% ] Duplicates remover:  1179
[ 00.248% ] Removing unused methods:  61
[ 00.138% ] Specializing return types:  34
[ 18.104% ] Soot Packs:  4458
[ 05.036% ] Soot Writing Output:  1240
```

Currently, the abc compiling process is divided into 26 phases as shown in Listing 2. We use abc internal timer profiling our 10 benchmarks and list the top 5 time consuming phases for each benchmark in Table 4.

benchmark	the top 5 time consuming phases				
	rank 1	rank 2	rank 3	rank 4	rank 5
hellono	Init. of Soot 41.080%	Polyglot phases 21.341%	Create polyglot compiler 10.333%	Update pattern matcher 6.200%	Soot Packs 4.260%
helloworld	Init. of Soot 30.534%	Polyglot phases 27.381%	Jimplification 8.117%	Create polyglot compiler 7.618%	Soot Packs 5.432%
asac	Soot Packs 20.959%	Weaving advice 13.967%	Polyglot phases 13.561%	Advice inlining 11.668%	Init. of Soot 8.470%
nullcheck	Advice inlining 33.093%	Duplicates remover 23.311%	Soot Packs 19.005%	Weaving advice 8.121%	Soot Writing Output 3.590%
eigenv	Weaving advice 32.207%	Soot Packs 27.052%	Advice inlining 23.433%	Duplicates remover 3.638%	Polyglot phases 2.621%
wig11	Polyglot phases 28.570%	Soot Packs 23.784%	Jimplification 11.146%	Weaving advice 9.151%	Soot Writing Output 8.489%
dcm-sim	Soot Packs 21.844%	Advice inlining 16.865%	Weaving advice 11.747%	Polyglot phases 10.726%	Soot Writing Output 9.020%
lod-sim	Weaving advice 32.519%	Soot Packs 23.313%	Boxing remover 19.521%	Polyglot phases 5.874%	Soot Writing Output 5.478%
weka	Polyglot phases 26.885%	Soot Packs 23.504%	Jimplification 16.192%	Soot Writing Output 9.232%	Weaving advice 9.128%
weka-tm	Polyglot phases 25.419%	Soot Packs 23.651%	Jimplification 17.147%	Weaving advice 9.253%	Compute advice lists 7.786%

Table 4: Top 5 time consuming phases for abc 1.2.0 version

Table 4 shows that there are several phases take large count of total compilation time for our test suite:

1. Polyglot phases: this phase takes large amount of compilation time for all benchmarks.
2. Soot Packs: similar as Polyglot phases, all benchmarks take great time at this phase.
3. Weaving advice: all benchmarks except two small benchmarks (hellono and helloworld) take great time at this phase, especially for eigenv and lod-sim benchmarks.
4. Advice Inlining: For those benchmarks with around advices ,nullcheck, eigenv, asac and dcm-sim benchmarks, abc spend large amount of time at this phase, especially for nullcheck benchmark which accounts 33% of total compilation time.
5. Jimplification: This phase accounts for the rank 3 time consuming phase for four benchmarks, helloworld, wig11, weka and weka-tm.
6. Duplicates remover: nullcheck and eigenv benchmark takes great time at this phase, especially for nullcheck benchmark account for 23.31% of total time.
7. Init. of Soot: This phase takes large part when compiling small benchmarks such as hellono and helloworld.[1]
8. Soot Writing Output: Although this phase is in the top 5 list, it accounts for small amount of total compilation time, from 3.590% to 9.232%.

The Weaving advice phase, Advice Inlining phase and Duplicates remover phase are quite closely related and account for most compilation time for eigenv and nullcheck benchmarks. In the later sections, we will dive into the performance issue related to those three phases.

[1]This phase takes constant time for all benchmarks.

4.2 HPROF

Sun's JDK provides a simple command line profiling tool `hprof` for heap and cpu profiling. By using `hprof`, users can request various types of heap and cpu profiling features from JVM. The `hprof` can be used to track down and isolate performance problems involving memory usage and inefficient code. The data generated by `hprof` can be in textual or binary format which can be used with tools like HAT [18]. The disadvantages of using this tool is slow and the generated file size is very big and it is very hard to interpret the result for memory usage.

4.3 JProfiler

JProfiler is a commercial tool for profiling Java programs which can provide graphical output. It can be used to find performance bottlenecks, pin down memory leaks and resolve threading issues. It is very powerful, we can set filters to only profile part of the program we are interested in. It provides many analysis views to give us a full picture of the program we are profiling. In our project, we used JProfiler 4.0.2 trial version.

4.4 Weaving aspects into abc

Another approach we took to tracking down the performance issue of abc is weaving advices into abc. We wrote around advices to record the execute time of each method in abc's abc.weaving.weaver package and used ajc to weave those advice into abc. When using the abc with weaved advices to compile the benchmarks, it will generate the report about the total execution time and number of invocations for each method in abc's abc.weaving.weaver package. The important characteristic of this approach is that we can profile part of the program where we are interested in. We also can archive similar report by setting profiling filters in JProfiler.

5 Observations and Improvement

In this section, we describes the process we took to detect and tackle the bottlenecks in abc. We started our work on abc 1.0.2 version [2] as described in Section 5.1. The rest of the sections describe the works based on the most recent abc release abc 1.2.0 version.
Abc is constructed base on Polyglot and Soot [6]. It consists of several packages: abc, soot, polyglot and so on. In this project, we mainly focus on the abc and soot packages. To improve abc compilation speed, we optimized those two packages respectively, as described in section 5.2 and 5.3.
Besides the manual optimization, we also tried to use the Soot optimization tool to automatically optimization abc. We described this approach in section 5.4.

[2] This is because when we start this project, summer 2005, abc 1.0.2 version is the newest release.

5.1 Abc-1.0.2 version observations and modifications

5.1.1 Profiling result of original abc-1.0.2 version

We list our profiling result for original abc 1.0.2 version in Table 5 and Trace 1. Table 5 shows the top 5 time consuming methods for compiling eigenv benchmark. The `count` column indicates how many times a particular stack trace was found to be active. The stack trace id is shown in `trace` column and the full qualified method name at the top stack trace is shown in `method` column. Trace 1 shows the stack trace for the most time consuming method.

rank	self	accum	count	trace	method
1	5.96%	5.96%	4531	19812	abc.weaving.weaver.AroundWeaver $Util.chainContainsLocal
2	2.10%	8.06%	1595	19118	java.util.LinkedList.listIterator
3	2.08%	10.14%	1578	19035	soot.util.HashChain$Link.insertAfter
4	2.00%	12.14%	1516	19117	java.util.HashMap.newValueIterator
5	1.32%	13.45%	1002	11294	java.util.HashMap.addEntry

Table 5: Top 5 time consuming methods by using Xrunhprof to profile abc-1.0.2 version compiling eigenv benchmark

Trace 1 No. 19812

```
abc.weaving.weaver.AroundWeaver$Util.chainContainsLocal(AroundWeaver.java:166)
abc.weaving.weaver.AroundWeaver$Util.setLocalName(AroundWeaver.java:323)
abc.weaving.weaver.AroundWeaver$Util.access$1300(AroundWeaver.java:119)
abc.weaving.weaver.AroundWeaver$AdviceMethod$ProceedMethod$AdviceApplicationInfo
   .copyStmtSequence(AroundWeaver.java:1216)
abc.weaving.weaver.AroundWeaver$AdviceMethod$ProceedMethod$AdviceApplicationInfo
   .doWeave(AroundWeaver.java:986)
abc.weaving.weaver.AroundWeaver$AdviceMethod$ProceedMethod.doWeave
   (AroundWeaver.java:793)
```

5.1.2 Code optimization

1. chainContainsLocal

   From the profiling result Table 5 and Trace 1, we can know that abc.weaving.weaver.AroundWeaver$Util.chainContainsLocal consume a lot of time during compiling eigenv benchmark.

   This method is quite straightforward as shown in Listing 3. It just iterates through the chain and compare whether the elements in the chain contains elements with the given name. So based on the optimizing strategy we adopted, always start from the easiest top few ones, we optimized this method first.

   After examining the AroundWeaver class, we found the actual type of input parameter is always soot.util.HashChain which uses a HashMap as the underlying structure and maintains a doubly-linked

Listing 3 Original code of chainContainsLocal() method in abc-1.0.2 version

```
private static boolean chainContainsLocal(Chain locals, String name) {
    Iterator it = locals.iterator();
    while (it.hasNext()) {
        if (((soot.Local) it.next()).getName().equals(name))
            return true;
    }
    return false;
}
```

list running through all of its entries. So the obvious solution to optimize this method is using the HashChain.contains() method to test the equals instead of iterating over the chain and compare elements one by one. Listing 4 shows the chainContainsLocal method after modification.

Listing 4 After modification, the code of chainContainsLocal() method in abc-1.0.2 version

```
private static boolean chainContainsLocal(Chain locals, String name) {
    HashChain hc = (HashChain)locals;
    return hc.contains(name);
}
```

After applied this modification, the timing results of compiling benchmarks are shown in Table 6. The `improvement` column is computed by (org - mod)/mod * 100.

benchmark	wall time			cpu time		
	org	mod	improvement(%)	org	mod	improvement(%)
hellono	2.6401	2.6551	-0.57	2.5826	2.615	-1.26
helloworld	3.4225	3.4301	-0.23	3.3701	3.4	-0.89
asac	8.805	8.54	3.01	8.7575	8.4775	3.2
nullcheck	27.3276	25.4351	**6.93**	27.1751	25.305	6.89
eigenv	75.915	25.8851	**65.91**	75.83	25.805	65.97
wig11	59.0625	59.335	-0.47	58.4526	58.59	-0.24
dcm-sim	18.03	18.0376	-0.05	17.9075	17.8325	0.42
lod-sim	39.41	40.275	-2.2	39.2751	40.1725	-2.29
weka	32.9975	33.0075	-0.04	32.7951	32.8376	-0.13

Table 6: Compilation speed comparison between abc-1.0.2 version and the abc-1.0.2 version with modified chainContainsLocal() method

From Table 6, we can see that this modification speed up the compilation time of eigenv benchmark greatly, around 66%. It also reduce the compilation time near 7% for nullcheck benchmark.

2. setLocalName

   Further analysis the above profiling result in Table 5 and Trace 1, we can see that it is abc.weaving.weaver.AroundWeaver$Util.setLocalName method which invokes the

abc.weaving.weaver.AroundWeaver$Util.chainContainsLocal() method. If we could reduce the number of invocations to chainContainsLocal method, we could also reduce the compilation time. Listing 5 shows that the original setLocalName in abc 1.0.2 version.

Listing 5 Original code of setLocalName() method in abc-1.0.2 version

```
private static void setLocalName(Chain locals, Local local,
                         String suggestedName){
   String name = suggestedName;
   int i = 0;
   while (AroundWeaver.Util.chainContainsLocal(locals, name)) {
       name = suggestedName + "$$" + (++i);
   }
   local.setName(name);
}
```

This method is to assign a local with non-duplicated name by first checking whether the local chain contains the local variable with the suggested name. If the suggested name exists reassign a new suggested name and check again until the chain does not contain the suggested name. The way to create the suggested name in this method is poor. For example, if we want to assign a local variable with suggested name ”book”, but the chain has already contains ”book0”, ”book1”, ... ”book99”. In such case, the setLocalName will invoke AroundWeaver.Util.chainContainsLocal method for 101 times before it can create a non-duplicated suggested name ”book100”. To solve this problem, we can use a static variable to help to generate unique local variable name. [3] Listing 6 shows the setLocalName method after modification.

Listing 6 After modification, the code of setLocalName() method in abc-1.0.2 version

```
private static long uniqueNameId = 0;
private static void setLocalName(Chain locals, Local local,
                         String suggestedName){
   String name = suggestedName + (++uniqueNameId);
   local.setName(name);
}
```

By doing this, we totally avoid the call to chainContainsLocal method. Table 7 shows that it speeds up the compilation speed for eigenv benchmark dramatically, nearly 70% compared with original abc. For the nullcheck benchmark, it reduced the compilation time nearly 7%.

The previous optimization was performed on an older version of abc, version 1.0.2. All of the remaining optimizations are done on the more recent version, abc 1.2.0.

[3] This performance problem has been solved since abc-1.1.0 version.

benchmark	wall time			cpu time		
	org	mod	improvement(%)	org	mod	improvement(%)
hellono	2.6401	2.6401	0.0	2.5826	2.6201	-1.46
helloworld	3.4225	3.4301	-0.23	3.3701	3.4076	-1.12
asac	8.805	8.6875	1.34	8.7575	8.625	1.52
nullcheck	27.3276	25.43	6.95	27.1751	25.34	6.76
eigenv	75.915	22.9751	69.74	75.83	22.925	69.77
wig11	59.0625	58.915	0.25	58.4526	58.3075	0.25
dcm-sim	18.03	17.9476	0.46	17.9075	17.8701	0.21
lod-sim	39.41	40.2001	-2.01	39.2751	40.075	-2.04
weka	32.9975	33.075	-0.24	32.7951	32.9151	-0.37

Table 7: Compilation time (in second) comparison between abc-1.0.2 version and the abc-1.0.2 version with modified setLocalName() method

5.2 Soot Modification

5.2.1 Profiling result before optimization

When using the Sun profiling tools `hprof`, we observed that java.util.HashMap.addEntry is the most time consuming operation for asac, eigenv and nullcheck benchmarks and is the second position for wig11 benchmark. Especially, it account for almost 11% of the total running time for eigenv benchmark. Here we only show the profiling results for eigenv benchmark as in Table 8, Trace 2 and Trace 3.

rank	self	accum	count	trace	method
1	10.99%	10.99%	30881	11345	java.util.HashMap.addEntry
2	9.70%	20.69%	27280	11344	java.util.HashMap.addEntry
3	5.59%	26.28%	15725	13548	java.util.HashMap.addEntry
4	2.25%	28.53%	6315	13544	java.util.HashMap.addEntry
5	1.88%	30.41%	5287	12721	java.util.AbstractList.iterator

Table 8: Top 5 time consuming methods by using Xrunhprof to profile abc-1.2.0 version compiling eigenv benchmark

Trace 2 No. 11345

```
java.util.HashMap.addEntry(HashMap.java:739)
java.util.HashMap.put(HashMap.java:392)
java.util.HashSet.add(HashSet.java:192)
soot.toolkits.scalar.SmartLocalDefs$LocalDefsAnalysis.
        flowThrough(SmartLocalDefs.java:146)
soot.toolkits.scalar.ForwardFlowAnalysis.
        doAnalysis(ForwardFlowAnalysis.java:165)
soot.toolkits.scalar.SmartLocalDefs$LocalDefsAnalysis.
        <init>(SmartLocalDefs.java:121)
```

Trace 3 No. 11344

```
java.util.HashMap.addEntry(HashMap.java:739)
java.util.HashMap.put(HashMap.java:392)
java.util.HashSet.add(HashSet.java:192)
java.util.AbstractCollection.addAll(AbstractCollection.java:319)
soot.toolkits.scalar.SmartLocalDefs\$LocalDefsAnalysis.
        copy(SmartLocalDefs.java:160)
soot.toolkits.scalar.ForwardFlowAnalysis.
        doAnalysis(ForwardFlowAnalysis.java:127)
```

5.2.2 HashSet substitution

Although the profiling result shows that the operation on HashMap is the bottleneck, from the trace result, we can identify the actual bottleneck is the operation on HashSet. In Sun JDK, the HashMap is used as back end to implement HashSet. According to our optimization strategy, the first try is to use some other implementation of HashSet to replace Sun's implementation of HashSet used by abc. The realistic problem is that the HashSet is used widely in abc and it is hard to replace all the use of HashSet in abc source code. However from the profiling result, we know that the most time consuming HashSet operation is invoked in SmartLocalDefs class. Thus, we can replace this HashSet use with another implementation in SmartLocalDefs class to see the effect. In this project, we tried THashSet, FastSet, IDHashSet and MyHashSet to replace HashSet implementation.

1. GNU Trove: THashSet

   We used the THashSet implementation provided by GNU Trove [11] (High performance collections for Java) package to replace HashSet in SmartLocalDefs.java. GNU Trove is claimed to be a fast, lightweight of implementations of the java.util Collections API. However the timing result shows that almost all benchmarks are slightly slow down after using THashSet instead of HashSet. We only get speed up for eigenv benchmark with the amount of 12% and for lod-sim benchmark with the amount of 1.8%. Table 9 shows the timing result. The slowdown for all other benchmarks is due to extra overhead of loading extra Trove java classes.

2. Javolution: FastSet

   Javolution [12] is a Java library for real-time and embedded systems. Its aim is to make the application faster and more time-predictable. In our experiment, we use Javolution 1.4 version. Table 10 shows the compilation speed of all benchmarks get worse performance after using FastSet to replace HashSet in SmartLocalDefs class. Especially for the lod-sim benchmark, we got 17% slowdown.

3. IDHashSet

   After examining SmartLocalDefs.java, ForwardFlowAnalysis.java, FlowAnalysis.java and Abstract-FlowAnalysis.java, we know that those classes just add the objects contained in the DirectedGraph object into HashSet or remove those objects from HashSet. They do not duplicate the objects or change the objects contained in the DirectedGraph object. That is, the objects stored in the Hash-Set have the character of reference-equality semantics. So we could use IdentityHashSet instead of HashSet. However, in Sun's JDK there is no IdentityHashSet but IdentityHashMap. The solution turned to use the IdentityHashMap to implement HashSet. Table 11 shows the timing result after using IDHashSet.

benchmark	wall time			cpu time		
	org	mod	improvement(%)	org	mod	improvement(%)
hellono	2.6426	2.685	-1.61	2.6301	2.6475	-0.67
helloworld	3.48	3.49	-0.29	3.4475	3.465	-0.51
asac	11.9676	12.0026	-0.3	11.8826	11.9276	-0.38
nullcheck	78.945	80.205	-1.6	78.7625	80.0251	-1.61
eigenv	65.9425	58.0451	**11.98**	65.83	57.985	11.92
wig11	60.0176	59.9325	**0.15**	59.205	59.2801	-0.13
dcm-sim	25.0401	25.4526	-1.65	24.9526	25.3651	-1.66
lod-sim	45.8626	45.0401	**1.8**	45.7201	44.9426	1.71
weka	32.9875	33.1326	-0.44	32.795	33.0175	-0.68
weka-tm	34.4325	34.6851	-0.74	34.25	34.4725	-0.65

Table 9: Compilation time comparison between abc-1.2.0 version and the modified abc-1.2.0 version using THashSet instead of HashSet in SmartLocalDefs

benchmark	wall time			cpu time		
	org	mod	improvement(%)	org	mod	improvement(%)
hellono	2.6426	2.7	-2.18	2.6301	2.67	-1.52
helloworld	3.48	3.585	-3.02	3.4475	3.525	-2.25
asac	11.9676	12.59	-5.21	11.8826	12.5125	-5.31
nullcheck	78.945	81.0551	-2.68	78.7625	80.9125	-2.73
eigenv	65.9425	66.05	-0.17	65.83	65.9626	-0.21
wig11	60.0176	61.2126	-2.0	59.205	60.4326	-2.08
dcm-sim	25.0401	25.8701	-3.32	24.9526	25.77	-3.28
lod-sim	45.8626	53.8525	**-17.43**	45.7201	53.7376	-17.54
weka	32.9875	34.2651	-3.88	32.795	34.11	-4.01
weka-tm	34.4325	35.3251	-2.6	34.25	35.1526	-2.64

Table 10: Compilation time comparison between abc-1.2.0 version and the modified abc-1.2.0 version using FastSet instead of HashSet in SmartLocalDefs

This result is similar as the one by using THashSet (as shown in Table 9) in that we only get notable speed up for one benchmark. This time, we get great improvement on null benchmark with 26%. However, we also slow down wig11, dcm-sim, lod-sim, weka, weka-tm benchmarks around 4% to 6%.

4. MyHashSet

   In the forward and backward flow analysis, the copy, merge and flowThrough operations add elements to or remove elements from HashSet which cause the set to allocate or remove Entry objects in the underlining implementation of HashSet. Since during the flow analysis, the add and remove elements are used frequently, is it possible that we can reuse those allocated Entry objects to improve the performance for the bottleneck of java.util.HashMap.addEntry?

   In order to do this, we need to maintain a list to record the deleted Entry objects in the underlying implementation of HashSet. When addEntry is called, we first check whether the deleted

benchmark	wall time			cpu time		
	org	mod	improvement(%)	org	mod	improvement(%)
hellono	2.6426	2.6401	0.1	2.6301	2.6151	0.58
helloworld	3.48	3.4826	-0.08	3.4475	3.4576	-0.3
asac	11.9676	12.2401	-2.28	11.8826	12.155	-2.3
nullcheck	78.945	58.495	**25.91**	78.7625	58.395	25.86
eigenv	65.9425	65.8825	0.1	65.83	65.805	0.04
wig11	60.0176	62.315	-3.83	59.205	60.7701	-2.65
dcm-sim	25.0401	25.8325	-3.17	24.9526	25.7175	-3.07
lod-sim	45.8626	47.9051	-4.46	45.7201	47.51	-3.92
weka	32.9875	34.8026	-5.51	32.795	34.5526	-5.36
weka-tm	34.4325	36.65	-6.45	34.25	36.06	-5.29

Table 11: Compilation time comparison between abc-1.2.0 version and the modified abc-1.2.0 version using IDHashSet instead of HashSet in SmartLocalDefs

list is empty or not. If the list is not empty, we just take one to use without creating a new Entry object. This solution consists several classes to implement: soot.toolkits.scalar.MyHashMap, soot.toolkits.scalar.MyHashSet, soot.toolkits.scalar.MyAbstractMap, soot.toolkits.scalar.MyLinkedHashSet and
soot.toolkits.scalar.MyLinkedHashMap.

We calculated the reused Entry objects during the compilation of benchmarks as shown in Table 12. We reused a lot of Entry objects for nullcheck, eigenv, lod-sim, weka and weka-tm benchmarks, especially for eigenv benchmark, around 740000.

Benchmark	Reused Entry Objects
hellono	0
helloworld	2
asac	3739
nullcheck	**21032**
eigenv	**739597**
wig11	4863
dcm-sim	4361
lod-sim	**20000**
weka	**22187**
weka-tm	**23254**

Table 12: Number of reused Entry objects in MyHashMap

However, when we profiling the abc with MyHashSet modification, we only get great speed up for nullcheck benchmark, around 29% as shown in Table 13. Although the number of reused Entry objects for eigenv benchark is the greatest in our test suite, we get 4% slowdown surprisingly. The benefits we get by reusing Entry objects may be offset by the cost to maintain the deleted Entry objects list and reset the entry fields for reuse. To demonstrate this point, we listed the addEntry method in MyHashMap class in Listing 7.

benchmark	wall time			cpu time		
	org	mod	improvement(%)	org	mod	improvement(%)
hellono	2.6426	2.6451	-0.1	2.6301	2.625	0.2
helloworld	3.48	3.52	-1.15	3.4475	3.4726	-0.73
asac	11.9676	11.9851	-0.15	11.8826	11.8875	-0.05
nullcheck	78.945	56.1151	**28.92**	78.7625	55.9726	28.94
eigenv	65.9425	68.9275	**-4.53**	65.83	68.8125	-4.54
wig11	60.0176	60.7726	-1.26	59.205	60.0226	-1.39
dcm-sim	25.0401	25.05	-0.04	24.9526	24.9775	-0.1
lod-sim	45.8626	47.66	-3.92	45.7201	47.5426	-3.99
weka	32.9875	33.0275	-0.13	32.795	32.8925	-0.3
weka-tm	34.4325	34.6801	-0.72	34.25	34.51	-0.76

Table 13: Compilation time comparison between abc-1.2.0 version and the modified abc-1.2.0 version using MyHashSet instead of HashSet in SmartLocalDefs

From Listing 7, we know that in order to reuse Entry objects we need first check whether the deleted list is empty or not. If it is not empty, we take out the first deleted Entry object from the list and reset the field values to reuse it. So the extra overhead caused by the process to reuse Entry objects may slow down benchmarks.

Listing 7 addEntry() method in the MyHashMap.java

```
void addEntry(int hash, Object key, Object value, int bucketIndex) {
    if (deletedEntryList != null) {
        //exists deleted entry objects, reuse one
        Entry e = deletedEntryList;
        deletedEntryList = deletedEntryList.next;
        e.resetEntry(hash, key, value, table[bucketIndex]);
        table[bucketIndex] = e;
    }else{ // create new Entry object
        table[bucketIndex] = new Entry(hash, key, value, table[bucketIndex]);
    }
    if (size++ >= threshold)
        resize(2 * table.length);
}
```

5.2.3 Code optimization

Another approach we used is to optimize the source code in SmartLocalDefs and related classes. Our main aim is to reduce the methods invocation which will finally caused HashMap.addEntry method be invoked. Another aim is to get rid of unnecessary method calls and avoid unnecessary class casting. The following are the code optimizations we took in SmartLocalDefs class and related classes.

1. SmartLocalDefs.LocalDefsAnalysis.flowThrow()

Listing 8 Original code of flowThrough() method in SmartLocalDefs class in Soot 2.2.3 version

```
protected void flowThrough(Object inValue, Object unit, Object outValue) {
    Unit u = (Unit) unit;
    HashSet in = (HashSet) inValue;
    HashSet out = (HashSet) outValue;
    out.clear();
    Set mask = (Set) unitToMask.get(u);
    for( Iterator inUIt = in.iterator(); inUIt.hasNext(); ) {
        final Unit inU = (Unit) inUIt.next();
        if( mask.contains(localDef(inU)) ) out.add(inU);
    }
    Local l = localDef(u);
    if( l != null ) {
        out.removeAll(defsOf(l));
        if(mask.contains(localDef(u))) out.add(u);
    }
}
```

In the original code (as shown in Listing 8), there are two places to invoke localDef() method with same parameter ”u”. This can be reduced to one method call by introduce a variable to store the result returned from localDef() method. Another potential performance penalty in this method is that it will add local ”inU” into HashSet ”out” even if ”inU” is contained in defsOf(l). Later on, it uses removeAll() to get rid of those added local elements contained in defsOf(l). To reduce the cost, we can check whether ”inU” is contained by defsOf(l) before add it into HashSet ”out” as shown in Listing 9.

Listing 9 Modified code of flowThrough() method in SmartLocalDefs class in Soot 2.2.3 version

```
protected void flowThrough(Object inValue, Object unit, Object outValue) {
    Unit u = (Unit) unit;
    HashSet in = (HashSet) inValue;
    HashSet out = (HashSet) outValue;
    out.clear();
    Set mask = (Set) unitToMask.get(u);
    Local l = localDef(u);
    HashSet allDefUnits = null;
    if (l == null){//add all units contained by mask
       for( Iterator inUIt = in.iterator(); inUIt.hasNext(); ) {
          final Unit inU = (Unit) inUIt.next();
          if( mask.contains(localDef(inU)) )
              out.add(inU);
       }
    } else {
    //check unit whether contained in allDefUnits before add into out set.
       allDefUnits = defsOf(l);
       for( Iterator inUIt = in.iterator(); inUIt.hasNext(); ) {
          if( mask.contains(localDef(inU)) ){
             //only add unit not contained by allDefUnits
             if ( allDefUnits.contains(inU))
                out.remove(inU);
             else
                out.add(inU);
          }
       }
       out.removeAll(allDefUnits);
       if(mask.contains(l)) out.add(u);
    }
}
```

2. SmartLocalDefs.LocalDefsAnalysis.copy()

Listing 10 Original code of copy() method in SmartLocalDefs class in Soot 2.2.3 version

```
protected void copy(Object source, Object dest) {
    HashSet sourceSet = (HashSet) source;
    HashSet destSet   = (HashSet) dest;
    destSet.clear();
    destSet.addAll(sourceSet);
}
```

In this method (as shown in Listing 10), it first clear the destination set then add all elements from the source set into destination set. If some elements in the source set already contained in the destination

set, we may waste time to remove the elements then add them again. So we use two iterations to achieve the copy operation. First iteration performs intersection of two sets. Second iteration adds all elements contained in source set but not contained in destination set into destination set. Listing 11 shows the copy method after modification.

Listing 11 Modified code of copy() method in SmartLocalDefs class in Soot 2.2.3 version

```
protected void copy(Object source, Object dest) {
    HashSet sourceSet = (HashSet) source;
    HashSet destSet   = (HashSet) dest;
    if (destSet.size() > 0)
      //retain all the elements contained by sourceSet
        destSet.retainAll(sourceSet);

    if (sourceSet.size() > 0) {
      //add the elements not contained by destSet
        for( Iterator its = sourceSet.iterator(); its.hasNext(); ) {
            Object o = its.next();
            if (!destSet.contains(o)){//need add this element.
                destSet.add(o);
            }
        }
    }
}
```

3. UnitGraph.getSuccsOf()

Listing 12 Original code of getSuccsOf() method in SmartLocalDefs class in Soot 2.2.3 version

```
public List getSuccsOf(Object u)
{
    if(!unitToSuccs.containsKey(u))
       throw new NoSuchElementException("Invalid unit " + u);
    return (List) unitToSuccs.get(u);
}
```

In this method (as shown in Listing 12), the returned result of unitToSuccs.get(u) itself can be used to check whether the map unitToSuccs contains key ”u” or not. If the returned result is null means the map contains no mapping for that key. So we can avoid the call to unitToSuccs.containsKey(u) method. Listing 13 shows the getSuccsOf method after modification.

4. BackwardFlowAnalysis.doAnalysis()

   In this method (as shown in Listing 14), the TreeSet implementation uses an integer comparator to sort the elements inserted into it. The integers of the elements are assigned according to the sequence that those elements appeared in the orderedUnits. It uses a HashMap to associate the elements with their integer values. This implementation has two performance issues: (1) it introduces the expense

Listing 13 Modified code of getSuccsOf() method in SmartLocalDefs class in Soot 2.2.3 version

```
   Modified code of getSuccsOf method in UnitGraph class:
public List getSuccsOf(Object u)
{
    List l = (List) unitToSuccs.get(u);
    if (l == null) throw new RuntimeException("Invalid unit " + u);
    return l;
}
```

Listing 14 Original code snippet of doAnalysis() method in BackwardFlowAnalysis class in Soot 2.2.3 version

```
final Map numbers = new HashMap();
List orderedUnits = new PseudoTopologicalOrderer().newList(graph);
int i = 1;
for( Iterator uIt = orderedUnits.iterator(); uIt.hasNext(); ) {
    final Object u = (Object) uIt.next();
    numbers.put(u, new Integer(i));
    i++;
}

TreeSet changedUnits = new TreeSet( new Comparator() {
    public int compare(Object o1, Object o2) {
        Integer i1 = (Integer) numbers.get(o1);
        Integer i2 = (Integer) numbers.get(o2);
        return -(i1.intValue() - i2.intValue());
    }
} );
```

of creating HashMap object and putting elements with their order values into HashMap object; (2) it uses the Boxing and Unboxing conversions for getting and setting the integer values in HashMap.

To avoid using a HashMap object, we could add an integer field to the object as the order value for the comparator. After doing experiments and checking the source code of Soot, we know that the elements added into TreeSet are Unit interface type objects. After further review the source code, we added the int field ”sortOrder” into AbstractUnit class which is the ancestor of all other classes which implements Unit interface or its subinterfaces. For the Jimple internal representation, the ancestor class AbstractStmt which represents the unit Statement also extends the AbstractUnit class. Once we determine the actual object type, the next thing we need to do is to downcast the Object into AbstractUnit type and use the integer field to compare in BackwardFlowAnalysis.doAnalysis() method. The modification is shown in Listing 15.

5. ForwardFlowAnalysis.doAnalysis()

   Similarly, we made the modification for doAnalysis method in ForwardFlowAnalysis class as we did for BackwardFlowAnalysis.doAnalysis().

Listing 15 Modified code snippet of doAnalysis() method in BackwardFlowAnalysis class in Soot 2.2.3 version

```
List orderedUnits = new PseudoTopologicalOrderer().newList(graph);
int i = 1;
for( Iterator uIt = orderedUnits.iterator(); uIt.hasNext(); ) {
    final AbstractUnit au = (AbstractUnit) uIt.next();
    au.sortOrder = i;
    i++;
}

//use a field (sortOrder) in AbstractUnit object to store the order value
//for the comparator instead of using HashMap to associate object with
//its order value.
TreeSet changedUnits = new TreeSet( new Comparator() {
    public int compare(Object o1, Object o2) {
        return -(((AbstractUnit)o1).sortOrder-((AbstractUnit)o2).sortOrder);
    }
} );
```

5.2.4 Performance of combined optimization

In the above two sections, we introduced various HashSet substitute implementations and the code optimizations in Soot. What is the performance after combining these two kind of optimizations and which HashSet implementation should we use in Soot? To answer those questions, we measured the performance of the various combination optimizations.

1. HashSet + Code Optimizations Since the original abc 1.2.0 version uses the HashSet in its implementation, this combination is actually the case that only use the code optimization on the original abc 1.2.0 version.

   Table 14 shows that we get the performance improved on all benchmarks. The most improvement we get is 5.4% for eigenv benchmark.

2. THashSet + Code Optimizations

   Table 15 shows that all code optimization we took only get some speed up on the benchmarks compared with no code optimization at all (as shown in Table 9). But we still slow down on hellono and helloworld benchmarks. Compared with the combination of HashSet plus code optimizations (as shown in Table 14), the combination of THashSet plus code optimizations get worse performance.

benchmark	wall time			cpu time		
	org	mod	improvement(%)	org	mod	improvement(%)
hellono	2.6426	2.6301	0.48	2.6301	2.59	1.53
helloworld	3.48	3.4576	0.65	3.4475	3.4175	0.88
asac	11.9676	11.53	3.66	11.8826	11.4625	3.54
nullcheck	78.945	75.375	4.53	78.7625	75.1025	4.65
eigenv	65.9425	62.3826	**5.4**	65.83	61.9425	5.91
wig11	60.0176	57.8825	3.56	59.205	56.8476	3.99
dcm-sim	25.0401	24.2126	3.31	24.9526	24.055	3.6
lod-sim	45.8626	43.51	5.13	45.7201	43.1401	5.65
weka	32.9875	31.48	4.57	32.795	31.1901	4.9
weka-tm	34.4325	33.2225	3.52	34.25	32.8525	4.09

Table 14: Compilation time comparison between original abc-1.2.0 version and the modified abc-1.2.0 version (using HashSet in SmartLocalDefs plus Code Optimizations)

benchmark	wall time			cpu time		
	org	mod	improvement(%)	org	mod	improvement(%)
hellono	2.6426	2.695	-1.99	2.6301	2.6575	-1.05
helloworld	3.48	3.525	-1.3	3.4475	3.4575	-0.3
asac	11.9676	11.7251	2.03	11.8826	11.6501	1.96
nullcheck	78.945	77.4075	1.95	78.7625	77.2401	1.94
eigenv	65.9425	60.54	**8.2**	65.83	60.4476	8.18
wig11	60.0176	58.6801	2.23	59.205	57.965	2.1
dcm-sim	25.0401	24.6401	1.6	24.9526	24.5226	1.73
lod-sim	45.8626	44.0426	3.97	45.7201	43.9325	3.91
weka	32.9875	32.25	2.24	32.795	32.0025	2.42
weka-tm	34.4325	33.3375	3.19	34.25	33.13	3.28

Table 15: Compilation time comparison between original abc-1.2.0 version and the modified abc-1.2.0 version (using THashSet in SmartLocalDefs plus Code Optimizations)

3. FastSet + Code Optimizations

benchmark	wall time			cpu time		
	org	mod	improvement(%)	org	mod	improvement(%)
hellono	2.6426	2.695	-1.99	2.6301	2.6601	-1.15
helloworld	3.48	3.5726	-2.67	3.4475	3.5176	-2.04
asac	11.9676	12.425	-3.83	11.8826	12.3626	-4.04
nullcheck	78.945	78.5901	**0.45**	78.7625	78.4326	0.42
eigenv	65.9425	64.1651	**2.7**	65.83	64.0726	2.67
wig11	60.0176	59.3501	**1.12**	59.205	58.7451	0.78
dcm-sim	25.0401	25.0725	-0.13	24.9526	24.9675	-0.06
lod-sim	45.8626	51.9351	-13.25	45.7201	51.885	-13.49
weka	32.9875	33.2001	-0.65	32.795	33.01	-0.66
weka-tm	34.4325	34.2675	0.48	34.25	34.1025	0.44

Table 16: Compilation time comparison between original abc-1.2.0 version and the modified abc-1.2.0 version (using FastSet in SmartLocalDefs plus Code Optimizations)

Table 16 shows that the compilation speeds of the benchmarks get a little faster compared with only use FastSet (as shown in Table 10). Now we get three benchmarks speed up: nullcheck, eigenv and wig11. Compared with the combination of HashSet plus code optimizations (as shown in Table 14), the combination of FastSet plus code optimizations get worse performance on all benchmarks.

4. MyHashSet + Code Optimizations

benchmark	wall time			cpu time		
	org	mod	improvement(%)	org	mod	improvement(%)
hellono	2.6426	2.6451	-0.1	2.6301	2.6125	0.67
helloworld	3.48	3.4751	0.15	3.4475	3.44	0.22
asac	11.9676	11.6175	2.93	11.8826	11.5351	2.93
nullcheck	78.945	55.2151	**30.06**	78.7625	54.4676	30.85
eigenv	65.9425	61.815	**6.26**	65.83	61.4675	6.63
wig11	60.0176	58.3101	2.85	59.205	57.0175	3.7
dcm-sim	25.0401	24.3201	2.88	24.9526	24.2176	2.95
lod-sim	45.8626	44.6076	2.74	45.7201	44.315	3.08
weka	32.9875	31.6151	4.17	32.795	31.3526	4.4
weka-tm	34.4325	33.2101	**3.56**	34.25	33.0751	3.44

Table 17: Compilation time comparison between original abc-1.2.0 version and the modified abc-1.2.0 version (using MyHashSet in SmartLocalDefs plus Code Optimizations)

Table 17 shows similar results as the combination of THashSet plus code optimizations (as shown in Table 15), speed up on almost all benchmarks compared with only use MyHashSet in SmartLocalDefs class (as shown in Table 13). When compared with the combination of HashSet plus code optimizations (as shown in Table 14), it get better performance on nullcheck, eigenv and weka-tm benchmarks. Especially for nullcheck benchmarks, around 25% speed up.

5. IDHashSet + Code Optimizations

benchmark	wall time			cpu time		
	org	mod	improvement(%)	org	mod	improvement(%)
hellono	2.6426	2.6401	0.1	2.6301	2.5926	1.43
helloworld	3.48	3.4825	-0.08	3.4475	3.4275	0.59
asac	11.9676	11.875	0.78	11.8826	11.8275	0.47
nullcheck	78.945	56.2476	**28.76**	78.7625	56.0601	28.83
eigenv	65.9425	64.9725	1.48	65.83	64.975	1.3
wig11	60.0176	60.9051	-1.48	59.205	59.235	-0.06
dcm-sim	25.0401	25.0176	0.09	24.9526	24.8925	0.25
lod-sim	45.8626	45.6026	0.57	45.7201	45.4451	0.61
weka	32.9875	33.1425	-0.47	32.795	32.9501	-0.48
weka-tm	34.4325	35.1151	-1.99	34.25	34.9426	-2.03

Table 18: Compilation time comparison between original abc-1.2.0 version and the modified abc-1.2.0 version (using IDHashSet in SmartLocalDefs plus Code Optimizations)

Table 18 shows that we only get notable improvement on nullcheck benchmark, around 29% speed up.

From the above experiments and analysis, both the combination of HashSet plus code optimizations and the combination of MyHashSet plus code optimizations are good solutions for the java.util.HashMap.addEntry bottleneck.

5.2.5 Profiling result after optimization

After applied the code optimization on abc 1.2.0 (the combination of HashSet plus code optimizations), we profiled all the benchmarks again and found that java.util.HashMap.addEntry is still the bottleneck for most benchmarks. But the results from Table 19 shows the percentages that java.util.HashMap.addEntry accounts for are all become smaller than the original result shown in Table 8, especially for eigenv benchmark which now only accounts for 7.69% of total time profiling, 3.3% percent down. And the total percentage of top 5 for compiling eigenv benchmark is from 30.41% down to 21.54%.

rank	self	accum	count	trace	method
1	7.69%	7.69%	17904	11322	java.util.HashMap.addEntry
2	5.51%	13.20%	12831	11318	java.util.HashMap.addEntry
3	4.79%	17.99%	11143	11319	java.util.HashMap.addEntry
4	2.34%	20.33%	5435	12679	java.util.AbstractList.iterator
5	1.22%	21.54%	2834	13501	java.util.HashMap.addEntry

Table 19: Top 5 time consuming methods by using Xrunhprof to profile abc-1.2.0 version compiling eigenv benchmark

5.3 Abc Modification

During profiling and comparing the generated class files, we found that abc does not generate identical codes sometimes. The mainly difference as we showed in using Soot to optimization abc in 5.2, the names of some inlined methods are different. After examining the source code about abc inlining and debugging the running result, we found that during around inlining abc produce many duplicated methods. And after finish inlining, abc normalized the inlined methods to compare the inlined methods and removed the duplicated ones.

The procedure to create new inlined methods and the comparison of two methods are both expensive. To create an new inlined method, it not only takes time to create method signature but also needs to copy each statement from the method to be inlined. And finally it also need to modify the original method invocation statement to invoke the new inlined method.

The way to compare whether two methods are identical is also expensive, it needs three steps:

1. Normalize method name and local variables,
2. Construct a string by contacting the method signature and statements in the method body,
3. Compare the String representation of the methods are identical or not.

If we could avoid creating the duplicated inline methods in the first place we can not only avoid creating new methods but also could reduce the number methods we need to compare. Another reason we could speed up abc by changing the algorithms of weaving and inlining phases is based on the fact that weaving and inlining stages are the most time consuming phases in the whole compilation process from our profiling result described in section 4.

In order to change the algorithms in weaving and inlining phases, we need to figure out the original ones.

The original procedure for around weaving:

1. For each matched joinpoint, assign a unique shadow id to identify it;
2. Extract the shadow codes to form a new method. To avoid confusing with shadow method, we call it the shadow extracted method. The naming of new created shadow extracted method uses the unique shadow id with ”shadow$” as prefix;
3. Replace the shadow codes with the method invocation to the shadow extracted method;
4. Move the shadow extracted method call into proceed method. The shadow extracted method call is insert into switch statement based on the unique shadow id in the proceed method;
5. Replace the shadow extracted method call with ”around$” advice method call which in turn invokes the proceed method call;

The original procedure of around advice inlining is for each shadow method which originally contains the matched joinpoint, recursively inline the methods which are called in its body.

According to the original procedure of around weaving, the method invocation contained in the shadow methods are ”around$” methods. So for around advice inlining, the process is as follows:

1. When inlining ”around$” methods invoked in shadow method, inline proceed methods which are called in the ”around$” methods into advice class;

2. Before inlining ”proceed$” methods, inline the ”around$” methods which is called in the proceed methods.

After recursively inlining the ”around$”, ”proceed$” methods, the body of ”proceed$” methods will be moved into ”around$” methods and the ”around$” methods are renamed as ”inline$” methods.

After weaving and inlining, the original abc used an additional optimization phase to remove duplicated methods and unused methods.

This implementation strategy is clean: at each phase, it only focuses on one thing without considering other tasks. However it also has its drawbacks as it increases compilation time.

In the original abc-1.2.0 version it does not check whether shadow extracted methods are identical or not when weaving around advices. So no matter whether two shadow extracted methods are identical or not, abc will generate all of them in the target classes. For example, in SortItemMain.jimple decompiled from SortItemMain.class generated by abc-1.2.0, there are two identical shadow extracted methods shadow$30 and shadow$45, as showing in the Listing 16.

In the modified abc, before creating a new shadow extracted method, we check whether there is any identical shadow extracted method already residing in the shadow class or not. If there is one, we will use that one without creation of an new one. By doing this, we can get several benefits:

1. Avoid creating a new shadow method when there exists duplicated one;
2. Save time in later phase that removes unused methods;
3. Reduce the target classes size.

   When we check the decompiled SortItemMain.class file generated by our modified abc-1.2.0 version, only shadow$30 is remained in it.

Another performance issue in the original procedure of around advice inlining is that it does not check whether the new created ”inline$” method exists in the advice class or not. So this cause abc generates many duplicated methods and uses an additional phase to remove those duplicated methods.

To improve the performance, when doing inlining, we first check whether there is an identical ”inline$” method in the target class or not by using the signature constructed by contacting three integer type arguments values, arguments types, method’s signature and target class name. If there exits one, we just reuse it instead of creating an new ”inline$” method.

Similarly, we can

1. Avoid creating a new ”inline$” method when there exists duplicated one;
2. Save time in later pass to check and remove duplicated methods.

In our test suite, asac, nullcheck and eigenv benchmarks all contain around advices that could be inlined. The result in Table 20 shows that we do get much improvement for those three benchmarks. For nullcheck benchmark, we get great improvement, around 32% speed up. We also get speed up around 8% for asac and 4% for eigenv benchmark.

Listing 16 Jimple code snippet in SortItemMain class generated by original abc-1.2.0 version

```
public static final java.lang.String shadow$30
(AroundAspects, java.lang.StringBuffer){
   AroundAspects r0;
   java.lang.StringBuffer r1;
   java.io.PrintStream $r3;
   java.lang.String $r4;
   r0 := @parameter0: AroundAspects;
   r1 := @parameter1: java.lang.StringBuffer;
   if r0 != null goto label0;
   staticinvoke <AroundAspects: AroundAspects aspectOf()>();

label0:
   $r3 = <java.lang.System: java.io.PrintStream err>;
   virtualinvoke $r3.<java.io.PrintStream:
       void println(java.lang.String)>("before method call");
   $r4 = virtualinvoke r1.<java.lang.StringBuffer:
       java.lang.String toString()>();
   return $r4;
}

public static final java.lang.String shadow$45
(AroundAspects, java.lang.StringBuffer){
  AroundAspects r0;
  java.lang.StringBuffer r1;
  java.io.PrintStream $r3;
  java.lang.String $r4;
  r0 := @parameter0: AroundAspects;
  r1 := @parameter1: java.lang.StringBuffer;
  if r0 != null goto label0;
  staticinvoke <AroundAspects: AroundAspects aspectOf()>();

label0:
  $r3 = <java.lang.System: java.io.PrintStream err>;
  virtualinvoke $r3.<java.io.PrintStream:
      void println(java.lang.String)>("before method call");
  $r4 = virtualinvoke r1.<java.lang.StringBuffer:
      java.lang.String toString()>();
  return $r4;
}
```

Besides the improvement on the compiling speed of those benchmarks, we also reduce the size of generated class files by removing the duplicated extracted shadow methods. [4] As shown in Table 21, for the bench-

[4] Here the ”duplicated” means that the two methods are same except the method name.

benchmark	wall time			cpu time		
	org	mod	improvement(%)	org	mod	improvement(%)
hellono	2.6426	2.6451	-0.1	2.6301	2.5875	1.62
helloworld	3.48	3.48	0.0	3.4475	3.4101	1.09
asac	11.9676	11.03	**7.84**	11.8826	10.935	7.98
nullcheck	78.945	53.695	**31.99**	78.7625	53.625	31.92
eigenv	65.9425	63.4751	**3.75**	65.83	63.355	3.76
wig11	60.0176	59.8301	0.32	59.205	59.0226	0.31
dcm-sim	25.0401	25.105	-0.26	24.9526	25.0051	-0.22
lod-sim	45.8626	46.0625	-0.44	45.7201	45.9401	-0.49
weka	32.9875	32.87	0.36	32.795	32.7125	0.26
weka-tm	34.4325	34.41	0.07	34.25	34.295	-0.14

Table 20: Compilation time comparison between original abc-1.2.0 version and the modified abc-1.2.0 version (avoiding duplicated methods)

marks with around advices (asac, nullcheck and eigenv benchmark)[5], we reduce the generated class files size greatly. We reduce the half class file size for eigenv benchmark, 22% for nullcheck and 15% for asac benchmark.

benchmark	generated classes file size		
	org	mod	reduced(%)
hellono	465	465	0
helloworld	2228	2228	0
asac	64781	54998	**15.1**
nullcheck	301964	236743	**21.6**
eigenv	157596	79955	**49.27**
wig11	761894	761894	0
dcm-sim	240105	237682	1.01
lod-sim	252020	252020	0
weka	313900	313900	0
weka-tm	326871	326871	0

Table 21: File size (in byte) comparison between the classes generated from original abc-1.2.0 version and the modified abc-1.2.0 version (avoiding duplicated methods)

5.4 Using Soot to optimize abc

Besides the manual optimization described in the previous sections, we also tried to use the Soot optimization tool to automatically optimize abc. The reason that we do this is that if the optimization tool can speed up abc then we can study what kind of optimizations performed by optimization tool and take advantage of them to manually optimize the abc source code.

[5]Although dcm-sim benchmark also contains around advices, the around advices are applied in very few places in the application. So we did not reduce the generated class file size greatly.

Since Soot itself is a Java optimization framework, we used it to optimize abc and see what is the effect of using Soot to optimize abc compiler.

5.4.1 Process to use Soot to optimize abc

The process to use Soot to optimize abc is as follows:

1. Compile abc and generate class files.
2. Use Soot to optimize those generated abc class files (only use -app option).
3. Copy soot/baf/toolkits/base/peephole.dat into corresponding Soot output directory.
4. Use the jar command to pack generate classes into a new abcsoot.jar.
5. Time the compilation time of abc with the new abcsoot.jar package.

5.4.2 Profiling result of Soot-optimized abc

After using Soot-optimized abc, the compilation speed of all benchmarks are reduced, especially for hellono, helloworld, asac benchmarks (speed up nearly 10.89%, 7.62% and 3.83% respectively). [6] The comparison result is shown in Table 22.

benchmark	wall time			cpu time		
	org	mod	improvement(%)	org	mod	improvement(%)
hellono	2.6426	2.355	10.89	2.6301	2.3025	12.46
helloworld	3.48	3.2151	7.62	3.4475	3.1825	7.69
asac	11.9676	11.51	3.83	11.8826	11.415	3.94
nullcheck	78.945	78.175	0.98	78.7625	78.0075	0.96
eigenv	65.9425	65.3276	0.94	65.83	65.2051	0.95
wig11	60.0176	59.5725	0.75	59.205	58.8101	0.67
dcm-sim	25.0401	24.5626	1.91	24.9526	24.455	2.0
lod-sim	45.8626	45.6876	0.39	45.7201	45.5726	0.33

Table 22: Compilation time comparison between abc-1.2.0 version and the Soot-optimized abc-1.2.0 version

5.4.3 Analysis of Soot-optimized abc

In order to know whether Soot-optimized abc generate identical class files with the ones generated by original abc, we wrote the program to automatically compare generated class files with original abc generated class files, decompile the class files into Jimple files if the class files are different and differentiate the Jimple files to generate report. The following are the differences observed by examining the generated report files.

1. the inlined method name are different: (as shown in Listing 17) [7]

[6] At the begining, when we running our tests on a single CPU computer we get great improvement on hellono, helloworld, asac and nullcheck benchmarks, nearly 40%, 33%, 15% and 31% respectively.

[7] The method body of inline126around$18 and inline$114$around$18 are same.

Listing 17 Difference of Jimple codes decompiled from SortTrace.class generated by original abc-1.2.0 version and the Soot-optimized abc-1.2.0 version

```
138c138  original abc-1.2.0
<  staticinvoke <SortTrace: java.lang.Object inline$126$around$18
   (SortTrace,int[],int,int,QSortAlgorithm)>(r4, r1, i0, i20, r0);
---      Soot-optimized abc-1.2.0
>  staticinvoke <SortTrace: java.lang.Object inline$114$around$18
   (SortTrace,int[],int,int,QSortAlgorithm)>(r4, r1, i0, i20, r0);
```

2. The order of some methods in the class files are different.

3. Some goto labels are different: As shown in Listing 18 and Listing 19, there are several differentness: the statement at ”label126:” , ”label28:” and ”label29:”. [8] Despite the fact that goto labels are different, the semantics of these two code snippets are same.

Listing 18 Jimple code of decompiled from the class file generated by original abc-1.2.0 version

```
label24:
   $r41 = <java.lang.System: java.io.PrintStream out>;
   virtualinvoke $r41.<java.io.PrintStream:
      void println(java.lang.String)>("Error writing to file");
   goto label28;
label25:
   $r42 := @caughtexception;
   r43 = $r42;
label26:
   goto label29;
label27:
   throw r43;
label28:
   if r30 == null goto label31;
   virtualinvoke r30.<java.io.PrintStream: void close()>();
   goto label31;
label29:
   if r30 == null goto label27;
   virtualinvoke r30.<java.io.PrintStream: void close()>();
   goto label27;
```

From above analysis, we know the Soot-optimized abc generates slightly different code from original abc. But the differences are minor and do not affect the functionality. So we can be sure that Soot-optimized abc generate class files are equal to original abc generated class files. Then the question comes up, why is Soot-optimized abc compiler faster compared with original abc compiler and what kind of optimization

[8] The target label31 codes are same.

Listing 19 Jimple code of decompiled from the class file generated by Soot-optimized abc-1.2.0 version

```
label24:
   $r41 = <java.lang.System: java.io.PrintStream out>;
   virtualinvoke $r41.<java.io.PrintStream:
      void println(java.lang.String)>("Error writing to file");
   goto label29;
label25:
   $r42 := @caughtexception;
   r43 = $r42;
label26:
   goto label28;
label27:
   throw r43;
label28:
   if r30 == null goto label27;
   virtualinvoke r30.<java.io.PrintStream: void close()>();
   goto label27;
label29:
   if r30 == null goto label31;
   virtualinvoke r30.<java.io.PrintStream: void close()>();
   goto label31;
```

happened by using Soot to optimize abc compiler? To answer these questions, we need to profile these two compilers and see which phase we get great improvement during compiling the benchmarks.

Table 23 shows the timing result at each phase for original abc 1.2.0 version and Soot-optimized abc 1.2.0 version. [9]

In order to identify which phase account for most of the overall improvement, based on Table 23, we compute the percentage of improvement on each phase account for total overall improvement by (org phase - Soot phase)/(org total time - Soot total time)*100. Table 24 shows the generated results. In this table, for each benchmark, the phase with largest positive value is the phase we get greatest improvement. If the values are negative at some phases, that means we get slow down at those phases.

Table 23 and Table 24 show that the Soot-optimized abc speeds up compilation at Init. of Soot phase, Loading Jars phase, Create Polyglot Compiler phase and Polyglot phase. For all other phases, Soot-optimized abc compiler did not get all benchmarks speed up. It speed up some benchmarks but at the same time it also slow down other benchmarks.

In order to know the reason that Soot-optimized abc get speed up at Init. of Soot phase for all benchmarks, we profiled the classes loaded at Init. of Soot phase. There are total 83 basic classes (93 for trace matching) are loaded at this phase for both original abc version and Soot-optimized abc. Those 83 basic classes are from java.io, java.lang, org.aspectbench.runtime and org.aspectj.lang packages. Comparing the loading time for each class, we see that Soot-optimized abc load many classes in shorter time compared with original abc did. We decompiled the class files related to Init. of Soot phase, soot.Scene class and soot.SootResolver class, but we did not see many differences between original abc version and Soot-optimized abc version.

[9] As the space limited, here we only list 6 benchmarks.

phase name	hellono		helloworld		asac		nullcheck		eigenv		wig11	
	org	Soot	org	Soot	org	Soot	org	Soot	org	Soot	org	Soot
Init. of Soot	974	818	978	816	980	818	971	823	980	817	972	824
Loading Jars	4	4	4	3	19	6	37	25	4	4	680	192
Create polyglot compiler	245	219	244	218	244	220	243	222	256	220	244	219
Polyglot phases	506	445	877	868	1569	1526	2550	2480	1676	1601	17074	16048
Initial Soot resolving	80	75	85	74	188	166	989	928	105	101	571	1394
Soot resolving	1	0	0	0	0	0	0	0	0	0	1	0
Aspect inheritance	0	0	2	2	2	2	1	1	2	2	1	1
Declare Parents	1	1	1	2	2	1	2	1	2	1	1	1
Intertype Adjuster	15	22	13	16	13	20	13	17	13	17	72	54
Jimplification	90	74	260	175	574	542	985	953	1030	1012	6661	6695
Fix up constructor calls	1	1	1	1	1	1	1	1	1	1	1	1
Update pattern matcher	147	132	84	67	198	180	106	100	76	60	56	64
Weave Initializers	9	8	7	8	9	8	21	13	7	8	73	51
Load shadow types	0	0	0	0	0	0	0	0	0	0	0	0
Compute advice lists	47	44	108	98	305	301	545	522	397	511	4798	4850
Add aspect code	12	14	20	22	13	14	14	14	20	18	14	14
Weaving advice	17	17	136	130	1616	1593	6301	6287	20593	20832	5469	5606
Exceptions check	3	2	5	4	18	18	39	37	586	31	82	75
Advice inlining	6	7	37	36	1350	1357	25675	25761	14983	15198	2860	1870
Interproc. constant propagator	11	8	1	4	147	145	2205	2209	1535	1578	72	73
Boxing remover	0	1	5	5	450	450	874	868	365	362	48	48
Duplicates remover	0	0	6	4	654	648	18086	18068	2326	2328	635	653
Removing unused methods	1	1	4	5	35	34	227	230	84	82	90	92
Specializing return types	0	0	0	0	19	20	170	162	60	59	0	1
Soot Packs	101	106	174	159	2425	2413	14745	13756	17297	17458	14214	15075
Soot Writing Output	100	107	151	149	739	749	2785	3799	1542	1562	5073	4900
total result	2371	2106	3203	2866	11570	11232	77585	77277	63940	63863	59762	58801

Table 23: Compilation time (in millisecond) comparison at each phase between original abc-1.2.0 version and the Soot-optimized abc-1.2.0 version

The main difference are the order of some declaration statements and the names of local variables. Finally, we realized that the real reason is that the library path we set to run those two compilers. When we use Soot to optimize abc, it will automatically include the application packages abc used and optimize them too. So the Soot-optimized abc jar file contains the tool packages it used such as Polyglot, Soot, Jimple and so on. Thus when run Soot-optimized abc, we did not set the tool packages in the classpath as we did for original abc. After we put the tool packages and original abc into one jar and set the classpath like we did for Soot to optimize abc, we did the experiment again. This time, Soot to optimize abc only get little speed up at Init. of Soot phase, from 0.3 to 3.6%.

To know why Soot-optimized abc get speed up in other phases for some benchmarks, we compared the Soot-optimized abc class files with original abc class files and decompiled the different files into Jimple codes. However we did not find many differences between the decompiled Jimple codes that could be the reason for speeding up the compilation speed.

6 Conclusion

In section 5.2, we described various soot optimizations and the combinations. According to the experimental results, we found that there is no obvious better one between using HashSet and MyHashSet in flow analysis. In this section, we use those two optimizations to combine with abc optimization to evaluate the overall improvements. In Table 25, we shows the individual optimization and two overall combination op-

phase name	the improvement(%) at each phase							
	hellono	helloworld	asac	nullcheck	eigenv	wig11	dcm-sim	lod-sim
Init. of Soot	58.87	48.08	47.93	48.06	211.69	15.41	51.63	704.55
Loading Jars	0.00	0.30	3.85	3.90	0.00	50.79	11.04	190.91
Create polyglot compiler	9.82	7.72	7.11	6.82	46.75	2.61	7.80	113.64
Polyglot phases	23.02	2.68	12.73	22.73	97.40	106.77	4.23	400.0
Initial Soot resolving	1.89	3.27	6.51	19.81	5.19	-85.64	19.16	309.1
Soot resolving	0.38	0.00	0.00	0.00	0.00	0.11	0.00	0.0
Aspect inheritance	0.00	0.00	0.00	0.00	0.00	0.00	-0.33	0.0
Declare Parents	0.00	-0.30	0.30	0.33	1.30	0.00	0.33	0.0
Intertype Adjuster	-2.65	-0.90	-2.08	-1.30	-5.19	1.88	0.98	-18.19
Jimplification	6.04	25.23	9.47	10.39	23.38	-3.54	-0.65	109.1
Fix up constructor calls	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.0
Update pattern matcher	5.67	5.05	5.33	1.95	20.78	-0.84	2.28	36.37
Weave Initializers	0.38	-0.30	0.30	2.60	-1.30	2.29	0.00	4.55
Load shadow types	0.00	0.00	0.00	0.00	0.00	0.00	0.00	4.55
Compute advice lists	1.14	2.97	1.19	7.47	-148.05	-5.42	-8.45	4.55
Add aspect code	-0.76	-0.60	-0.30	0.00	2.60	0.00	0.00	-4.55
Weaving advice	0.00	1.79	6.81	4.55	310.39	-14.26	-6.17	504.55
Exceptions check	0.38	0.30	0.00	0.65	720.78	0.73	0.65	4.55
Advice inlining	-0.38	0.30	-2.08	-27.93	-279.22	103.02	-7.47	1613.64
Interproc. constant propagator	1.14	-0.90	0.60	-1.30	-55.84	-0.11	0.00	0.0
Boxing remover	0.00	0.00	0.00	1.95	3.90	0.00	2.28	-4018.19
Duplicates remover	0.00	0.60	1.78	5.85	-2.60	-1.88	-0.65	0.0
Removing unused methods	0.00	-0.30	0.30	-0.98	2.60	-0.21	0.00	4.55
Specializing return types	0.00	0.00	-0.30	2.60	1.30	0.00	0.33	0.0
Soot Packs	-1.89	4.46	3.56	321.11	-209.09	-89.60	1.30	213.64
Soot Writing Output	-2.65	0.60	-2.96	-329.23	-25.97	18.01	21.76	-77.28

Table 24: Improvement at each phase by using Soot-optimized abc-1.2.0 version

timizations. We can see that when using the combination of MyHashSet, code optimization and abc around inlining (”All 2” column in Table 25), the compilation speed of almost all benchmarks is slower than the using of the combination of HashSet, code optimization and abc around inlining (”All 1” column in Table 25). The only exception is nullcheck benchmark where the compilation speed of using ”All 1” is around 24% slower than ”All 2”. [10]

So we choose ”All 1” as our optimization solution for abc. By using this solution, we get 35% speed up when compile nullcheck benchmark, 12% for eigenv benchmark and 10% for asac benchmark.

As to Soot-optimized abc, it only get speed up benchmarks slightly after we reset the classpath setting to keep consistent with original abc. So it is not worth to deploy a soot-optimized abc.

Another conclusion we get by doing this project is that to speed up a application the most efficient way is to change the inefficient algorithms. By changing the inefficient algorithms you may get unexpected great speed up as we did in this project.

7 Future Work

In this project, although we detected and relieved some bottlenecks during the process of abc compilation, there are still rooms to further optimize abc. Here I list two issues that may be worth further addressing in the future.

[10]The compilation time of using ”All 1” is 51.69s. The compilation time of using ”All 2” is 39.035s.

benchmark	org wall time	wall time improvement(%)					
		Soot Optimization			abc around inlining (d)	All 1: (a) + (d)	All 2: (c) + (d)
		HashSet + code opt (a)	MyHashSet (b)	MyHashSet + code opt (c)			
hellono	2.6426	0.48	-0.10	-0.10	-0.10	0.10	-0.19
helloworld	3.4800	0.65	-1.15	0.15	0.00	1.08	-0.08
asac	11.9676	3.66	-0.15	2.93	7.84	**10.47**	10.35
nullcheck	78.9450	4.53	28.92	30.06	31.99	**34.53**	**50.56**
eigenv	65.9425	5.40	-4.53	6.26	3.75	**12.08**	9.81
wig11	60.0176	3.56	-1.26	2.85	0.32	4.40	4.01
dcm-sim	25.0401	3.31	-0.04	2.88	-0.26	3.35	3.47
lod-sim	45.8626	5.13	-3.92	2.74	-0.44	5.13	4.51
weka	32.9875	4.57	-0.13	4.17	0.36	4.69	4.61
weka-tm	34.4325	3.52	-0.72	3.56	0.07	3.90	3.90

Table 25: Compilation time comparison between original abc-1.2.0 version and the abc-1.2.0 version with all modifications combined

- Flow Analysis

  In this project, although we relieve the severity of the bottlenecks related to Flow Analysis, there still may be better solution to fully remove them. In this project, we also tried to use bit vector to replace the HashSet. But due to the variation of the number of elements between benchmarks, this attempt is less efficient than using HashSet.

- Inline ”shadow$” methods

  Currently, abc keeps shadow extracted methods in the shadow class in order to keep it consistent with the original source code arrangement: advices are in advice classes and the shadow extracted methods are kept in the shadow classes. In the decompiled Jimple code, we can see many short ”shadow$xxx” method in the shadow classes. If we can inline those small extracted shadow methods into advice classes instead of keeping them in shadow classes, it should speed up the execution of the generated classes. However the drawback of doing this is that it may slow down the compilation speed and increase the generated class file size.

References

[1] abc: The AspectBench Compiler for AspectJ, ”http://abc.comlab.ox.ac.uk/introduction”

[2] Soot: a Java Optimization Framework, ”http://www.sable.mcgill.ca/soot/”

[3] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhotak, Ondrej Lhotak, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble, ”Building the abc AspectJ compiler with Polyglot and Soot”, Technical Report abc-2004-4, Dec. 2004

[4] Sascha Kuzin, ”Efficient Implementation of Around-Advice for the AspectBench Compiler”, MSc dissertation, Oxford University, September 2004

[5] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhotak, Ondrej Lhotak, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble, ”Optimising AspectJ”, PLDI 2005, Chicago, USA, June 2005

[6] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhotak, Ondrej Lhotak, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble, ”abc: An extensible AspectJ compiler”, AOSD 2005, Chicago, USA, March 2005

[7] AspectJ Eclipse project, ”http://www.eclipse.org/aspectj/”

[8] JProfiler, ”http://www.ej-technologies.com/products/jprofiler/overview.html”, EJ Technologies

[9] Jack Shiraza, ”Java Performance Tuning”, 2nd Edition, O’Reilly, 2003

[10] John Jorgensen, ”Speed the Plow - Improving the performance of Soot”, CS621 coruse project report, April 2001

[11] L.Hendren, C.Verbrugge, O.deMoor, and G.Sittampalam, B.Dufour, C.Goard, ”Measuring the dynamic behaviour of aspectj programs”, Sable Technical Report 2004-2, March 2004

[12] GNU Trove: High performance collections for Java, ”http://trove4j.sourceforge.net/”

[13] Javolution: Java library for real-Time, embedded and high-performance applications, ”http://javolution.org/”

[14] Polyglot, ”http://www.cs.cornell.edu/projects/polyglot/”

[15] Ivan Kiselev, ”Aspect-Oriented Programming with AspectJ”, Sams, 2002

[16] PARC: the origin of AspectJ project, ”http://www.parc.com/research/projects/aspectj/default.html”

[17] The HPROF Profiler Agent, ”http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html#hprof”

[18] HAT: Heap Analysis Tool, ”https://hat.dev.java.net/”