McGill University
School of Computer Science
Sable Research Group

Common Subexpression Removal in Soot

Sable Technical Report 2000-1

Patrick Lam

May 1, 2000

Abstract

Everyone who knows something about optimizing compilers knows about common subexpression elimination. It is used to remove redundant computations, which usually improves the execution time of a program. The transformation is quite simple to describe, and so common subexpression elimination must, of course, therefore be trivial to implement. In this work, we discuss an implementation of common subexpression elimination for Java, using the Soot framework. Our implementation passes the Soot sentinel suite. Experimental results are presented, showing that common subexpression elimination slightly speeds up Java programs.

1 Introduction

All serious compilers include common subexpression elimination. Until now, the Soot framework has lacked such a thing. This project fills in the gap. We have implemented common subexpression elimination in Soot and provide experimental results measuring its impact.

This project recaps the standard available expressions analysis, then describes our implementation of common subexpression elimination based on it. We then discuss some issues which made the implementation of common subexpression elimination not quite trivial. Next, we provide many experimental results, showing the limited, but visible, effect of common subexpression elimination. Finally, we conclude and discuss future work.

2 The Available Expressions Analysis

For completeness, we include a description of *available expressions*, which is the flow analysis needed to do common subexpression elimination. This has entered into common knowledge for compilers people; however, an account of available expressions is in [Muc97]. We will also describe what our implementation of this analysis actually computes. This section also includes, as a bonus, a description of a simple side-effect tester, used in our analysis.

The available expressions analysis acts on the universe of "expressions" in the program; for instance, $x + y$ or $y - 8$ might be available expressions. We want to know, at a program point p, which expressions e have been computed on all paths. We also require that none of the operands to e have been modified between the computation point and p.

In our implementation, we mean by "expressions" the following things:

- all instances of the Jimple class `Expr`; e.g. `AddExpr, LengthExpr`, except for `NewExpr`, `NewArrayExpr`, `NewMultiArrayExpr`; we also exclude all `InvokeExpr`s.
- all instances of the Soot class `FieldRef`. We have `InstanceFieldRef`s and `StaticFieldRef`s to contend with.

The presence of the latter significantly improves the potential gains realizable by this transformation; it also imposes some type of side-effect testing.

We do not deal with array references (instances of `ArrayRef`). We expect that adding array references to the universe of available expressions would not give significant improvement, as the index of the reference should usually vary quite erratically.

The universe of expressions is gathered by collecting everything that appears on the right-hand side of an `AssignStmt`. It cannot be constructed explicitly; see section 4.2 for a discussion.

This analysis is a forward all-paths one; the merge operator is intersection. In the initial approximation, the start node has no expressions available, while at all other nodes, all expressions are available.

The dataflow equations are fairly simple. At each assignment statement, we simply generate the expression or field reference on the right-hand side, unless it is a `NewExpr`, `NewArrayExpr`, `NewMultiArrayExpr` or a `InvokeExpr`. The simple form of Jimple guarantees that this captures all computed expressions.

To handle kills at u, we remove all field references which could possibly be written to by u, and also all expressions using values which can possibly be written to by u.

This gives the following equation:

$$\mathrm{OUT}(s) = (\mathrm{IN}(s) \cup \mathrm{gen}(s)) \cap \mathrm{kill}(s)$$

where gen and kill are as described above.

2.1 Our Side Effect Analysis

Of course, the above statement requires some side-effect analysis! We implemented a simple one to start with; the interface should allow any other side-effect analysis to easily be plugged in.

In fact, our side-effect analysis is the simplest correct one which offers some hope of being useful.

If a unit u contains an invoke expression, then all values v which are not local variables can be written to.

Otherwise, Soot's "def boxes" are almost all we need. In particular, each unit u has a list of units defined at u. We iterate through this list, checking that neither v nor any value used in v is defined at u.

Furthermore, if we are checking that a field reference `r0.<foo: int f>` cannot be written by a unit u, then we must check that u does not contain any write to a field `<foo: int f>`, regardless of base.

This is not new – the same rules were used, for instance, in the aggregator. This implementation is cleaner, since we have separated out the side effect information and put it in its own class.

3 Common Subexpression Elimination

Once we have the available expressions information, eliminating common subexpressions is quite simple. Here is our strategy for eliminating common subexpressions from a method m:

- Compute the available expressions information.
- Iterate through the statements of m, looking for a computation of some expression e at a statement s, with e available at s.

- If found, replace the computation of e by a reference to a temporary local variable `cseTmpN` at both computation sites, and add a statement computing e before the first computation site.

An example of a common subexpression elimination follows:

```
int f;

public void foo()
{
    x = this.f;
    y = this.f * 2;
}
```

We can see that the second read to `this.f` is redundant. We present the following before-and-after Jimp picture:

```
x = this.f;                $cseTmp0 = this.f;
$i0 = this.f;        ⇒     x = $cseTmp0;
y = $i0 * 2;               y = $cseTmp0 * 2;
```

We have thus eliminated an unnecessary field access.

Note that the actual implementation has to deal with some object-equality issues. For instance, two different instances of `x * 2` would not `.equals()` each other. We discuss this problem in section 4.1.

Some would advocate a basic-block based analysis for available expressions, thus giving faster common subexpression elimination. This is slightly less precise, and we will show in section 5.2 that our analysis is already sufficiently fast.

We assume that the program under consideration has no concurrency. In [Lam00], a technique for extending sequential flow analyses to automatically handle concurrency is presented.

4 The Perils of Implementation

There were several nontrivial impediments to an implementation of common subexpression elimination. We describe them in this section.

4.1 The Object Equality Problem

We are now ready to actually carry out common subexpression elimination. Say that we have the following code:

```
l0:  x = y + z;
l1:  w = y + z;
```

It is clear that $y + z$ should be available at `l1`, and so we can eliminate the common subexpression.

Caveat: The two $y + z$ expressions are represented by different objects. How do we test their equality?

- Object equality is not good enough; it would never tell us about any possible eliminations, because all `Expr` objects are distinct.
- Overriding `.equals()` to mean "structurally equal" is a really bad idea. Here are two potential problems.

  Given an `Expr`, we can create a map back to the containing unit, so that we don't need to carry around the unit where an expression was computed in our abstraction. This can no longer be done if we equate all structurally equal values.

  Also, if carried out to its logical conclusion, we would have to make `.equals()` mean structural equality on units. This is disastrous. It would make it impossible to put two `goto l1` statements in a `Chain`!
- We thus choose to define an `EquivTo` interface, with an `.equivTo()` method. We extend all `Value` objects to implement this interface.

The chosen solution was somewhat painful to implement. However, we expect that it will turn out to be essential for other expression-based analyses in the future. This code has been contributed to the main codebase.

4.2 Exploding Universes

An earlier implementation of available expressions was extremely slow, as opposed to simply being slow. Blame for this was assigned to the creation of huge `FlowUniverse`s, containing every single expression referenced in a class. In order to avoid this, the flow universe is never explicitly created; we instead use `ArraySparseSet`, which allows us to refer to objects as needed.

To do flow analysis on this universe, we can still create the generate sets in a preliminary phase. In particular, we populate a map from `Unit` to `ArraySet`s, containing all of the generated objects at each unit. However, the number of killed or preserved objects at each unit is prohibitively large; we instead iterate through the flow set and kill objects, as described in the analysis rules, on-the-fly.

There is, however, yet another difficulty. To compute the least fixed point for an all-paths analysis, it is necessary to initially assume $\top$ everywhere except for the start node. This would appear to be impossible in the presence of implicit universes.

Fortunately, some clever use of object-oriented technology saves the day. We create an implementation of `FlowSet` called `ToppedSet`. This set delegates all of its behaviour to a `FlowSet` object specified in its constructor – we use `ArraySparseSet` in our implementation – but it can also be set to $\top$ using the `setTop()` call. If $s = \top$, then for any topped set x, $x \cup s = \top$, while $x \cap s = x$. This does just the right thing.

4.3 Soot Bugs and Oddities

This project exposed two Soot bugs. This is not surprising, as a project of this size is certain to have bugs, regardless of the amount of testing performed.

The introduction of $\top$ also prompted the addition of code to the flow function preventing one from flowing through a unit when the IN set equals $\top$. Surprisingly, this condition was triggered by the `jack_i` benchmark. It turns out that Jimplification generates `nop` instructions for unreachable code. A standard pass in Soot performs nop elimination, leaving empty traps and `r2 = @caughtexception` statements in its wake. The latter are bad for flow analysis, sending $\top$ to unsuspecting `flowThrough` functions. We had to add an unreachable code elimination phase to eliminate this source of trouble.

The second bug is perhaps not quite a bug, but certainly qualifies as unexpected behaviour. To compute the kills at a statement, we obviously put the flow set into a list and iterate over it, removing items as necessary. However, unlike everywhere else in Soot, this list happens to be backed; removing items from the list also removes them from the `FlowSet`, and breaks the iteration without producing a `ConcurrentModificationException`. This bug was fixed by cloning the list produced from the `FlowSet`.

4.4 Sentinels are Insensitive

Some rather major bugs in the implementation managed to get past large parts of the Sentinel suite.

In available expressions, we must generate expressions, and then kill expressions. An earlier implementation found that the reverse order – kill, then gen – only broke at most 6 benchmarks, and some of these broke because of other reasons.

Another bug was an incomplete implementation of side effect testing. In particular, we initially forgot to deal with this situation:

```
foo o, p;

o = new foo();
p = o;
x = o.f;
p.f = 5;
y = o.f;
```

Clearly, the read of `o.f` is not available after the assignment to `p.f`; this was discussed in section 2.1. This was not detected at all by the Sentinel – no benchmark failed because of this error. When corrected, only 10 of the 29 benchmarks tested had fewer common subexpressions, and in all cases, at most 4 incorrect common subexpressions were detected.

5 Experimental Results

We now present a plethora of experimental results for common subexpression elimination. It was claimed that the current implementation of cse does not slow down Soot excessively. We present running times for Soot, to substantiate this claim. We also count the number of common subexpressions eliminated in a number of test cases. Finally, we examine the effect of common subexpression elimination on the running times of programs – we compare the running time after Sootification with and without common subexpression elimination, when used with either -O or -W.

5.1 Correctness

If we are to distribute our common subexpression eliminator with Soot, we would very much like for the implementation to be correct. Ensuring its correctness is not obvious; however, we have tested our implementation with the complete Soot Sentinel suite. It passes every one of the 267 benchmarks in the Sentinel suite[1].

5.2 Soot speed

Soot is slow. Nevertheless, we want to avoid making Soot excessively slow, especially when the payoff is unclear. This prompted the use of implicit flow universes. Did they keep Soot from being really really slow? We offer some Soot running times in figure 1

These times were not scientifically collected; each of them is from a single invocation of Soot. However, they provide a descriptive idea of how much time generally gets spent eliminating common subexpressions. Typically there is no difference, although in the worst case – excluding outliers – sootification time can increase by about 25%; the percentage is calculated by taking

$$\frac{\text{time with cse} - \text{time without cse}}{\text{time without cse}} \times 100$$

Mysteriously, sometimes Soot is faster after common subexpression elimination.

We have observed experimental error of up to 10% of execution time.

5.3 Counting common subexpressions eliminated

It is fairly easy to print out how many common subexpressions Soot actually manages to remove. These numbers are also in table 1. There appears to be no correlation between running time of common subexpression elimination and the number of common subexpressions removed.

Clearly, inlining strictly increased the number of common subexpressions available, although not by an especially exciting amount.

5.4 Dynamic measurements

The bottom line, for any practical compilers work, is the speedup result. How much faster do programs get? In this section, we present statistics on how much speedup common subexpression elimination can give.

It is highly likely that, by itself, common subexpression elimination will not have much impact. Aggressive side-effect information should be required.

We present results comparing the speedup obtained by running Soot for all entries in the following table:

-O, no cse	-O, with cse
-W, no cse	-W, with cse

[1] Note that mpegaudio_i fails under the Borland `javacomp` JIT compiler, but succeeds with the Sun JIT compiler.

	-O	-O cse	% time	# cses	-W	-W cse	% time	# cses
boyer	10:34	10:34	0	0	10:39	10:51	2	0
check	3:00	3:07	4	97	3:30	3:55	12	104
check_i	1:18	1:20	3	12	1:23	1:23	0	12
compress	2:25	2:37	8	114	2:49	3:28	23	141
compress_i	1:51	2:08	15	69	2:11	1:57	-12	70
db	2:20	2:24	3	91	3:08	2:52	-9	98
db_i	1:50	1:47	-3	30	1:53	2:00	6	31
illness	0:30	0:32	7	35	0:34	0:39	15	36
jack	4:41	4:42	0	128	6:49	6:45	-1	151
jack_i	6:59	7:15	4	83	6:58	7:10	3	87
javac	8:47	8:57	2	643	14:07	21:22	51	831
javac_i	10:51	10:23	-4	870	15:01	15:57	6	879
javasrc-p	7:52	8:05	3	335	14:46	16:26	11	620
jess	5:51	5:35	5	271	6:44	7:17	8	379
jess_i	6:10	5:14	-15	377	6:52	6:09	-12	412
jpat-p	0:24	0:26	8	42	0:29	0:29	0	47
kawac	8:45	10:50	24	716	15:44	16:30	5	871
lexgen	1:18	1:22	5	3	1:59	2:30	26	5
machineSim	0:27	0:33	22	25	0:31	0:40	29	66
mpegaudio	7:48	9:37	24	521	9:16	11:06	20	563
mpegaudio_i	8:31	11:35	35	621	9:28	12:18	30	632
mtrt	3:14	3:18	2	153	4:10	5:12	25	285
mtrt_i	2:40	3:08	18	309	3:27	3:25	-1	316
puzzle	0:26	0:30	15	72	0:29	0:31	7	72
raytrace	3:19	3:09	-5	153	4:20	4:59	15	285
raytrace_i	2:48	3:00	7	309	3:21	3:24	2	316
sablecc	8:24	9:48	17	500	13:49	14:32	5	595
soot	9:38	9:42	0	543	22:44	19:16	-15	1031

Table 1: Sootification times with and without CSE; static counts

Table 2: Timing results for benchmarks

	-S	-O	rat	-Ocse	rat	-W	rat	-Wcse	rat
compress	70.51	69.85	0.99	67.73	0.96	66.60	0.94	64.49	0.91
db	111.38	111.36	1.00	110.98	1.00	111.40	1.00	111.17	1.00
jack	38.89	39.24	1.01	38.83	1.00	38.32	0.99	38.18	0.98
javac	47.38	47.15	1.00	46.24	0.98	46.06	0.97	44.84	0.95
javasrc-p	45.80	46.06	1.01	44.53	0.97	44.98	0.98	43.65	0.95
jess	33.75	32.54	0.96	33.06	0.98	30.95	0.92	32.38	0.96
jpat-p	45.39	46.46	1.02	45.33	1.00	45.37	1.00	45.47	1.00
mpegaudio	58.53	60.39	1.03	56.15	0.96	57.67	0.99	55.04	0.94
raytrace	30.94	30.72	0.99	31.05	1.00	21.80	0.71	21.87	0.71
sablecc-w	32.09	32.30	1.01	31.44	0.98	31.52	0.98	30.52	0.95
schroeder-s	20.58	–	–	–	–	20.55	1.00	20.22	0.98
soot-c	34.57	34.58	1.00	33.60	0.97	33.38	0.97	32.11	0.93
average			1.00		0.98		0.95		0.94

We compare the speedup obtained with and without cse. We expect that, with -W inlining enabled, we will get the most significant speedups. Table 2 contains the results. Numbers presented are from executing each benchmark five times on unloaded dual-processor PII-400 machines with at least 256M of RAM, running Linux kernel 2.2.8 and the Sun JDK 1.2 FCS release for Linux, along with the Borland `javacomp` JIT engine. Each benchmark was executed 10 times, and the average time taken after discarding the fastest and slowest times. Most of the reported runs had an execution time variation not exceeding 0.10 seconds. We present execution fractions; lower is better.

Note that `schroeder-s` fails with the Borland JIT after -O.

There is a small impact on compress and mpegaudio without inlining, but otherwise combining common subexpression elimination without inlining only produces negligible speedups; on average, we observe an execution fraction of 0.98 versus 1.00 without cse. Note that no programs execute more slowly after common subexpression elimination. With inlining, we obtain an additional 2 to 5 percent speedup – still relatively minor, but measurable. The average execution fraction in this case is 0.94, compared to 0.95 with inlining alone. We expect that better side effect information would give better results. With our current conservative side-effect information, it wasn't reasonable to think that we could get significant speedup.

6 Conclusions and Future Work

We have presented an implementation of common subexpression elimination, presented some of the problems associated with such an implementation, and shown experimental results showing a small but measurable impact from this transformation.

The logical next step would be to implement partial redundancy elimination [Muc97][KRS92]. There already exists a preliminary implementation of PRE; it simply needs to be completed and debugged. We expect that this would not take much time.

In this work, we have contributed a number of improvements to the Soot framework. Available expressions can now be computed. This is the first expression-based analysis done with Soot; it required the development of an alternate notion of equivalence for Soot objects. Furthermore, a clean framework for using side-effect information is now available. Finally, a new type of flow set was introduced, adding $\top$ to any other flow set implementation.

This work would not benefit from attributes; the optimizations can be expressed in the bytecode itself. However, improved side effect information, as planned for this summer, could greatly increase the reach of common subexpression elimination, and make it into a viable optimization technique.

References

[KRS92] J. Knoop, O. Ruething, and B. Steffen. Lazy code motion. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, volume 27, pages 224–234, San Francisco, CA, June 1992.

[Lam00] Patrick Lam. Dataflow analysis for concurrent programs. Master's thesis, McGill University, July 2000.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, San Francisco, California, 1997.