Soot phase options

Patrick Lam (plam@sable.mcgill.ca)
Feng Qian (fqian@sable.mcgill.ca)

April 26, 2002

Soot supports the powerful, but initially confusing, notion of "phase options". This document will permit the reader to successfully use the Soot phase options.

Soot's execution is divided into a number of phases. Building the JimpleBody is a phase (called `jb`), and it has a number of subphases, like aggregation of stack variables (`jb.asv`).

Soot allows the user to specify options for each phase; these options will change the behaviour of the phase. This is specified by giving Soot the command-line option `-p phase.name option:value`. For instance, to instruct Soot to use original names in Jimple, we would invoke Soot like this:

```
[plam@cannanore test] java soot.Main foo -p jb use-original-names
```

Unless specified otherwise, all options are boolean; allowed values are "true" or "false". When an option is omitted, the default value is "false"; specifying an option without a value is the same as saying "true".

All transformers accept the option "`disabled`", which, when set to `true`, causes the given transformer to not execute.

Soot transformers are expected to be classes extending either `BodyTransformer` or `SceneTransformer`. In either case, an `internalTransform` method on the transformer must be overridden to provide an implementation which carries out some transformation.

These transformers belong to a `Pack`. The `Pack` keeps a collection of transformers, and can execute them, in order, when called. To add a transformer to some `Pack` without modifying Soot itself, create your own class, which modifies the `Packs` as needed and then calls `soot.Main`.

The remainder of this document describes the various transformations belonging to the various Packs in Release 1 of Soot.

Contents

1 JimpleBody creation

This phase is active during `JimpleBody` creation; Soot will always start by creating `JimpleBody`'s from a method source – either coffi, for reading `.class` files, or the jimple parser, for reading `.jimple` files.

Phase name	`jb`
Class name	`soot.jimple.JimpleBody`

Recognized options

`no-splitting` Soot will not split the locals by use-def webs while generating the Jimple body.

`no-typing` Soot will not resolve the types of local variables.

`aggregate-all-locals` Soot will aggregate both regular locals and stack locals. (See the `jop.cp` phase for more information). This options will suppress the 'no-aggregating' option.

`no-aggregating` If `'aggregate-all-locals'` is not specified, then disable the aggregation of locals for Jimple Body's. This subsumes `'only-stack-locals'` in the 'jb.asv' phase.

`use-original-names` Soot attempts to find and use the original names from the method source. If this is not specified, Soot gives standard names to local variables, according to the variable type.

`pack-locals` Soot will pack local names tightly, using a graph coloring algorithm.

`no-cp` Soot will not apply the copy propagator to Jimple Body's.

`no-nop-elimination` Soot will not remove nop instructions.

`no-unreachable-code-elimination` Soot will not remove unreachable code.

`verbatim` Do not apply any options.

1.1 Stack-variable aggregation

Phase name	`jb.asv`
Class name	`soot.jimple.toolkits.base.Aggregator`

This phase handles the aggregation of stack variables for Jimple. For a full description of aggregation, see [VRGH+00]. Briefly, aggregation finds instances where some expression has a single use; it replaces the use with the expression itself.

This phase is deactivated by the `'no-aggregating'` option in the `jb` phase.

Recognized option

`only-stack-locals` Aggregate values stored in stack locals only.

1.2 Unsplit-originals local packer

Phase name	`jb.ulp`
Class name	`soot.toolkits.scalar.LocalPacker`

This phase only executes when the 'use-original-names' option is chosen for the 'jb' phase. It unsplits the locals according to the original names found for them.

Recognized option

`unsplit-original-locals` (default: `true`) Calls the LocalPacker to implement the use-original-names option.

1.3 Local name standardizer

Phase name	`jb.lns`
Class name	`soot.jimple.toolkits.scalar.LocalNameStandardizer`

This phase assigns standard names to local variables. It only executes when 'use-original-names' is not chosen.

Recognized option

`only-stack-locals` (default: `true`) Standardizes only stack local variables' names.

1.4 Copy propagator

Phase name	`jb.cp`
Class name	`soot.jimple.toolkits.scalar.CopyPropagator`

This phase provides a cascaded copy propagator. It is executed only when '`no-cp`' is not chosen in the '`jb`' phase.

If it encounters situations of the form: A: a = ...; B: ... x = a; C:... use (x); where a has only one definition, and x has only one definition (B), then it can propagate immediately without checking between B and C for redefinitions of a (namely A) because they cannot occur. In this case the propagator is global.

Otherwise, if a has multiple definitions then it only checks for redefinitions of constants and copies in extended basic blocks.

From bytecode, we get some number of declared locals; we call these "regular locals". In Jimple, we have converted the stack elements to locals. The new locals thus introduced are called "stack locals". These locals have names which usually begin with $.

The default behaviour in this phase is to propagate only on the 'stack' locals.

Recognized options

`only-regular-locals` Copy propagation only occurs on the "regular" locals.

`only-stack-locals` (default: `true`) Copy propagation only occurs on the "stack" locals.

1.5 Dead assignment eliminator

Phase name	`jb.dae`
Class name	`soot.jimple.toolkits.scalar.DeadAssignmentEliminator`

This phase eliminates assignment statements (to locals) with no uses.

Recognized option

`only-stack-locals` (default: `true`) Only eliminates dead assignments to stack locals.

2 Jimple Transformation Pack

Soot will always apply the contents of the Jimple transformation pack to each method under analysis. This pack is called `jtp`. There are no transformations in this pack in an unmodified version of Soot.

3 Jimple Optimization Pack

When Soot is given the `-O` command-line option, the `JimpleOptimizationPack` is applied to every JimpleBody in an application class. This section lists the default transformations in the `JimpleOptimizationPack`.

3.1 Common subexpression elimination

Phase name	`jop.cse`
Class name	`soot.jimple.toolkits.scalar.NaiveCommonSubexpressionElimination`

Runs an available expressions analysis on a body, then eliminates common subexpressions.

This implementation is especially slow, as it does not run on basic blocks. A better implementation (which wouldn't catch every single common subexpression, but would get most) would use basic blocks instead.

It is also slow because the flow universe is explicitly created; it need not be. A better implementation would implicitly compute the kill sets at every node.

Because of the current slowness, this transformation is not enabled in the default settings. To enable it, specify `-p jop.cse disabled:false` on the command line.

Recognized options None.

3.2 Busy code motion

Phase name	`jop.bcm`
Class name	`soot.jimple.toolkits.scalar.pre.BusyCodeMotion`

Busy Code Motion is a straightforward implementation of Partial Redundancy Elimination. This implementation is not very aggressive. The Lazy Code Motion, described in section 3.3 is an improved version of the Busy Code Motion, and should be used instead of it.

Busy Code Motion is not enabled by default. To enable it, specify `-p jop.bcm disabled:false` on the command line.

Recognized options None.

3.3 Lazy code motion

Phase name	`jop.lcm`
Class name	`soot.jimple.toolkits.scalar.pre.LazyCodeMotion`

Lazy Code Motion is the enhanced version of the Busy Code Motion, a Partial Redundancy Eliminator. Before doing Partial Redundancy Elimination, this optimization performs loop inversion (turning `while` loops into `do while` loops inside an `if` statement). This allows the Partial Redundancy Eliminator to optimize loop invariants of `while` loops.

By default, this transformation is disabled. To enable it, specify `-p jop.lcm disabled:false` on the command line.

Recognized options

`safe:`**STRING** (default: `safe`) one of `safe`, `medium` or `unsafe`. This option controls how fields and exception-throwing statements are treated.

- `safe` is safe, but only considers additions, subtractions and multiplications.
- `medium` is unsafe in multi-threaded environment, as already performed field accesses are reused.
- `unsafe` moves exception-throwing statements, and reorders them. They are potentially moved out of `try-catch`-blocks.

`unroll` (default: `true`) if `true`, loop inversion is performed before doing the transformation.

3.4 Copy propagator

Phase name	`jop.cp`
Class name	`soot.jimple.toolkits.scalar.CopyPropagator`

See section 1.4 for a description of copy propagation.
The default behaviour here is to propagate on all locals.

Recognized options

`only-regular-locals` Copy propagation only occurs on the "regular" locals.

`only-stack-locals` Copy propagation only occurs on the "stack" locals.

3.5 Constant propagator and folder

Phase name	`jop.cpf`
Class name	`soot.jimple.toolkits.scalar.ConstantPropagatorAndFolder`

This phase does constant propagation and folding. Constant folding is the compile-time evaluation of constant expressions (i.e. 2 * 3).

Recognized options None.

3.6 Conditional branch folder

Phase name	`jop.cbf`
Class name	`soot.jimple.toolkits.scalar.ConditionalBranchFolder`

Statically evaluates the condition-expression of Jimple `IfStmt`s. If the condition is identically 'true' or 'false', changes the conditional branch instruction to a 'goto' statement.

Recognized options None.

3.7 Dead assignment eliminator

Phase name	`jop.dae`
Class name	`soot.jimple.toolkits.scalar.DeadAssignmentEliminator`

This phase eliminates assignment statements (to locals) with no uses. See section 1.5.
In this incarnation, the default value for `only-stack-locals` is `false`.

3.8 Unreachable code eliminator

Phase names	jop.uce1, jop.uce2
Class name	soot.jimple.toolkits.scalar.UnreachableCodeEliminator

Removes unreachable codes and empty traps.

Recognized options None.

3.9 Unconditional branch folder

Phase name	jop.ubf1, jop.ubf2
Class name	soot.jimple.toolkits.scalar.UnconditionalBranchFolder

Removes unnecessary 'goto' statements from a JimpleBody.

If a GotoStmt's target is the next instruction, then it is removed. If a GotoStmt x's target is another GotoStmt, with target y, then x's target can be changed to y's target.

If some IfStmt's target is a GotoStmt, then the IfStmt's target can be updated to the GotoStmt's target.

(These situations could result from other optimizations; after folding branches, we might generate more unreachable code.)

Recognized options None.

3.10 Unused local eliminator

Phase name	jop.ule
Class name	soot.jimple.toolkits.scalar.UnusedLocalEliminator

Removes locals with no uses in the method body.

Recognized options None.

4 Whole-Jimple Transformation Pack

Soot can do whole-program analyses. For the current version of Soot, this means that Jimple bodies are created for each method in the application, and analyses run on this set of Jimple bodies. The application consists of one class, specified on the command-line, plus all classes referenced (directly or indirectly) by it. It excludes classes in java.*, javax.*, and sun.*. This mode is triggered by the --app option.

In whole-program mode, Soot will always apply the contents of the Whole-Jimple transformation pack to each method under analysis. This occurs after all Jimple bodies have been created. This pack is called wjtp. In an unmodified version of Soot, the only transformation in wjtp is the Spark pointer analysis kit. Spark has many options, which are listed at http://www.sable.mcgill.ca/soot/tutorial/phase-options/spark.ps.

5 Whole-Jimple Optimization Pack

To run optimizing transformations on the whole program, use the -W command-line option. This tells Soot that the whole-jimple optimization pack is to be applied (phase name wjop).

The default behaviour of this Pack has static method binding disabled and static inlining enabled. To reverse this, give the options -p wjop.smb disabled:false -p wjop.si disabled.

By default, static method binding and static inlining uses Class Hierarchy Analysis (CHA) to identify monomophic call sites. Changing the VTA-passes option can cause them to use Variable-Type Analysis (VTA) once or several times rather than CHA.

5.1 Static method binding

Phase name	`wjop.smb`
Class name	`soot.jimple.toolkits.invoke.StaticMethodBinder`

Static method binding uses CHA or VTA to statically bind monomorphic call sites. That is, smb takes the call graph returned by CHA or VTA; if the analysis result shows that any virtual invoke statement in the Jimple bodies actually only calls one method, then a static copy of the method is made, and the virtual invoke is changed to a static invoke.

Recognized options

`insert-null-checks` The receiver object is checked for nullness before the target method is invoked. If the target is null, then a NullPointer exception is thrown.

`insert-redundant-casts` Inserts extra casts for the verifier. The verifier will complain if the target uses 'this' (so we have to pass an extra parameter), and the argument passed to the method is not the same type. For instance, `Bottle.price_static` is a method which takes a Cost object, and Cost is an interface implemented by Bottle. We must then cast the Cost to a Bottle before passing it to `price_static`.

`allowed-modifier-changes:STRING` (default STRING: unsafe) Determines what changes in visibility modifiers are allowed. "unsafe" modifies the visibility on code so that all inlining is permitted; some IllegalAccessErrors may be missed. "safe" preserves the exact meaning of the analysed program, and "none" changes no modifiers whatsoever.

`VTA-passes:INT` (default INT: 0) By defult, static method binding uses CHA. This option can cause VTA to run one or more times as specified by the parameter.

5.2 Static inlining

Phase name	`wjop.si`
Class name	`soot.jimple.toolkits.invoke.StaticInliner`

The StaticInliner talkes an call graph returned by CHA or VTA and visits all call sites in the application in a bottom-up fashion, inlining invoke statements which is determined to be monomorphic by analysis result. Note that the modifier "static" is supposed to be compared to a (not-currently-implemented) profile-guided inliner.

Recognized options

`insert-null-checks` As in `StaticMethodBinder`.

`insert-redundant-casts` As in `StaticMethodBinder`.

`allowed-modifier-changes` As in `StaticMethodBinder`.

`expansion-factor:FLOAT` (default FLOAT: 3) Determines the maximum allowed expansion of a method. Inlining will cause the method to grow by a factor of no more than expansion-factor.

`max-container-size:INT` (default INT: 5000) Determines the maximum number of Jimple statements for a container method. If a method has more than this number of Jimple statements, then no methods will be inlined into it.

`max-inlinee-size:INT` (default INT: 20) Determines the maximum number of Jimple statements for an inlinee method. If a method has more than this number of Jimple statements, then it will not be inlined into other methods.

`VTA-passes:INT` (default INT: 0) This option can cause VTA to run one or more times as specified by the parameter.

6 Annotating class files

Soot has a number of phase options to configure the annotation process. Array bounds check and null pointer check detection have separate phases and phase options.

6.1 Null pointer check options

The null pointer check analysis has the phase name `jtp.npc`. It has one phase option (aside from the default `disabled` option).

`-p jtp.npc onlyarrayref` By default, all bytecodes that need null pointer checks are annotated with the analysis result. When this option is set to true, Soot will annotate only array-referencing bytecodes with null pointer check information; other bytecodes, such as `getfield` and `putfield`, will not be annotated.

6.2 Array bounds check options

The array bounds check analysis has the phase name `jtp.abc`. If whole-program analysis is required, an extra phase `wjtp2.ra` for finding rectangular arrays occurs.

`-p jtp.abc with-cse` The analysis will consider common subexpressions. For example, consider the situation where `r1` is assigned `a*b`; later, `r2` is assigned `a*b`, where both `a` and `b` have not been changed between the two statements. The analysis can conclude that `r2` has the same value as `r1`. Experiments show that this option can improve the result slightly.

`-p jtp.abc with-arrayref` With this option enabled, array references can be considered as common subexpressions; however, we are more conservative when writing into an array, because array objects may be aliased. NOTE: We also assume that the application in a single-threaded program or in a synchronized block. That is, an array element may not be changed by other threads between two array references.

`-p jtp.abc with-fieldref` The analysis treats field references (static and instance) as common subexpressions. The restrictions from the '`with-arrayref`' option also apply.

`-p jtp.abc with-classfield` This option makes the analysis work on the class level. The algorithm analyzes 'final' or 'private' class fields first. It can recognize the fields that hold array objects with constant length. In an application using lots of array fields, this option can improve the analysis results dramatically.

`-p jtp.abc with-all` A macro. Instead of typing a long string of phase options, this option will turn on all options of the phase "`jtp.abc`".

`-p jtp.abc with-rectarray, -p wjtp2.ra with-wholeapp` These two options are used together to make Soot run the whole-program analysis for rectangular array objects. This analysis is based on the call graph, and it usually takes a long time. If the application uses rectangular arrays, these options can improve the analysis result.

7 Generating Jasmin

This is not a phase in the same sense as the others; notably, it does not belong to any `Pack`.

Phase name	`Jimple.JasminClass`
Class name	`soot.jimple.JasminClass`

This class is used in the generation of a `jasmin` file from a `SootClass` containing either Jimple or Grimp bodies.

For more detail about peepholes, see [VRGH+00].

Recognized option

`no-peephole` Peephole optimizations are disabled when generating Jasmin.

History

- March 16, 2000: Initial version.
- October 6, 2000: Added new phase options for Soot 1.2.0 concerning annotation.
- April, 2002: Added busy code motion and lazy code motion options.

References

[VRGH+00] Raja Vallée-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot framework: is it feasible?. In *Compiler Construction, 9th International Conference, CC 2000*, April 2000.