

Soot phase options

Patrick Lam (plam@sable.mcgill.ca)
Feng Qian (fqian@sable.mcgill.ca)
Ondřej Lhoták (olhotak@sable.mcgill.ca)
John Jorgensen

June 18, 2003

Soot supports the powerful—but initially confusing—notion of “phase options”. This document aims to clear up the confusion so you can exploit the power of phase options.

Soot’s execution is divided into a number of phases. For example, `JimpleBodys` are built by a phase called `jb`, which is itself comprised of subphases, such as the aggregation of local variables (`jb.a`).

Phase options provide a way for you to change the behaviour of a phase from the Soot command-line. They take the form `-p phase.name option:value`. For instance, to instruct Soot to use original names in Jimple, we would invoke Soot like this:

```
java soot.Main foo -p jb use-original-names:true
```

Multiple option-value pairs may be specified in a single `-p` option separated by commas. For example,

```
java soot.Main foo -p cg.spark verbose:true,on-fly-cg:true
```

There are five types of phase options:

1. Boolean options take the values “true” and “false”; if you specify the name of a boolean option without adding a value for it, “true” is assumed.
2. Multi-valued options take a value from a set of allowed values specific to that option.
3. Integer options take an integer value.
4. Floating point options take a floating point number as their value.
5. String options take an arbitrary string as their value.

Each option has a default value which is used if the option is not specified on the command line.

All phases and subphases accept the option “enabled”, which must be “true” for the phase or subphase to execute. To save you some typing, the pseudo-options “on” and “off” are equivalent to “enabled:true” and “enabled:false”, respectively. In addition, specifying any options for a phase automatically enables that phase.

Adding your own subphases

Within Soot, each phase is implemented by a `Pack`. The `Pack` is a collection of transformers, each corresponding to a subphase of the phase implemented by the `Pack`. When the `Pack` is called, it executes each of its transformers in order.

Soot transformers are usually instances of classes that extend `BodyTransformer` or `SceneTransformer`. In either case, the transformer class must override the `internalTransform` method, providing an implementation which carries out some transformation on the code being analyzed.

To add a transformer to some `Pack` without modifying Soot itself, create your own class which changes the contents of the `Packs` to meet your requirements and then calls `soot.Main`.

The remainder of this document describes the transformations belonging to Soot’s various `Packs` and their corresponding phase options.

Contents

1	Jimple Body Creation (jb)	2
1.1	Local Splitter (jb.ls)	2
1.2	Jimple Local Aggregator (jb.a)	2
1.3	Unused Local Eliminator (jb.ule)	2
1.4	Type Assigner (jb.tr)	2
1.5	Unsplit-originals Local Packer (jb.ulp)	3
1.6	Local Name Standardizer (jb.lns)	3
1.7	Copy Propagator (jb.cp)	3
1.8	Dead Assignment Eliminator (jb.dae)	4
1.9	Post-copy propagation Unused Local Eliminator (jb.cp-ule)	4
1.10	Local Packer (jb.lp)	4
1.11	Nop Eliminator (jb.ne)	4
1.12	Unreachable Code Eliminator (jb.uce)	4
2	Call Graph Constructor (cg)	4
2.1	Class Hierarchy Analysis (cg.cha)	5
2.2	Spark (cg.spark)	5
2.2.1	Spark General Options	6
2.2.2	Spark Pointer Assignment Graph Building Options	6
2.2.3	Spark Pointer Assignment Graph Simplification Options	7
2.2.4	Spark Points-To Set Flowing Options	7
2.2.5	Spark Output Options	9
3	Whole-Jimple Transformation Pack (wjtp)	10
4	Whole-Jimple Optimization Pack (wjop)	10
4.1	Static Method Binder (wjop.smb)	10
4.2	Static Inliner (wjop.si)	11
5	Whole-Jimple Annotation Pack (wjap)	12
5.1	Rectangular Array Finder (wjap.ra)	12
6	Shimple Control (shimple)	12
7	Shimple Transformation Pack (stp)	13
8	Shimple Optimization Pack (sop)	13
8.1	Shimple Constant Propagator and Folder (sop.cpf)	13
9	Jimple Transformation Pack (jtp)	13
10	Jimple Optimization Pack (jop)	14
10.1	Common Subexpression Eliminator (jop.cse)	14
10.2	Busy Code Motion (jop.bcm)	14
10.3	Lazy Code Motion (jop.lcm)	15
10.4	Copy Propagator (jop.cp)	16
10.5	Jimple Constant Propagator and Folder (jop.cpf)	16
10.6	Conditional Branch Folder (jop.cbf)	16
10.7	Dead Assignment Eliminator (jop.dae)	16
10.8	Unreachable Code Eliminator 1 (jop.uce1)	16
10.9	Unconditional Branch Folder 1 (jop.ubf1)	17
10.10	Unreachable Code Eliminator 2 (jop.uce2)	17
10.11	Unconditional Branch Folder 2 (jop.ubf2)	17

10.12	Unused Local Eliminator (<code>jop.ule</code>)	17
11	Jimple Annotation Pack (<code>jap</code>)	17
11.1	Null Pointer Checker (<code>jap.npc</code>)	18
11.2	Null Pointer Colourer (<code>jap.npcolorer</code>)	18
11.3	Array Bound Checker (<code>jap.abc</code>)	18
11.4	Profiling Generator (<code>jap.profiling</code>)	19
11.5	Side Effect tagger (<code>jap.sea</code>)	19
11.6	Field Read/Write Tagger (<code>jap.fieldrw</code>)	19
11.7	Call Graph Tagger (<code>jap.cgtagger</code>)	20
11.8	Parity Tagger (<code>jap.parity</code>)	20
12	Grimp Body Creation (<code>gb</code>)	20
12.1	Grimp Pre-folding Aggregator (<code>gb.a1</code>)	20
12.2	Grimp Constructor Folder (<code>gb.cf</code>)	21
12.3	Grimp Post-folding Aggregator (<code>gb.a2</code>)	21
12.4	Grimp Unused Local Eliminator (<code>gb.ule</code>)	21
13	Grimp Optimization (<code>gop</code>)	22
14	Baf Body Creation (<code>bb</code>)	22
14.1	Load Store Optimizer (<code>bb.lso</code>)	22
14.2	Peephole Optimizer (<code>bb.pho</code>)	23
14.3	Unused Local Eliminator (<code>bb.ule</code>)	23
14.4	Local Packer (<code>bb.lp</code>)	23
15	Baf Optimization (<code>bop</code>)	23
16	Tag Aggregator (<code>tag</code>)	24
16.1	Line Number Tag Aggregator (<code>tag.ln</code>)	24
16.2	Array Bounds and Null Pointer Check Tag Aggregator (<code>tag.an</code>)	24
16.3	Dependence Tag Aggregator (<code>tag.dep</code>)	24
16.4	Field Read/Write Tag Aggregator (<code>tag.fieldrw</code>)	24

1 Jimple Body Creation (`jb`)

Jimple Body Creation creates a `JimpleBody` for each input method, using either `coffi`, to read `.class` files, or the jimple parser, to read `.jimple` files.

Accepted phase options:

Enabled (`enabled`) (default value: `true`)

Use Original Names (`use-original-names`) (default value: `false`)

Retain the original names for local variables when the source includes those names. Otherwise, Soot gives variables generic names based on their types.

1.1 Local Splitter (`jb.ls`)

The Local Splitter identifies DU-UD webs for local variables and introduces new variables so that each disjoint web is associated with a single local.

Accepted phase options:

Enabled (`enabled`) (default value: `true`)

1.2 Jimple Local Aggregator (jb.a)

The Jimple Local Aggregator removes some unnecessary copies by combining local variables. Essentially, it finds definitions which have only a single use and, if it is safe to do so, removes the original definition after replacing the use with the definition's right-hand side.

At this stage in `JimpleBody` construction, local aggregation serves largely to remove the copies to and from stack variables which simulate load and store instructions in the original bytecode.

Accepted phase options:

Enabled (`enabled`) (default value: `true`)

Only Stack Locals (`only-stack-locals`) (default value: `true`)

Only aggregate locals that represent stack locations in the original bytecode. (Stack locals can be distinguished in Jimple by the `$` character with which their names begin.)

1.3 Unused Local Eliminator (jb.ule)

The Unused Local Eliminator removes any unused locals from the method.

Accepted phase options:

Enabled (`enabled`) (default value: `true`)

1.4 Type Assigner (jb.tr)

The Type Assigner gives local variables types which will accommodate the values stored in them over the course of the method.

Accepted phase options:

Enabled (`enabled`) (default value: `true`)

1.5 Unsplit-originals Local Packer (jb.ulp)

The Unsplit-originals Local Packer executes only when the `'use-original-names'` option is chosen for the `'jb'` phase. The Local Packer attempts to minimize the number of local variables required in a method by reusing the same variable for disjoint DU-UD webs. Conceptually, it is the inverse of the Local Splitter.

Accepted phase options:

Enabled (`enabled`) (default value: `true`)

Unsplit Original Locals (`unsplit-original-locals`) (default value: `true`)

Use the variable names in the original source as a guide when determining how to share local variables among non-interfering variable usages. This recombines named locals which were split by the Local Splitter.

1.6 Local Name Standardizer (jb.lns)

The Local Name Standardizer assigns generic names to local variables.

Accepted phase options:

Enabled (`enabled`) (default value: `true`)

Only Stack Locals (`only-stack-locals`) (default value: `false`)

Only standardizes the names of variables that represent stack locations in the original bytecode. This becomes the default when the ‘`use-original-names`’ option is specified for the ‘`jb`’ phase.

1.7 Copy Propagator (`jb.cp`)

This phase performs cascaded copy propagation.

If the propagator encounters situations of the form:

```
A: a = ...;
    ...
B: x = a;
    ...
C: ... = ... x;
```

where `a` and `x` are each defined only once (at `A` and `B`, respectively), then it can propagate immediately without checking between `B` and `C` for redefinitions of `a`. In this case the propagator is global.

Otherwise, if `a` has multiple definitions then the propagator checks for redefinitions and propagates copies only within extended basic blocks.

Accepted phase options:

Enabled (`enabled`) (default value: `true`)

Only Regular Locals (`only-regular-locals`) (default value: `false`)

Only propagate copies through “regular” locals, that is, those declared in the source bytecode.

Only Stack Locals (`only-stack-locals`) (default value: `true`)

Only propagate copies through locals that represent stack locations in the original bytecode.

1.8 Dead Assignment Eliminator (`jb.dae`)

The Dead Assignment Eliminator eliminates assignment statements to locals whose values are not subsequently used, unless evaluating the right-hand side of the assignment may cause side-effects.

Accepted phase options:

Enabled (`enabled`) (default value: `true`)

Only Stack Locals (`only-stack-locals`) (default value: `true`)

Only eliminate dead assignments to locals that represent stack locations in the original bytecode.

1.9 Post-copy propagation Unused Local Eliminator (`jb.cp-ule`)

This phase removes any locals that are unused after copy propagation.

Accepted phase options:

Enabled (`enabled`) (default value: `true`)

1.10 Local Packer (jb.lp)

The Local Packer attempts to minimize the number of local variables required in a method by reusing the same variable for disjoint DU-UD webs. Conceptually, it is the inverse of the Local Splitter.

Accepted phase options:

Enabled (enabled) (default value: `false`)

Unsplit Original Locals (unsplit-original-locals) (default value: `false`)

Use the variable names in the original source as a guide when determining how to share local variables across non-interfering variable usages. This recombines named locals which were split by the Local Splitter.

1.11 Nop Eliminator (jb.ne)

The Nop Eliminator removes `nop` statements from the method.

Accepted phase options:

Enabled (enabled) (default value: `true`)

1.12 Unreachable Code Eliminator (jb.uce)

The Unreachable Code Eliminator removes unreachable code and traps whose catch blocks are empty.

Accepted phase options:

Enabled (enabled) (default value: `true`)

2 Call Graph Constructor (cg)

The Call Graph Constructor computes a call graph for whole program analysis. When this pack finishes, a call graph is available in the Scene. The different phases in this pack are different ways to construct the call graph. Exactly one phase in this pack must be enabled; Soot will raise an error otherwise.

Accepted phase options:

Enabled (enabled) (default value: `true`)

Safe forName (safe-forname) (default value: `true`)

When a program calls `Class.forName()`, the named class is resolved, and its static initializer executed. In many cases, it cannot be determined statically which class will be loaded, and which static initializer executed. When this option is set to `true`, Soot will conservatively assume that any static initializer could be executed. This may make the call graph very large. When this option is set to `false`, any calls to `Class.forName()` for which the class cannot be determined statically are assumed to call no static initializers.

Safe newInstance (safe-newinstance) (default value: `true`)

When a program calls `Class.newInstance()`, a new object is created and its constructor executed. Soot does not determine statically which type of object will be created, and which constructor executed. When this option is set to `true`, Soot will conservatively assume that any constructor could be executed. This may make the call graph very large. When this option is set to `false`, any calls to `Class.newInstance()` are assumed not to call the constructor of the created object.

Verbose (verbose) (default value: `false`)

Due to the effects of native methods and reflection, it may not always be possible to construct a fully conservative call graph. Setting this option to true causes Soot to point out the parts of the call graph that may be incomplete, so that they can be checked by hand.

All Application Class Methods Reachable (all-reachable) (default value: `false`)

When this option is false, the call graph is built starting at a set of entry points, and only methods reachable from those entry points are processed. Unreachable methods will not have any call graph edges generated out of them. Setting this option to true makes Soot consider all methods of application classes to be reachable, so call edges are generated for all of them. This leads to a larger call graph. For program visualization purposes, it is sometimes desirable to include edges from unreachable methods; although these methods are unreachable in the version being analyzed, they may become reachable if the program is modified.

Trim Static Initializer Edges (trim-clinit) (default value: `true`)

The call graph contains an edge from each statement that could trigger execution of a static initializer to that static initializer. However, each static initializer is triggered only once. When this option is enabled, after the call graph is built, an intra-procedural analysis is performed to detect static initializer edges leading to methods that must have already been executed. Since these static initializers cannot be executed again, the corresponding call graph edges are removed from the call graph.

2.1 Class Hierarchy Analysis (`cg.cha`)

This phase uses Class Hierarchy Analysis to generate a call graph.

Accepted phase options:

Enabled (enabled) (default value: `true`)

Verbose (verbose) (default value: `false`)

Setting this option to true causes Soot to print out statistics about the call graph computed by this phase, such as the number of methods determined to be reachable.

2.2 Spark (`cg.spark`)

Spark is a flexible points-to analysis framework. Aside from building a call graph, it also generates information about the targets of pointers. For details about Spark, please see Ondrej Lhotak's M.Sc. thesis.

Accepted phase options:

Enabled (enabled) (default value: `false`)

2.2.1 Spark General Options

Accepted phase options:

Verbose (verbose) (default value: `false`)

When this option is set to true, Spark prints detailed information about its execution.

Ignore Types Entirely (ignore-types) (default value: `false`)

When this option is set to true, all parts of Spark completely ignore declared types of variables and casts.

Force Garbage Collections (`force-gc`) (default value: `false`)

When this option is set to true, calls to `System.gc()` will be made at various points to allow memory usage to be measured.

Pre Jimplify (`pre-jimplify`) (default value: `false`)

When this option is set to true, Spark converts all available methods to Jimple before starting the points-to analysis. This allows the Jimplification time to be separated from the points-to time. However, it increases the total time and memory requirement, because all methods are Jimplified, rather than only those deemed reachable by the points-to analysis.

2.2.2 Spark Pointer Assignment Graph Building Options

Accepted phase options:

VTA (`vta`) (default value: `false`)

Setting VTA to true has the effect of setting field-based, types-for-sites, and simplify-sccs to true, and on-fly-cg to false, to simulate Variable Type Analysis, described in our OOPSLA 2000 paper. Note that the algorithm differs from the original VTA in that it handles array elements more precisely.

RTA (`rta`) (default value: `false`)

Setting RTA to true sets types-for-sites to true, and causes Spark to use a single points-to set for all variables, giving Rapid Type Analysis.

Field Based (`field-based`) (default value: `false`)

When this option is set to true, fields are represented by variable (Green) nodes, and the object that the field belongs to is ignored (all objects are lumped together), giving a field-based analysis. Otherwise, fields are represented by field reference (Red) nodes, and the objects that they belong to are distinguished, giving a field-sensitive analysis.

Types For Sites (`types-for-sites`) (default value: `false`)

When this option is set to true, types rather than allocation sites are used as the elements of the points-to sets.

Merge String Buffer (`merge-stringbuffer`) (default value: `true`)

When this option is set to true, all allocation sites creating `java.lang.StringBuffer` objects are grouped together as a single allocation site.

Simulate Natives (`simulate-natives`) (default value: `true`)

When this option is set to true, the effects of native methods in the standard Java class library are simulated.

Simple Edges Bidirectional (`simple-edges-bidirectional`) (default value: `false`)

When this option is set to true, all edges connecting variable (Green) nodes are made bidirectional, as in Steensgaard's analysis.

On Fly Call Graph (`on-fly-cg`) (default value: `true`)

When this option is set to true, the call graph is computed on-the-fly as points-to information is computed. Otherwise, an initial CHA approximation to the call graph is used.

Parms As Fields (`parms-as-fields`) (default value: `false`)

When this option is set to true, parameters to methods are represented as fields (Red nodes) of the `this` object; otherwise, parameters are represented as variable (Green) nodes.

Returns As Fields (`returns-as-fields`) (default value: `false`)

When this option is set to true, return values from methods are represented as fields (Red nodes) of the `this` object; otherwise, return values are represented as variable (Green) nodes.

2.2.3 Spark Pointer Assignment Graph Simplification Options

Accepted phase options:

Simplify Offline (`simplify-offline`) (default value: `false`)

When this option is set to true, variable (Green) nodes which form single-entry subgraphs (so they must have the same points-to set) are merged before propagation begins.

Simplify SCCs (`simplify-sccs`) (default value: `false`)

When this option is set to true, variable (Green) nodes which form strongly-connected components (so they must have the same points-to set) are merged before propagation begins.

Ignore Types For SCCs (`ignore-types-for-sccs`) (default value: `false`)

When this option is set to true, when collapsing strongly-connected components, nodes forming SCCs are collapsed regardless of their declared type. The collapsed SCC is given the most general type of all the nodes in the component.

When this option is set to false, only edges connecting nodes of the same type are considered when detecting SCCs.

This option has no effect unless `simplify-sccs` is true.

2.2.4 Spark Points-To Set Flowing Options

Accepted phase options:

Propagator (`propagator`) (default value: `worklist`)

This option tells Spark which propagation algorithm to use.

Possible values:

<code>iter</code>	Iter is a simple, iterative algorithm, which propagates everything until the graph does not change.
<code>worklist</code>	Worklist is a worklist-based algorithm that tries to do as little work as possible. This is currently the fastest algorithm.
<code>cycle</code>	This algorithm finds cycles in the PAG on-the-fly. It is not yet finished.
<code>merge</code>	Merge is an algorithm that merges all concrete field (yellow) nodes with their corresponding field reference (red) nodes. This algorithm is not yet finished.
<code>alias</code>	Alias is an alias-edge based algorithm. This algorithm tends to take the least memory for very large problems, because it does not represent explicitly points-to sets of fields of heap objects.
<code>none</code>	None means that propagation is not done; the graph is only built and simplified. This is useful if an external solver is being used to perform the propagation.

Set Implementation (`set-impl`) (default value: `double`)

Select an implementation of points-to sets for Spark to use.

Possible values:

<code>hash</code>	Hash is an implementation based on Java's built-in hash-set.
<code>bit</code>	Bit is an implementation using a bit vector.

hybrid	Hybrid is an implementation that keeps an explicit list of up to 16 elements, and switches to a bit-vector when the set gets larger than this.
array	Array is an implementation that keeps the elements of the points-to set in a sorted array. Set membership is tested using binary search, and set union and intersection are computed using an algorithm based on the merge step from merge sort.
double	Double is an implementation that itself uses a pair of sets for each points-to set. The first set in the pair stores new pointed-to objects that have not yet been propagated, while the second set stores old pointed-to objects that have been propagated and need not be reconsidered. This allows the propagation algorithms to be incremental, often speeding them up significantly.
shared	This is a bit-vector representation, in which duplicate bit-vectors are found and stored only once to save memory.

Double Set Old (double-set-old) (default value: hybrid)

Select an implementation for sets of old objects in the double points-to set implementation.

This option has no effect unless Set Implementation is set to double.

Possible values:

hash	Hash is an implementation based on Java's built-in hash-set.
bit	Bit is an implementation using a bit vector.
hybrid	Hybrid is an implementation that keeps an explicit list of up to 16 elements, and switches to a bit-vector when the set gets larger than this.
array	Array is an implementation that keeps the elements of the points-to set in a sorted array. Set membership is tested using binary search, and set union and intersection are computed using an algorithm based on the merge step from merge sort.
shared	This is a bit-vector representation, in which duplicate bit-vectors are found and stored only once to save memory.

Double Set New (double-set-new) (default value: hybrid)

Select an implementation for sets of new objects in the double points-to set implementation.

This option has no effect unless Set Implementation is set to double.

Possible values:

hash	Hash is an implementation based on Java's built-in hash-set.
bit	Bit is an implementation using a bit vector.
hybrid	Hybrid is an implementation that keeps an explicit list of up to 16 elements, and switches to a bit-vector when the set gets larger than this.
array	Array is an implementation that keeps the elements of the points-to set in a sorted array. Set membership is tested using binary search, and set union and intersection are computed using an algorithm based on the merge step from merge sort.
shared	This is a bit-vector representation, in which duplicate bit-vectors are found and stored only once to save memory.

2.2.5 Spark Output Options

Accepted phase options:

Dump HTML (`dump-html`) (default value: `false`)

When this option is set to true, a browseable HTML representation of the pointer assignment graph is output to a file called `pag.jar` after the analysis completes. Note that this representation is typically very large.

Dump PAG (`dump-pag`) (default value: `false`)

When this option is set to true, a representation of the pointer assignment graph suitable for processing with other solvers (such as the BDD-based solver) is output before the analysis begins.

Dump Solution (`dump-solution`) (default value: `false`)

When this option is set to true, a representation of the resulting points-to sets is dumped. The format is similar to that of the Dump PAG option, and is therefore suitable for comparison with the results of other solvers.

Topological Sort (`topo-sort`) (default value: `false`)

When this option is set to true, the representation dumped by the Dump PAG option is dumped with the variable (green) nodes in (pseudo-)topological order.

This option has no effect unless Dump PAG is true.

Dump Types (`dump-types`) (default value: `true`)

When this option is set to true, the representation dumped by the Dump PAG option includes type information for all nodes.

This option has no effect unless Dump PAG is true.

Class Method Var (`class-method-var`) (default value: `true`)

When this option is set to true, the representation dumped by the Dump PAG option represents nodes by numbering each class, method, and variable within the method separately, rather than assigning a single integer to each node.

This option has no effect unless Dump PAG is true. Setting Class Method Var to true has the effect of setting Topological Sort to false.

Dump Answer (`dump-answer`) (default value: `false`)

When this option is set to true, the computed reaching types for each variable are dumped to a file, so that they can be compared with the results of other analyses (such as the old VTA).

Add Tags (`add-tags`) (default value: `false`)

When this option is set to true, the results of the analysis are encoded within tags and printed with the resulting Jimple code.

Calculate Set Mass (`set-mass`) (default value: `false`)

When this option is set to true, Spark computes and prints various cryptic statistics about the size of the points-to sets computed.

3 Whole-Jimple Transformation Pack (wjtp)

Soot can perform whole-program analyses. In whole-program mode, Soot applies the contents of the Whole-Jimple Transformation Pack to the scene as a whole after constructing a call graph for the program.

In an unmodified copy of Soot the Whole-Jimple Transformation Pack is empty.

Accepted phase options:

Enabled (`enabled`) (default value: `true`)

4 Whole-Jimple Optimization Pack (wjop)

If Soot is running in whole program mode and the Whole-Jimple Optimization Pack is enabled, the pack's transformations are applied to the scene as a whole after construction of the call graph and application of any enabled Whole-Jimple Transformations.

Accepted phase options:

Enabled (`enabled`) (default value: `false`)

4.1 Static Method Binder (`wjop.smb`)

The Static Method Binder statically binds monomorphic call sites. That is, it searches the call graph for virtual method invocations that can be determined statically to call only a single implementation of the called method. Then it replaces such virtual invocations with invocations of a static copy of the single called implementation.

Accepted phase options:

Enabled (`enabled`) (default value: `false`)

Insert Null Checks (`insert-null-checks`) (default value: `true`)

Insert a check that, before invoking the static copy of the target method, throws a `NullPointerException` if the receiver object is null. This ensures that static method binding does not eliminate exceptions which would have occurred in its absence.

Insert Redundant Casts (`insert-redundant-casts`) (default value: `true`)

Insert extra casts for the Java bytecode verifier. If the target method uses its `this` parameter, a reference to the receiver object must be passed to the static copy of the target method. The verifier may complain if the declared type of the receiver parameter does not match the type implementing the target method.

Say, for example, that `Singer` is an interface declaring the `sing()` method and that the call graph shows all receiver objects at a particular call site, `singer.sing()` (with `singer` declared as a `Singer`) are in fact `Bird` objects (`Bird` being a class that implements `Singer`). The virtual call `singer.sing()` is effectively replaced with the static call `Bird.staticsing(singer)`. `Bird.staticsing()` may perform operations on its parameter which are only allowed on `Birds`, rather than `Singers`. The Insert Redundant Casts option inserts a cast of `singer` to the `Bird` type, to prevent complaints from the verifier.

Allowed Modifier Changes (`allowed-modifier-changes`) (default value: `unsafe`)

Specify which changes in visibility modifiers are allowed.

Possible values:

<code>unsafe</code>	Modify the visibility on code so that all inlining is permitted.
<code>safe</code>	Preserve the exact meaning of the analyzed program.
<code>none</code>	Change no modifiers whatsoever.

4.2 Static Inliner (wjop.si)

The Static Inliner visits all call sites in the call graph in a bottom-up fashion, replacing monomorphic calls with inlined copies of the invoked methods.

Accepted phase options:

Enabled (`enabled`) (default value: `true`)

Insert Null Checks (`insert-null-checks`) (default value: `true`)

Insert, before the inlined body of the target method, a check that throws a `NullPointerException` if the receiver object is null. This ensures that inlining will not eliminate exceptions which would have occurred in its absence.

Insert Redundant Casts (`insert-redundant-casts`) (default value: `true`)

Insert extra casts for the Java bytecode verifier. The verifier may complain if the inlined method uses `this` and the declared type of the receiver of the call being inlined is different from the type implementing the target method being inlined.

Say, for example, that `Singer` is an interface declaring the `sing()` method and that the call graph shows that all receiver objects at a particular call site, `singer.sing()` (with `singer` declared as a `Singer`) are in fact `Bird` objects (`Bird` being a class that implements `Singer`). The implementation of `Bird.sing()` may perform operations on `this` which are only allowed on `Birds`, rather than `Singers`. The Insert Redundant Casts option ensures that this cannot lead to verification errors, by inserting a cast of `bird` to the `Bird` type before inlining the body of `Bird.sing()`.

Allowed Modifier Changes (`allowed-modifier-changes`) (default value: `unsafe`)

Specify which changes in visibility modifiers are allowed.

Possible values:

<code>unsafe</code>	Modify the visibility on code so that all inlining is permitted.
<code>safe</code>	Preserve the exact meaning of the analyzed program.
<code>none</code>	Change no modifiers whatsoever.

Expansion Factor (`expansion-factor`) (default value: 3)

Determines the maximum allowed expansion of a method. Inlining will cause the method to grow by a factor of no more than the Expansion Factor.

Max Container Size (`max-container-size`) (default value: 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 (`max-inlinee-size`) (default value: 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.

5 Whole-Jimple Annotation Pack (wjap)

Some analyses do not transform Jimple body directly, but annotate statements or values with tags. Whole-Jimple annotation pack provides a place for annotation-oriented analyses in whole program mode.

Accepted phase options:

Enabled (`enabled`) (default value: `true`)

5.1 Rectangular Array Finder (`wjap.ra`)

The Rectangular Array Finder traverses Jimple statements based on the static call graph, and finds array variables which always hold rectangular two-dimensional array objects.

In Java, a multi-dimensional array is an array of arrays, which means the shape of the array can be ragged. Nevertheless, many applications use rectangular arrays. Knowing that an array is rectangular can be very helpful in proving safe array bounds checks.

The Rectangular Array Finder does not change the program being analyzed. Its results are used by the Array Bound Checker.

Accepted phase options:

Enabled (`enabled`) (default value: `false`)

6 Shimple Control (`shimple`)

Shimple Control sets parameters which apply throughout the creation and manipulation of Shimple bodies. Shimple is Soot's SSA representation.

Accepted phase options:

Enabled (`enabled`) (default value: `true`)

Phi Node Elimination Optimizations (`phi-elim-opt`) (default value: `post`)

Possible values:

<code>none</code>	Do not optimize as part of Phi node elimination, either before or after eliminating Phi nodes. This is useful for monitoring and understanding the behaviour of Shimple optimizations and transformations.
<code>pre</code>	Perform some optimizations, such as dead code elimination and local aggregation, before eliminating Phi nodes. This appears to be less effective than post-optimization, but the option is provided for future testing and investigation.
<code>post</code>	Perform some optimizations, such as dead code elimination and local aggregation, after eliminating Phi nodes. This appears to be more effective than post-optimization.
<code>pre-and-post</code>	If enabled, applies recommended optimizations such as dead code elimination and local aggregation both before and after Phi node elimination. Provided for experimentation.

Local Name Standardization (`standard-local-names`) (default value: `false`)

If enabled, the Local Name Standardizer is applied whenever Shimple creates new locals. Normally, Shimple will retain the original local names as far as possible and use an underscore notation to denote SSA subscripts. This transformation does not otherwise affect Shimple behaviour.

Debugging Output (`debug`) (default value: `false`)

If enabled, Soot may print out warnings and messages useful for debugging the Shimple module. Automatically enabled by the global debug switch.

7 Shimple Transformation Pack (stp)

When the Shimple representation is produced, Soot applies the contents of the Shimple Transformation Pack to each method under analysis. This pack contains no transformations in an unmodified version of Soot.

Accepted phase options:

Enabled (`enabled`) (default value: `true`)

8 Shimple Optimization Pack (sop)

The Shimple Optimization Pack contains transformations that perform optimizations on Shimple, Soot's SSA representation.

Accepted phase options:

Enabled (`enabled`) (default value: `false`)

8.1 Shimple Constant Propagator and Folder (`sop.cpf`)

A powerful constant propagator and folder based on an algorithm sketched by Cytron et al that takes conditional control flow into account. This optimization demonstrates some of the benefits of SSA – particularly the fact that Phi nodes represent natural merge points in the control flow.

Accepted phase options:

Enabled (`enabled`) (default value: `true`)

Prune Control Flow Graph (`prune-cfg`) (default value: `true`)

Conditional branching statements that are found to branch unconditionally (or fall through) are replaced with unconditional branches (or removed). This transformation exposes more opportunities for dead code removal.

9 Jimple Transformation Pack (jtp)

Soot applies the contents of the Jimple Transformation Pack to each method under analysis. This pack contains no transformations in an unmodified version of Soot.

Accepted phase options:

Enabled (`enabled`) (default value: `true`)

10 Jimple Optimization Pack (jop)

When Soot's `Optimize` option is on, Soot applies the Jimple Optimization Pack to every `JimpleBody` in application classes. This section lists the default transformations in the Jimple Optimization Pack.

Accepted phase options:

Enabled (`enabled`) (default value: `false`)

10.1 Common Subexpression Eliminator (jop.cse)

The Common Subexpression Eliminator runs an available expressions analysis on the method body, then eliminates common subexpressions.

This implementation is especially slow, as it runs on individual statements rather than on basic blocks. A better implementation (which would find most common subexpressions, but not all) would use basic blocks instead.

This implementation 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 its current slowness, this transformation is not enabled by default.

Accepted phase options:

Enabled (`enabled`) (default value: `false`)

Naive Side Effect Tester (`naive-side-effect`) (default value: `false`)

If Naive Side Effect Tester is `true`, the Common Subexpression Eliminator uses the conservative side effect information provided by the `NaiveSideEffectTester` class, even if interprocedural information about side effects is available.

The naive side effect analysis is based solely on the information available locally about a statement. It assumes, for example, that any method call has the potential to write and read all instance and static fields in the program.

If Naive Side Effect Tester is set to `false` and Soot is in whole program mode, then the Common Subexpression Eliminator uses the side effect information provided by the `PASideEffectTester` class. `PASideEffectTester` uses a points-to analysis to determine which fields and statics may be written or read by a given statement.

If whole program analysis is not performed, naive side effect information is used regardless of the setting of Naive Side Effect Tester.

10.2 Busy Code Motion (jop.bcm)

Busy Code Motion is a straightforward implementation of Partial Redundancy Elimination. This implementation is not very aggressive. Lazy Code Motion is an improved version which should be used instead of Busy Code Motion.

Accepted phase options:

Enabled (`enabled`) (default value: `false`)

Naive Side Effect Tester (`naive-side-effect`) (default value: `false`)

If Naive Side Effect Tester is set to `true`, Busy Code Motion uses the conservative side effect information provided by the `NaiveSideEffectTester` class, even if interprocedural information about side effects is available.

The naive side effect analysis is based solely on the information available locally about a statement. It assumes, for example, that any method call has the potential to write and read all instance and static fields in the program.

If Naive Side Effect Tester is set to `false` and Soot is in whole program mode, then Busy Code Motion uses the side effect information provided by the `PASideEffectTester` class. `PASideEffectTester` uses a points-to analysis to determine which fields and statics may be written or read by a given statement.

If whole program analysis is not performed, naive side effect information is used regardless of the setting of Naive Side Effect Tester.

10.3 Lazy Code Motion (jop.lcm)

Lazy Code Motion is an enhanced version of 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.

Accepted phase options:

Enabled (`enabled`) (default value: `false`)

Safety (`safety`) (default value: `safe`)

This option controls which fields and statements are candidates for code motion.

Possible values:

<code>safe</code>	Safe, but only considers moving additions, subtractions and multiplications.
<code>medium</code>	Unsafe in multi-threaded programs, as it may reuse the values read from field accesses.
<code>unsafe</code>	May violate Java's exception semantics, as it may move or reorder exception-throwing statements, potentially outside of <code>try-catch</code> blocks.

Unroll (`unroll`) (default value: `true`)

If `true`, perform loop inversion before doing the transformation.

Naive Side Effect Tester (`naive-side-effect`) (default value: `false`)

If Naive Side Effect Tester is set to `true`, Lazy Code Motion uses the conservative side effect information provided by the `NaiveSideEffectTester` class, even if interprocedural information about side effects is available.

The naive side effect analysis is based solely on the information available locally about a statement. It assumes, for example, that any method call has the potential to write and read all instance and static fields in the program.

If Naive Side Effect Tester is set to `false` and Soot is in whole program mode, then Lazy Code Motion uses the side effect information provided by the `PASideEffectTester` class. `PASideEffectTester` uses a points-to analysis to determine which fields and statics may be written or read by a given statement.

If whole program analysis is not performed, naive side effect information is used regardless of the setting of Naive Side Effect Tester.

10.4 Copy Propagator (jop.cp)

This phase performs cascaded copy propagation.

Accepted phase options:

Enabled (`enabled`) (default value: `true`)

Only Regular Locals (`only-regular-locals`) (default value: `false`)

Only propagate copies through “regular” locals, that is, those declared in the source bytecode.

Only Stack Locals (`only-stack-locals`) (default value: `false`)

Only propagate copies through locals that represent stack locations in the original bytecode.

10.5 Jimple Constant Propagator and Folder (jop.cpf)

The Jimple Constant Propagator and Folder evaluates any expressions consisting entirely of compile-time constants, for example `2 * 3`, and replaces the expression with the constant result, in this case `6`.

Accepted phase options:

Enabled (`enabled`) (default value: `true`)

10.6 Conditional Branch Folder (jop.cbf)

The Conditional Branch Folder statically evaluates the conditional expression of Jimple `if` statements. If the condition is identically `true` or `false`, the Folder replaces the conditional branch statement with an unconditional `goto` statement.

Accepted phase options:

Enabled (`enabled`) (default value: `true`)

10.7 Dead Assignment Eliminator (jop.dae)

The Dead Assignment Eliminator eliminates assignment statements to locals whose values are not subsequently used, unless evaluating the right-hand side of the assignment may cause side-effects.

Accepted phase options:

Enabled (`enabled`) (default value: `true`)

Only Stack Locals (`only-stack-locals`) (default value: `false`)

Only eliminate dead assignments to locals that represent stack locations in the original bytecode.

10.8 Unreachable Code Eliminator 1 (jop.uce1)

The Unreachable Code Eliminator removes unreachable code and traps whose catch blocks are empty.

Accepted phase options:

Enabled (`enabled`) (default value: `true`)

10.9 Unconditional Branch Folder 1 (jop.ubf1)

The Unconditional Branch Folder removes unnecessary `goto` statements from a `JimpleBody`.

If a `goto` statement's target is the next instruction, then the statement is removed. If a `goto`'s target is another `goto`, with target `y`, then the first statement's target is changed to `y`.

If some `if` statement's target is a `goto` statement, then the `if`'s target can be replaced with the `goto`'s target.

(These situations can result from other optimizations, and branch folding may itself generate more unreachable code.)

Accepted phase options:

Enabled (`enabled`) (default value: `true`)

10.10 Unreachable Code Eliminator 2 (jop.uce2)

Another iteration of the Unreachable Code Eliminator.

Accepted phase options:

Enabled (`enabled`) (default value: `true`)

10.11 Unconditional Branch Folder 2 (`jop.ubf2`)

Another iteration of the Unconditional Branch Folder.

Accepted phase options:

Enabled (`enabled`) (default value: `true`)

10.12 Unused Local Eliminator (`jop.ule`)

The Unused Local Eliminator phase removes any unused locals from the method.

Accepted phase options:

Enabled (`enabled`) (default value: `true`)

11 Jimple Annotation Pack (`jap`)

The Jimple Annotation Pack contains phases which add annotations to Jimple bodies individually (as opposed to the Whole-Jimple Annotation Pack, which adds annotations based on the analysis of the whole program).

Accepted phase options:

Enabled (`enabled`) (default value: `true`)

11.1 Null Pointer Checker (`jap.npc`)

The Null Pointer Checker finds instructions which have the potential to throw `NullPointerException` and adds annotations indicating whether or not the pointer being dereferenced can be determined statically not to be null.

Accepted phase options:

Enabled (`enabled`) (default value: `false`)

Only Array Ref (`only-array-ref`) (default value: `false`)

Annotate only array-referencing instructions, instead of all instructions that need null pointer checks.

Profiling (`profiling`) (default value: `false`)

Insert profiling instructions that at runtime count the number of eliminated safe null pointer checks. The inserted profiling code assumes the existence of a `MultiCounter` class implementing the methods invoked. For details, see the `NullPointerChecker` source code.

11.2 Null Pointer Colourer (`jap.npcolorer`)

Produce colour tags that the Soot plug-in for Eclipse can use to highlight null and non-null references.

Accepted phase options:

Enabled (`enabled`) (default value: `false`)

11.3 Array Bound Checker (`jap.abc`)

The Array Bound Checker performs a static analysis to determine which array bounds checks may safely be eliminated and then annotates statements with the results of the analysis.

If Soot is in whole-program mode, the Array Bound Checker can use the results provided by the Rectangular Array Finder.

Accepted phase options:

Enabled (`enabled`) (default value: `false`)

With All (`with-all`) (default value: `false`)

Setting the With All option to true is equivalent to setting each of With CSE, With Array Ref, With Field Ref, With Class Field, and With Rectangular Array to true.

With Common Sub-expressions (`with-cse`) (default value: `false`)

The analysis will consider common subexpressions. For example, consider the situation where `r1` is assigned `a*b`; later, `r2` is assigned `a*b`, where neither `a` nor `b` have 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.

With Array References (`with-arrayref`) (default value: `false`)

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. We also assume that the application is single-threaded or that the array references occur in a synchronized block. That is, we assume that an array element may not be changed by other threads between two array references.

With Field References (`with-fieldref`) (default value: `false`)

The analysis treats field references (static and instance) as common subexpressions; however, we are more conservative when writing to a field, because the base of the field reference may be aliased. We also assume that the application is single-threaded or that the field references occur in a synchronized block. That is, we assume that a field may not be changed by other threads between two field references.

With Class Field (`with-classfield`) (default value: `false`)

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 of constant length. In an application using lots of array fields, this option can improve the analysis results dramatically.

With Rectangular Array (`with-rectarray`) (default value: `false`)

This option is used together with `wjap.ra` 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.

Profiling (`profiling`) (default value: `false`)

Profile the results of array bounds check analysis. The inserted profiling code assumes the existence of a `MultiCounter` class implementing the methods invoked. For details, see the `ArrayBoundsChecker` source code.

11.4 Profiling Generator (`jap.profiling`)

The Profiling Generator inserts the method invocations required to initialize and to report the results of any profiling performed by the Null Pointer Checker and Array Bound Checker. Users of the Profiling Generator must provide a `MultiCounter` class implementing the methods invoked. For details, see the `ProfilingGenerator` source code.

Accepted phase options:

Enabled (`enabled`) (default value: `false`)

Not Main Entry (`notmainentry`) (default value: `false`)

Insert the calls to the `MultiCounter` at the beginning and end of methods with the signature `long runBenchmark(java.lang.String[])` instead of the signature `void main(java.lang.String[])`.

11.5 Side Effect tagger (`jap.sea`)

The Side Effect Tagger uses the active invoke graph to produce side-effect attributes, as described in the Spark thesis, chapter 6.

Accepted phase options:

Enabled (`enabled`) (default value: `false`)

Build naive dependence graph (`naive`) (default value: `false`)

When set to true, the dependence graph is built with a node for each statement, without merging the nodes for equivalent statements. This makes it possible to measure the effect of merging nodes for equivalent statements on the size of the dependence graph.

11.6 Field Read/Write Tagger (`jap.fieldrw`)

The Field Read/Write Tagger uses the active invoke graph to produce tags indicating which fields may be read or written by each statement, including invoke statements.

Accepted phase options:

Enabled (`enabled`) (default value: `false`)

Maximum number of fields (`threshold`) (default value: 100)

If a statement reads/writes more than this number of fields, no tag will be produced for it, in order to keep the size of the tags reasonable.

11.7 Call Graph Tagger (`jap.cgtagger`)

The Call Graph Tagger produces `LinkTags` based on the call graph. The Eclipse plugin uses these tags to produce linked popup lists which indicate the source and target methods of the statement. Selecting a link from the list moves the cursor to the indicated method.

Accepted phase options:

Enabled (`enabled`) (default value: `false`)

11.8 Parity Tagger (`jap.parity`)

The Parity Tagger produces `StringTags` and `ColorTags` indicating the parity of a variable (even, odd, top, or bottom). The eclipse plugin can use tooltips and variable colouring to display the information in these tags. For example, even variables (such as `x` in `x = 2`) are coloured yellow.

Accepted phase options:

Enabled (`enabled`) (default value: `false`)

12 Grimp Body Creation (gb)

The Grimp Body Creation phase creates a `GrimpBody` for each source method. It is run only if the output format is `grimp` or `grimple`, or if class files are being output and the `Via Grimp` option has been specified.

Accepted phase options:

Enabled (`enabled`) (default value: `true`)

12.1 Grimp Pre-folding Aggregator (gb.a1)

The Grimp Pre-folding Aggregator combines some local variables, finding definitions with only a single use and removing the definition after replacing the use with the definition's right-hand side, if it is safe to do so. While the mechanism is the same as that employed by the Jimple Local Aggregator, there is more scope for aggregation because of Grimp's more complicated expressions.

Accepted phase options:

Enabled (`enabled`) (default value: `true`)

Only Stack Locals (`only-stack-locals`) (default value: `true`)

Aggregate only values stored in stack locals.

12.2 Grimp Constructor Folder (gb.cf)

The Grimp Constructor Folder combines `new` statements with the `specialinvoke` statement that calls the new object's constructor. For example, it turns

```
r2 = new java.util.ArrayList;  
r2.<init>();
```

into

```
r2 = new java.util.ArrayList();
```

Accepted phase options:

Enabled (`enabled`) (default value: `true`)

12.3 Grimp Post-folding Aggregator (gb.a2)

The Grimp Post-folding Aggregator combines local variables after constructors have been folded. Constructor folding typically introduces new opportunities for aggregation, since when a sequence of instructions like

```
r2 = new java.util.ArrayList;  
r2.<init>();  
r3 = r2
```

is replaced by

```
r2 = new java.util.ArrayList();
r3 = r2
```

the invocation of `<init>` no longer represents a potential side-effect separating the two definitions, so they can be combined into

```
r3 = new java.util.ArrayList();
```

(assuming there are no subsequent uses of `r2`).

Accepted phase options:

Enabled (enabled) (default value: `true`)

Only Stack Locals (only-stack-locals) (default value: `true`)

Aggregate only values stored in stack locals.

12.4 Grimp Unused Local Eliminator (gb.ule)

This phase removes any locals that are unused after constructor folding and aggregation.

Accepted phase options:

Enabled (enabled) (default value: `true`)

13 Grimp Optimization (gop)

The Grimp Optimization pack performs optimizations on `GrimpBodys` (currently there are no optimizations performed specifically on `GrimpBodys`, and the pack is empty). It is run only if the output format is `grimp` or `grimple`, or if class files are being output and the Via Grimp option has been specified.

Accepted phase options:

Enabled (enabled) (default value: `false`)

14 Baf Body Creation (bb)

The Baf Body Creation phase creates a `BafBody` from each source method. It is run if the output format is `baf` or `b`, or if class files are being output and the Via Grimp option has not been specified.

Accepted phase options:

Enabled (enabled) (default value: `true`)

14.1 Load Store Optimizer (bb.lso)

The Load Store Optimizer replaces some combinations of loads to and stores from local variables with stack instructions. A simple example would be the replacement of

```
store.r $r2;  
load.r $r2;
```

with

```
dup1.r
```

in cases where the value of `$r2` is not used subsequently.

Accepted phase options:

Enabled (`enabled`) (default value: `true`)

Debug (`debug`) (default value: `false`)

Produces voluminous debugging output describing the progress of the load store optimizer.

Inter (`inter`) (default value: `false`)

Enables two simple inter-block optimizations which attempt to keep some variables on the stack between blocks. Both are intended to catch `if`-like constructions where control flow branches temporarily into two paths that converge at a later point.

sl (`s1`) (default value: `true`)

Enables an optimization which attempts to eliminate `store/load` pairs.

sl2 (`s12`) (default value: `false`)

Enables an a second pass of the optimization which attempts to eliminate `store/load` pairs.

sll (`s11`) (default value: `true`)

Enables an optimization which attempts to eliminate `store/load/load` trios with some variant of `dup`.

sll2 (`s112`) (default value: `false`)

Enables an a second pass of the optimization which attempts to eliminate `store/load/load` trios with some variant of `dup`.

14.2 Peephole Optimizer (bb.pho)

Applies peephole optimizations to the Baf intermediate representation. Individual optimizations must be implemented by classes implementing the `Peephole` interface. The Peephole Optimizer reads the names of the `Peephole` classes at runtime from the file `peephole.dat` and loads them dynamically. Then it continues to apply the `Peephole`s repeatedly until none of them are able to perform any further optimizations.

Soot provides only one `Peephole`, named `ExamplePeephole`, which is not enabled by the delivered `peephole.dat` file. `ExamplePeephole` removes all `checkcast` instructions.

Accepted phase options:

Enabled (`enabled`) (default value: `true`)

14.3 Unused Local Eliminator (`bb.ule`)

This phase removes any locals that are unused after load store optimization and peephole optimization.

Accepted phase options:

Enabled (`enabled`) (default value: `true`)

14.4 Local Packer (`bb.lp`)

The Local Packer attempts to minimize the number of local variables required in a method by reusing the same variable for disjoint DU-UD webs. Conceptually, it is the inverse of the Local Splitter.

Accepted phase options:

Enabled (`enabled`) (default value: `true`)

Unsplit Original Locals (`unsplit-original-locals`) (default value: `false`)

Use the variable names in the original source as a guide when determining how to share local variables across non-interfering variable usages. This recombines named locals which were split by the Local Splitter.

15 Baf Optimization (`bop`)

The Baf Optimization pack performs optimizations on `BafBodys` (currently there are no optimizations performed specifically on `BafBodys`, and the pack is empty). It is run only if the output format is `baf` or `b`, or if class files are being output and the `Via Grimp` option has not been specified.

Accepted phase options:

Enabled (`enabled`) (default value: `false`)

16 Tag Aggregator (`tag`)

The Tag Aggregator pack aggregates tags attached to individual units into a code attribute for each method, so that these attributes can be encoded in Java class files.

Accepted phase options:

Enabled (`enabled`) (default value: `true`)

16.1 Line Number Tag Aggregator (`tag.ln`)

The Line Number Tag Aggregator aggregates line number tags.

Accepted phase options:

Enabled (`enabled`) (default value: `false`)

16.2 Array Bounds and Null Pointer Check Tag Aggregator (`tag.an`)

The Array Bounds and Null Pointer Tag Aggregator aggregates tags produced by the Array Bound Checker and Null Pointer Checker.

Accepted phase options:

Enabled (`enabled`) (default value: `false`)

16.3 Dependence Tag Aggregator (`tag.dep`)

The Dependence Tag Aggregator aggregates tags produced by the Side Effect Tagger.

Accepted phase options:

Enabled (`enabled`) (default value: `false`)

16.4 Field Read/Write Tag Aggregator (`tag.fieldrw`)

The Field Read/Write Tag Aggregator aggregates field read/write tags produced by the Field Read/Write Tagger, phase `jap.fieldrw`.

Accepted phase options:

Enabled (`enabled`) (default value: `false`)