Soot command-line options

CONTENTS

SYNOPSIS

Soot is invoked as follows:

```
java javaOptions soot.Main [ sootOption* ] classname*
```

DESCRIPTION

This manual documents the command line options of the Soot bytecode compiler/optimizer tool. In essence, it tells you what you can use to replace the *sootOption* placeholder which appears in the SYNOPSIS.

The descriptions of Soot options talk about three categories of classes: argument classes, application classes, and library classes.

Argument classes are those you specify explicitly to Soot. When you use Soot's command line interface, argument classes are those classes which are either listed explicitly on the command line or found in a directory specified with the `-process-dir` option. When you use the Soot's Eclipse plug-in, argument classes are those which you selected before starting Soot from the Navigator popup menu, or all classes in the current project if you started Soot from the Project menu.

Application classes are classes that Soot analyzes, transforms, and turns into output files.

Library classes are classes which are referred to, directly or indirectly, by the application classes, but which are not themselves application classes. Soot resolves these classes and reads `.class` or `.jimple` source files for them, but it does not perform transformations on library classes or write output files for them.

All argument classes are necessarily application classes. When Soot is not in ``application mode'', argument classes are the only application classes; other classes referenced from the argument classes become library classes.

When Soot is in application mode, every class referenced from the argument classes, directly or indirectly, is also an application class, unless its package name indicates that it is part of the standard Java runtime system.

Users may fine-tune the designation of application and library classes using the Application Mode Options.

Here is a simple example to clarify things. Suppose your program consists of three class files generated from the following source:

```
// UI.java
interface UI {
public void display(String msg);
}

// HelloWorld.java
class HelloWorld {
public static void main(String[] arg) {
UI ui = new TextUI();
ui.display("Hello World");
}
}

// TextUI.java
import java.io.*;
class TextUI implements UI {
public void display(String msg) {
System.out.println(msg);
}
}
```

If you run

```
java soot.Main HelloWorld
```

`HelloWorld` is the only argument class and the only application class. `UI` and `TextUI` are library classes, along with `java.lang.System`, `java.lang.String`, `java.io.PrintStream`, and a host of other classes from the Java runtime system that get dragged in indirectly by the references to `String` and `System.out`.

If you run

```
java soot.Main --app HelloWorld
```

`HelloWorld` remains the only argument class, but the application classes include `UI` and `TextUI` as well as `HelloWorld`. `java.lang.System` et. al. remain library classes.

If you run

```
java soot.Main -i java. --app HelloWorld
```

`HelloWorld` is still the only argument class, but the set of application classes includes the referenced Java runtime classes in packages whose names start with `java.` as well as `HelloWorld`, `UI`, and `textUI`. The set of library classes includes the referenced classes from other packages in the Java runtime.

OPTIONS

General Options

`-jasmin-backend`	Use the Jasmin back end for generating Java bytecode (instead of using ASM).
`-h` `-help`	Display help and exit
`-pl` `-phase-list`	Print list of available phases
`-ph` *phase* `-phase-help` *phase*	Print help for specified *phase*

-version	Display version information and exit
-v -verbose	Verbose mode
-interactive-mode	Run in interactive mode
-unfriendly-mode	Allow Soot to run with no command-line options
-app	Run in application mode
-w -whole-program	Run in whole-program mode
-ws -whole-shimple	Run in whole-shimple mode
-fly -on-the-fly	Run in on-the-fly mode
-validate	Run internal validation on bodies
-debug	Print various Soot debugging info
-debug-resolver	Print debugging info from SootResolver
-ignore-resolving-levels	Ignore mismatching resolving levels
-weak-map-structures	Use weak references in Scene to prevent memory leakage when removing many classes/methods/locals

Input Options

-cp *path* -soot-class-path *path* -soot-classpath *path*	Use *path* as the classpath for finding classes.
-soot-modulepath *modulepath*	Use *modulepath* as the modulepath for finding classes.
-dotnet-nativehost-path *arg*	Use to locate your NativeHost Java JNI library.
-pp -prepend-classpath	Prepend the given soot classpath to the default classpath.
-ice -ignore-classpath-errors	Ignores invalid entries on the Soot classpath.
-process-multiple-dex	Process all DEX files found in APK.
-search-dex-in-archives	Also includes Jar and Zip files when searching for DEX files under the provided classpath.
-process-path *dir* -process-dir *dir*	Process all classes found in *dir* (but not classes within JAR files in *dir*, use process-jar-dir for that)
-process-jar-dir *dir*	Process all classes found in JAR files found in *dir*
-virtualedges-path *arg*	Path to virtual edges configuration used in call graphs
-derive-java-version	Java version for output and internal processing will be derived from the given input classes
-oaat	From the process-dir, processes one class at a time.
-android-jars *path*	Use *path* as the path for finding the android.jar file
-force-android-jar *path*	Force Soot to use *path* as the path for the android.jar file.
-android-api-version *version*	Force Soot to use *version* as the API version when readin in APK or DEX files.
-ast-metrics	Compute AST Metrics if performing java to jimple
-src-prec *format*	c class only-class Sets source precedence to *format* files

	`J  jimple`
	`java`
	`apk`
	`apk-class-jimple  apk-c-j`
	`dotnet`	

`-full-resolver`	Force transitive resolving of referenced classes	
`-ignore-methodsource-error`	Ignore errors from method source and return empty jimple body	
`-resolve-all-dotnet-methods`	Resolve all dotnet methods, such as unsafe methods	
`-allow-phantom-refs`	Allow unresolved classes; may cause errors	
`-allow-phantom-elms`	Allow phantom methods and fields in non-phantom classes	
`-allow-cg-errors`	Allow Errors during callgraph construction	
`-no-bodies-for-excluded`	Do not load bodies for excluded classes	
`-j2me`	Use J2ME mode; changes assignment of types	
`-main-class` *`class`*	Sets the main class for whole-program analysis.	
`-polyglot`	Use Java 1.4 Polyglot frontend instead of JastAdd	
`-permissive-resolving`	Use alternative sources when classes cannot be found using the normal resolving strategy	
`-drop-bodies-after-load`	Drop the method source after it has served its purpose of loading the method body	
`-nc` `-native-code`	Enables native methods to be concrete. Needed for analyzing the Java Native Interface.	

Output Options

`-d` *`dir`*
`-output-dir` *`dir`*
	Store output files in *dir*	

`-f` *`format`*
`-output-format` *`format`*
	`J  jimple`
	`j  jimp`
	`S  shimple`
	`s  shimp`
	`B  baf`
	`b`
	`G  grimple`
	`g  grimp`
	`X  xml`
	`dex`
	`force-dex`
	`n  none`
	`jasmin`
	`c  class`
	`d  dava`
	`t  template`
	`a  asm`
		Set output format for Soot

`-java-version` *`version`*	`default`
	`1.1  1`
	`1.2  2`
	`1.3  3`
	`1.4  4`
	`1.5  5`
	`1.6  6`
	`1.7  7`
	`1.8  8`
	`1.9  9`
	`1.10  10`
	`1.11  11`
	`1.12  12`
		Force Java version of bytecode generated by Soot.

-outjar -output-jar	Make output dir a Jar file instead of dir
-hierarchy-dirs	Generate class hierarchy directories for Jimple/Shimple
-xml-attributes	Save tags to XML attributes for Eclipse
-print-tags -print-tags-in-output	Print tags in output files after stmt
-no-output-source-file-attribute	Don't output Source File Attribute when producing class files
-no-output-inner-classes-attribute	Don't output inner classes attribute in class files
-dump-body *phaseName*	Dump the internal representation of each method before and after phase *phaseName*
-dump-cfg *phaseName*	Dump the internal representation of each CFG constructed during phase *phaseName*
-show-exception-dests	Include exception destination edges as well as CFG edges in dumped CFGs
-gzip	GZip IR output files
-force-overwrite	Force Overwrite Output Files

Processing Options

-plugin *file*	Load all plugins found in *file*	
-wrong-staticness *arg*	fail ignore fix fixstrict	Ignores or fixes errors due to wrong staticness
-field-type-mismatches *arg*	fail ignore null	Specifies how errors shall be handled when resolving field references with mismatching types
-p *phase opt:val* -phase-option *phase opt:val*	Set *phase*'s *opt* option to *value*	
-t *num* -num-threads *num*	Force Soot to use *num* threads when transforming classes.	
-O -optimize	Perform intraprocedural optimizations	
-W -whole-optimize	Perform whole program optimizations	
-via-grimp	Convert to bytecode via Grimp instead of via Baf	
-via-shimple	Enable Shimple SSA representation	
-throw-analysis *arg*	pedantic unit dalvik dotnet auto-select	
-check-init-ta *arg* -check-init-throw-analysis *arg*	auto pedantic unit dalvik dotnet	
-omit-excepting-unit-edges	Omit CFG edges to handlers from excepting units which lack side effects	
-trim-cfgs	Trim unrealizable exceptional edges from CFGs	

`-ire` `-ignore-resolution-errors`	Does not throw an exception when a program references an undeclared field or method.

Application Mode Options

`-i` *pkg* `-include` *pkg*	Include classes in *pkg* as application classes
`-x` *pkg* `-exclude` *pkg*	Exclude classes in *pkg* from application classes
`-include-all`	Set default excluded packages to empty list
`-dynamic-class` *class*	Note that *class* may be loaded dynamically
`-dynamic-dir` *dir*	Mark all classes in *dir* as potentially dynamic
`-dynamic-package` *pkg*	Marks classes in *pkg* as potentially dynamic

Input Attribute Options

`-keep-line-number`	Keep line number tables
`-keep-bytecode-offset` `-keep-offset`	Attach bytecode offset to IR

Output Attribute Options

`-write-local-annotations`	Write out debug annotations on local names

Annotation Options

`-annot-purity`	Emit purity attributes
`-annot-nullpointer`	Emit null pointer attributes
`-annot-arraybounds`	Emit array bounds check attributes
`-annot-side-effect`	Emit side-effect attributes
`-annot-fieldrw`	Emit field read/write attributes

Miscellaneous Options

`-time`	Report time required for transformations
`-subtract-gc`	Subtract gc from time
`-no-writeout-body-releasing`	Disables the release of method bodies after writeout. This flag is used internally.

PHASES AND PHASE OPTIONS

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, `JimpleBody`s 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.

Jimple Body Creation (jb)

Jimple Body Creation creates a JimpleBody for each input method, using either asm, 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.

- **Preserve source-level annotations** (preserve-source-annotations)
(default value: `false`)

  Preserves annotations of retention type SOURCE. (for everything but package and local variable annotations)

- **Stabilize local names** (stabilize-local-names)
(default value: `false`)

  Make sure that local names are stable between runs. This requires re-normalizing all local names after the standard transformations, sorting them, and padding all local names with leading zeros up to the maximum number of digits in the local with the highest integer value. This can negatively impact performance. This option automatically sets "sort-locals" in "jb.lns" during the second re-normalization pass.

- **Model LambdaMetafactory** (model-lambdametafactory)
(default value: `true`)

  When the asm bytecode frontend is used and this option is set to true, Soot creates an implementation of the LambdaMetafactory for each dynamic invoke and replaces the original dynamic invoke by a static invocation of the factory's bootstrap method. This allows the call-graph generation to find the lambda body reachable, i.e., call-graphs contain paths from the invocation of a functional interface to the lambda body implementing this interface. Note that this procedure is not reversed when writing-out. Therefore, written-out code will contain the created LambdaMetafactories and instrumented calls to the corresponding bootstrap methods.

Duplicate CatchAll Trap Remover (jb.dtr)

This transformer detects cases in which the same code block is covered by two different catch all traps with different exception handlers (A and B), and if there is at the same time a third catch all trap that covers the second handler B and jumps to A, then the second trap is unnecessary, because it is already covered by a combination of the other two traps. This transformer removes the unnecessary trap.

Accepted phase options:

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

Empty Switch Eliminator (jb.ese)

The Empty Switch Eliminator detects and removes switch statements that have no data labels. Instead, the code is transformed to contain a single jump statement to the default label.

Accepted phase options:

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

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`)

Shared Initialization Local Splitter (jb.sils)

Accepted phase options:

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

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.)

Unused Local Eliminator (jb.ule)

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

Accepted phase options:

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

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`)

- **Use older type assigner** (use-older-type-assigner)
  (default value: `false`)

This enables the older type assigner that was in use until May 2008. The current type assigner is a reimplementation by Ben Bellamy that uses an entirely new and faster algorithm which always assigns the most narrow type possible. If compare-type-assigners is on, this option causes the older type assigner to execute first. (Otherwise the newer one is executed first.)

- **Compare type assigners** (compare-type-assigners)
  (default value: `false`)

  Enables comparison (both runtime and results) of Ben Bellamy's type assigner with the older type assigner that was in Soot.

- **Ignore Nullpointer Dereferences** (ignore-nullpointer-dereferences)
  (default value: `false`)

  If this option is enabled, Soot wiil not check whether the base object of a virtual method call can only be null. This will lead to the null_type pseudo type being used in your Jimple code.

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.

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.

- **Sort Locals** (sort-locals)
  (default value: `false`)

  First sorts the locals alphabetically by the string representation of their type. Then if there are two locals with the same type, it uses the only other source of structurally stable information (i.e. the instructions themselves) to produce an ordering for the locals that remains consistent between different soot instances. It achieves this by determining the position of a local's first occurrence in the instruction's list of definition statements. This position is then used to sort the locals with the same type in an ascending order. The local names are then appended with leading zeros so that all local names have the same length (i.e., the same number of digits) as the largest local name in each individual method body.

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.

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.

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`)

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.

Nop Eliminator (jb.ne)

The Nop Eliminator removes nop statements from the method.

Accepted phase options:

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

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`)

- **Remove unreachable traps** (remove-unreachable-traps)
  (default value: `true`)

  Remove exception table entries when none of the protected instructions can throw the exception being caught.

Trap Tightener (jb.tt)

The Trap Tightener changes the area protected by each exception handler, so that it begins with the first instruction in the old protected area which is actually capable of throwing an exception caught by the handler, and ends just after the last instruction in the old protected area which can throw an exception caught by the handler. This reduces the chance of producing unverifiable code as a byproduct of pruning exceptional control flow within CFGs.

Accepted phase options:

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

Conditional Branch Folder (jb.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`)

Java To Jimple Body Creation (jj)

Jimple Body Creation creates a JimpleBody for each input method, using polyglot, to read .java files.

Accepted phase options:

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

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

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

Local Splitter (jj.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: `false`)

Shared Initialization Local Splitter (jj.sils)

Accepted phase options:

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

Jimple Local Aggregator (jj.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.)

Unused Local Eliminator (jj.ule)

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

Accepted phase options:

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

Type Assigner (jj.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: `false`)

Unsplit-originals Local Packer (jj.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: `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 among non-interfering variable usages. This recombines named locals which were split by the Local Splitter.

Local Name Standardizer (jj.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.

Copy Propagator (jj.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.

Dead Assignment Eliminator (jj.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.

Post-copy propagation Unused Local Eliminator (jj.cp-ule)

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

Accepted phase options:

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

Local Packer (jj.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.

Nop Eliminator (jj.ne)

The Nop Eliminator removes nop statements from the method.

Accepted phase options:

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

Unreachable Code Eliminator (jj.uce)

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

Accepted phase options:

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

Whole Jimple Pre-processing Pack (wjpp)

This pack allows you to insert pre-processors that are run before call-graph construction. Only enabled in whole-program mode.

Accepted phase options:

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

Constant reflective method invocation base transformer (wjpp.cimbt)

When using the types-for-invoke option of the cg phase, problems might occur if the base object of a call to Method.invoke() (the first argument) is a string constant. This option replaces all string constants of such calls by locals and, therefore, allows the static resolution of reflective call targets on constant string objects.

Accepted phase options:

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

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

Whole Shimple Pre-processing Pack (wspp)

This pack allows you to insert pre-processors that are run before call-graph construction. Only enabled in whole-program Shimple mode. In an unmodified copy of Soot, this pack is empty.

Accepted phase options:

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

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: `false`)

  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: `false`)

  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.

- **Library mode** (library)
  (default value: `disabled`)

  Specifies whether the target classes should be treated as an application or a library. If library mode is disabled (default), the call graph construction assumes that the target is an application and starts the construction from the specified entry points (main method by default). Under the assumption that the target is a library, possible call edges might be missing in the call graph. The two different library modes add theses missing calls to the call graph and differ only in the view of the class hierachy (hierachy of target library or possible extended hierachy). If simulate-natives is also set, the results of native methods are also set to any sub type of the declared return type.

  Possible values:

`disabled` Call(and pointer assignment) graph construction treat the target classes as application starting from the entry points.

`any-subtype` On library analysis it has to be assumed, that a possible client can call any method or access any field, to which he has the access rights (default public/protected but can be set with soot.Scene#setClientAccessibilityOracle). In this mode types of any accessible field, method parameter, this local, or caugth exception is set to any possible sub type according to the class hierachy of the target library. If simulate-natives is also set, the results of native methods are also set to any sub type of the declared return type.

`signature-resolution` On library analysis it has to be assumed, that a possible client can call any method or access any field, to which he has the access rights (default public/protected but can be set with soot.Scene#setClientAccessibilityOracle). In this mode types of any accessible field, method parameter, this local, or caugth exception is set to any possible sub type according to a possible extended class hierarchy of the target library. Whenever any sub type of a specific type is considered as receiver for a method to call and the base type is an interface, calls to existing methods with matching signature (possible implementation of method to call) are also added. As Javas' subtyping allows contra-variance for return types and co-variance for parameters when overriding a method, these cases are also considered here. Example: Classes A, B (B sub type of A), interface I with method public A foo(B b); and a class C with method public B foo(A a) { ... }. The extended class hierachy will contain C as possible implementation of I. If simulate-natives is also set, the results of native methods are also set to any possible sub type of the declared return type.

- **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.

- **JDK version** (jdkver)
  (default value: `3`)

  This option sets the JDK version of the standard library being analyzed so that Soot can simulate the native methods in the specific version of the library. The default, 3, refers to Java 1.3.x.

- **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.

- **Implicit Entry Points** (implicit-entry)
  (default value: `true`)

  When this option is true, methods that are called implicitly by the VM are considered entry points of the call graph. When it is false, these methods are not considered entry points, leading to a possibly incomplete call graph.

- **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.

- **Reflection Log** (reflection-log)

  Load a reflection log from the given file and use this log to resolve reflective call sites. Note that when a log is given, the following other options have no effect: safe-forname, safe-newinstance.

- **Guarding strategy** (guards)
  (default value: `ignore`)

  Using a reflection log is only sound for method executions that were logged. Executing the program differently may be unsound. Soot can insert guards at program points for which the reflection log contains no information. When these points are reached (because the program is executed differently) then the follwing will happen, depending on the value of this flag. ignore: no guard is inserted, the program executes normally but under unsound assumptions. print: the program prints a stack trace when reaching a porgram location that was not traced but continues to run. throw (default): the program throws an Error instead.

- **Types for invoke** (types-for-invoke)
  (default value: `false`)

  For each call to Method.invoke(), use the possible types of the first receiver argument and the possible types stored in the second argument array to resolve calls to Method.invoke(). This strategy makes no attempt to resolve reflectively invoked static methods. Currently only works for context insensitive pointer analyses.

- **Resolve Abstract Classes with No Children** (resolve-all-abstract-invokes)
  (default value: `false`)

  Normally, if a method is invoked on a class that is abstract and said class does not have any children in the Scene, the method invoke will not be resolved to any concrete methods even if the abstract class or its parent classes contain a concrete declaration of the method. This is because without any non-abstract children it is impossible to tell if the resolution is correct (since any child may override any non-private method in any of its parent classes). However, sometimes it is necessary to resolve methods in such situations (e.g. when analyzing libraries or incomplete code). This forces all methods invoked on abstract classes to be resolved if there exists a parent class with a concrete definition of the method even if there are no non-abstract children of the abstract class.

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.

- **AppOnly** (apponly)
  (default value: `false`)

  Setting this option to true causes Soot to only consider application classes when building the callgraph. The resulting callgraph will be inherently unsound. Still, this option can make sense if performance optimization and memory reduction are your primary goal.

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`)

- **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.

- **AppOnly** (apponly)
  (default value: `false`)

  Setting this option to true causes Soot to only consider application classes when building the callgraph. The resulting callgraph will be inherently unsound. Still, this option can make sense if performance optimization and memory reduction are your primary goal.

- **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.

- **Propagate All String Constants** (string-constants)
  (default value: `false`)

  When this option is set to false, Spark only distinguishes string constants that may be the name of a class loaded dynamically using reflection, and all other string constants are lumped together into a single string constant node. Setting this option to true causes all string constants to be propagated individually.

- **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.

- **Treat EMPTY as Alloc** (empties-as-allocs)
  (default value: `false`)

  When this option is set to true, Spark treats references to EMPTYSET, EMPTYMAP, and EMPTYLIST as allocation sites for HashSet, HashMap and LinkedList objects respectively, and references to Hashtable.emptyIterator as allocation sites for Hashtable.EmptyIterator. This enables subsequent analyses to differentiate different uses of Java's immutable empty collections.

- **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.

- **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.

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

  This option tells Spark which propagation algorithm to use.

  Possible values:

`iter`	Iter is a simple, iterative algorithm, which propagates everything until the graph does not change.
`worklist`	Worklist is a worklist-based algorithm that tries to do as little work as possible. This is currently the fastest algorithm.
`cycle`	This algorithm finds cycles in the PAG on-the-fly. It is not yet finished.
`merge`	Merge is an algorithm that merges all concrete field (yellow) nodes with their corresponding field reference (red) nodes. This algorithm is not yet finished.
`alias`	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.
`none`	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:

`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.
`heintze`	Heintze's representation has elements represented by a bit-vector + a small 'overflow' list of some maximum number of elements. The bit-vectors can be shared by multiple points-to sets, while the overflow lists are not.
`sharedlist`	Shared List stores its elements in a linked list, and might share its tail with other similar points-to sets.
`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.

- **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.
`heintze`	Heintze's representation has elements represented by a bit-vector + a small 'overflow' list of some maximum number of elements. The bit-vectors can be shared by multiple points-to sets, while the overflow lists are not.
`sharedlist`	Shared List stores its elements in a linked list, and might share its tail with other similar points-to sets.

- **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.
`heintze`	Heintze's representation has elements represented by a bit-vector + a small 'overflow' list of some maximum number of elements. The bit-vectors can be shared by multiple points-to sets, while the overflow lists are not.
`sharedlist`	Shared List stores its elements in a linked list, and might share its tail with other similar points-to sets.

- **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.

- **Demand-driven refinement-based context-sensitive points-to analysis** (cs-demand)
  (default value: `false`)

  When this option is set to true, Manu Sridharan's demand-driven, refinement-based points-to analysis (PLDI 06) is applied after Spark was run.

- **Create lazy points-to sets** (lazy-pts)
  (default value: `true`)

  When this option is disabled, context information is computed for every query to the reachingObjects method. When it is enabled, a call to reachingObjects returns a lazy wrapper object that contains a context-insensitive points-to set. This set is then automatically refined with context information when necessary, i.e. when we try to determine the intersection with another points-to set and this intersection seems to be non-empty.

- **Maximal traversal** (traversal)
  (default value: `75000`)

Make the analysis traverse at most this number of nodes per query. This quota is evenly shared between multiple passes (see next option).

- **Maximal number of passes** (passes)
  (default value: `10`)

  Perform at most this number of refinement iterations. Each iteration traverses at most (traverse / passes) nodes.

- **Geometric, context-sensitive points-to analysis** (geom-pta)
  (default value: `false`)

  This switch enables/disables the geometric analysis.

- **Encoding methodology used** (geom-encoding)
  (default value: `Geom`)

  This switch specifies the encoding methodology used in the analysis. All possible options are: Geom, HeapIns, PtIns. The efficiency order is (from slow to fast) Geom - HeapIns - PtIns, but the precision order is the reverse.

  Possible values:

`Geom`	Geometric Encoding.
`HeapIns`	Heap Insensitive Encoding. Omit the heap context range term in the encoded representation, and in turn, we assume all the contexts for this heap object are used.
`PtIns`	Pointer Insensitive Encoding. Similar to HeapIns, but we omit the pointer context range term.

- **Worklist type** (geom-worklist)
  (default value: `PQ`)

  Specifies the worklist used for selecting the next propagation pointer. All possible options are: PQ, FIFO. They stand for the priority queue (sorted by the last fire time and topology order) and FIFO queue.

  Possible values:

`PQ`	Priority Queue (sorted by the last fire time and topology order)
`FIFO`	FIFO Queue

- **Verbose dump file** (geom-dump-verbose)
  (default value:)

  If you want to save the geomPTA analysis information for future analysis, please provide a file name.

- **Verification file** (geom-verify-name)
  (default value:)

  If you want to compare the precision of the points-to results with other solvers (e.g. Paddle), you can use the 'verify-file' to specify the list of methods (soot method signature format) that are reachable by that solver. During the internal evaluations (see the option geom-eval), we only consider the methods that are common to both solvers.

- **Precision evaluation methodologies** (geom-eval)
  (default value: `0`)

  We internally provide some precision evaluation methodologies and classify the evaluation strength into three levels. If level is 0, we do nothing. If level is 1, we report the statistical information about the points-to result. If level is 2, we perform the virtual callsite resolution, static cast safety and all-pairs alias evaluations.

- **Transform to context-insensitive result** (geom-trans)
  (default value: `false`)

If you stick to working with SPARK, you can use this option to transform the context sensitive result to insensitive result. After the transformation, the context sensitive points-to quries cannot be answered.

- **Fractional parameter** (geom-frac-base)
  (default value: `40`)

  This option specifies the fractional parameter, which manually balances the precision and the performance. Smaller value means better performance and worse precision.

- **Blocking strategy for recursive calls** (geom-blocking)
  (default value: `true`)

  Blocking strategy is a 1CFA model for recursive calls. This model significantly improves the precision.

- **Iterations** (geom-runs)
  (default value: `1`)

  We can run multiple times of the geometric analysis to continuously improve the analysis precision.

- **Pointers processed by geomPTA** (geom-app-only)
  (default value: `true`)

  When this option is true, geomPTA only processes the pointers in library functions (java.*, sun.*, and etc.) that potentially impact the points-to information of pointers in application code, the pointers in application code, and the base pointers at virtual callsites.

Paddle (cg.paddle)

Paddle is a BDD-based interprocedural analysis framework. It includes points-to analysis, call graph construction, and various client analyses.

Accepted phase options:

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

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

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

- **Configuration** (conf)
  (default value: `ofcg`)

  Selects the configuration of points-to analysis and call graph construction to be used in Paddle.

  Possible values:

`ofcg`	Performs points-to analysis and builds call graph together, on-the-fly.
`cha`	Builds only a call graph using Class Hieararchy Analysis, and performs no points-to analysis.
`cha-aot`	First builds a call graph using CHA, then uses the call graph in a fixed-call-graph points-to analysis.
`ofcg-aot`	First builds a call graph on-the-fly during a points-to analysis, then uses the resulting call graph to perform a second points-to analysis with a fixed call graph.
`cha-context-aot`	First builds a call graph using CHA, then makes it context-sensitive using the technique described by Calman and Zhu in PLDI 04, then uses the call graph in a fixed-call-graph points-to analysis.

`ofcg-context-aot`	First builds a call graph on-the-fly during a points-to analysis, then makes it context-sensitive using the technique described by Calman and Zhu in PLDI 04, then uses the resulting call graph to perform a second points-to analysis with a fixed call graph.
`cha-context`	First builds a call graph using CHA, then makes it context-sensitive using the technique described by Calman and Zhu in PLDI 04. Does not produce points-to information.
`ofcg-context`	First builds a call graph on-the-fly during a points-to analysis, then makes it context-sensitive using the technique described by Calman and Zhu in PLDI 04. Does not perform a subsequent points-to analysis.

- **Use BDDs** (bdd)
  (default value: `false`)

  Causes Paddle to use BDD versions of its components

- **Variable ordering** (order)
  (default value: `32`)

  Selects one of the BDD variable orderings hard-coded in Paddle.

- **Dynamic reordering** (dynamic-order)

  Allows the BDD package to perform dynamic variable ordering.

- **Profile** (profile)
  (default value: `false`)

  Turns on JeddProfiler for profiling BDD operations.

- **Verbose GC** (verbosegc)
  (default value: `false`)

  Print memory usage at each BDD garbage collection.

- **Worklist Implementation** (q)
  (default value: `auto`)

  Select the implementation of worklists to be used in Paddle.

  Possible values:

`auto`	When the bdd option is true, the BDD-based worklist implementation will be used. When the bdd option is false, the Traditional worklist implementation will be used.
`trad`	Normal worklist queue implementation
`bdd`	BDD-based queue implementation
`debug`	An implementation of worklists that includes both traditional and BDD-based implementations, and signals an error whenever their contents differ.
`trace`	A worklist implementation that prints out all tuples added to every worklist.
`numtrace`	A worklist implementation that prints out the number of tuples added to each worklist after each operation.

- **Backend** (backend)
  (default value: `auto`)

  This option tells Paddle which implementation of BDDs to use.

  Possible values:

`auto`	When the bdd option is true, the BuDDy backend will be used. When the bdd option is false, the backend will be set to none, to avoid loading any BDD backend.
`buddy`	Use BuDDy implementation of BDDs.

`cudd`	Use CUDD implementation of BDDs.
`sable`	Use SableJBDD implementation of BDDs.
`javabdd`	Use JavaBDD implementation of BDDs.
`none`	Don't use any BDD backend. Any attempted use of BDDs will cause Paddle to crash.

- **BDD Nodes** (bdd-nodes)
  (default value: `0`)

  This option specifies the number of BDD nodes to be used by the BDD backend. A value of 0 causes the backend to start with one million nodes, and allocate more as required. A value other than zero causes the backend to start with the specified size, and prevents it from ever allocating any more nodes.

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

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

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

  When this option is set to true, Paddle 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.

- **Context abstraction** (context)
  (default value: `insens`)

  This option tells Paddle which level of context-sensitivity to use in constructing the call graph.

  Possible values:

`insens`	Builds a context-insensitive call graph.
`1cfa`	Builds a 1-CFA call graph.
`kcfa`	Builds a k-CFA call graph.
`objsens`	Builds an object-sensitive call graph.
`kobjsens`	Builds a context-sensitive call graph where the context is a string of up to k receiver objects.
`uniqkobjsens`	Builds a context-sensitive call graph where the context is a string of up to k unique receiver objects. If the receiver of a call already appears in the context string, the context string is just reused as is.
`threadkobjsens`	Experimental option for thread-entry-point sensitivity.

- **Context length (k)** (k)
  (default value: `2`)

  The maximum length of call string or receiver object string used as context.

- **Context-sensitive Heap Locations** (context-heap)
  (default value: `false`)

  When this option is set to true, the context-sensitivity level that is set for the context-sensitive call graph and for pointer variables is also used to model heap locations context-sensitively. When this option is false, heap locations are modelled context-insensitively regardless of the context-sensitivity level.

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

Setting RTA to true sets types-for-sites to true, and causes Paddle 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. Allocation sites creating a java.lang.StringBuilder object are also grouped together as a single allocation site.

- **Propagate All String Constants** (string-constants)
  (default value: `false`)

  When this option is set to false, Paddle only distinguishes string constants that may be the name of a class loaded dynamically using reflection, and all other string constants are lumped together into a single string constant node. Setting this option to true causes all string constants to be propagated individually.

- **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.

- **Global Nodes in Simulated Natives** (global-nodes-in-natives)
  (default value: `false`)

  The simulations of native methods such as System.arraycopy() use temporary local variable nodes. Setting this switch to true causes them to use global variable nodes instead, reducing precision. The switch exists only to make it possible to measure this effect on precision; there is no other practical reason to set it to true.

- **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.

- **this Pointer Assignment Edge** (this-edges)
  (default value: `false`)

  When constructing a call graph on-the-fly during points-to analysis, Paddle normally propagates only those receivers that cause a method to be invoked to the this pointer of the method. When this option is set to true, however, Paddle instead models flow of receivers as an assignnment edge from the receiver at the call site to the this pointer of the method, reducing precision.

- **Precise newInstance** (precise-newinstance)
  (default value: `true`)

  Normally, newInstance() calls are treated as if they may return an object of any type. Setting this option to true causes them to be treated as if they return only objects of the type of some dynamic class.

- **Propagator** (propagator)
  (default value: `auto`)

  This option tells Paddle which propagation algorithm to use.

  Possible values:

  `auto` When the bdd option is true, the Incremental BDD propagation algorithm will be used. When the bdd option is false, the Worklist propagation algorithm will be used.

  `iter` Iter is a simple, iterative algorithm, which propagates everything until the graph does not change.

  `worklist` Worklist is a worklist-based algorithm that tries to do as little work as possible. This is currently the fastest algorithm.

  `alias` 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.

  `bdd` BDD is a propagator that stores points-to sets in binary decision diagrams.

  `incbdd` A propagator that stores points-to sets in binary decision diagrams, and propagates them incrementally.

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

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

  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.

  `heintze` Heintze's representation has elements represented by a bit-vector + a small 'overflow' list of some maximum number of elements. The bit-vectors can be shared by multiple points-to sets, while the overflow lists are not.

  `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.

- **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.

`heintze` Heintze's representation has elements represented by a bit-vector + a small 'overflow' list of some maximum number of elements. The bit-vectors can be shared by multiple points-to sets, while the overflow lists are not.

- **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.

  `heintze` Heintze's representation has elements represented by a bit-vector + a small 'overflow' list of some maximum number of elements. The bit-vectors can be shared by multiple points-to sets, while the overflow lists are not.

- **Print Context Counts** (context-counts)
  (default value: `false`)

  Causes Paddle to print the number of contexts for each method and call edge, and the number of equivalence classes of contexts for each variable node.

- **Print Context Counts (Totals only)** (total-context-counts)
  (default value: `false`)

  Causes Paddle to print the number of contexts and number of context equivalence classes.

- **Method Context Counts (Totals only)** (method-context-counts)
  (default value: `false`)

  Causes Paddle to print the number of contexts and number of context equivalence classes split out by method. Requires total-context-counts to also be turned on.

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

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

- **Number nodes** (number-nodes)
  (default value: `true`)

  When printing debug information about nodes, this option causes the node number of each node to be printed.

Whole Shimple Transformation Pack (wstp)

Soot can perform whole-program analyses. In whole-shimple mode, Soot applies the contents of the Whole-Shimple Transformation Pack to the scene as a whole after constructing a call graph for the program. In an unmodified copy of Soot the Whole-Shimple Transformation Pack is empty.

Accepted phase options:

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

Whole Shimple Optimization Pack (wsop)

If Soot is running in whole shimple mode and the Whole-Shimple 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-Shimple Transformations. In an unmodified copy of Soot the Whole-Shimple Optimization Pack is empty.

Accepted phase options:

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

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.

Accepted phase options:

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

May Happen in Parallel Analyses (wjtp.mhp)

May Happen in Parallel (MHP) Analyses determine what program statements may be run by different threads concurrently. This phase does not perform any transformation.

Accepted phase options:

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

Lock Allocator (wjtp.tn)

The Lock Allocator finds critical sections (synchronized regions) in Java programs and assigns locks for execution on both optimistic and pessimistic JVMs. It can also be used to analyze the existing locks.

Accepted phase options:

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

- **Locking Scheme** (locking-scheme)
  (default value: `medium-grained`)

Selects the granularity of the generated lock allocation

Possible values:

`medium-grained` Try to identify transactional regions that can employ a dynamic lock to increase parallelism. All side effects must be protected by a single object. This locking scheme aims to approximate typical Java Monitor usage.

`coarse-grained` Insert static objects into the program for synchronization. One object will be used for each group of conflicting synchronized regions. This locking scheme achieves code-level locking.

`single-static` Insert one static object into the program for synchronization for all transactional regions. This locking scheme is for research purposes.

`leave-original` Analyse the existing lock structure of the program, but do not change it. With one of the print options, this can be useful for comparison between the original program and one of the generated locking schemes.

- **Perform Deadlock Avoidance** (avoid-deadlock)
  (default value: `true`)

  Perform Deadlock Avoidance by enforcing a lock ordering where necessary.

- **Use Open Nesting** (open-nesting)
  (default value: `true`)

  Use an open nesting model, where inner transactions are allowed to commit independently of any outer transaction.

- **Perform May-Happen-in-Parallel Analysis** (do-mhp)
  (default value: `true`)

  Perform a May-Happen-in-Parallel analysis to assist in allocating locks.

- **Perform Local Objects Analysis** (do-tlo)
  (default value: `true`)

  Perform a Local-Objects analysis to assist in allocating locks.

- **Print Topological Graph** (print-graph)
  (default value: `false`)

  Print a topological graph of the program's transactions in the format used by the graphviz package.

- **Print Table** (print-table)
  (default value: `false`)

  Print a table of information about the program's transactions.

- **Print Debugging Info** (print-debug)
  (default value: `false`)

  Print debugging info, including every statement visited.

Rename duplicated classes (wjtp.rdc)

Rename duplicated classes when the file system is not case sensitive. If the file system is case sensitive, this phase does nothing.

Accepted phase options:

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

- **FixedClassNames** (fcn)

  Use this parameter to set some class names unchangable even they are duplicated. The fixed class name list cannot contain duplicated class names. Using '-' to split multiple class names (e.g., fcn:a.b.c-a.b.d).

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`)

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:

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

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`)

- **Reconstruct Jimple body after inlining** (rerun-jb)
  (default value: `true`)

  When a method with array parameters is inlined, its variables may need to be assigned different types than they had in the original method to produce compilable code. When this option is set, Soot re-runs the Jimple Body pack on each method body which has had another method inlined into it so that the typing algorithm can reassign the types.

- **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:

`unsafe`	Modify the visibility on code so that all inlining is permitted.
`safe`	Preserve the exact meaning of the analyzed program.
`none`	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.

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`)

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`)

Unreachable Method Tagger (wjap.umt)

Uses the call graph to determine which methods are unreachable and adds color tags so they can be highlighted in a source browser.

Accepted phase options:

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

Unreachable Fields Tagger (wjap.uft)

Uses the call graph to determine which fields are unreachable and adds color tags so they can be highlighted in a source browser.

Accepted phase options:

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

Tightest Qualifiers Tagger (wjap.tqt)

Determines which methods and fields have qualifiers that could be tightened. For example: if a field or method has the qualifier of public but is only used within the declaring class it could be private. This, this field or method is tagged with color tags so that the results can be highlighted in a source browser.

Accepted phase options:

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

Call Graph Grapher (wjap.cgg)

Creates graphical call graph.

Accepted phase options:

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

- **Show Library Methods** (show-lib-meths)
(default value: `false`)

Purity Analysis [AM] (wjap.purity)

Purity anaysis implemented by Antoine Mine and based on the paper A Combined Pointer and Purity Analysis for Java Programs by Alexandru Salcianu and Martin Rinard.

Accepted phase options:

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

- **Dump one .dot files for each method summary** (dump-summaries)
(default value: `true`)

- **Dump .dot call-graph annotated with method summaries (huge)** (dump-cg)
(default value: `false`)

- **Dump one .dot for each intra-procedural method analysis (long)** (dump-intra)
(default value: `false`)

- **Print analysis results** (print)
(default value: `true`)

- **Annotate class files** (annotate)
(default value: `true`)

- **Be (quite) verbose** (verbose)
(default value: `false`)

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`)

- **Shimple Node Elimination Optimizations** (node-elim-opt)
  (default value: `true`)

  Perform some optimizations, such as dead code elimination and local aggregation, before/after eliminating nodes.

- **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.

- **Extended SSA (SSI)** (extended)
  (default value: `false`)

  If enabled, Shimple will create extended SSA (SSI) form.

- **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.

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`)

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`)

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.

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`)

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`)

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.

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.

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:

`safe`	Safe, but only considers moving additions, subtractions and multiplications.
`medium`	Unsafe in multi-threaded programs, as it may reuse the values read from field accesses.

`unsafe` May violate Java's exception semantics, as it may move or reorder exception-throwing statements, potentially outside of try-catch 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.

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.

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`)

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`)

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 Tag Dead Code** (only-tag)
  (default value: `false`)

  Only tag dead assignment statements instead of eliminaing them.
- **Only Stack Locals** (only-stack-locals)
  (default value: `false`)

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

Null Check Eliminator (jop.nce)

Replaces statements 'if(x!=null) goto y' with 'goto y' if x is known to be non-null or with 'nop' if it is known to be null, etc. Generates dead code and is hence followed by unreachable code elimination. Disabled by default because it can be expensive on methods with many locals.

Accepted phase options:

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

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`)
- **Remove unreachable traps** (remove-unreachable-traps)
  (default value: `false`)

  Remove exception table entries when none of the protected instructions can throw the exception being caught.

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`)

Unreachable Code Eliminator 2 (jop.uce2)

Another iteration of the Unreachable Code Eliminator.

Accepted phase options:

- **Enabled** (enabled)
  (default value: `true`)
- **Remove unreachable traps** (remove-unreachable-traps)
  (default value: `false`)

  Remove exception table entries when none of the protected instructions can throw the exception being caught.

Unconditional Branch Folder 2 (jop.ubf2)

Another iteration of the Unconditional Branch Folder.

Accepted phase options:

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

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`)

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`)

Null Pointer Checker (jap.npc)

The Null Pointer Checker finds instruction which have the potential to throw NullPointerExceptions 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.

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`)

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.

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

  Add color tags to the results of the array bounds check analysis.

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[]).

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.

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.

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`)

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`)

Parameter Alias Tagger (jap.pat)

For each method with parameters of reference type, this tagger indicates the aliasing relationships between the parameters using colour tags. Parameters that may be aliased are the same colour. Parameters that may not be aliased are in different colours.

Accepted phase options:

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

Live Variables Tagger (jap.lvtagger)

Colors live variables.

Accepted phase options:

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

Reaching Defs Tagger (jap.rdtagger)

For each use of a local in a stmt creates a link to the reaching def.

Accepted phase options:

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

Cast Elimination Check Tagger (jap.che)

Indicates whether cast checks can be eliminated.

Accepted phase options:

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

Unreachable Method Transformer (jap.umt)

When the whole-program analysis determines a method to be unreachable, this transformer inserts an assertion into the method to check that it is indeed unreachable.

Accepted phase options:

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

Loop Invariant Tagger (jap.lit)

An expression whose operands are constant or have reaching definitions from outside the loop body are tagged as loop invariant.

Accepted phase options:

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

Available Expressions Tagger (jap.aet)

A each statement a set of available expressions is after the statement is added as a tag.

Accepted phase options:

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

- **Kind** (kind)
  (default value: `optimistic`)

  Possible values:
  `optimistic`
  `pessimistic`

Dominators Tagger (jap.dmt)

Provides link tags at a statement to all of the satements dominators.

Accepted phase options:

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

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`)

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.

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`)

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.

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`)

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`)

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 asm or a, or if class files are being output and the Via Grimp option has not been specified.

Accepted phase options:

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

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

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

- **sl2** (sl2)
  (default value: `false`)

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

- **sll** (sll)
  (default value: `true`)

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

- **sll2** (sll2)
  (default value: `false`)

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

Store Chain Optimizer (bb.sco)

The store chain optimizer detects chains of push/store pairs that write to the same variable and only retains the last store. It removes the unnecessary previous push/stores that are subsequently overwritten.

Accepted phase options:

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

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 Peepholes 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`)

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`)

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.

Nop Eliminator (bb.ne)

The Nop Eliminator removes nop instructions from the method.

Accepted phase options:

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

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 asm or a, or if class files are being output and the Via Grimp option has not been specified.

Accepted phase options:

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

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`)

Line Number Tag Aggregator (tag.ln)

The Line Number Tag Aggregator aggregates line number tags.

Accepted phase options:

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

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`)

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`)

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`)

Dava Body Creation (db)

The decompile (Dava) option is set using the -f dava options in Soot. Options provided by Dava are added to this dummy phase so as not to clutter the soot general arguments. -p db (option name):(value) will be used to set all required values for Dava.

Accepted phase options:

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

- **Source** (source-is-javac)
  (default value: `true`)

  check out soot.dava.toolkits.base.misc.ThrowFinder In short we want to ensure that if there are throw exception info in the class file dava uses this info.

Transformations (db.transformations)

The transformations implemented using AST Traversal and structural flow analses on Dava's AST

Accepted phase options:

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

Renamer (db.renamer)

If set, the renaming analyses implemented in Dava are applied to each method body being decompiled. The analyses use heuristics to choose potentially better names for local variables. (As of February 14th 2006, work is still under progress on these analyses (dava.toolkits.base.renamer).

Accepted phase options:

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

De-obfuscate (db.deobfuscate)

Certain analyses make sense only when the bytecode is obfuscated code. There are plans to implement such analyses and apply them on methods only if this flag is set. Dead Code elimination which includes removing code guarded by some condition which is always false or always true is one such analysis. Another suggested analysis is giving default names to classes and fields. Onfuscators love to use weird names for fields and classes and even a simple re-naming of these could be a good help to the user. Another more advanced analysis would be to check for redundant constant fields added by obfuscators and then remove uses of these constant fields from the code.

Accepted phase options:

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

Force Recompilability (db.force-recompile)

While decompiling we have to be clear what our aim is: do we want to convert bytecode to Java syntax and stay as close to the actual execution of bytecode or do we want recompilably Java source representing the bytecode. This distinction is important because some restrictions present in Java source are absent from the bytecode. Examples of this include that fact that in Java a call to a constructor or super needs to be the first statement in a constructors body. This restriction is absent from the bytecode. Similarly final fields HAVE to be initialized once and only once in either the static initializer (static fields) or all the constructors (non-static fields). Additionally the fields should be initialized on all possible execution paths. These restrictions are again absent from the bytecode. In doing a one-one conversion of bytecode to Java source then no attempt should be made to fix any of these and similar problems in the Java source. However, if the aim is to get recompilable code then these and similar issues need to be fixed. Setting the force-recompilability flag will ensure that the decompiler tries its best to produce recompilable Java source.

Accepted phase options:

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