Using Side-Effect Attributes

Anatole Le (ale44@sable.mcgill.ca)

March 18, 2006

This note explains how to use Soot annotation options to add side-effect attributes in class files and how these attributes can be used in JIT or ahead-of-time compilers.

1 Side-Effects

Side-effect analysis provides an approximation of the set of memory locations that each instruction may read or write. This analysis can optimize code by eliminating redundant loads and stores. It has been observed in the past for languages such as Modula-3, C or Java that the use of side-effect information in compilers does have a significant impact on performance.

Soot can perform static analyses for computing side-effect information with different levels of precision. The simplest, least precise side-effect analysis computed in Soot uses Class Hierarchy Analysis (CHA) for an approximation of the call graph and method summaries of fields read and written. More precise (though more expensive) side-effect analyses make use of call graph and points-to information computed by Spark. The results of these analyses can then be encoded in class files as attributes, and ahead-of-time or JIT compilers can use it to improve optimizations such as common sub-expression elimination, load elimination, dead store removal and loop-invariant code motion.

Soot also supports user-defined attributes. The process of adding new analyses and attributes is documented in "Adding attributes to class files via Soot".

2 Annotation Options for Side-Effects in Soot

2.1 Options

Soot has a command-line option `-annot-side-effect` to annotate class files with side-effect attributes. Since a side-effect analysis requires a call-graph, options whole-program mode (`-w`) and application mode (`-app`) must also be specified. These three options are required to perform Soot's simplest side-effect analysis, which we call `CHA`.

More precise side-effect analyses that make use of points-to analysis can be computed using various Spark options such as `on-fly-cg` and `field-based`. When `on-fly-cg` is true, the call graph is computed on the fly (otf) as the points-to analysis is performed. When it is false, it is computed ahead-of-time (aot) using CHA. The `field-based` option specifies whether the points-to analysis is field-based (fb) or field-sensitive (fs). A description of the different Spark options can be found in Ondrej Lhotak's M.Sc. thesis and in the Soot Phase Options document.

The figure below gives an overview of the relative precision of the variations, with precision increasing from bottom to top.

-w -app -annot-side-effect

With these options enabled, a simple side-effect analysis is computed using a CHA call graph and method field read/write summaries are built. No points-to analysis is performed to differentiate fields from different objects. Side-effect information is then annotated in class files attributes.

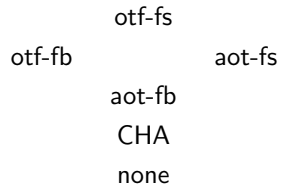

Figure 1: Relative Precision of Analysis Variations

-p cg.spark enabled -annot-side-effect
The analysis uses Spark to compute points-to analysis. By default it builds the call graph on-the-fly using points-to information and performs a field-sensitive analysis. Side-effect information is then computed using the points-to analysis, and encoded in class files.

-p cg.spark enabled,on-fly-cg:false,field-based -annot-side-effect
The analysis uses Spark to compute points-to analysis. The `on-fly-cg:false` and `field-based` options specify Spark to construct the call graph ahead-of-time using CHA, and to perform a field-based analysis. Side-effect information is then computed using the points-to analysis, and encoded in class files.

2.2 Examples

Annotate class files with Soot's simple side-effect analysis.

```
java -Xmx400m soot.Main -w -app -annot-side-effect
    spec.benchmarks._201_compress.Main
```

Annotate class files with a side-effect analysis using Spark. The call graph is built ahead-of-time (aot) and the points-to analysis is field-based (fb).

```
java -Xmx400m soot.Main -w -app -p cg.spark enabled,on-fly-cg:false,field-based
    -annot-side-effect spec.benchmarks._201_compress.Main
```

Annotate class files with a side-effect analysis using Spark. The call graph is built ahead-of-time (aot) and the points-to analysis is field-sensitive (fs).

```
java -Xmx400m soot.Main -w -app -p cg.spark enabled,on-fly-cg:false
   -annot-side-effect spec.benchmarks._201_compress.Main
```

Annotate class files with a side-effect analysis using Spark. The call graph is built on-the-fly (otf) and the analysis is field-based (fb).

```
java -Xmx400m soot.Main -w -app -p cg.spark enabled,field-based
   -annot-side-effect spec.benchmarks._201_compress.Main
```

Annotate class files with a side-effect analysis using Spark. The call graph is built on-the-fly (otf) and the analysis is field-sensitive (fs).

```
java -Xmx400m soot.Main -w -app -p cg.spark enabled -annot-side-effect
    spec.benchmarks._201_compress.Main
```

```
SideEffectAttribute:

  RecordCount
  (2 bytes)

  BytecodeOffset   ReadSet   WriteSet   ExtraByte
  (2 bytes)        (2 bytes) (2 bytes)  (1 byte)

  BytecodeOffset   ReadSet   WriteSet   ExtraByte
  (2 bytes)        (2 bytes) (2 bytes)  (1 byte)

  ...


DependenceGraph:

  Set         Set
  (2 bytes)   (2 bytes)

  Set         Set
  (2 bytes)   (2 bytes)

  ...
```

Figure 2: Side-Effect Attribute Format

3 Side-Effect Attribute Format

Figure 2 shows the format of side-effect attributes in class files. Each method is associated with two attributes. The first one, `SideEffectAttribute`, maps each bytecode that has side-effects to a read and write set. The extra byte contains a bit that indicates whether a bytecode explicitly or implicitly invokes a native method, and other bits for future use. The second attribute, `DependenceGraph`, denotes which read and write sets have dependences.

4 Example of Using Side-Effect Attributes

In Figure 3, we show sample code and the resulting encoding of side-effect information. Method `foo` contains instructions that, once compiled to bytecode, would include two *putfield*, two *invokevirtual*, and one *getfield* bytecodes at offset 2, 7, 11, 16 and 20. In the side-effect attribute, there is no entry for `a.nothing()` (offset 11) since this call has no side-effect. In method `foo`, it is clear from the code that there is a write-write dependence between statements `a.g = 4` and `a.setG( 5 )`. This dependence is given in the attributes by mapping the write set of `a.g = 4` to 1 and `a.setG( 5 )` to 2 in the SideEffectAttribute at offsets 7 and 16 respectively, and specifying that sets 1 and 2 have a dependence in the DependenceGraph attribute. Now, since there is no dependence between statements `a.f = 3` and subsequent statements, the load in statement `int i = a.f` can be eliminated by a copy of its previous assigned value (i.e. 3).

Other information

Our technical report contains detailed explanations of the side-effect variations, and how side-effect attributes can be used in the presence of method inlining.

```
class A {
  int f;
  int g;
  void setF( int n ) { this.f = n; }
  void setG( int n ) { this.g = n; }
  void nothing() {}

  void foo( A a ) {
      a.f = 3;      // putfield      at offset  2
      a.g = 4;      // putfield      at offset  7
      a.nothing();  // invokevirtual at offset 11
      a.setG( 5 );  // invokevirtual at offset 16
      int i = a.f;  // invokevirtual at offset 20
  }

  public static void main(String[] args) {
      foo( new A() );
  }
}

SideEffectAttribute (method foo):
  RecordCount: 4
  Offset  ReadSet  WriteSet
       2       -1         0
       7       -1         1
      16       -1         2
      20        0        -1

DependenceGraph (method foo):
  Set  Set
    1    2
```

Figure 3: Example of Side-Effect Attribute

Change log

- Feb 27, 2005: Initial version.