A Brief Overview of Shimple

Navindra Umanee (navindra@cs.mcgill.ca)

June 6, 2003

This document briefly describes Shimple, an SSA variant of Soot's Jimple internal representation. It assumes general knowledge of Soot, Jimple and SSA form. You may wish to jump directly to the walk-through section for a demonstration of why you might be interested in using Shimple either by implementing analyses or by applying existing optimizations.

1 Why Shimple?

Static Single Assignment (SSA) form guarantees a single static definition point for every variable used in a program, thereby significantly simplifying as well as enabling certain analyses.

Shimple provides you with an IR in SSA form that is almost entirely identical to Jimple except for the introduction of Phi nodes. The idea is that Shimple can be treated almost identically to Jimple with the added benefits of SSA.

For example, the additional variable splitting due to SSA form may turn a previously flow-insensitive analysis into a flow-sensitive one with little or no additional work.

2 Hacking Overview

The public API of Shimple is fully described in the Soot API documentation. In particular, in the soot.shimple package, the Shimple class provides the Shimple grammar constructors and various utility functions, the ShimpleBody class describes Shimple bodies and PhiExpr provides the interface to Phi expressions.

Use/Definition and Definition/Use chains for Shimple bodies can be obtained with ShimpleLocalDefs or ShimpleLocalUses available in package soot.shimple.toolkits.scalar.

Currently, the only Shimple optimization available is the powerful, control flow aware, SConstantPropagatorAndFolder. The latter as well as some simple Shimple analyses are available in package soot.shimple.toolkits.scalar. Please consult the Soot source for details.

3 Usage Options

For a full description of the options and phases pertaining to Shimple, please consult the primary Soot option and phase documentation.

4 Command Line Walk Through

For fun, you may wish to run Shimple from the command-line and study its output. Consider the following (compiled) Java code of the `ShimpleTest` class:

```
public int test()
{
    int x = 100;
```

```
    while(doIt){
        if(x < 200)
            x = 100;
        else
            x = 200;
    }

    return x;
}
```

4.1 Producing Jimple

If you produce Jimple with 'soot.Main -f jimple ShimpleTest' you obtain the following code for the
test() method:

```
        i0 = 100;
        goto label2;

    label0:
        if i0 >= 200 goto label1;

        i0 = 100;
        goto label2;

    label1:
        i0 = 200;

    label2:
        if $z0 != 0 goto label0;

        return i0;
```

4.2 Producing Shimple

To produce Shimple instead, use 'soot.Main -f shimple ShimpleTest':

```
        i0 = 100;
(0)     goto label2;

    label0:
        if i0_1 >= 200 goto label1;

        i0_2 = 100;
(1)     goto label2;

    label1:
(2)     i0_3 = 200;

    label2:
        i0_1 = Phi(i0 #0, i0_3 #2, i0_2 #1);
        if $z0 != 0 goto label0;
```

```
        return i0_1;
```

The difference between the Jimple and Shimple output is that the latter guarantees unique local definition points in the program (for scalars). Notice here that the variable `i0` has been split into the four variables `i0`, `i0_1`, `i0_2` and `i0_3`, each with a unique definition point.

We have also introduced a Phi node. You can read 'i0_1 = Phi(i0 #0, i0_3 #2, i0_2 #1)' as saying that `i0_1` will be assigned value `i0` if control flow comes from unit #0 *or* it will be assigned `i0_3` if control flow comes from unit #2 *or...* and so on.

If you have a prejudice against variable names with underscores, you may use 'soot.Main -f shimple -p shimple standard-local-names ShimpleTest' instead so that Shimple applies the Local Name Standardizer each time new locals are introduced.

Feel free to skip the following digression and move on to the next subsection.

4.2.1   A Digression on Shimple Pointers

Because Soot represents the body of a method internally as a Unit chain, we need to store explicit pointers (such as `#0` and `#1` in the above example) to keep track of the control flow predecessors of the Phi statements.

Shimple's internal implementation of PatchingChain attempts to move and maintain these pointers in a manner that will be as transparent as possible to the user. For example, in the simplest case, if a statement is appended to block:

```
      label0:
(1)       i0_1 = 1000;
```

to obtain:

```
      label0:
          i0_1 = 1000;
(1)       new_stmt;
```

Shimple will automatically move the `#1` pointer down to the new statement since it is in the same basic block.

The intent is to provide maximum flexibility for code motion optimizations as well as other transformations. In this case, `i0_1 = 1000` is free to move up or down the Unit chain as long as the new location dominates the original CFG block it was in.

4.3   Producing Jimple, Again

Since we eventually have to get rid of those pesky Phi nodes, you may wish to see what the code looks like after going from Jimple to Shimple and back again to Jimple. Do this with 'java soot.Main -f jimple --via-shimple ShimpleTest':

```
        i0_1 = 100;
        goto label2;

    label0:
        if i0_1 >= 200 goto label1;

        i0_1 = 100;
        goto label2;

    label1:
```

```
    i0_1 = 200;

label2:
    if $z0 != 0 goto label0;

    return i0_1;
```

Happily, in this case, the Jimple produced looks exactly like the original Jimple code. As usual you may specify '-p shimple standard-local-names' if the underscores hurt your eyes; they are otherwise quite harmless.

To understand what's really going on when Shimple eliminates Phi nodes, you can tell it to eliminate them naively with 'soot.Main -f jimple --via-shimple -p shimple phi-elim-opt:none ShimpleTest':

```
    i0 = 100;
    i0_1 = i0;
    goto label2;

label0:
    if i0_1 >= 200 goto label1;

    i0_2 = 100;
    i0_1 = i0_2;
    goto label2;

label1:
    i0_3 = 200;
    i0_1 = i0_3;

label2:
    if $z0 != 0 goto label0;

    return i0_1;
```

Now you can see that all Shimple did was to replace the Phi node with equivalent copy statements.

4.4   Applying Shimple Optimizations

So, what good is Shimple?

If you were paying attention, you may have noticed that despite the control flow and different assignments to x, x is in fact a constant 100 and is only ever used by a single return statement. In other words, all the computations are quite useless and need to be optimized away.

Let's try to apply Jimple's Constant Propagator and Folder. In fact, to be fair, let's try all the available Jimple optimizations activated with 'soot.Main -f jimple -O ShimpleTest':

```
    i0 = 100;
    goto label2;

label0:
    if i0 >= 200 goto label1;

    i0 = 100;
    goto label2;
```

```
    label1:
        i0 = 200;

    label2:
        if $z0 != 0 goto label0;

        return i0;
```

As you can see in this case, the Jimple optimizations failed to deduce that x is a constant. Jimple may be forgiven for this since reasoning about control flow is not always an easy task to automate. Shimple, on the other hand, encodes control flow information explicitly in the Phi nodes thereby allowing optimizations to make use of the information.

Currently, the only optimization we have implemented specifically for Shimple is a version of the powerful constant propagation algorithm sketched by the Cytron et el. The implementation can reason about control flow and can currently handle the conditional branching statements `IfStmt`, `TableSwitchStmt` and `LookupSwitchStmt`.

Let's apply it with '`soot.Main -f jimple --via-shimple -O ShimpleTest`':

```
    label0:
        if $z0 != 0 goto label0;

        return 100;
```

Et voila, Shimple optimized out the x variable completely. What happened is that the optimization propagated reachable definitions to the Phi node and then noticed that the Phi node was useless (because it made a selection from identical values) and therefore trimmed it out. In the process, dead code was exposed which was ultimately eliminated.

To understand what is really going on, you can look at the output from '`soot.Main -f shimple -p sop on`', '`soot.Main -f shimple -p sop on -p sop.cpf prune-cfg:false`' and '`soot.Main -f jimple --via-shimple -p shimple phi-elim-opt:none -p sop on -p sop.cpf prune-cfg:false`' on this and other examples:

```
public int test2()
{
    int i = 1000;
    int j = 1000;

    while(doIt){
        if(i == j){
            if(anotherIt)
                i = j;
            else
                j = i;
        }
        else{
            i = 5;
            j = 6;
        }
    }

    return i + j;
}
```

5

5 Thanks and Credits

Thanks and credits go alphabetically to Laurie Hendren, John Jorgensen, Patrick Lam and Ondrej Lhotak for helping with the design of Shimple and general implementation issues. I am most grateful for their help and advice.

6 Future Work

Much more work on Shimple is planned as the project is likely to morph into a Master's thesis. Some thoughts currently include investigating the various scalar SSA variants as well as heap, array and possibly concurrent forms of SSA. The Shimple architecture and implementation will therefore evolve quite a bit internally, but as far as possible we will try to maintain backwards-compatibility for the public interfaces.

Suggestions, improvements and bug reports/fixes welcome! Please send these either to the Soot mailing list at soot-list@sable.mcgill.ca, or directly to myself at navindra@cs.mcgill.ca.

6.1 Partial To-Do List

- Implement more SSA-based analyses.

- Add timers and profiling code.

- Make internal analyses more useful and generic for external use.

- Adopt an Strategy-type pattern for SSA builder modules, etc.

- Implement a Shimple parser.

- Provide an interface for CFG manipulations that intelligently updates Phi nodes.

- Implement a Control Dependence Graph? Any interest in that?

6.2 Known Issues

A vague description of a couple of known issues follows. You may ignore this section completely since regular usage of Shimple should not be affected in general.

6.2.1 Issue 1

One issue is related to Phi nodes inserted at the beginning of try blocks which are subsequently used by Phi nodes in the corresponding handler block. Fortunately, although the code produced looks strange it is not incorrect. Example:

```
    label1:
        i0_2 = Phi(i0 #0, i0_1 #1);
(2)     i1 = 4 / 0;
(3)     i0_3 = i0_2 / 0;
        i2 = i0_3;

     label2:
        goto label4;

     label3:
        $r0 := @caughtexception;
        i0_4 = Phi(i0_1 #1, i0 #0, i0_2 #2, i0_3 #3);
```

```
        catch java.lang.Exception from label1 to label2 with label3;
```

The `#2` pointer of the second Phi node really should be pointing directly at the first Phi node instead of at the statement following it (since the latter may throw an exception and branch to the handler block). Fortunately, in these cases the second Phi node will always be pointing directly at the predecessors of the first Phi node as well (`#0` and `#1` in this example), rendering the matter moot. This glitch will be eliminated in a future release.

6.2.2   Issue 2

Another issue is related to the Shimple patching algorithm. In the rare case that control flow falls through from an if statement to a try block and the if statement has a pointer to it:

```
        value = 1000;
(1)     if (whatever) goto label100;

label1:
        first_trap_statement;
        ...
        goto label100;
label2:
        $r0 := @caughtexception;
        i0 = Phi(value #1, ...);
```

In the above, it may be desirable to move the `#1` pointer down if a Unit happens to be inserted after the if statement. Although Shimple is smart enough to do this in most known cases, it currently misses the one case where control flows from an if statement in a non-exceptional context to an exceptional context.

This is not a big problem for most people unless an exotic code motion algorithm (currently non-existent in Soot) attempts to move the definition of `value` below the if statement for some reason.

History

- June 6, 2003: Initial version.

- June 17, 2003: Updated for Soot 2.0.1.