E. MIPS and the SPIM Simulator

E.1 Introduction

Besides being able to compile *j--* source programs to JVM bytecode, the *j--* compiler can also produce native code for the MIPS architecture. The compiler does not actually produce binary code that can be run on such a machine, but it produces human-readable assembly code that can be executed by a program that simulates a MIPS machine. The advantage of using a simulator is that one can easily debug the translated program, and does not need access to an actual MIPS machine in order to run the program. The simulator that we will use is called SPIM (MIPS spelled backwards). Chapter 7 describes the MIPS architecture and the SPIM simulator in some detail. In this appendix, we describe how one can obtain and run SPIM, compile *j--* programs to SPIM code, and extend the JVM to SPIM translator to convert more JVM instructions to SPIM code. The MIPS (and so SPIM) instruction set is described very nicely in (Larus, 2009).

E.2 Obtaining and Running SPIM

SPIM implements both a command-line and a graphical user interface (GUI). The GUI for SPIM, called QtSpim, is developed using the Qt UI Framework[1]. Qt being cross-platform, QtSpim will run on Windows, Linux, and Mac OS X.

One can either download the compiled version of SPIM for Windows, Mac OS X, or Debian-based (including Ubuntu) Linux from:

`http://sourceforge.net/projects/spimsimulator/files/`

or build SPIM from source files obtained from:

`http://spimsimulator.svn.sourceforge.net/viewvc/spimsimulator/`

The README file in the source distribution documents the installation steps.

E.3 Compiling j-- Programs to SPIM Code

As we have seen before, the *j--* compiler supports the following command-line syntax:

```
Usage: j-- <options> <source file>
where possible options include:
  -t Only tokenize input and print tokens to STDOUT
```

[1] A cross-platform application and UI framework with APIs for C++ programming and Qt Quick for rapid UI creation.

```
  -p Only parse input and print AST to STDOUT
  -pa Only parse and pre-analyze input and print AST to STDOUT
  -a Only parse, pre-analyze, and analyze input and print AST to
STDOUT
  -s <naive|linear|graph> Generate SPIM code
  -r <num> Max. physical registers (4-18) available for
allocation; default = 8
  -d <dir> Specify where to place output files; default = .
```

In order to compile the *j--* source program to SPIM code, one must specify the **`-s`** and the optional **`-r`** switches along with appropriate arguments. The **`-s`** switch tells the compiler that we want SPIM code for output and the associated argument specifies the register allocation mechanism that must be used; the options are: "naive" for a naïve round robin allocation, "linear" for allocation using the linear scan algorithm, or "graph"[2] for allocation using graph coloring algorithm. See chapter 8 for a detailed description of these register allocation algorithms. The optional **`-r`** switch forces the compiler to allocate only up to a certain number of physical registers, specified as argument; the argument can take values between 4 and 18, inclusive. If this switch is not specified, then a default value of 8 is used. Registers are allocated starting at $t0[3].

For example, here is the command line syntax for compiling the **`Factorial`** program to SPIM code allocating up to 18 (the maximum) physical registers using the naïve allocation procedure:

```
> $j/bin/j-- -s naive -r 18 $j/tests/spim/Factorial.java
```

The compiler first translates the source program to JVM bytecode in memory, which in turn is translated into SPIM code. The compiler prints to **`STDOUT`** the details of the translation process for each method, such as, the basic block structure with JVM instructions represented as tuples, the high-level (HIR) instructions in each basic block, the low-level (LIR) instructions for each basic block before allocation of physical registers to virtual registers, the intervals and associated ranges for each register, and the LIR instructions for each basic block after the physical registers have been allocated and spills have been handled where needed. In addition, the compiler also produces a target .s file containing the SPIM code (**`Factorial.s`** in above example) in the folder in which the command is run. One can specify an alternate folder name for the target .s file using the **`-d`** command-line switch.

Once the .s file has been produced, there are several ways to execute it using the SPIM simulator. The simplest way is to run the following command:

---

[2] Register allocation using graph coloring is not yet implemented in the compiler. It is left an exercise. See chapter 8.

[3] That is a total of up to18 registers ($t0, $t1, …, $t9, $s0, …, $s7) available for allocation.

```
> spim -file <input file>
```

where **spim** is the command-line version of the SPIM simulator. The **-file** argument specifies the name of the file with a .s extension that contains the SPIM code to execute. For example, we can execute the **Factorial.s** file from above as:

```
> spim -file Factorial.s
```

which produces the following output:

```
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
5040
5040
```

Yet another way of running the .s file is to launch the **spim** command-line interpreter, and at the **(spim)** prompt, use **spim** commands **load** and **run** to load and execute a .s file. The **quit()** command exits the **spim** interpreter. For example, **Factorial.s** can be executed as follows:

```
> spim
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
(spim) load "Factorial.s"
(spim) run
5040
5040
(spim) quit()
>
```

For a complete list of all the **spim** commands that are supported by the command-line interpreter, execute the command **help** at the **(spim)** prompt.

Finally, one can load a .s file into the QtSpim user interface and click the run button to execute the program. QtSpim offers a convenient interface for inspecting values in registers and memory, and also facilitates easy debugging of programs by stepping through the instructions, one at a time.

E.4 Extending the JVM to SPIM Translator

The *j--* compiler translates only a limited number of the JVM instructions to SPIM code, just enough to compile and run the four *j--* programs (**`Factorial`**, **`GCD`**, **`Fibonacci`**, and **`Formals`**) under **`$j/tests/spim`**. Nevertheless, it provides sufficient machinery with which one could build on to translate more instructions to SPIM code.

Unlike the *j--* programs that are compiled for the JVM, the ones that target SPIM cannot import any Java libraries. Any low-level library support for SPIM must be made available in the form of SPIM runtime files; a .s file and a corresponding Java wrapper for each library. For example, if *j--* programs that target SPIM need to perform input/output (IO) operations, it must import **`spim.SPIM`** which is a Java wrapper for **`SPIM.s`** file, the runtime library for IO operations in SPIM. These two files can be found under **`$j/src/spim`**. The Java wrapper only needs to provide the function headers and not the implementation, since it is only needed to make the compiler happy. The actual implementation of the functions resides in the .s file. The compiler copies the runtime .s files verbatim to the end of compiled .s program. This saves us the trouble of loading the runtime files separately into SPIM before loading the compiled .s program.

If one needs to extend the SPIM runtime to, say, support objects, strings, and arrays, then for each one of them, a .s file and a Java wrapper must be implemented under **`$j/src/spim`**, and the following code in **`Nemitter.write()`** must be updated to copy the new runtime libraries to the end of the compiled *j--* program.

```
// Emit SPIM runtime code; just SPIM.s for now.
String[] libs = { "SPIM.s" };
out.printf("# SPIM Runtime\n\n");
for (String lib : libs) {
    ...
}
```

See Chapter 7 for hints on providing run-time representational support for objects, strings, and arrays.

Converting additional JVM instructions, besides the ones that already are, into SPIM code, involves the following steps:

- Adding/modifying the HIR representation: This is done in **`$j/src/jminusminus/NHIRInstruction.java`**. Each class that represents an HIR instruction inherits from the base class **`NHIRInstruction`**. The additions will be to the constructor of the HIR instruction; the inherited **`toLir()`** method that converts and returns the low-level (LIR) representation; and the inherited **`toString()`** method that returns a string representation for the HIR instruction.

- Adding/modifying the LIR representation: This is done in **`$j/src/jminusminus/NLIRInstruction.java`**. Each class representing an LIR instruction inherits from the base class **`NLIRInstruction`**. The additions will be to the constructor of the LIR instruction; the inherited **`allocatePhysicalRegisters()`** method that replaces references to virtual registers in the LIR instruction with references to physical registers and generates LIR spill instructions where needed; the **`toSpim()`** method that emits SPIM instructions to an output stream (a .s file); and the **`toString()`** method that returns a string representation for the LIR instruction.

Both **`NHIRInstruction`** and **`NLIRInstruction`** have plenty of code in place that can serve as a guideline in implementing the above steps for the new JVM instructions being translated to SPIM code.

Further Reading

The website (Larus, 2000-2010) for the SPIM simulator provides a wealth of information about the simulator, especially on how to obtain, compile, install, and run the simulator.

For a detailed description of the SPIM simulator, its memory usage, procedure call convention, exceptions and interrupts, input and output, and the MIPS assembly language, see (Larus, 2009).