# Automating Instruction Selector Generation in Jikes RVM

Chujiao Ma, Adam Fidel, Tim Richards, J. Eliot B. Moss

August 26, 2009

**Abstract**

As the compiler evolve with the modern technology, general tools to easily develop compiler architecture becomes a necessity, especially for architectural and IR extensions. While there are existing general and framework-independent tools for compiler front-end, such tools do not exist for back end. Such tools for back-end generation is difficult because it will need to work for a wide variety of intermediate representations (IRs) and targets.

For this, we propose Gist, a universal program that takes in a compiler IR and a machine instruction set architecture and generate general instruction selection patterns that can then be adapted to any compiler framework. More specifically, the focus of this paper is on generation instruction selectors for Jikes RVM baseline compiler and PowerPC architecture.

## 1 Introduction

Innovation in micro-architecture and experiments in novel extensions require tools that allow compiler writers to easily design different compiler components. While numerous tools exists for developing compiler front-end functions such as parsing and syntax tree construction, there are limited number of tools for dealing with back-end tasks. The main component of the compiler back-end is the instruction selector, which is responsible for selecting what instruction to use, such as add, load, store and such. Currently the instruction selector in various compilers must be handwritten, involving thousands of instructions and prone to human error. While the selector can be automated, there are no easy way to automate instruction selectors that are compiler framework independent. This, is the problem we are trying to solve with Gist.

Gist is a tool intended for automatic generation of compiler selectors that can easily be adapted to any compilers. It is universal because it automatically generates instruction selection patterns for any combination of target instruction set and compiler IR. It is also easy to use because the compiler writer only have to write an adapter, which is magnitudes simpler than an instruction selector, in order to use GIST. It takes in IR description and target description, both written in the same description language, then outputs a generic description of instruction selection, which is tailored to a specific compiler, Jikes RVM baseline compiler in this case, through an adapter.

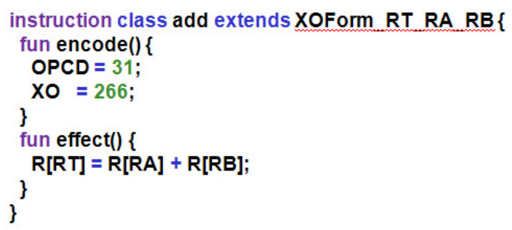
## 2 Background

There has been considerable progress made in automating the construction of compiler front-end components. As for tools to automate construction of compiler back-end, there are a few that made the effort to be compiler and language independent. One of the tools is the bottom-up rewrite system techniques (BURS), generate pattern in a given expression tree using tree rewriting. While the process is independent of the IR and ISA, the pattern specifications are dependent on both, which makes it difficult to be applied to other compiler frameworks. Fraser ([4]) uses a rule based production system to generate code generators automatically but it is compiler dependent. Gist, on the other hand, is language and compiler independent since both the IR and ISA descriptions are written in a framework neutral language, CISL. CISL is specifically designed for the automatic generation of simulators and compiler back-end. It is a class-based language with a Java style syntax aimed at simplicity and extensibility.

The compiler we are currently focusing on is Jikes RVM baseline compiler. Jikes Research Virtual Machine is open source and used for experimenting with virtual machine technologies, therefore an appropriate option for testing automatic instruction selection generation. The compiler it uses is called the baseline compiler, which is a non-optimizing just-in-time compiler. Next, we will show how Gist works, run and test it in the baseline compiler.

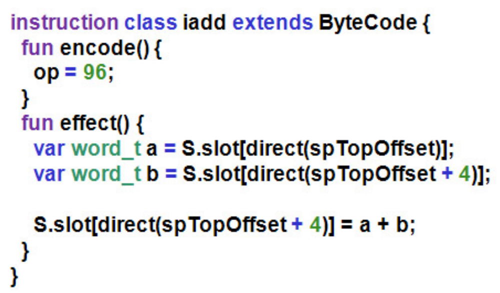
### 2.1 How Gist Works

Gist takes in compiler IR instructions and the target architecture descriptions, both of which are written in the framework neutral language, CISL. Figure 1 shows an example of the target description, iadd from PowerPC, and Figure 2 shows iadd from baseline compiler, both described in CISL.



```
instruction class add extends XOForm_RT_RA_RB {
  fun encode() {
    OPCD = 31;
    XO   = 266;
  }
  fun effect() {
    R[RT] = R[RA] + R[RB];
  }
}
```

Figure 1: Target description for GIST (PowerPC).

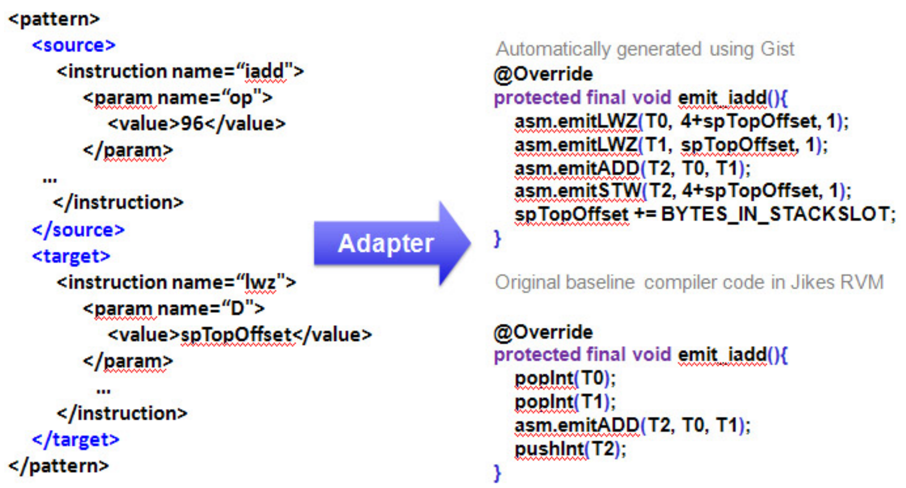


```
instruction class iadd extends ByteCode {
 fun encode() {
   op = 96;
 }
 fun effect() {
   var word_t a = S.slot[direct(spTopOffset)];
   var word_t b = S.slot[direct(spTopOffset + 4)];

   S.slot[direct(spTopOffset + 4)] = a + b;
 }
}
```

Figure 2: Compiler description for GIST (Jikes RVM Baseline compiler).

After covering most of the baseline compiler and PowerPC descriptions, we input both files into Gist. Gist uses a heuristic search to produce multiple matches for each baseline compiler and PowerPC description. For each match, the sequence with the fewest instruction, assumed to be the most efficient, are selected to be part of the final output. For descriptions with no matches, no output is generated. After Gist finished matching the descriptions, it produces an XML file with generated outputs for the description. the XML is a simple and generic format that is compiler and language independent. As shown in Figure 3, the machine descriptions in XML format are processed by compiler-specific adapters to produce instruction selector components that can be plugged in to an existing framework of the compiler.



```
<pattern>
  <source>
    <instruction name="iadd">
      <param name="op">
        <value>96</value>
      </param>
   ...
    </instruction>
  </source>
  <target>
    <instruction name="lwz">
      <param name="D">
        <value>spTopOffset</value>
      </param>
        ...
    </instruction>
  </target>
</pattern>
```

Automatically generated using Gist

```
@Override
protected final void emit_iadd(){
   asm.emitLWZ(T0, 4+spTopOffset, 1);
   asm.emitLWZ(T1, spTopOffset, 1);
   asm.emitADD(T2, T0, T1);
   asm.emitSTW(T2, 4+spTopOffset, 1);
   spTopOffset += BYTES_IN_STACKSLOT;
}
```

Original baseline compiler code in Jikes RVM

```
@Override
protected final void emit_iadd(){
   popInt(T0);
   popInt(T1);
   asm.emitADD(T2, T0, T1);
   pushInt(T2);
}
```

Figure 3: The adapter that takes the generic GIST outputs and formats it for a specific compiler framework.

To use Gist on a specific compiler, all that the compiler programmer have to do is to write an adapter that essentially parse the XML file into the format of the instruction selector for that specific compiler, which is much less intensive than writing thousands lines of code. In this case, we wrote an adapter parses these patterns and generates code to be used in Jikes RVMs baseline compiler.

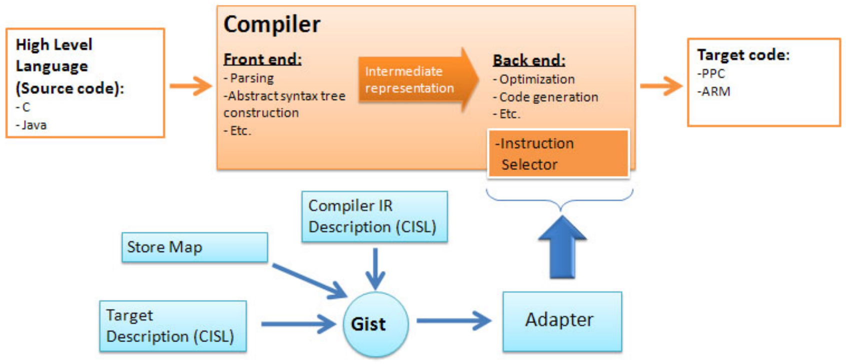


Figure 4: Diagram of the gist process and how it fits in the compiler.

Figure 4 illustrates how Gist fits in with respect to the compiler. Gist is independent of the compiler framework and automatically generates machine descriptions that are language and compiler independent. The compiler writer writes an adapter that tailors the results into the format required by the compiler framework. Then the output of the adapter is plugged into the compiler, replacing the original handwritten machine descriptions.

## 2.2 Tests and Results

The goal of Gist is to replace the instruction selector component of the compiler. As such, we will show the effectiveness of the generated instruction selector by comparing its results when plugged into existing framework with results of the original instruction selector. Currently Gist is able to cover 69 % of the bytecodes in the baseline compiler. While this does not seem to be much, the bytecode covered most of the commonly used descriptions. Of the remaining uncovered bytecodes, most of them rely on runtime information, which cannot be statistically matched. To demonstrate that the instruction selector generated by Gist does not hinder the efficiency of the compiler,we decided to test the runtime of the original instruction selector, and the instruction selector with components replaced by Gist.

Since Jikes RVM is a JIT compiler, we used a standard Java runtime benchmarks package called DaCapo. To confirm the validity of the results, we did ten iterations of all benchmark tests for the original implementation and the Gist generated instruction selector. The runtime slowdown of the generated vs. original, averaged over ten trials, is shown below in Figure 5.

While the generated instruction selector performed better in some tests as shown by the graph, it is small enough to be statistically insignificant and the

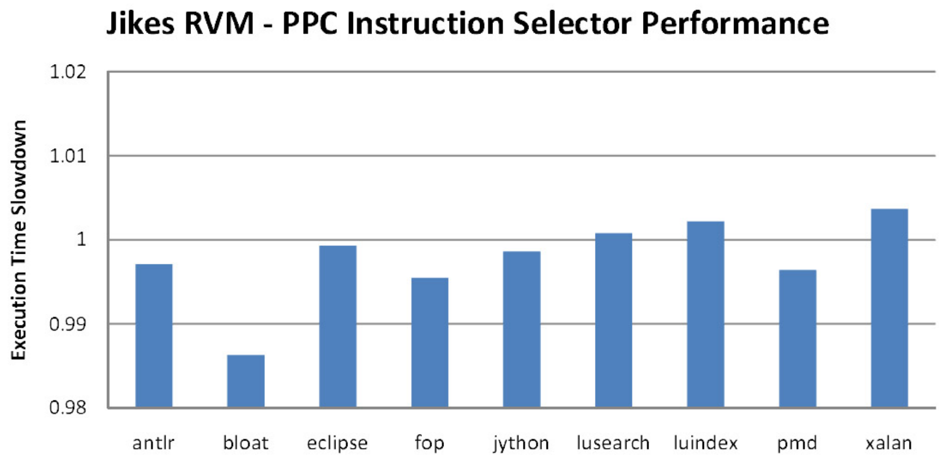


Figure 5: Jikes RVM-PPC slowdown of the DeCapo benchmarks for generated vs. original (lower = better).

generated instruction selector did not cover and replace all descriptions.

## 2.3 Conclusion and Future Work

This demonstrated that Gist perform just as well as the original selectors. Gist does not hinder the performance of the compiler while being easy to use, bypassing the tediousness and human error-prone problem of the original selector.

Gist is more universal than prior works and outputs components that match the performance of those in existing compilers. In addition to expanding the bytecode currently covered for the Jikes RVM and PowerPC instruction selector, we would also like to extend Gist to other compilers and ISAs, which currently includes LCC, MIPS, ARM and such.

## References

[1] J. Eliot B. Moss, Trek Palmer, Timothy Richards, Edward K. Walters II, Charles C. Weems. *CISL: A Class-based Machine Description Language for Co-generation of Compilers and Simulators.* International Journal of Parallel Programming, 2005.

[2] J. Eliot B. Moss, Trek Palmer, Timothy Richards, Edward K. Walters II, Charles C. Weems. *CMDL: A Class-based Machine Description Language for Co-generation of Compilers and Simulators.* IPDPS Workshop on Parallel and Distributed Computing Issues in Next Generation Software, 2004.

[3] J. Eliot B. Moss, Charles C. Weems, Timothy Richards. *The CoGenT Project: Co-Generating Compilers and Simulators For Dynamically Compiled Languages.* IPDPS Workshop on Parallel and Distributed Computing Issues in Next Generation Software, 2003.

[4] C. W. Fraser. Automatic Generation of Code Generators. PhD thesis, Yale University, 1977.