Automated Instruction Selector Generation with Jikes RVM

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

Dept. of Computer Science, University of Massachusetts Amherst
{afidel,chujiao,richards,moss,}@cs.umass.edu

Abstract

As novel and major extensions are added to machine micro-architectures, compiler intermediate representations and virtual machines, the need for automatic generation of back-end compiler components to support these extensions grows dramatically. Tools to assist in developing compiler front-ends currently exist, yet there has been no focus on sufficiently generic and framework-independent tools for compiler back-ends. Gist, a tool created by researchers at the University of Massachusetts Amherst, attempts to bridge this gap; specifically, it generates an instruction selector given a compiler intermediate representation (IR) and a target machine instruction set architecture (ISA). Other research efforts fixed one side of the problem, either the source IR or the target ISA, which only presents a non-generic solution and reduces the problem's difficulty tremendously. By introducing a universal mechanism to describe a machine instruction set architecture and a compiler's IR, Gist is general enough to generate instruction selection patterns for any combination of ISA and IR that can be adequately described by this mechanism. The focus of this research is on generating instruction selectors specifically for the Jikes RVM baseline compiler with respect to the PowerPC architecture.

1 Introduction

Considerable research effort has been poured into automatic generation of compiler components, yet much of the focus has been on the front-end work such as parsing and scanning. One of the main phases of a compiler back-end is the instruction selector. In essence, for each compiler intermediate representation (IR) or virtual machine (VM) operation, the instruction selector emits a sequence of target instructions that attempt to replicate the operation semantically. Gist is a tool that generates instruction selection patterns given a description of the target machine, either the IR or VM specification and a mapping of the memories between the two. These descriptions are provided using a sufficiently expressive machine description language CISL (Cogent Instruction Set Language). Despite its name, CISL is used to describe both the target machine instruction set architecture (ISA) and the source IR. A memory mapping is then needed to specify, for example, that a certain target register is to be used to store a stack pointer, or that certain addressing modes from the source correspond in a particular manner on the target.

In the spirit of generic and framework-independent instruction selection generation, Gist generates instruction selection patterns in strict XML form. These can be adapted to certain compiler frameworks with negligible amount of effort. Our contention is that allowing the combination of Gist and a compiler-specific adapter to create an instruction selector provides a quick and easy way to construct this compiler component that is easily provable to be correct and less prone to human error.

The need for automated compiler component generation is apparent: advances in current or completely new micro-architectures require tremendous modification of existing compilers to support these changes. Similarly, novel programming language features or entirely new languages that target a virtual machine will also require reworking of the compiler back-end and the instruction selector specifically. Innovative efforts dealing with such changes might be prohibited due to the high cost of reworking the compiler. Automating instruction selection with Gist promotes experimentation and research for this type of innovation.

Jikes RVM [1] is a Java virtual machine used primarily for research purposes. It is the perfect candidate for automatic instruction selection generation due to its rich runtime semantics and its open source foundation. We will be working with what is called the Baseline compiler in Jikes which is a non-optimizing just-in-time compiler. The rest of the paper will describe our attempt to generate a functional instruction selector for Jikes RVM targeting the PowerPC ISA using Gist.

2 Related Work

As mentioned previously, most of the success enjoyed by automatic compiler component generation dealt with front-end components. An automatic way to generate a compiler parser was introduced in 1979 with `yacc` [6], which built its parsers using grammars similar to Backus-Naur Form. ANother Tool for Language Recognition (ANTLR) [8] is another parser generator similar to `yacc` that uses LL(*) parsing.

Dealing with the back-end has proven to be a much greater challenge. Cattell [3] proposed a way to automatically generate code generators using a set of axioms to rewrite the compiler's IR to a syntax tree representing the semantics on the target ISA. Because the description language is essentially the compiler's IR, Cattell's work was heavily dependent on the compiler framework. Gist differs in its commitment to generality; it relies on machine descriptions of both the compiler IR and target ISA that are independent of each other. Sharing a similar pitfall with regards to generic descriptions, Ceng et al [4] formalized a method to retarget the C compiler using instruction semantic models. Although not tied to a specific framework, Ceng's research relied on C specifically, and it is unclear whether their approach will hold for other languages with rich runtime semantics such as Java.

3 Background

When dealing with normal compiler writing, the back-end components are usually built by hand, allowing more room for human-induced errors to be brought into play. Gist automates the construction of a compiler's instruction selector given adequate description of a compiler IR and a target ISA in the description language CISL.

During the Gist process, there is a sequence of clearly defined steps to get from the concept to a working instruction selector that can be plugged into an existing compiler framework. First, identify the compiler IR or virtual machine specification from which to find matches. This will be known as the source. Also, identify the machine instruction set architecture that will be the end target of instruction emits. Our research focused on Jikes RVM, which uses the Java virtual machine bytecode specification, for the source and the PowerPC micro-architecture for the target.

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

Figure 1: CISL description of the `iadd` bytecode for the JVM specification

Next, sufficiently describe the semantics of both the target and source in CISL. It is worth noting that for the target, the description does not have to be fully exhaustive. For example, when working with the matching between Jikes RVM and the PowerPC architecture, it is not necessary to provide semantic descriptions for PowerPC's vector instructions as the Java virtual machine (bytecode) specification has no VM instructions dealing specifically with vectors. Figure 1 shows a semantic description of the Jikes RVM bytecode iadd. Because Jikes RVM is a stack-based virtual machine, all operations are performed on stack elements. With the iadd instruction, two integers are popped off of the stack, added, and the sum is placed back on the stack. This operation can be succinctly expressed in CISL using very few lines of code. Note that in the figure, syntactical information is also provided, yet it is not necessary for the process of instruction selection. This is because CISL descriptions can be, and have been in the past [7], used to generate assemblers, dissassemblers, and functional simulators.

The descriptions provided in the previous step are independent of any specific pairing of target and source. After an adequate amount of machine descriptions are provided for the source and target, the next step is to specify the mapping

of memories between the two. This mapping is specific for each target/source pair. An example of a memory mapping would be denoting that some temporary registers utilized in the source description map explicitly to a set of nonvolatile general purpose registers on the target. Pertaining to the Jikes RVM/PowerPC mapping, we directly denoted that the frame pointer used in Jikes RVM, which holds the base address of the operation stack, is realized by register 1 in the PowerPC.

After providing the memory mapping and the descriptions for the source and the target, Gist will attempt to find instruction selection patterns for each source instruction using a heuristic greedy best-first search. A heuristic search is necessary because of the exponential growth of the search space due to the Cartesian product-like nature of the source size and target size. Recent research [5] also suggests that searching for a target pattern to semantically replicate a source IR is recursively undecidable. It is also for this reason that automatic generation of instruction selection patterns must use a heuristic search.

Once Gist finds a pattern for each source instruction, it is only a matter of developing a small adapter that takes in these instruction selection patterns in XML form and outputs functioning code that is final instruction selector. This instruction selector can be plugged directly into the compiler framework and tested by simply compiling valid code and observing the behavior of the resulting program.

4 Method

Our goal for the project was to cover a large range of instruction types for the JVM specification: integer instructions, long instructions, single- and double-precision floating point instructions, stack manipulation instructions, type conversion instructions, array access instructions, and stack access instructions. Because of the stack-based nature of Jikes RVM and the fact that PowerPC is a RISC architecture that provides no inherent stack operations or addressing modes dealing with memory-to-memory operations, almost all of the instruction selection patterns have to match a large sequence (three or more) of instructions on the target.

In Jikes RVM, all data that is used in the basic instructions are stored in an operand stack which has a quantum data size of 32-bits. In our CISL description of the source, we based all memory access as being word-aligned to match this. The PowerPC target, being a RISC architecture, has to load data word-by-word from memory into its register file of 32-bit registers, perform register-to-register operations on that data, and store the results back into memory one word at a time.

CISL provides a mechanism to deal with compiler constants or virtual machines constants without directly interacting with them. For matching with Jikes RVM, this is essential due to its stack pointer being a constant named spTopOffset. Jikes RVM is unique amongst Java virtual machines because it itself is written in Java. In our description, for instructions dealing with the stack

(i.e., virtually every instruction) we can explicitly reference this variable which is formally declared in the Baseline compiler as `public int spTopOffset`. For the CISL description of the bytecode `iadd` in Figure 1, notice how we are describing stack access using this compiler variable and not anything that is formally described in the JVM specification. This concept of using outside information related to the frameworks themselves shows how flexible CISL can be at describing the compiler IR or VM specification.

After we outlined and provided sufficient machine descriptions for both Jikes RVM and PowerPC to cover the broad range of instructions discussed previously, we executed Gist to produce the instruction selection patterns in XML form. Taking as input these XML patterns is a small adapter written in Java which parses the metadata and outputs a functional instruction selector for the Jikes RVM Baseline compiler. Because Gist itself is written in Java, these framework-specific adapters can seamlessly be integrated into Gist, completing the entire process from machine descriptions to valid instruction selector.

The adapters make extensive use of the `StringTemplate` Java library which was written by Terence Parr, the creator of ANTLR. ANTLR [8] is a tool that aids in automating a compiler's front-end, so it is only fitting that we will use a related tool to assist in automating back-end components. In essence, the adapters' main functionality is to convert these XML patterns into forms that the internal compiler framework expects. Since Jikes RVM is written in Java, the Jikes adapter produces code that represents Java methods, one per bytecode. These methods must take the form of `protected final void emit_<bytecode>` where <bytecode> is the actual name of the VM operation.

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

Figure 2: Original emit sequence for the `iadd` bytecode in the Baseline compiler

Figure 2 illustrates the original version of the `iadd` bytecode in the Baseline compiler. Note that the `asm` library that is referenced in the code is the actual PowerPC machine library that emits instructions to the instruction queue. `popInt` and `pushInt` are internal methods that simply call load/store instructions to the machine and handle stack pointer arithemetic. Figure 3 shows the same bytecode that is automatically generated by Gist. Notice how there is less method invocation overhead as the generated version simply invokes all the `asm` methods directly instead of encapsulating them in separate method calls. The adapter also handles stack manipulation, which is framework dependent.

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

Figure 3: Gist generated emit sequence for the `iadd` bytecode

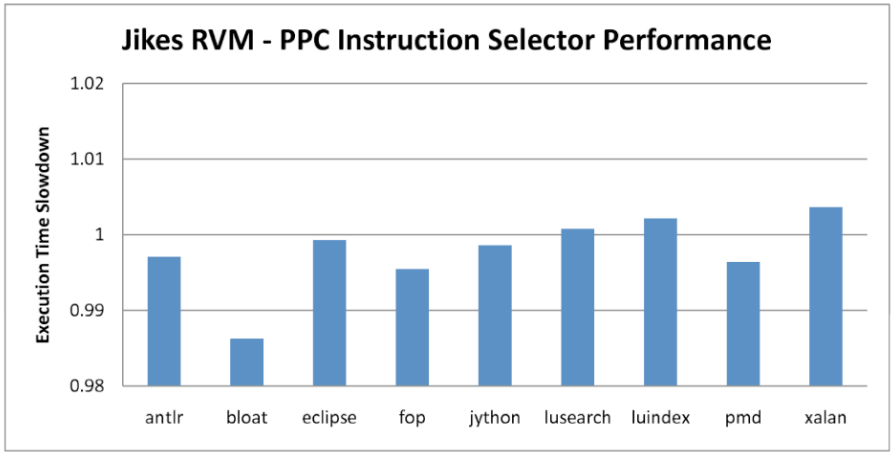

Figure 4: Slowdown of DaCapo tests on generated compiler (lower = better)

5 Results

As our goal was to generate a fully functional instruction selector for the Jikes RVM targeting PowerPC, there are many reasons why we cannot cover the full spectrum of the JVM specification. One of the major problems is that many JVM operations rely on runtime information, which cannot be statically matched for obvious reasons. The original implementation of the Jikes RVM Baseline compiler contains 165 emit methods for the set of JVM bytecodes. Of these 165, we consider 123 bytecodes for matching. We discard consideration for the 42 bytecodes based on the fact that they either contain integrated run time semantics, have ambiguous semantics in the JVM specification, or they are not supported on the PowerPC architecture.

Gist is able to match semantics and generate valid instruction selector emit methods for 85 bytecodes. This is a 69% coverage rate for the bytecodes that we consider. Although this percentage may seem low at first glance, consider that the bytecodes covered span a broad range of JVM capability outlined previously.

Because Gist was able to generate an actual instruction selector, we were

able to plug the Java source code into the Baseline compiler and observe its performance. The Baseline compiler is a JIT compiler, so we can run standard benchmarks and test their runtimes as opposed to testing static compile times. We chose to perform tests using a standard benchmarking suite, DaCapo [2]. Figure 4 shows the relative slowdown of the benchmarks on the generated Baseline compiler vs. its original implementation, averaged over ten trials. Note that although for some tests, our implementation shows a small improvement in runtime, we attribute this as within the point of being statistically insignificant.

6 Conclusion

Gist introduces an automatic and framework-independent method of instruction selector generation for compiler back-ends. We demonstrate this methodology by building a functional instruction selector for the Jikes RVM baseline compiler targeting the PowerPC architecture. Code generated by our instruction selector is functionally identical to code emitted by the original implementation of the Baseline compiler, and the instruction selector itself has a runtime nearly identical to the original.

References

[1] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The jikes research virtual machine project: building an open-source research community. *IBM Syst. J.*, 44(2):399–417, 2005.

[2] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The dacapo benchmarks: java benchmarking development and analysis. *SIGPLAN Not.*, 41(10):169–190, 2006.

[3] R. G. G. Cattell. *Formalization and automatic derivation of code generators.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1978.

[4] J. Ceng, M. Hohenauer, R. Leupers, G. Ascheid, H. Meyr, and G. Braun. C compiler retargeting based on instruction semantics models. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 1150–1155, Washington, DC, USA, 2005. IEEE Computer Society.

[5] J. Dias. *Automatically Generating the Back End of a Compiler Using Declarative Machine Descriptions.* PhD thesis, Harvard University, 2008.

[6] S. C. Johnson. *Unix Programmer's Manual*, volume 2b. 1979.

[7] J. E. B. Moss, T. Palmer, T. Richards, E. K. Walters, II, and C. C. Weems. Cisl: a class-based machine description language for co-generation of compilers and simulators. *Int. J. Parallel Program.*, 33(2):231–246, 2005.

[8] T. J. Parr and R. W. Quong. Antlr: a predicated-ll(k) parser generator. *Softw. Pract. Exper.*, 25(7):789–810, 1995.