Instructor Selector Generation from Architecture Description

Miloslav Trmač[1], Adam Husár[1], Jan Hranáč[2], Tomáš Hruška[1], and Karel Masařík[1]

1 **Brno University of Technology, Faculty of Information Technology**
`{itrmac, ihusar, hruska, masarik}@fit.vutbr.cz`

2 **ApS Brno, s.r.o.**
`hranac@aps-brno.cz`

Abstract

We describe an automated way to generate data for a practical LLVM instruction selector based on machine-generated description of the target architecture at register transfer level.

The generated instruction selector can handle arbitrarily complex machine instructions with no internal control flow, and can automatically find and take advantage of arithmetic properties of an instructions, specialized pseudo-registers and special cases of immediate operands.

Digital Object Identifier 10.4230/OASIcs.MEMICS.2010.109

1 Introduction

Application-specific processors are often used in embedded applications with large production quantities due to the speed and low power consumption they can provide at modest cost compared to using a generic CPU with higher execution speed. The Lissom project[8] developed at the Brno University of Technology aims to provide a full development environment for iterative hardware-software codesign, allowing embedded system developers to rapidly experiment with application-specific architecture facilities by automatically generating software development tools (assembler, disassembler, linker, simulator, C compiler) and a hardware prototype. Typically, the embedded system developer would experiment with changing the CPU, e.g. adding specialized instructions, and evaluate each experiment by using the Lissom software to regenerate the tool chain, recompile the application, and test its performance on a simulator.

This effort includes automatic generation of a C compiler. We have decided to base this work on the open-source LLVM project[6], reusing its existing front-end (which converts input in C into the LLVM internal representation), its optimization passes, and the provided infrastructure for implementing back-ends (which convert the internal representation into assembler or binary code).

In the compiler front-end it is only necessary to provide information about the application binary interface (ABI) of the target platform; front-ends are already prepared for this, so we only need to describe data types in a predetermined format. The LLVM optimization passes work purely on the internal representation without considering the target architecture. Most of the work therefore involves the compiler back-end, which is described in this article, focusing primarily on the instruction selection component.

2 Related Work

A full-featured compiler generator is described in [5]: from an architecture description in a language called LISA, it generates input for the commercial CoSy compiler development system. LISA is suitable primarily for accurate simulation, so compiler generation is not automatic: register properties and instruction scheduling information is extracted automatically, but the user must manually provide instruction behavior descriptions using a provided GUI. If the LISA input is changed, the information in the GUI may become obsolete and needs to be updated manually. In contrast, in our approach the instruction selector is generated automatically from the same input file used for generating other tools, shortening the edit-compile-evaluate cycle of architecture design exploration.

An approach that does not ask the user to specify the instruction selector described in [2]. The necessary instruction operation descriptions are extracted from an architecture description automatically. After phases that simplify the instruction behavior descriptions, the descriptions are matched against manually prepared patterns to find instructions that can be used in the initial instruction selection pass of `gcc` (`gcc` does not use a tree pattern selector, and requires named templates for the atomic operations). Unlike our approach, instructions that combine several basic operations are not supported.

An entirely different approach is not to try to support arbitrary user-specified architectures. Such systems are based on a specific CPU architecture, and provide a combined hardware/software compiler to the user. Given an input program, the compiler produces description of hardware (e.g. in VHDL) that implements some parts of the program, and executable code for the base architecture augmented with the described hardware, that implements the rest.[9] Such systems can produce good results if the base architecture is suitable, but are difficult to adapt when the base architecture needs to be replaced.

3 ISAC and Instruction Semantics Extraction

ISAC is an architecture description language (ADL) based originally on LISA.[7] It is a mixed ADL, meaning that it allows to describe both architecture and microarchitecture.

The architectural description consists of registers, memories, and instruction set description. Registers and memories are described as global C language variables. Instruction set is described hierarchically, because most instructions can use the same register or immediate operands. Two main constructs are used to describe the instruction set: The `OPERATION` construct allows to describe parts of instruction's syntax, binary encoding, and semantics. Instances of other operations or groups may be used in an operation specification. `GROUP` is used to describe situations where an instance used in an operation can be one of a set of operations or groups. An example of a "store byte" instruction is in listing 1.

Listing 1 Example of a "store byte" instruction with indirect adressing mode

```
# This is an simplified example, an operation usually represents
# multiple instructions (e.g. also "store half-word", and
# "store word").
OPERATION instr_direct_loadstore {
  INSTANCE register ALIAS {rt, base};
  INSTANCE signed_imm16 ALIAS {offset};
  ASSEMBLER { "SB" rt "," offset "(" base ")" };    # Syntax
  CODING { 0b100100 base rt offset };               # Binary encoding
  BEHAVIOR {                            # Instruction behavior
    int addr = regs[base] + offset;     # regs is an array of
    char val = regs[rt] & 0xFF;         #   general-purpose registers
    mem[addr] = val;             # mem represents memory address space
  };
}
```

The compiler back-end needs to identify each particular target instruction, but there is no notion of an instruction in the ISAC language. What can be used here, is that the assembly syntax description is based on context-free grammars. If we generate all words of the assembly language, we get a list of instructions. For each instruction the corresponding behavior in C is generated as well. This C code representing instruction semantics is then converted to a common format, a few simple optimizations that simplify the semantics are performed, and this result is passed to the back-end generator. The result for the example "store byte" instruction can be seen in listing 2.

Listing 2 Instruction semantics example, corresponding to listing 1

```
# Temporary variable names were changed to make the example easier to
# follow.
instr instr_direct_loadstore__op_sb__gpr_std__simm16__gpr_std__,
  %imm = i16 immop(1);
  %Rx = i32 regop(cl0, 0);
  %Rx_trunc = trunc(%Rx, i32 8);
  %imm_ext = sext(%imm, i32 32);
  %Ry = i32 regop(cl0, 2);
  %addr = add(%Ry, %imm_ext);
  store(%Rx_trunc, %addr);
, "SB" 0 "," 1 "(" 2 ")", 1  # Assembler syntax
```

4 Instruction Selector Generation

Instruction selection is the largest component of a LLVM compiler back-end. Its purpose is to convert an input program from a target-independent internal representation into a lower-level representation that deals with instructions of the target architecture instead of generic operations.

4.1 Instruction Semantics Format

The primary result of the instruction semantics extraction process described in section 3 is a list of instructions. In contrast to the human concept of a "single instruction", where all binary formats that use the same assembler mnemonics are considered a single instruction, we define a single instruction as a maximal set of binary encodings within which semantics and binary encoding can change only by substituting one register by another register from the same register class, or one constant by another constant of the same bit width and format.

For example, `load R1 = [R1 + R2]` and `load [R1 + imm]` are different instructions although they share the "load" mnemonics. Also, `add R1 = R2 + R3` and `add R1 = R2 + R0` (where `R0` denotes a read-only pseudo-register with zero value) are different instructions: they share the same binary format, but the semantics of one is "add values of two registers", and the semantics of the other is "copy a register value." In contrast, `cmov if(R1) R2 = R3` is (necessarily) considered a single instruction, where the semantics depends on the value of a register, not on its identity.

The extracted semantics is a sequence of *atomic operations*, using an unbounded number of temporary variables. Only limited control flow is supported: an `if` operation can be used to delineate a conditionally-executed set of atomic operations, but control flow can not return to the "main path" after executing the `if` body. Loops are not supported, so the control flow graph can form at most a tree rooted at entry of the semantics. In this control flow format, it is easy to arrange temporary variables to form the industry-standard SSA representation[4]. In contrast, the semantics can modify a single physical register repeatedly, so physical registers are not in SSA form.

4.2 Basic LLVM Instruction Selector

LLVM uses a tree pattern matching instruction selector, which can take advantage of complex instructions, as long as they have only one result. In contrast to research in this area, which suggests using bottom-up analysis with dynamic programming and generating globally optimal instruction selections (within the assumed cost model)[1, 3], LLVM uses a top-down, only locally optimal selector, sacrificing code quality for flexibility and speed. The selector is automatically generated from instruction descriptions, but it also allows adding additional C++ code to handle more complex cases.

Description of each instruction in LLVM includes its assembler format, (variable) operands, effects on other (fixed) registers, and other information, e.g. flags describing important behavior, and optionally binary format of the instruction. Instructions that should be handled by the generic instruction selector must also include semantics description in an expression tree form.

Instructions for which the extracted semantics naturally describe an expression tree (single externally-observable output—either register assignment or a side effect, no conditionals) can be converted to the LLVM format by treating the use-def links in the linear instruction semantics description as parent-child links in an expression tree. Instructions that change a register value conditionally can be converted into a corresponding LLVM operation as well, by implicitly constructing the C "?:" operator within the semantics.

An example LLVM instruction description is provided in listing 3.

Listing 3 LLVM instruction description example, corresponding to listing 2

```
def instr_direct_loadstore__op_sb__gpr_std__simm16__gpr_std__:
  LissomInst
  <(outs), (ins cl0:$op0, cl0:$op2, Si32i16imm:$op1),    # Operands
   "SB $op0 , $op1 ( $op2 )",                             # Assembler
   [(truncstorei8 cl0:$op0,                               # Expression
                 (add cl0:$op2, (i32 sextimm16:$op1)))]>{}
```

This approach can handle a large number of instructions, including instructions that combine several operations. On the other hand, this alone is insufficient on almost all architectures, because many architectures do not provide some of the "atomic" operations in a pure form, and if the "atomic" operations are not available to the instruction selector, there are likely to be programs that cannot be compiled.

4.3 Instructions with Multiple Outputs

Many architectures support instructions that provide more than one output (in this section, "output" means storing a value in a register, or a side effect). These instructions are often primarily used for only one of the outputs. This includes almost all instructions on architectures that use a flags register. To handle these cases, any instruction that has more than one output is *cloned.* One clone is created for each output, and in each clone, the other outputs are made invisible to LLVM (setting of a register is replaced by an annotation that the register is clobbered by an indeterminate value). If one of the outputs can not safely be made invisible (e.g. a jump), the clone is discarded. Thus, a single `xor Rx = Ry ^ Rz` instruction on an architecture with a flags register is cloned into `xor_reg`, which sets `Rx` and clobbers the flags register, and `xor_flags`, which sets the flags register and clobbers `Rx`. The LLVM instruction selector can use the `xor_reg` clone and thus automatically take advantage of the instruction.

4.4 Specialized Instruction Outputs

On many architectures some atomic operations are available only as a part of a more generic instruction. For example, an architecture might provide only a single flexible `load` instruction: `load Rx = [Ry + Rz * imm1 + imm2]` (where values of `imm1` and `imm2` are often limited in range). By choosing `imm1 = 1`, `imm2 = 0`, or `imm1 = imm2 = 0`, we can get the simpler `load Rx = [Ry + Rz]`, and `load Rx = [Ry]`, respectively. To handle these cases, we attempt to specialize each instruction with chosen constant values.

First, promising values of immediate operands are collected: Basic dominator optimizations (dead code deletion, copy propagation, constant folding, arithmetic simplification) are performed on the semantics, and each atomic operation that refers to an immediate operand is examined for values of the operand that could allow constant folding the operation. For example, values `0` and `1` are used for multiplication operands, or values `0` and `~0` (of correct width) are used for operands of bit-wise operations. Other atomic operations, including other arithmetic operations, comparisons, and `if`, are handled similarly. Finally, the value `0` is always added, simply because it so often leads to simplification.

After all candidate values are collected for each immediate operand, the instruction is cloned: one clone is created for each possible assignment of candidate values to respective immediate operands (including the cases when a specific value is not assigned to some of the operands). Each clone is then re-optimized: if the optimization does not simplify the instruction semantics, the clone is discarded. The remaining clones are treated exactly the same as "native" instructions (e.g. an LLVM description is generated for them).

4.5 The Instruction Set as a Whole

In addition to cooperating with the LLVM instruction selector, LLVM needs some information about the overall structure of the instruction set.

Most important is the LLVM "legalization" pass, whose purpose is to modify the input program to only use operations that are available in the target architecture, e.g. converting a 64-bit multiplication into a sequence of 32-bit multiplications and additions. Unfortunately LLVM is not able to extract the required information about available instructions from the individual instruction descriptions, so this information has to be generated separately.

Second, the LLVM instruction selector build process can not handle instructions sets in which two or more instructions match the same expression subtree; the backend author must explicitly select the instruction that the instruction selector will use for the subtree. (The other instructions can still be made available to LLVM—and used perhaps through specialized built-in functions—but their description must not contain the expression subtree, making them unavailable to the instruction selector.) To do this, structurally identical expression subtrees are identified in the backend generator, and a single instruction is chosen from each set of duplicates.

Finally, LLVM needs to generate some target instructions after instruction selection has finished, notably instructions for moves and memory accesses necessary for register allocation and spilling. These instructions are located by looking for instructions matching a specific form of atomic operations that do not have any unwanted side effects.

5 Other Backend Tasks

In addition to instruction selection, a LLVM backend needs to provide information to the register allocator, mainly description of register classes and lists of registers unavailable for allocation. This information is already provided in the extracted instruction semantics, based primarily on user's annotations in the source ISAC file.

We do not currently extract enough information about the pipeline and usage of functional units by instructions to implement effective instruction scheduling.

Finally, a back-end must implement handling of function frames, function calls, parameter passing, and other ABI-dependent transformations. We handle this by looking for instructions matching a specific form of atomic operations, similar to the case of target instructions used after instruction selection. The specific ABI can not be automatically determined, and means to allow the user to specify it are currently being added to ISAC. Without such information, the backend generator automatically generates a reasonable ABI by examining the existing instructions (e.g. looking for "return" or "call"), or, on very regular architectures with little built-in function call support, by looking for indirect addressing modes suitable for managing a stack manually.

6 Experimental Results

We have used the back-end generator on a restricted model of the MIPS architecture developed for the purpose. The semantics extraction implementation resulted in a description of 139 individual instructions (using the definition of "instruction" given in section 4.1). Out of these 139 instructions, 62 could be directly used by the LLVM instruction selector, remaining instructions required additional handling and conversion. Because the architecture includes a `R0` pseudo-register equal to a immediate value of 0, the instruction extraction process recognized a large number of instruction variants involving the `R0` register that ultimately resulted in the same LLVM semantics: in particular, semantics of 17 instructions was `Rx = 0`, and semantics of 11 instructions was `Rx = Ry`.

In our model, the only cloned instructions with multiple outputs were variants of division and multiplication.

Specializing instructions using specific values of immediate operands based on the original 139 instructions resulted in 26 new instruction clones, with 17 distinct classes of semantics for the purpose of the LLVM instruction selector. Similarly to the handling of the `R0` register in semantics extraction, the most frequent specializations resulted in simple register-to register copy and register assignment, but this process also identified ways to provide the primitive load/store operations (e.g. `LW Rx, 0(Ry)` - load a 16-bit value from address given by register `Ry` to register `Rx`), which are necessary both to guarantee ability of the instruction selector to handle arbitrary programs, and to implement register spilling in the register allocator.

7 Conclusion and Future Work

In this article we have presented primarily our approach to automatic generation of a LLVM instruction selector. While the other components of the LLVM backend generator are sufficient to create a working MIPS backend, implementation experience with more architectures is necessary before we can be confident in the viability of our approach and before we can present it in detail.

Our goals for future work include extracting enough information for instruction scheduling, which will also allow choosing the best possible instruction for the instruction selector when there are several alternatives. Extensions of LLVM to take advantage of SIMD instructions and WLIW architectures are already under development. We also intend to test performance of code created by the generated backend using industry-relevant benchmarks, and improve our backend generator based on detailed analysis of the compiled code for these benchmarks.

Acknowledgements This research was supported by the grants of MPO Czech Republic FR-TI1/038 and by the Research Plan MSM No. 0021630528. We would also like to thank the anonymous reviewers for their valuable comments.

References

1 Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions of Programming Languages and Systems*, 11(4):491–516, October 1989.

2 Soubhik Bhattacharya. Generation of gcc backend from Sim-nML processor description. Master's thesis, Indian Institute of Technology, Kapur, July 2001.

3 Todd A. Proebsting Christopher W. Fraser, David R. Hanson. Engineering a simple, efficient code generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, September 1992.

4 Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions of Programming Languages and Systems*, 13(4):451–490, October 1991.

5 Manuel Hohenauer, Hanno Scharwaechter, Kingshuk Karuri, Oliver Wahlen, Tim Kogel, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, and Gunnar Braun. A methodology and tool suite for C compiler generation from ADL processor models. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 1530–1591, 2004.

6 Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, March 2004.

7 Roman Lukáš, Tomáš Hruška, Dušan Kolář, and Karel Masařík. Two-way deterministic translation and its usage in practice. In *Proceedings of 8th Spring International Conference - ISIM'05*, pages 101–107, 2005.

8 Karel Masařík, Tomáš Hruška, and Dušan Kolář. Language and development environment for microprocessor design of embedded systems. In *Proceedings of IFAC Workshop on Programmable Devices and Embedded Systems PDeS 2006*, pages 120–125. Faculty of Electrical Engineering and Communication BUT, 2006.

9 Tim Sander, Aditya Vishnubhotla Vijay, and Sorin A. Huss. HW/SW codesignflow with LLVM. 2008 LLVM Developers' Meeting, August 2008.