Automatic C Compiler Generation from Architecture Description Language ISAC

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

1 **Brno University of Technology, Faculty of Information Technology**
Bozetechova 2, Brno, Czech Republic
{ihusar, itrmac, hruska, masarik, kolar, iprikryl}@fit.vutbr.cz

2 **ApS Brno, s.r.o.**
Purkynova 93a, Brno, Czech Republic
hranac@aps-brno.cz

Abstract

This paper deals with retargetable compiler generation. After an introduction to application-specific instruction set processor design and a review of code generation in compiler backends, ISAC architecture description language is introduced. Automatic approach to instruction semantics extraction from ISAC models which result is usable for backend generation is presented. This approach was successfully tested on three models of MIPS, ARM and TI MSP430 architectures. Further backend generation process that uses extracted instruction is semantics presented. This process was currently tested on the MIPS architecture and some preliminary results are shown.

Digital Object Identifier 10.4230/OASIcs.MEMICS.2010.47

1 Introduction

As semiconductor process node technology advances and allows Moore's law to be still valid, chip designers are faced with a problem how to make a chip that conforms to given performance, power, and cost requirements, but still can be used in many different devices in order to alleviate non-recurring engineering costs (chip design and mask manufacturing) that rise tremendously with each new technology process node.

Electronic System Level (ESL) methodologies try to lower design costs by providing guidelines for SoC (System on Chip) and MPSoC (Multiprocessor SoC) design. One ESL methodology for MPSoC design presented in [5] comprises of several steps, where the most important ones are target application analysis, task partitioning to specific processors, and optimization of such specific processors to suit performance, power and cost requirements. Processors optimized for a certain task are called *Application Specific Instruction-set Processors* (ASIPs).

When an ASIP is designed, the target application is analyzed and hot spots are found. New instructions that accelerate frequent computations are added to the ASIP's instruction set and the application is compiled and analyzed again. This process, often called *compiler-in-the-loop* ASIP design [2], is iteratively repeated until requirements are satisfied. To allow such optimization process, compiler, assembler, and simulator for the current ASIP architecture must be available.

Project Lissom running at the Brno University of Technology approaches this problem by providing an development environment for application-specific instruction processor (ASIP) design and optimization. Using *Architecture Description Language* (ADL) *ISAC* [7], the user can describe both the architecture (instruction set, registers and memories) and

Sixth Doctoral Workshop on Math. and Eng. Methods in Computer Science (MEMICS'10)—Selected Papers.
Editors: L. Matyska, M. Kozubek, T. Vojnar, P. Zemčík, D. Antoš; pp. 47–53

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

microarchitecture (usually pipelined processor implementation). From this description can all the needed tools and hardware description be generated. C language compiler is one of the essential tools and this paper presents a novel approach to higher-level programming language compiler generation from an ADL model.

2 Related Work

2.1 Retargetable Compilers

Retargetable compilers are higher-level programming language compilers that can be adapted to compile for different architectures.

Compilers are usually divided into three components. A frontend first parses input language, performs semantic checks and generates intermediate program representation (IR). Then a midend (also called optimizer or middle-end) is applied and performs mostly target-independent optimizations on IR. Resulting IR is passed to a backend (also called code generator) whose task is to transform the IR into target architecture program.

Examples of retargetable compilers are gcc, LLVM, CoSy, SUIF, lcc, Trimaran, LANCE, and SPAM. Target features may vary substantially and because of flexibility and needs for some target-specific modifications is adapting these listed compilers for a different target requiring extensive compiler expertise and it is up to the tool developers to make the compiler retargetting based on an ADL model as user-retargetable as possible.

2.2 Automatical Compiler Retargeting using an ADL Model

Instruction selection pass is usually performed in a backend as one of the first passes. The purpose is to transform input intermediate representation that uses compiler's IR instructions to a representation with target instructions. Instruction selection pass is generated by so-called code generator generators from some instruction description, where instruction semantics is in form of a tree or DAG (Directed Acyclic Graph) patterns.

Instruction selection pass is the most problematic pass to generate from an ADL model, and in this paper we will focus only on it. For example in LISATek Processor Designer suite, one of the most advanced ASIP design environments used in practice, significant manual effort to create instruction selection patterns from LISA language model is needed [3]. In a recent book C Compilers for ASIPs: Automatic Compiler Generation with LISA [2], they state that to derive instruction selection patterns from instruction behavior in C "is quite difficult, if not impossible". In the first compiler generator version, patterns were described using graphical interface and this caused the semantics information to be stored outside the LISA model in a special format. To partially overcome this problem, a special section SEMANTICS for patterns description was introduced, however, the instruction semantics is still present in the model twice, once in SEMANTICS and once in BEHAVIOR. A similar approach is used in Tensilica TIE [5], where two types of description: one for simulation and for hardware generation, second for compiler generation, are used. Review of other approaches to compiler generation can be found in [8], and also in [2].

We are convinced that a potential ADL user is usually familiar with the C language and to specify instruction behavior using this language is very convenient, better than to learn a new specification language with new syntax and a set of operations. However, as the LISA approach shows, to extract instruction semantics from C language description suitable for compiler generation is difficult and even is such a large project was not solved.

This paper comes with a novel solution that approaches this problem and that allows to extract automatically instruction selection patterns. Further, the process of compiler backend generation that uses such extracted instruction semantics is briefly presented.

3 ISAC Language

The ISAC (Instruction Set Architecture C) language falls into the category of mixed ADLs and allows both to describe architecture and microarchitecture. For purposes of this paper, we will consider only the architectural description. The ISAC language is originally based on the language LISA [1] where we simplified syntax and improved some constructs. Processor architecture consists of register, memory, and instruction set specification.

Description of each instruction consists of textual and binary representation and also of its semantics (behavior). For most existing instruction sets, many of instruction features are similar (like register and immediate operands, or conditional predicates), so for conciseness, the description is hierarchical and it is based on translational context-free grammars. There are two main constructs used to describe the instruction set. The first one is OPERATION, where parts of instruction's syntax, coding, and semantics are described. Construct GROUP is used to describe situations where an instance in an operation can be one of a set of operations or groups. One special group and operation modifier was introduced to the ISAC language because of compiler generation. It is a keyword REPRESENTS and tells that this group or operation is a register operand.

Example description of MIPS architecture instructions ADD and SUB is in fig. 1. Names of sections ASSEMBLER, CODING, and EXPRESSION were abbreviated to ASM, COD, and EXPR, also binary encoding was slightly modified in this example.

```
OPERATION reg REPRESENTS regs {
  ASM { "R" regnum=#U }; COD { regnum=0bx[5] }; EXPR { regnum; } }
OPERATION opc_add { ASM{ "ADD" }; COD{ 0b10 }; EXPR{ 0x2; };}
OPERATION opc_sub { ASM{ "SUB" }; COD{ 0b11 }; EXPR{ 0x3; };}
GROUP opc = opc_add, opc_sub;
OPERATION instr {
  INSTANCE reg ALIAS {rd, rs, rt}; INSTANCE opc;
  ASM { opc rd "," rs "," rt }; // Assembly syntax
  COD { 0b00 rs rt rd opc };    // Binary coding
  BEHAVIOR {                    // Instruction behavior described using C
    switch (opc) {
      case 0x2: regs[rd] = regs[rs] + regs[rt]; break;
      case 0x3: regs[rd] = regs[rs] - regs[rt]; break;
}};}
```

Figure 1 Description of MIPS instructions ADD and SUB in ISAC

To generate a C compiler, we need to extract instruction semantics and syntax for each instruction and then use it to generate compiler backend. We will look at these two steps in the following sections.

```
instr instr__opc_add__reg__reg__reg__, // Name
  %R1 = i32 regop(cl0, 1);             // Semantics
  %R2 = i32 regop(cl0, 2);
  %add = add(%R1, %R2);
  regop(cl0, 0) = i32 %add;,
  "ADD" 0 "," 1 "," 2                  // Syntax
```

Figure 2 Extracted instruction ADD with its semantics and syntax

4 Instruction Semantics Extraction

In the ISAC language is the instruction set described hierarchically using context-free grammars, there is no notion of an instruction present. However, to be able to identify particular instructions in the backend, we need to extract a set of instructions. To get such a set, we simply generate the assembly language from the assembly language grammar obtained from the model. Absence of cycles in this grammar is ensured by the ISAC language compiler, therefore the generated language is finite. Detailed information on grammar extraction from an ISAC model can be found in [6]. For our example in fig. 1, we get a language consisting of two words "ADD reg , reg , reg" and "SUB reg , reg , reg".

We also need unique instruction identification and instruction semantics. To obtain this, we construct a finite automaton with three types of terminals on transitions: operation names, assembly syntax parts, and instance names and parts of semantics in language C. Each path from the starting state to a final state then represents one instruction. For each such path we separately concatenate operation names, assembly terminals, and we also create C code that represents the instruction semantics.

Like this we obtain the instruction syntax and semantics, the only problem is that the form of semantics in the C language we retrieved from the ISAC model is neither suitable for instruction selection pass generation nor for other analyses.

But we can simplify it. We parse this code, then apply optimizations like constant propagation and dead code elimination. Further, memory and register accesses are identified. This way we obtain semantics representation that is usable for instruction selection pass generation. This process is fully automatic and we need no to add to the model information about instructions specific only for compiler generation. This approach overcomes possible inconsistency problems when behavior is described twice in LISA approach (described in subsection 2.2).

The result for our example can be seen in fig. 2. Semantics is described using our SSA-based intermediate representation, auxiliary variables have prefix `%` and `cl0` is an identifier that specifies general-purpose register class.

In this example, instruction semantics is described as a simple DAG with two register input operands on leaves. Register values are added and stored into another register operand. As operations in semantics description can be standard arithmetic, register read/write, and memory load/store operations used. Conditional execution is expressed using operation `if`, and jumps with operation `br`.

5 Retargetable Backend Generation

We have decided to base our work on the open-source LLVM compiler[4]. Only trivial modifications are necessary in the frontend, most of the work involves the backend (which

```
def instr__opc_add__reg__reg__reg__:
   LissomInst<
    (outs cl0:$op0), (ins cl0:$op1, cl0:$op2),     // Operands
    "ADD $op0 , $op1 , $op2",                      // Syntax
    [(set cl0:$op0, (add cl0:$op1, cl0:$op2))]>;   // Selection pattern
```

Figure 3 LLVM instruction description example

generates the actual assembler output).

The largest component of the LLVM backend is *instruction selector*, which converts an input program from a target-independent representation into a lower-level representation that deals with instructions of the target architecture. LLVM uses a tree pattern matching instruction selector, which can take advantage of complex instructions, as long as they generate only one result. The instruction selector is automatically generated from instruction descriptions, they include an expression tree representation of the semantics to match, but it also allows adding C++ code to handle more complex cases.

An example of instruction description that is generated from example in fig. 2 is provided in fig. 3.

LLVM also needs some information about the overall structure of the instruction set. Most important is the *legalization* pass, which modifies the input program to only use operations that are available in the target architecture. Unfortunately LLVM cannot extract the required information from the individual instruction descriptions, so this information is generated separately.

Fuirther 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 finding instructions matching a specific form of operations, that do not have any unwanted side effects.

Finally, we generate code handling function frames, function calls, parameter passing, and other transformations dependent on the architecture Application Binary Interface (ABI) describing calling conventions and register allocation rules. Means to allow the user to specify the ABI are currently being added to the ISAC language. In absence of such information, the backend generator automatically generates a reasonable ABI by examining the existing instructions, e.g. looking for a “return” or “call” instruction.

6 Results and Future Work

Program that extracts compiler generator information from ISAC model was implemented and tested on architectural models of 2 32-bit general-purpose processors MIPS (MIPS32, Release 1) and ARM (ARMv5) and a 16-bit microcontroller MSP430 from Texas Instruments. MIPS and ARM models describe all basic instructions from their instruction set without any extensions and co-processors, model of MSP430 is complete and describes all the instructions this instruction set provides. Results are shown in table 1. All tests were run on Intel Core2 Quad 9550 @2.83GHz, Fedora 9, x86_64, only one core was used. Semantics extraction program was compiled with gcc 4.4.1 using -O3. Total execution time is an average from 5 runs and the standard deviation was ± 3%.

Each instruction of the ARM architecture can have one of 15 different predicates and may use one of 8 different addressing modes and this is the reason why the extracted instructions count is so high. The behavior of most instructions of the the MSP430 architecture is described

using just one ISAC operation that contains large switches. For each such instruction is lots of code generated and this causes high relative extraction time.

We cannot compare these counts of extracted instructions to other approaches, because in the available publications on related work, intermediate instruction-set forms are neglected, and directly the results of generated compilers are presented.

Table 1 Transformation time and count of generated instructions for ISAC models of MIPS and ARM architectures

	MIPS	ARM	MSP430
ISAC lines	1110	1450	2040
C lines	610	1190	665
Count of extracted instructions	281	5741	718
Extraction time	0.5 s	35.5 s	16.0 s

When creating the ISAC model, the designer must be careful about using correct data types, otherwise unnecessary data type conversions in selection patterns are generated. Inspection of generated patterns can reveal diverse bugs in instruction behavior. A tool that graphically displays extracted patterns was developed and this and this can greatly aid in processor design verification.

Extracted semantics for the MIPS architecture was used to generate MIPS LLVM backend. This backend was first working at the time of writing this paper, therefore we present here only preliminary results for a simple program that calculates the Fibonacci sequence.

This program was compiled for MIPS by compilers GCC 4.4.1, and CLANG 1.0 with LLVM 2.8 (CLANG is a frontend that generates intermediate representation for LLVM). Input for our generated backed was obtained by compiling source code with CLANG 1.0 and then optimized (for -O3) with LLVM 2.8 optimizer. Resulting assembly code was then assembled and simulated using tools automatically generated from the MIPS ISAC model (e.g. [7]). Cycle counts needed to execute the program are shown in table 2.

Table 2 Preliminary results for backend generator for the MIPS architecture, values are cycle counts needed to simulate compiled program that calculates the Fibonacci sequence

	GCC	LLVM	Backend generated from the ISAC model
No optimizations (-O0)	1991	2200	1416
All optimizations (-O3)	508	506	913

Current plans for the future are: to allow the user to specify ABI, support for predicated execution, arbitrary bit-width integral data types, floating and vector (SIMD) variables and operations. Also we will improve the backend generator according to semantics description extracted from ARM, MSP430, and other models.

7 Conclusion

This paper presents an approach to higher-level language compiler generation. Backend is the part of compiler, where most of target-specific transformations is done and to accelerate application-specific processor architecture development, we need to generate compiler backend as automatically as possible.

First, architecture description language ISAC is briefly presented. Further, translation from ISAC architecture model to the compiler generation model is described. This approach overcomes problems caused by possible architecture model inconsistencies when one type of description is used for simulation and hardware generation and another type is used for compiler generation as is in similar projects usual. Inspection of extracted patterns can also point to some bugs that may be present in the architecture model and leads the user to use exact data types. Usage of exact data types also results in a model usable for efficient hardware generation.

Instruction semantics extraction from ISAC is fully automatic and it was tested on architectures MIPS, ARM, and MSP430. Extracted information was used to generate a backend for the MIPS architecture and some preliminary results were presented.

Acknowledgments

This research was supported by the grants of MPO Czech Republic FR-TI1/038, by the Research Plan MSM No. 0021630528, by the doctoral grant GA CR 102/09/H042, by the BUT FIT grant FIT-SS-10-1, and by the European project SMECY.

References

1 Andreas Hoffmann, Heinrich Meyr, and Rainer Leupers. *Architecture Exploration for Embedded Processors with Lisa*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

2 Manuel Hohenauer and Rainer Leupers. *C Compilers for ASIPs: Automatic Compiler Generation with LISA*. Springer Publishing Company, Incorporated, 2009.

3 Paolo Ienne and Rainer Leupers, editors. *Customizable Embedded Processors*. Morgan Kaufmann, 2007.

4 Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 75, Washington, DC, USA, 2004. IEEE Computer Society.

5 Steve Leibson. *Designing SOCs with Configured Cores: Unleashing the Tensilica Xtensa and Diamond Cores (Systems on Silicon)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

6 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.

7 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.

8 Prabhat Mishra and Nikil Dutt, editors. *Processor Description Languages*. Morgan Kaufmann, 2008.