ACCGen: An Automatic ArchC Compiler Generator

Rafael Auler, Paulo Cesar Centoducatte, and Edson Borin
Computer Systems Laboratory - LSC
Institute of Computing, University of Campinas - UNICAMP
Campinas, São Paulo, Brazil 13083-852
{auler, ducatte, edson}@ic.unicamp.br

Abstract

The current level of circuit integration led to complex designs encompassing full systems on a single chip, known as System-on-a-Chip (SoC). In order to predict the best design options and reduce the design costs, designers are required to perform a large design space exploration on early stages of the design. To speed up this process, Electronic Design Automation (EDA) tools are employed to model and experiment with the system. ArchC is an “Architecture Description Language” (ADL) and a set of tools that can be leveraged to automatically build SoC simulators based on high-level system models, enabling easy and fast design space exploration in early stages of the design. Currently, ArchC is capable of automatically generating hardware simulators, assemblers, and linkers for a given architecture model. In this work, we present ACCGen, an automatic Compiler Generator for ArchC, the missing link on the automatic generation of compiler toolchains for ArchC. Our experimental results show that compilers generated by ACCGen are correct for Mibench applications. They compare, as well, the generated code quality with LLVM and gcc, two well-known open-source compilers. We also show that ACCGen is fast and has little impact on the design space exploration turnaround time, allowing the designer to, using an easy and fully automated workflow, completely assess the outcome of architectural changes in less than 2 minutes.

1. Introduction

The increasing level of circuit integration has made it possible for industry to build entire systems on a single die, giving rise to the System-on-a-Chip (SoC) revolution. This trend generated demand for sophisticated tools to enable whole system simulation at a high level of abstraction. Looking forward to address this problem, researchers have proposed the use of Architecture Description Languages (ADLs) [1], capable of describing a design at high level of abstraction and at the same time being flexible enough to allow easy and fast design space exploration.

In order to explore the interface with which the software drives the hardware, the Instruction Set Architecture (ISA), designers must be able to easily specify the high-level hardware architecture of the processor. ADL descriptions are well known to address the problem of automatic functional simulators generation and RTL hardware synthesis using high-level descriptions, but there is still too much engineering effort put into adapting the software to run on the new platform.

The semantic gap between the processor ISA and the software language is bridged by a compiler, and when a new ISA is proposed or modified, the software team has the burden of the tedious, manual, and error prone task of porting an existing

compiler to the new architecture. Despite efforts to make compilers easier to retarget, it is still a task that requires an expert both in compiler project and target architecture domains.

Modern compilers are organized in order to improve code reutilization of a common set of algorithms for multiple targets and multiple source languages. Each *front end* parses one source language and each *back end* produces code to a different target processor. One would expect it to be easy to retarget a compiler, since the software team just needs to write a new back end for an existing compiler project. But, in practice, this task requires developers to understand not only the target processor, but also the compiler Application Programming Interface (API), which is a considerable effort and is likely to add a prohibitive delay into the design exploration phase.

The algorithms used in compiler back ends are well known and established. They handle the instruction selection, instruction scheduling, and register allocation problems. In fact, most of back ends share a lot of common code, differing mostly on the few parts of the code that are dependent on the target processor architecture. In this work, we explore algorithms to automate the generation of compiler back end code by extracting the necessary target processor architecture information available on ADL models. This complex task requires either manual steps in the workflow or large generation times due to the use of automatic programming algorithms. We present a both fully automatic and fast approach.

ArchC [2] is an ADL and a set of tools that, given an ADL processor model, automatically produce processor simulators compatible with the SystemC language [3], which enables the resulting simulator to be integrated with other SystemC hardware models into a larger platform. ArchC has also a rich set of description features that allow the platform designer to embed assembly language syntax annotations into the ADL processor model and automatically generate assemblers, linkers and debuggers [4], [5] based on the GNU binutils framework [6].

The LLVM compiler infrastructure [7] is a collection of modular and reusable components to implement compilation toolchains. The components include modules for code optimization, analysis, and code generation to target specific code. LLVM may also be used as a complete static compiler that performs intraprocedural and interprocedural optimizations if the user builds the tools to this end. It is also remarkably well documented and has been widely adopted by academia and industry as a production quality compiler.

The goal of this work is to enable the automatic synthesis of LLVM back ends using the ArchC processor model. The contributions of the paper can be summarized as follows:

- We designed an automatic instruction selection pass generator and a minimal LLVM back end code skeleton generator that are used to automatically build a working LLVM back end without the need to program any line of code;

Figure 1. The ArchC design exploration framework.

- We propose extensions to the ArchC ADL needed to allow designers to describe the required information for compiler generation by using solely information available on the processor architecture manual;
- We show results in which the designer is able to generate a simulator and the complete compiler toolchain in less than 2 minutes, allowing the designer to rapidly assess the outcome of architectural changes. This is only possible by using an approach that is both fast and fully automated;
- We also compare the software produced by the automatic compiler with the software produced by manually tuned production compilers.

The remainder of this paper is organized as follows. Section 2 provides an overview of the whole compilation system. Section 3 discuss the LLVM infrastructure. Section 4 describes the ArchC ADL. Section 5 shows the experimental results. Finally, Section 6 lists the related work and Section 7 concludes the paper.

2. Overview

ArchC [2] is an architecture description language and a set of tools that enables rapid platform design exploration through automatic generation of simulation tools. Figure 1 illustrates the overall ArchC design exploration framework.

The ArchC framework helps the designer building the processor simulator and the compilation toolchain, which will be used to compile the platform software. As an example, the `acsim` tool automatically generates a SystemC processor simulator based solely on the ArchC processor model. The generated simulator can be then combined with other SystemC hardware models to compose a larger platform simulator.

The ArchC compilation toolchains are composed mostly by three tools: the compiler, the assembler and the linker. All of these tools are architecture dependent and must be generated or modified when the architecture model changes. For example, inserting a new instruction on the architecture model requires the assembler to recognize the new instruction assembly syntax. The compiler should also be aware of the new instruction in order to generate it when compiling code. Baldassin et al. [4] extended the ArchC abstract model with annotations to enable automatic generation of assemblers, linkers, disassemblers and debuggers. Their tool (`acbinutils`), later integrated into the ArchC project, enabled the designers to automatically generate most of the tools present on the compilation and debugging toolchains.

Despite previous efforts, the automatic compiler generation is still a missing link on the ArchC compilation toolchains. The work presented in this paper fills this gap by introducing

ACCGen, an automatic Compiler Generator for ArchC. The ACCGen tool leverages the LLVM compiler infrastructure [7] to automatically build compilers for ArchC models. The next section presents an overview of the LLVM infrastructure.

3. The LLVM Infrastructure

The LLVM infrastructure is a collection of modular and reusable components that can be connected together to compose compilation tools, like a static compiler. The components include modules for code optimization, analysis and code generation to target specific code.

3.1. General Organization

Modern compilers are typically organized in three major parts: front end, machine independent code optimizations and back end. The front end is responsible for translating the source code from a high-level language, like C++, into an intermediate representation that is source language and target architecture agnostic. The intermediate representation (IR) can then be optimized with machine independent code optimizers. The back end is in charge of translating the IR code into the target machine assembly code. This organization allows compilers to be enhanced to compile from new source languages or to new target architectures simply by attaching a new front end or a new back end. LLVM based static compilers, like `llvm-gcc` and `clang` (actually named after their front end names), are also organized in this way.

Figure 2 illustrates the overall organization of the `llvm-gcc` compiler, a LLVM based compiler that uses a modified version of the GNU gcc [8] as the front end.

Figure 2. Overall organization of the `llvm-gcc` compiler.

The `llvm-gcc` compiler works as follows: the `llvm-gcc` tool (front end) first compiles the source code into *LLVM bitcode*, the LLVM IR. Then, the LLVM `opt` tool can be leveraged to apply several machine independent optimizations to the LLVM bitcode. Finally, the `llc` tool (back end) generates the

assembly code for the target machine. As described before, the GNU assembler (as) and the GNU linker (ld) assemble and link the assembly code together with other object and library code to form the final executable.

Our automatic Compiler Generator for ArchC, the ACCGen tool, builds on top of the `llvm-gcc` compiler toolchain to automatically generate compilers for ArchC processor models. The front end and the LLVM bitcode are target architecture agnostic and only the back end (`llc`) needs to be generated. The next section describes the back end of the LLVM infrastructure.

3.2. LLVM back end

The LLVM project is designed to concentrate target specific information on the back end of the compiler, yet allowing the programmer to implement custom algorithms for specific instruction selection phases. This code is written in C++ in its majority, while a small fraction uses the *tablegen language*, created specifically for the LLVM project in order to express declarative information about the target architecture and its instructions. In this sense, LLVM itself already implements a code generator up to a certain degree, since the programmer does not need to write the instruction selector itself.

The tablegen approach generates patterns directly as described by the user, similar to TWIG [9], an instruction selector generator. However, tablegen often cannot handle special cases and these must be custom specified with C++ code bypassing the tablegen specification. Our approach, on the other hand, reduces the amount of information required to create the back end and does not require the designer to have C++ or LLVM API knowledge. ACCGen is fully automatic and does not need manual C++ code customizations. This is possible because we use a search algorithm to find LLVM pattern implementations without user assistance.

It is a common practice among programmers who need to write a new back end to copy and paste the code from a working back end and work on incremental changes until it is adapted to the new target. This approach works well because back ends in LLVM, albeit designed to contain only target specific information, still share a lot of common code. To avoid burdening our automatic back end generator with tasks which involve always generating the same code for every back end, we wrote a target independent LLVM back end code skeleton, which is the base code shared by every automatically generated back end.

The LLVM back end class hierarchy is already provided with the back end skeleton. The automatic generated code excerpts are inserted into sections marked with specific tags demanding target specific behavior. Our code generator engine generates C++ code that uses the LLVM library and emits target instructions. After the search engine decides which sequence of target instructions best represents a computation, this sequence is translated to the C++ code in charge of emitting the corresponding instructions.

4. An Extended ArchC ADL Model

In order to retarget the LLVM compiler back end, we need a restrict set of information about the target architecture. Some of this information is already available in an ADL machine model while other does not. The most important missing information that is frequently omitted from ADLs is certainly the ABI convention.

The ABI must be enforced by the back end when manipulating the machine stack or when making procedure calls. This kind of information is compiler specific (sometimes

tied directly to the input programming language, such as the definition of the number of registers used for the `double` data type in C), does not describe any hardware and thus is naturally absent from ADL models. For this reason, it is usually necessary to extend the ADL to support compiler generation, except in cases when these decisions are made automatically and consequently the generated compiler is not compliant with other compilers ABI. In these cases, the code generated by the generated compiler may not be linked with previously existent code or code generated by other compilers. In this sense, we chose to extend the ArchC language with these necessary information to generate ABI compliant compilers.

It is important to choose the right degree of compiler specific details that the ADL model should contain. On the one hand, an ADL annotated with detailed compiler information may allow the user to fine tune the back end. On the other hand, this may come at the expense of complicating the machine description, rendering the ADL useless for a non compiler-expert. As pointed out earlier, even LLVM's tablegen language itself may be used as a form of back end automation, although tablegen's approach requires so much customization that only an expert programmer with knowledge of the LLVM API would handle it. On the other hand, an ADL annotated with little or no compiler information may allow a designer with no compiler knowledge to generate a back end for a new ADL model. However, in this case, the code produced by the automatically generated back end may not be ABI compliant or may suffer from poor performance.

We stick to the principle that the ACCGen tools should require the minimum amount of information from platform designer when generating the new compiler. The designer should not put more effort in describing compiler specific aspects than she would put when programming a custom back end, moreover, it should not be necessary to have any knowledge of LLVM specifically. This constraint is important because the ADL must not be tied to any particular compiler implementation, but should focus in describing machine aspects.

The following list is a collection of information that one must know when generating a new compiler back end:

- Instructions codification format and assembly syntax: Since the ArchC ADL ArchC already has automatic assembler generation [4], we chose to rely on this generated assembler and generate a back end functional up to assembly emission. Therefore, our ACCGen tool requires only the assembly syntax information, already present on the ArchC ADL.
- Instructions identification: Trivially obtained from a classic ADL model.
- Instructions semantic: This is the declarative information that states what each instruction performs. It is the building block to enable target machine assembly code emission.
- Data layout: Pointer sizes, data alignment, and endianness.
- Register file: Allows to find if a specific data entity is directly supported in the register file and is also used for register allocation purposes.
- Calling convention: When a call to another function needs to be translated, the back end must know detailed information about how to perform this call in the target machine language

4.1. Preexistent Information

Part of the information needed to retarget a compiler to a given architecture overlaps with information required by classic ADL models to generate simulators, making the integration of

simulation generation and compiler generation an interesting approach.

The ArchC language organizes the model information in two different ways. The first one retains structural information expressing the list of hardware resources available on the platform. The second one is a file with a description of the instruction set architecture: the instruction formats, the instructions themselves and the behavior of each instruction. The behavior is written using the SystemC language and describes how the hardware resources interact in order to complete its processing. ArchC generates the processor simulator as a SystemC object itself and can be easily connected with other SystemC components to form large platforms.

Figure 3 shows an excerpt of the ISA information for the MIPS architecture. The instruction formats and definitions use annotations resembling C++, familiar to SystemC designers. The `ac_format` keyword is used to define a new instruction format and specify how many bits each field occupies. The `set_asm` and `set_decoder` keywords specify the assembly syntax and what value is expected by the simulator decoder in each field.

```
1 ac_format Rtype = "%op:6 %rs:5 %rt:5 %rd:5
2           %shamt:5 %funct:6";
3 ac_instr <Rtype> add, addu;
4
5 add.set_asm("add %reg,%reg,%reg", rd, rs, rt);
6 add.set_decoder(op=0x00, funct=0x20);
7
8 addu.set_asm("addu %reg,%reg,%reg", rd, rs, rt);
9 addu.set_decoder(op=0x00, funct=0x21);
```

Figure 3. MIPS ArchC model example, ISA section.

These annotations are already used by the ArchC simulator and assembler generators. Our ACCGen tool uses this information to (1) extract individual instruction identification and (2) learn how to produce assembly code for the target architecture and how to use each instruction operand. This eliminates the need to deal directly with instruction codification and the generated assembly code will be already compatible with the ArchC generated assembler.

4.2. Instruction Semantics

Figure 4 shows how the designer may specify an instruction behavior in ArchC using arbitrary C++ code. This code snippet is used by ArchC to build an instruction-set simulator (ISS) able to interpret programs written for the target machine. The symbol `RB` must be previously listed as a hardware resource available as a register file. `rd`, `rs` and `rt` are operands declared in line 1 of Figure 3. One of the advantages of this approach is that it is easy for the designer to modify or generate a new simulator and even add some extra C++ code to collect statistics about the target code running on the simulator. While there are ADL approaches that restricts their instruction behavior definition language to RTL-synthesizable constructs in favor of hardware synthesizing tools, it is common practice to allow the designer to program instructions with a regular programming language, as in the LISA ADL [10].

Instruction semantics are the basic information required by the compiler back end to map the IR code to the target architecture instructions. However, automatically inferring instruction semantics out of the aforementioned instruction behavior can be a very hard task. Therefore, we add new constructs to the ArchC ADL that allows the system platform designer to specify the instruction semantics.

```
1 void ac_behavior( add )
2 {
3     RB[rd] = RB[rs] + RB[rt];
4 };
```

Figure 4. Instruction behavior description example using ArchC.

We designed a new RTL (Register Transfer List) based instruction semantics language that captures how each processor instruction interacts to compute new data or to change the control flow. Figure 5 shows how the semantics of a typical add instruction is represented. Leaf nodes represent inputs or outputs and thus are connected to a storage element of the processor. The union of all such elements constitutes the processor state and, in this way, the semantic tree expresses how the instruction changes the processor state.

Figure 5. The semantic tree for the add instruction of MIPS: (a) Graphical form; (b) Linear representation.

The leaf nodes also have type information. The tag `GPR` following the leaf names states that it is a general purpose register and as a consequence this instruction semantic describes an add instruction which operates with registers. Intermediate nodes represent operations identified by different names.

Figure 6 shows how the designer may define a particular instruction semantic tree. The linear representation for the tree is written along with a static cost number. The string `addu` providing the instruction identification has the same name declared inside the ISA ArchC model.

```
1 define instruction addu semantic as (
2     (transfer Op1:GPR (+ Op2:GPR Op3:GPR));
3 ) cost 1;
```

Figure 6. Definition of the semantic tree for the MIPS addu instruction.

The tree leaves need to be linked with the assembly operands declared on the ArchC `set_asm` construct, also used by the assembler generator. In this way, the generated compiler knows how to properly emit this instruction when necessary. A leaf node is defined with two strings separated by colon. The first specifies the name. If this name finishes with a number, this number identifies the operand that will be used in the assembly syntax string when writing this instruction in assembly forms. As an example let's assume the MIPS addu

instruction semantic, shown in Figure 6. It has three leaf nodes: Op1:GPR, Op2:GPR, and Op3:GPR. The number 1, at the destination node Op1:GPR, indicates that this node should be mapped to the first operand of the MIPS assembly syntax.

4.2.1. Specific References and Semantic Forests

Leaf names may directly correspond to a particular assembly syntax operand. However, it is often the case that an instruction has codified in its behavior a hard-coded register that may not be freely selected among an entire storage class, but may be equally treated as an operand for an intermediate node. Consider, as an example, the MIPS multiply (`mult`) instruction whose semantic forest is specified in Figure 7. In this case, the semantic is separated in two different trees and, as a consequence, constitutes a semantic forest. The first one specifies that the low part of the multiplication result is always written in `lo`, a MIPS special register, while the second one binds the 32 bit high part to the `hi` special register.

```
1 define instruction mult semantic as (
2   (transfer Lo:SPECIAL (* Op1:GPR Op2:GPR));
3   (transfer Hi:SPECIAL (mulhs Op1:GPR Op2:GPR));
4 ) cost 1;
```

Figure 7. Semantic tree definition for the MIPS `mult` instruction.

4.2.2. Literal Operand Binding

Leaf nodes in our behavior specification language strictly specify storage elements of the platform. In this sense, the operands for the assembly syntax are frequently linked directly to leaf nodes. Nevertheless, sometimes the designer may want to bind a literal value to a specific assembly operand whenever a particular semantic tree is chosen to perform a computation and, therefore, requiring this instruction instantiation in assembly language code.

Figure 8 presents the case of the ARM `and` instruction. The first line shows how the instruction assembly syntax is declared in the ArchC ISA file. The syntax string uses the percentage character to delimit a new operand in a similar way to the C standard library `scanf` function. By reading this line we know this instruction assembly syntax may take up to 5 operands. The most important operands are the last three, also referred in the semantic tree by using node names ending with numbers (Op3, Op4 and Op5, which are all registers). On the other hand, the first two operands are static in this semantic tree. The first one is the ARM condition to execute the instruction and is always the empty string. This means the instruction execution do not depend on the program status register (PSR). The second one is the `s` flag. When set, the user must put the `s` suffix after the mnemonic and this means this data processing instruction will update the PSR. Both operands must be set to empty string if the back end needs to use this instruction as represented by the given semantic tree. In order to allow this, the user may start the tree with the construct `let operand = "string"`. This sentence statically binds the literal string to the specified operand.

Note in the previous example that the `and` instruction is being underutilized since the semantic tree provided does not fully capture the instruction behavior. The most obvious evidence is that we statically bound two operands to empty strings. Frequently it is convenient to use only a fraction of an instruction computation. Therefore, it is interesting to express an instruction with multiple semantic trees, either because this instruction alters multiple storage elements (in which case

```
1 and1.set_asm("and%[cond]sf %reg, %reg, %reg",
2 // rest of line omitted
3
4 define instruction and1 semantic as (
5   let Op1 = "", Op2 = "" in
6   (transfer Op3:GPR (and Op4:GPR Op5:GPR);
7 ) cost 1;
```

Figure 8. Simplified semantic tree definition for the ARM `and` instruction.

each semantic tree corresponds to the computing of a given destination) or because it may be interesting to the back end to use the instruction ignoring part of its behavior. However, partially using some instruction may not be as trivial as it seems, and in some cases it may be necessary to introduce side effect compensation code in order to eliminate unwanted instruction behavior for a given application. In this case, the user may fully describe the instruction behavior and let the search engine find a way to use this instruction while eliminating unwanted effects.

4.3. A Fast Search Engine

The search engine is an algorithm responsible for searching how to implement a given program fragment using machine instructions that we only know its semantic tree. We employ an expression tree rewriting mechanism to change semantic tree nodes of one or more instructions until there is a match with the target program fragment we wish to implement. Using this approach, we are able to discover the implementation of 57 pre-selected LLVM code tiles that composes a fully capable LLVM instruction selector. Other program fragments are also needed for stack adjustment, stack element access, global variable access, function prologue, function epilogue, and function call. These are not handled by the LLVM instruction selector and are processed in later back end phases. Our search engine uses tree rewriting because this allows it to be simple and fast as in Cattell approach [11], yet powerful enough to retarget LLVM. It does not perform an extensive search as an superoptimizer [12] due to time constraints. This compromise proved to be fast and sufficient for compiler pattern automatic programming. To further increase speed, our algorithm performs independent searches in parallel when the host machine has a multicore processor.

Other constructs added to the ADL model trivially specifies the remaining information listed in the opening of this section as necessary for compiler retargeting. These extensive details and the search engine full description are available to the interested reader [13].

5. Experimental results

We devote this section to present the experimental results used to show our generated compiler produces correct code, our compiler generator is fast and we compare our generated compiler with gcc and LLVM regarding output code quality.

5.1. Turnaround Time

Figure 9 shows typical delays in the tool synthesis process for ArchC when an architecture modification is committed in the model and the framework must be rebuilt. We conducted this experiment in an Intel Core 2 Quad Q6600 with 4GB RAM memory. In this example, we use an ARM model and we are modifying its instructions iteratively, automatically generating the whole test framework and testing the impact of our architecture modifications.

Figure 10. Validation workflow used for the ARM back end.

Figure 9. Timings for each process in the test framework generation after a modification in the model is changed.

Being the most complex component in the toolchain, the compiler generation workflow can easily become the critical path (shown in bold in the illustration) in the process of synthesizing tools in charge of helping the designer explore the project design space. This is especially true for fully automatic unsupervised generation as proposed here, since we need to use expensive search algorithms to allow program synthesis and take implementation decisions about the instruction selector patterns. On the other hand, this approach reduces the effort required from the designer since a human takes much more time than a computer to take these decisions. Furthermore, ACCGen is designed to be fast and allow many design space exploration iterations, completing the synthesis for the ARM model in 24 seconds on average. Rebuilding LLVM with the newly generated back end C++ files takes additional 38 seconds. On average, ACCGen generates 5 thousand lines of code for the back end and a large fraction of the compilation time is in fact due to linking the code to already compiled LLVM libraries.

Then, we combine the compiler and the retargeted assembler to compile the platform application source code. This time depends on the application complexity, but it is typically very fast for simple embedded applications. At this moment we produced the platform software now using our new ISA

modifications and we are ready to test the performance impact of this decision using the generated simulator. A medium-sized experiment simulating a trace of 400 million instructions takes about 30 seconds on the ArchC generated ARM simulator. This means the designer is free to analyze the platform performance and start a new round of architecture modifications in less than 2 minutes.

5.2. Back end Correctness

Using our search engine we managed to generate the instruction selector for four different RISC architectures: ARM, MIPS, Sparc V8 and PowerPC. Although being able to compile bare machine programs, ABI compliance is still work in progress for the last three machines. The ABI compliance gives us the ability to link the produced programs with existing C libraries and test real world C programs. We completed this goal for the ARM architecture and we provide a detailed analysis of the results in this section.

We used the ArchC simulation and binary tools generation infrastructure [2], [4] to verify the generated ARM back end. There is no easy way to check if a whole compiler toolchain is completely correct, so we built a test infrastructure using Mibench [14] programs and observed their output. This involves a long process starting at the program C source code and ending with the program output. If the program output is the same compared to running it in a standard Intel IA-32 desktop environment, the program was compiled and executed without failure.

Figure 10 exposes the validation workflow used for the ARM architecture. The C source code for a Mibench program is the input to a LLVM front end. We used llvm-gcc version 4.2.1. This front end converts the C source code to LLVM intermediate representation language, bc (*bitcode*) files. At this point, we can already start analyzing the program using LLVM bitcode disassembly tools, or apply target machine agnostic optimizations using the LLVM opt tool. Whether the bitcode is optimized or not, it is used as input for the LLVM llc tool. This is the LLVM back end. The mechanism described by this paper automatically generates the majority of the code used in llc, the other part comes from the LLVM common library. This infrastructure tests whether llc is generating correct assembly (translated from LLVM intermediate representation) or not. The LLVM version used is 2.5, not recent but stable version that our compiler generator targets.

The output from llc is an assembly language source file. To process this, we used GNU binutils ARM assembler and linker version 2.21. Also, since we are compiling C source files depending on the C standard library, we also link the GNU C library glibc version 2.3.6 together.

At this stage, we recur to ArchC version 2.1 simulator generation tools to automatically synthesize a SystemC compatible functional ARM simulator. We use this simulator to run our

test executable. It emulates a Linux operating system with a thin syscall wrapper that redirects program requests to the host operating system. Using this approach, the output of the program will be delivered to the host operating system and we can compare it with the output for the same program running on a golden model. Table 1 shows the tested Mibench programs for the ARM architecture model. The next section will discuss the performance of the compiled programs.

Program	Description
Quick Sort	Sorting using the <code>qsort</code> standard C library function.
Bit Count	Tests a set of algorithms for bit counting.
CRC32	CRC algorithm for hash calculation.
ADPCM	Signal modulation used in telecommunications.
Dijkstra	Classic algorithm for finding the shortest path in a positive weighted graph.
Patricia	Patricie trie data structure implementation used for storage of information identified by sparse sets of key values.
Rijndael	<i>Advanced Encryption Standard</i> (AES) cryptography algorithm.
SHA	Secure hash function used in cryptography.

Table 1. Tested Mibench programs.

5.3. Generated Code Performance

We measured performance using the number of ARM executed instructions needed to complete the program with the given input. Figure 11 shows this number for each tested program. We eliminated from this count the instructions belonging to the C standard library because this library was precompiled and, in this way, would report the performance of code generated by another compiler. We chose this metric because the instruction count is very close to real hardware performance for ARM in-order chips and working sets that fit in cache.

The graph has two versions, the first one is using Mibench small input and the second one uses Mibench large inputs. In this graph we compare three compilers. The first one, in dark bars, is the ACCGen automatically generated compiler. The second is using the exact same version of LLVM, but using a manually codified ARM back end provided by the LLVM community. The last one is gcc version 3.4.5 (which matches the toolchain for the used glibc 2.3.6, adequate for embedded targets). These results used the target agnostic optimization tool (opt) applied to the intermediate language LLVM bytecode with the “-O3” flag and gcc is also used with “-O3” flag (optimizations enabled).

The generated code performance tested for LLVM manual and synthetic back ends both used the same front end, because this experiment assess solely the back end quality rather than the full compilation toolchain capabilities. Changing the front end would introduce additional factors that could make it difficult to analyze what performance differences arises by using the automated approach. The gcc, on the other hand, uses a different toolchain and its performance report must be analyzed with these considerations in mind.

Some programs compiled with LLVM manual or automatic back end failed to produce correct output and we omitted their bars. On average, ACCGen generated back end code executes 2 times more instructions than the manually codified LLVM

(a) Small input

(b) Large input

Figure 11. Number of executed instructions during the simulation of Mibench programs with three different compilers.

back end. LLVM manual back end code, in turn, executes 1.11 times more instructions than gcc. Our experimental data shows that the automatic approach can’t beat the manual back ends (gcc and LLVM), but the generated back end is far simpler to produce and doesn’t need LLVM or C++ knowledge. Our decision in favor of complete automation impacts the generated code performance, but this impact doesn’t negatively affects the design space exploration of new instructions because the generated compiler successfully recognizes and uses an arbitrary instruction set. It should be observed that a production quality compiler used in building final software needs to be further tuned, and we believe this can be done using a simple peephole optimizer. These performance discrepancies occurs because some programs are matched by the instruction selector using well implemented patterns, while others may abuse occurrences of bad implemented patterns, those for whom the search engine couldn’t automatically find an efficient target machine implementation due to time constraints. In this way, using a generated peephole optimizer can reduce the gap between automatically and manually codified back end code quality.

6. Related work

The problem of automatic synthesis of compilers based on ADL models has been extensively explored since the decade of 1990, and its foundations are based on earlier research on code generator generation. Examples of compiler generation capable ADLs are EXPRESSION [15] and LISA [10]. These ADLs were conceived to allow high-level hardware description and later extended to generate toolchains. EXPRESSION, for instance, can generate VHDL models, but its compiler retargetability support requires nontrivial compiler expertise

from the designer. As opposed to our approach, the designer must manually implement compiler back end patterns. The work presented by Brandner et al. [16] is an example of an ADL designed from scratch to address compiler retargetability. Instead of using program synthesis techniques [12], as in our work, that provide a powerful reasoning engine, they define *templates*, a set of tree patterns useful when some IR fragment could not be directly implemented with a single instruction.

Ceng et al. [17] proposed a LISA ADL extension to enable compiler generation. However, their approach is not fully automatic as proposed in our work, since some patterns need to be hand-written. They define instruction semantics specification to extract instructions behavior information and retarget Cosy, a proprietary compiler suite. ACCGen, our ArchC based compiler generator, targets LLVM, a powerful open source compiler that is on par with gcc.

Dias et al. [18], [19] explored the complete automation of the instruction selector generation using approaches similar to ours. They propose to ease the work of compiler retargeting by automatically generating instruction selector for the C— framework. Their main idea is to define tiles (analogous to our LLVM patterns) and maintain a pool of computable expressions expanded by a search algorithm until the tiles implementation is discovered. Code generated from `gcc -O1` is 2% to 30% faster than code generated from `Quick C—`, their automatically retargeted research compiler. However, it is unclear their time impact on the design exploration cycle. They also do not present a full architecture exploration framework as in our work, which enables easily analyzing the impact of a new instruction on a software workload.

The roots of back end automatic inference begin with Cattell [11]. Our search has similarities with Cattell's in the use of transformation rules, but we have implemented different heuristics and created a framework for parallel hardware to make it faster. Also, we adapted our framework to completely retarget the LLVM compiler and integrated it into the ArchC design space exploration workflow.

7. Conclusions

In this paper, we present ACCGen, a compiler generator for the ArchC ADL. We carefully reduce the information put into an ADL model for compiler generation purposes to the essential in order to make designing as easy as possible. We present the information model used to extend ArchC that is used to automatically infer the rest of the information needed to retarget a production quality compiler like LLVM.

The ARM compiler generated by ACCGen proved to be correct in compiling Mibench applications and is adequate for design space exploration. We also show that the integration of ACCGen in the regular ArchC design flow allows fast design space exploration, giving the designer the ability to test architectural changes impact on the whole platform in less than 2 minutes.

References

- [1] P. Mishra and N. Dutt, *Processor Description Languages*. Volume I. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.
- [2] R. Azevedo, S. Rigo, M. Bartholomeu, G. Araujo, C. Araujo, and E. Barros, "The ArchC architecture description language and tools," *Int. J. Parallel Program.*, vol. 33, no. 5, pp. 453–484, 2005.
- [3] T. Grotker, *System Design with SystemC*. Norwell, MA, USA: Kluwer Academic Publishers, 2002.
- [4] A. Baldassin, P. Centoducatte, S. Rigo, D. Casarotto, L. C. V. Santos, M. Schultz, and O. Furtado, "An open-source binary utility generator," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 13, no. 2, pp. 1–17, 2008.
- [5] R. Auler, A. Baldassin, and P. Centoducatte, "Automatic Architecture Description Language (ADL)-based toolchain generation: the dynamic linking framework," in *SBLP '10: The 14th Brazilian Symposium on Programming Languages*, 2010.
- [6] R. H. Pesch and J. M. Osier, *The GNU binary utilities*, Free Software Foundation, Inc., May 1993, version 2.15.
- [7] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO '04: Proceedings of the international symposium on Code generation and optimization*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75+.
- [8] R. Stallman, *Using the GNU compiler collection*, Free Software Foundation, Inc., May 2004, for GCC version 3.4.3.
- [9] A. V. Aho, M. Ganapathi, and S. W. K. Tjiang, "Code generation using tree matching and dynamic programming," *ACM Trans. Program. Lang. Syst.*, vol. 11, no. 4, 1989.
- [10] A. Hoffmann, H. Meyr, and R. Leupers, *Architecture exploration for embedded processors with LISA*. Kluwer Academic Publishers, 2002.
- [11] R. G. Cattell, "Automatic derivation of code generators from machine descriptions," *ACM Trans. Program. Lang. Syst.*, vol. 2, pp. 173–190, April 1980.
- [12] S. Gulwani, "Dimensions in program synthesis," in *Proceedings of the 12th international ACM SIGPLAN symposium on Princ. and pract. of decl. program.*, ser. PPDP '10, 2010.
- [13] R. Auler, "ADL-Based Automatic Backend Generation," Master's thesis, In Portuguese, Institute of Computing, University of Campinas, September 2011.
- [14] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proceedings of the 4th Annual IEEE Workshop on Workload Characterization*, December 2001, pp. 3–14.
- [15] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau, "EXPRESSION: A language for architecture exploration through compiler/simulator retargetability," *Design, Automation and Test in Europe Conference and Exhibition*, vol. 0, p. 485, 1999.
- [16] F. Brandner, D. Ebner, and A. Krall, "Compiler generation from structural architecture descriptions," in *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, ser. CASES '07. New York, NY, USA: ACM, 2007, pp. 13–22.
- [17] 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*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 1150–1155.
- [18] J. a. Dias and N. Ramsey, "Automatically generating instruction selectors using declarative machine descriptions," *SIGPLAN Not.*, vol. 45, pp. 403–416, January 2010.
- [19] N. Ramsey and J. a. Dias, "Resourceable, retargetable, modular instruction selection using a machine-independent, type-based tiling of low-level intermediate code," in *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '11. New York, NY, USA: ACM, 2011, pp. 575–586.