확장형 컴퓨팅 구조를 위한 범용 컴파일러 설계

타로파 에마뉴엘, 이원종, 바슨 스리니, 한탁돈
연세대학교 컴퓨터 과학과
{etaropa[O], airtight}@yonsei.ac.kr
{srini, hantack}@cs.yonsei.ac.kr

Designing a Generic Compiler for Scalable Computing Fabric

Emanuel Taropa[0], Won-Jong Lee, Vason P. Srini, Tack-Don Han
Department of Computer Science, Yonsei University

Abstract

A blooming area of processor design is represented by scalable computing fabric. As the structure of the processors developed using scalable computing fabric evolved from simple programmable units to processors supporting change of flow instructions and function calls, an increasing interest is in developing the compiling technology that will allow us to harness not only the full power of their hardware but also to target multiple architectures. In this paper we present the front-end of a generic compiler, able to accept a various source languages and transform them in a common intermediate representation.

1. Introduction

Initially designed to replace pin-point functionality, FPGAs evolved into having an impressive computational density among today's programmable chips. Combining multiple FPGAs into complex designs yields nowadays a *high-performance, cost-effective* solution to the computational challenges raised by the ever increasing size of the datasets to be processed.

The scalability and performance of FPGA based systems comes from harnessing the spatial parallelism obtained by implementing a large number of simple functional units on the same FPGA. Their design is thus simple, efficient, self-contained and easy to templetize.

As top of the line FPGAs[1] can achieve performance and power efficiency close to the one offered by ASICs while being fully programmable, an increasing interest is in developing a compiler technology able to target scalable computing fabric (SCF) [2] resulted from using platform FPGAs. This compiler needs to be *extensible, modular, re-usable* and *efficient.*

Retargetable, optimizing compilers for modern architectures have been previously analyzed [3], [4] and implemented [5], [6]. Clustered VLIW [7] and reconfigurable architectures and their associated compilers [8] have been extensively researched during the past few years. Although these compilers generate efficient code for their target architectures, they suffer on the configurability and re-usability sides.

In this paper we describe a generic compiler front-end able to accept a comprehensive array of high level source languages (Cg, Matlab, HPF) and to translate them in a common intermediate representation (IR), fit for high-level optimizations and easily targetable to virtual component libraries. Momentarily in a prototype stage, where basic features of the compiled languages are supported and restrictions are imposed on typing rules, the compiler will support the full specification of the source languages and generate optimized FPGA representations of source programs. The front-end design is detailed by section 2. Section 3 describes the main structures used by the IR and motivates our choices. Section 4 contains a sample source program, its transformed representation in IR and its VCL template. The paper concludes with section 5 where conclusions are drawn and future directions are presented.

2. Front-end Design

An essential part in the compilation chain, the front-end contains the array of transformations starting with the lexical analysis of the source language(s) and ending with the Medium Level IR (MIR) generation. Most of the high-level optimizations are implemented starting from the High Level IR (HIR) and ending with close to machine specific optimizations performed over the MIR. These optimizations greatly influence the quality of the code generated by the machine dependent back-end (or in our case by the virtual component selector).

As developing language we used Java [9] and as IDE the open source Eclipse [10]. For generating the lexical analyzer and parser code, we used ANTLR [11] which accelerated the process of writing the grammars and their productions for the compiled source languages. A representation of the compilation chain is given in Figure 1. Our compiling architecture is represented in Figure 2.

2.1 Lexical Analyzer

The lexical analysis phase was straight forward. We used publicly available grammars describing the structure of the compiled source languages. The ANTLR definition of the lexical analyzers, for each of the compiled source languages simply followed the publicly available grammars, making few integration modifications. The output of the lexical analysis is a stream of tokens, not including comments and spacing characters, is fed to the syntax analyzer.

2.2 Syntactical Analyzer

Similarly to the lexical analysis, for the syntax analysis phase we had to develop multiple analyzers, for each of the compiled source languages. Using ANTLR we were able to have relatively readable parsers that were self contained and offered a unitary interface to the constructed abstract syntax trees (ASTs).

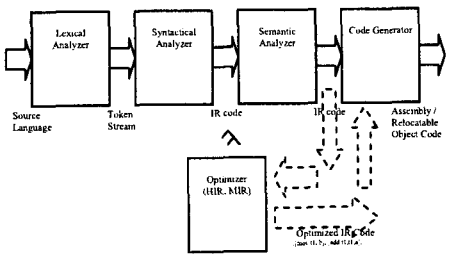

Figure 1: The compilation chain, broken into main passes.

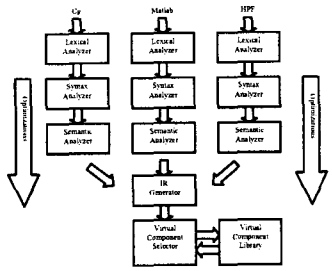

Figure 2: The compiling architecture.

A crucial role for the parsing efficiency is played by the symbol table (ST). Organized in multiple levels corresponding with the scope of the compiled languages, the ST is a unique component used by all compilation chains. A good equilibrium between space and efficiency is realized using a hashing structure, where the in-memory address of each node is used as a hash key. For our single threaded, single memory, compilation model this worked fine. For a distributed compilation system which may potentially run on multiple machines the memory address of an object obviously no longer suffices. A naming scheme including the machine name and the memory address can then be used. This would trade space (by increasing the hash key size) for efficiency (keys will remain unique).

The output of the syntax analyzer is represented by an AST. This represents the structure of the source language at a very high-level, keeping all the information about loops, function calls and array indexing. The advantage of structuring the source code as an AST is revealed by the next compilation phases, where by traversals of this tree we realize type inference and type promotion and high-level optimizations.

2.3 Semantic Analyzer

The AST produced by the syntactical analyzer is used as input for this compilation phase. The output of this phase is represented by a "decorated" AST, which is an AST containing type information. As different typing rules apply for each of the compiled languages, we had to build three different semantic analyzers. The type promoters had a common structure by deriving from the generic TreeWalker class offered by ANTLR.

Out of the three languages, MATLAB's type inference was the most difficult. For this initial stage of development, we support a subset of MATLAB's type promotion mechanism. An example of compiling MATLAB into an FPGA is given in [12].

For a rapid prototyping stage, we supported first the fundamental types in all of the three compiled languages. We wrote type inference rules for the following types: *byte* (or character, with restrictions), *integer* (represented on 16, 32, 48 and 64 bits), *floating point numbers* in single and double precision. All these types are supported both at single variable level as well as within arrays and assignments to these arrays. Implicit casting from a type to another one is supported as described by each source language typing rules. We do not allow implicit casting for composed types and inherently, the supported explicit casting is implemented as reinterpret casting.

2.4 IR Code Generator

The decorated AST produced by the semantic analyzer is then traversed and transformed in MIR. The unification point of the three compile chains, the IR code generator outputs three-address statements, easy to map on virtual components.

2.5 Virtual Component Selector

The virtual component selector (VCS) interacts with a virtual component library (VCL). Initially, the VCL contains the templates for basic operations (i.e. templates for adders, multipliers, registers, buses etc). Each of the components is characterized by various parameters, most important being the size of the occupied area on the FPGA, the power consumption and the latency. Being configurable, these parameters help us estimate the cost of the resulting design in the context of a given technology.

Reading the optimized IR, the VCS queries the VCL for available components. If none is found, a new component expressing the required transformation should be created and stored in the VCL.

3. IR Constructs and Optimizations

The IR code generator starts with walking the decorated AST and transforming it to a linear (or canonic) form. A simple example is illustrated in Figure 3 which presents the transformation of a loop node in MIR.

The main abstractions used for MIR are similar to the ones presented in [5]. We present the most important in Table 1.

Instruction Type	MIR
ASSIGN	V <- Expression
GOTO	<goto: label>
IF conditional_expression THEN	<if: cond, label_true_block, label_false_block>
CALL	<call: procname, args>
RETURN	<return: v?>
SEQUENCE	<sequence: MIR instructions>

Table 1: Some of the standard resources / instructions and their translation in MIR.

As the common MIR is suitable for most optimizations done in compilers, we used it to evaluate the following: *dependence analysis* and *dependence graphs* – for various graph algorithms we developed our own library, *tail call optimization, register allocation and assignment.*

Using the same MIR for multiple source languages forced us to identify common traits across them and to construct the IR following them. We combined array indexing and referencing, calling conventions (with the typing subset imposed to MATLAB and inlining Cg procedures), loop expressions (with subsets for loops expression in HPF) and expression evaluation (again, taking into account the typing subset imposed to MATLAB). One of the most important gains we had using the generic MIR design is that supporting another source language is fairly trivial, by

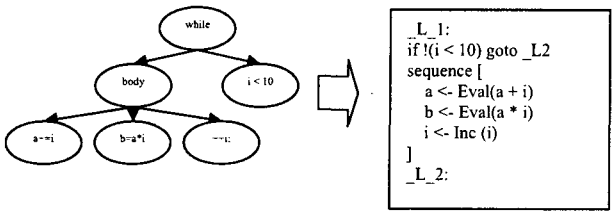

Figure 3: Compiling a WHILE loop in MIR. "Eval" stands for the expansion of its argument of type expression in MIR. (e.g. Eval (i + 1) ⇔ tempvar_n <- i + 1, where tempvar_n is a generated temporary corresponding to "i+1" 's parent node in the decorated AST)

implementing only the first three links of our compiling chain: the lexer, the parser and the type promoter.

Some of the optimizations we are currently implementing include: *static single assignment, strength reduction, loop unrolling and loop fusion.* We target these optimizations for their high impact on the quality of the generated design.

4. Examples of Transformations

Out of the whole array of transformations applied to the source language, we selected a few illustrating the critical parts of the compile chain: *type inference, MIR generation and virtual component selection.*

Let's consider the following simple source program:

```
int16 i = 0; float a = 0.0; double b = 1.0;
while (i < 10) {
    i += 1;
    a += i;
    b = a * i;
}
```

Its transformed representation in MIR is illustrated by Figure 3. Trivially, type inference applied for the AST inner nodes corresponding to the operators of the source statements implicitly casts the induction variable from int16 to single and double precision floating point number. The encompassing type is then promoted to the respective operators nodes ('+' has float type and '*' has double).

The MIR code is then fed to the VCS which interacts with VCL. Let's assume we already have pre-generated in the VCL the template for adders and multipliers. The resulting virtual component template is given below:

```
IntRegister[16] R0;
FloatRegister[32] R1;
FloatRegister[64] R2;
Adder(Float [R1], Int[R0], Float [R1]);
Multiplier(Float [R1], Int [R0], Double[R3]);
```

5. Conclusions and Future Work

We have presented a compiling architecture accepting various high level source languages and targeting a virtual component library. Having a modular design, the compilation chain is easily modifiable and high level optimizations are easy to implement and apply to the IR code. The effect of these optimizations is directly reflected by the efficiency of the generated components and final design, implementing in FPGA the initial high level program.

The current status of the implementation is only an initial one, as only subsets of the source languages are supported (especially for MATLAB's type inference and promotion mechanism) and as the VCS and VCL chain doesn't yet support combining complex operations into composed components synthesized from the initial basic blocks.

The completion of the VCL and VCS chain along with the generation of complex designs and their implementation in FPGA are our next goals.

Acknowledgments

The authors whish to thank the Advanced Compiler Design class (spring 2005) students from Yonsei University, for their industriousness and diligence in tackling the most delicate problems in designing subparts of the generic compiler. Special thanks go to Yi Jaeheon, Kang Jun-Seok, Tom Yum.

REFERENCES

[1] Virtex II and Virtex II Pro Platform FPGAs - http://www.xilinx.com/bvdocs/publications/ds083.pdf

[2] Dataflux Systems Inc, Berkeley http://www.datafluxsystems.com/index.php?content=technology/scf

[3] A.V. Aho, R. Sethi, J.D. Ullman, S. Guthe, T. Ertl – Compilers: Principles, Techniques and Tools,Addison-Wesley, 2nd edition, 1986

[4] Steven S. Muchnick – Advanced Compiler Design Implementation, Morgan Kauffman, 1997

[5] Z. Bozkus, A.N. Choudhary, G. Fox, T. Haupt, S. Ranka – Fortran 90D/{HPF} compiler for distributed memory {MIMD} computers: design, implementation, and performance result, Supercomputing, 1993, pp. 351 – 360

[6] W. Mark and S. Glanville and K. Akeley – CG: A system for programming graphics hardware in a C-like language, ACM Transactions on Graphics, August 2003

[7] A. Therechko, M. Garg, H. Corporaal: Evaluation of Speed and Area of Clustered VLIW Processors, 18th International Conference on VLSI Design, pp. 557 – 563, 2005.

[8] Y. Qian, S. Carr, P. Sweany - Optimizing loop performance for clustered VLIW architectures, Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques, pp. 271 – 280, 2002

[9] Sun – The Java Programming Language Reference http://java.sun.com/j2se/1.4.2/docs/api/index.html

[10] Eclipse – An Open Source IDE http://www.eclipse.org

[11] ANTLR – Parser Generator http://www.antlr.org

[12] Xilinx – Xilinx System Generator Users Guide Version 8.1 http://www.xilinx.com/products/software/sysgen/app_docs/user_guide.htm, Chapter 3, Section 8.