COMPILER CONSTRUCTION FOR A NETWORK IDENTIFICATION

Sumit Soni*¹, Pravesh Kumar²

**Corresponding Author: -*

Abstracts:

The Compiler Construction is mainly used to improve the network processing system through the advance advance compiler design. The name Compiler is primarily used for programs that translate source code from a high –level programming language to a lower level language. The advance compiler design is useful in the industrial application like telecommunication. Our approach to increase the flexibility, productivity and usability of the system to target with the minimal effort. we use the bit level addressing for the network processor we show how a compiler backend has been design and also improve the feature of the compiler and fully operational. A compiler is a computer program (or set of programs) that transforms source code written in a programming language (the source language) into another computer language (the target language, often having a binary form known as object code. The most common reason for wanting to transform source code is to create an executable program. scheduling, code generation.

Keywords: *Hardware and software, compilers, network processor, reconfiguration*

Distributed under Creative Commons CC BY-NC 4.0 OPEN ACCESS

INTRODUCTION:

A program that translates from a low-level language to a higher level one is a decompile. A program that translates between high-level languages is usually called a Source-to-source compiler or transpiler. A language rewriter is usually a program that translates the form of expressions without a change of language. The term compiler-compiler is sometimes used to refer to a parser generator a tool often used to help create the lexer and parser.

A compiler is likely to perform many or all of the following operations: lexical analysis, preprocessing, parsing, semantic analysis, code generation, and code optimization. Program faults caused by incorrect compiler behavior can be very difficult to track down and work around; therefore, compiler implementors invest significant effort to ensure compiler correctness

Executing a program written in a high-level programming language must be compiled that is, translated into the object program. Today, however, compilers can be built with much less effort. we have to improve the compiler for network processor. The purpose of this paper is to show how an efficient compiler for an advanced architecture has been implemented and to describe the required machine-specific code generation techniques. More specifically, we show how bit packet processing is made available to the programmer at the C level and how the register allocator needs to be designed to handle variable-length bit packets in registers, which is not directly possible by different techniques. The compiler then analyses the program, partitions it into hardware and software, and generates data paths for the reconfigurable hardware. We use the code generation for the final phase of the compilation .it convert the intermediate code in to machine instructions. We use the code generation for irregular data path. The compiler is design for the network processor, the network processor is known as the bit - Serial protocol processing

The design of advance compiler has been motivated by the growing need for new high bandwidth communication equipment in networks (e.g., Internet routers and Ethernet adapters) as well as in telecommunication (e.g., ISDN and DSL). The corresponding communication protocols mostly employ bit- stream-oriented data formats. The bit streams consist of packets of different length, there are variable length header packets and payload packets.

A major system design problem in this area is that the required high bandwidth leaves only a very short time frame for processing each bit packet arriving at a network node. Even contemporary high-end programmable processors can hardly keep pace with the required real-time performance, not to mention the issue of computational efficiency.

In the figure the transmitter transmits the data to the memory and then the memory takes the token which is provide by the compiler phase (lexical analyzer). then the data is sent to the buffer where the all data will be stored in the buffer after that the signal will be send to the network stream and then goes to the receiver end the same process will occurs.

Compiler Architecture:

Like most other compilers, the NP compiler is subdivided into a frontend and a backend part. The frontend is responsible for source code analysis, generation of an intermediate representation, and machine-independent optimizations, while the backend maps the machine-independent intermediate representation into machine-specific assembly code. The current compiler partitions the application into software and hardware and produces Data Flow Graphs (DFG) for the latter. Although it works satisfactorily and obtains good results, it has potential for improvements. Particularly, the heuristics for the selection of program partitions for hardware, as well as the generation of data paths can profit from further work. The compiler is consisted of different phases

Overview

Compilers perform translation. Every non-trivial translation requires analysis and synthesis

Both analysis and synthesis are made up of internal phases.

During semantic analysis we have to check legality rules and while doing so, we tie up the pieces of the syntax tree (by resolving identifier references, inserting cast operations **for implicit coercions, etc.**) to form a semantic graph.

c-code:

this is the code in which the user gives their input in the help of c programming. This extended software part is finally output as C- code. The data paths are pre-placed by a dedicated tool developed specifically for this purpose. Routing is performed by the standard Xilinx M3 suite.

Lexical analysis:

the lexical analyzer is the interface between the source program and the compiler. the lexical analyzer read the source program one character at a time. Carving the source program into a sequence of atomic unit called 'TOKEN'. Each token represents a sequence of characters that can be treated as a single logic entity

IF (5.EQ.MAX) GO TO 100

We find the following eight tokens:

IF; (; 5; EQ.; MAX;); GOTO; 100.

OPTIMIZATION:

Object programs that are frequently executed should be fast and small certain compiler have with in them a phase that tries to apply transformation to the output of intermediate code generator. The term optimization in this context is a complete misnomer, since there is no algorithmic way of producing a target program Our first optimization steps rely on proven Machine-independent methods. We employ Scalarization, Software Pipelining, Code Movement and other techniques. In this manner, we reduce memory accesses in loops and attempt to create inside loops having a high degree of instruction level parallelism.

Code generation:

The code generation phase converts the intermediate code into a sequence of machine instruction. a single-minded code generator might map the statement $A := B = C$ into machine code sequence

```
LOAD B
ADD C
STORE A
```

Selection of Hardware Building Blocks:

the compiler selects operations for a later hardware implementation on the high-level representation. This is feasible since HW operations will not be affected by lower-level transformations. For evaluating the HW- suitability of SW operations, we rely on estimation data available through the FLAME interface. This flexible interface allows access to hardware-specific libraries, and offers hardware-relevant data such as area and time requirements of operations, availability of synthesizable components as well as chip resources. Partitioning relies mainly on the data collected in this step.

TARGET ARCHITECTURE:

the overall architecture of our target machine, the Infineon NP. The NP core shows a 16-bit RISC-like basic architecture with 12 general-purpose registers and special extensions for bit-level data access. This principle is illustrated in Fig. 3. The NP instruction set permits performing arithmetic logic unit (ALU) computations on bit packets that are not aligned at the processor word length. A packet may be stored in any bit index sub range of a register and a packet may even span up to two different registers. In this way, protocol processing can be adapted to the required variable packet lengths instead of the fixed machine word length. However, this *packet-level addressing* is only possible within registers, not within memory. Therefore, partial bit streams have to be loaded from memory into registers before processing on the ALU can take place. The size and position of the different bit fields are statically known from the C source code for each specific application. In order to enable packet-level addressing of unaligned data, the NP instruction set permits the specification of offsets and operand lengths within registers. The general instruction format is as follows:

CMD reg1.off, reg2.off, width.

“CMD” denotes the assembly command, “reg1.off” and “reg2.off” denote argument registers with a corresponding offset, and “width” is the bit width of the operation to be performed. The use is shown in Fig. 5: any bit packet is addressed by means of the corresponding register number, its offset within the register, and the packet bit width. If offset plus width are larger than the register word length (16 bits), then the packet spans over two registers (without increasing the access latency, though). This feature is very challenging especially from a compiler designer’s viewpoint. The width of argument and result packets must be identical and one of the two argument registers is also the result location of any ALU operation. Therefore, two offsets and one width parameter per instruction are sufficient.

Fig2. Processing of variable-length bit packet.

Fig 3. Data layout in memory and registers. Bit packets are not aligned at memory or register word lengths.

Code selection:

The code selector uses the widespread technique of tree pattern matching with dynamic programming for mapping data flow trees (DFTs) into assembly code. The basic idea in this approach is to represent the target machine instruction set in the form of a cost-attributed tree grammar, and parsing each given DFT with respect to that grammar. As a result, an optimum derivation for the given cost metric, and hence an optimal code selection, are obtained. The runtime complexity is only linear in the DFT size.

RESULT

The advance compiler for the NP described in THESE sections is fully operational. The performance of the generated code has been measured by means of a cycle-true NP instruction-set simulator for a set of test programs. As may be expected, the quality of compiler-generated code as compared to handwritten assembly code. the adv compiler operate the NP in different phases which is already describe.

When using CKFs without specific knowledge of the application, the performance overhead of compiled code may be several hundred percent, which is clearly not acceptable for the intended application domain.

CONCLUSION AND FUTURE WORK:

We have a lot of problem in NP. the compiler challenges encountered for NPs; a new class is created that allow for efficient protocol processing by means of packet-level addressing. We have described the implementation of a advance compiler for a real-life industrial NP. In order to make packet-level addressing accessible at the C language level, the main concepts in this compiler are the use of CKFs and a special register allocation technique, and helpful in the

telecommunication these techniques work in practice, so that the processor features are well exploited. Although the detailed implementation is machine-specific, we believe that the main techniques can be easily ported to similar NPs for which a growing compiler demand may be expected in the future. Ongoing work deals with gradually replacing the pragmatic approach of CKFs with more sophisticated code selection techniques, capable of directly mapping complex bit masking operations into single machine instructions. This will be enabled by the use of special tree grammars that model the instruction

REFERENCES.

- [1].principles of compiler design by Alfred v Aho, Jeffery D. Ullman
- [2].Xtensa RISC processor, Tensilica Inc. [Online]. Available: <http://www.tensilica.com>
- [3].3. R. Leupers, *Code Optimization Techniques for Embedded Processors*. Norwell, MA: Kluwer, 2000.