Code Generation and Optimization for the Flow-based Network Processor based on LLVM[1]

SangHee Lee*, Hokyoon Lee*, Seon Wook Kim*, Hwanjo Heo**, Jongdae Park**
*School of Electrical and Computer Engineering, Korea University
**Net. Computing Convergence Research Team, ETRI
e-mail: {lshron, hokyoon79, seon}@korea.ac.kr, {hwanjo, parkjd}@etri.re.kr

Abstract

A network processor (NP) is an application-specific instruction-set processor for fast and efficient packet processing. There are many issues in compiler's code generation and optimization due to NP's hardware constraints and special hardware support. In this paper, we describe in detail how to resolve the issues. Our compiler was developed on LLVM 3.0 and the NP target was our in-house network processor which consists of 32 64-bit RISC processors and supports multi-context with special hardware structures. Our compiler incurs only 9.36% code size overhead over hand-written code while satisfying QoS, and the generated code was tested on a real packet processing hardware, called S20 for code verification and performance evaluation.

1. Introduction

With the advent of ubiquitous computing, traffic volumes for Internet Protocol (IP) have been dramatically increasing for both fixed and mobile networks [6]. On this protocol, all information in forms of voice, images, video, text, and so on are transformed into sequences of packets, and then the packets are delivered through high speed network links where network nodes share link capacity among many clients and route traffic efficiently. Until recently, the network nodes were made by fixed ASICs. However, with the wide range of requirements for network nodes over time, the custom hardware has become a very expensive solution for service providers. For this reason, the trend has been to give more and more emphasis on programmability inside packet processors, and finally a network processor (NP) appears [15]. Currently a large number of network processors are commercially available in a market, such as Intel IXP2800 [5], EZchip NP-4 [4], and so on.

Even though we rely on great functionalities of a network processor for network node development, the software development is not trivial. Most vendors provide C compilers with optimized libraries, but they recommend a programmer to use micro codes instead of high-level languages [3]. The complex architecture and resource constraint on NPs make compiler development difficult. For example, network processors consist of a multi-context structure and need special instructions for handling the packet processing and bit unit operations.

In this paper, we introduce C compiler construction and optimization for our network processor, called OmniFlow [14], based on LLVM 3.0 [13, 12]. Similar to other network processors [10] the OmniFlow processor has several constraints for compiler development. This processor does not support some popular instructions such as multiplication, load with sign extension, etc. A data memory, a stack memory, and a function call are not allowed, and instruction code size is limited. Also, when we perform load/store packet or flow blocks, registers have to be serially allocated. In this paper, we describe how to overcome these constraints and support various optimizations in detail. We also built an assembler based on the GNU binutils [11]. For performance evaluation, we tested network packet translation with S20 chassis [2] and IXIA's IxN2X network testing system [1]. S20 chassis is a router that contains the OmniFlow processor, and IxN2X is a throughput measuring instrument. Our compiler incurs only 9.36% code size overhead over hand-generated code while satisfying QoS.

This paper is organized as follows: Section 2 describes our OmniFlow processor organization and critical issues for compiler construction due to hardware constraints. Section 3 explains code generation and optimization, and Section 4 shows the correctness verification and performance evaluation. Finally, conclusion is made in Section 5.

2. Omniflow Network Processor

The OmniFlow processor is a network processor, which is an array of 32 64-bit RISC processors specialized for packet processing and multi-context operation. Each context has a register file, a program counter, and a context controller state machine. The register file has 64 32-bit registers.

Different from compiler construction for general processors, there are critical issues to be solved for the compiler development as follows:

1. We need to support special instructions such as bit field manipulation and packet load/store instructions. On the contrary, the target processor does not provide widely used instructions in general processors such as multiplication, truncate store, and sign extended load.
2. Our target hardware does not use any data memory for data and stack sections, but only for packet and flow. Therefore, a programmer cannot use local and global variables. Similarly, the target hardware does not allow a register to be spilled because there is no stack.
3. The target processor does not use a procedure call, and therefore, a caller/callee convention is not used.
4. The target hardware has the limited size of a code memory, which allows only 1024 lines of assembly codes.

[1] This work has been supported by ETRI OmniFlow processor compiler toolchain performance improvement (EA20121517).

3. OmniFlow Processor Compiler Development

We used LLVM 3.0 for our compiler construction. The modified and extended components of the LLVM compiler framework are shown in Figure 1.

We modified the frontend for overlay optimization and registration of target dependent information. LLVM IR and SCCP (Sparse Conditional Constant Propagation) [9] optimization were extended for supporting the overlay optimization. We made many efforts for developing the backend such as 1) describing target information and selection rules, 2) supporting special instructions such as overlay, packet load/store, maskcmp etc, 3) printing error messages for unsupported instructions, 4) delay slot optimization. We explain these implementations in detail in the following subsections.

3.1 Frontend and Extended LLVM IR

We used Clang 3.0 [8] for our frontend built in the LLVM package. In the frontend we registered the OmniFlow target specific information class and its associated information such as endian, data format, and register names. This information is used for IR code generation.

Also, we performed overlay optimization in the frontend. The overlay instructions handle bit assignment and shift operations that are frequently used for packet processing. We introduced overlay IR for the instructions. Our compiler analyzes LLVM IR (AST, Abstract Syntax Tree) and builds the overlay IR.

3.2 Backend

In the backend, compiler construction and modification steps are the following: First, we registered the processor information such as the processor name, endian, and data format. Second, we described register information. We set dummy numbers to a stack pointer register and a frame pointer register. Also, a return address register was not defined because a function call is not allowed. Third, we defined target processor instruction sets. We needed to define selection rules for overlay instructions, maskcmp instructions, and special instructions such as packet load/store. Also, error messages should be printed out for unsupported instructions such as multiplication, truncate store, etc. Finally, in the backend, we described the OmniFlow assembly string formats for code generation. For conditional branch instructions, MASKCMP and MASKCMPI, there are different patterns depending on their condition value.

3.3 Code Optimization

We support three code optimizations: overlay optimization for bit field movement, maskcmp optimization for bit field comparison, and delay slot optimization for removing pipeline stalls. We explain the optimizations in detail in the following subsections.

3.3.1 Overlay Optimization

Figure 2 explains formats and operations about overlay instructions, OVERLAYR and OVERLAYL. Each register (Rs and Rd in Figure 2) has a bit field mask for extracting or merging a bit field value. The OVERLAYL instruction performs the following steps at once.

1. Extract the source register bit field value.
2. Initialize the destination register bit field value.
3. Shift the extracted source registers value to the left.
4. Merge the two bit field values and, save this result at destination register.

The use of overlay instructions can reduce the generated assembly code size and execution time significantly. So, in order to generate as many overlay instructions as possible, we modified the frontend. We found three patterns of bit field operations from AST which can be applied to overlay instructions.

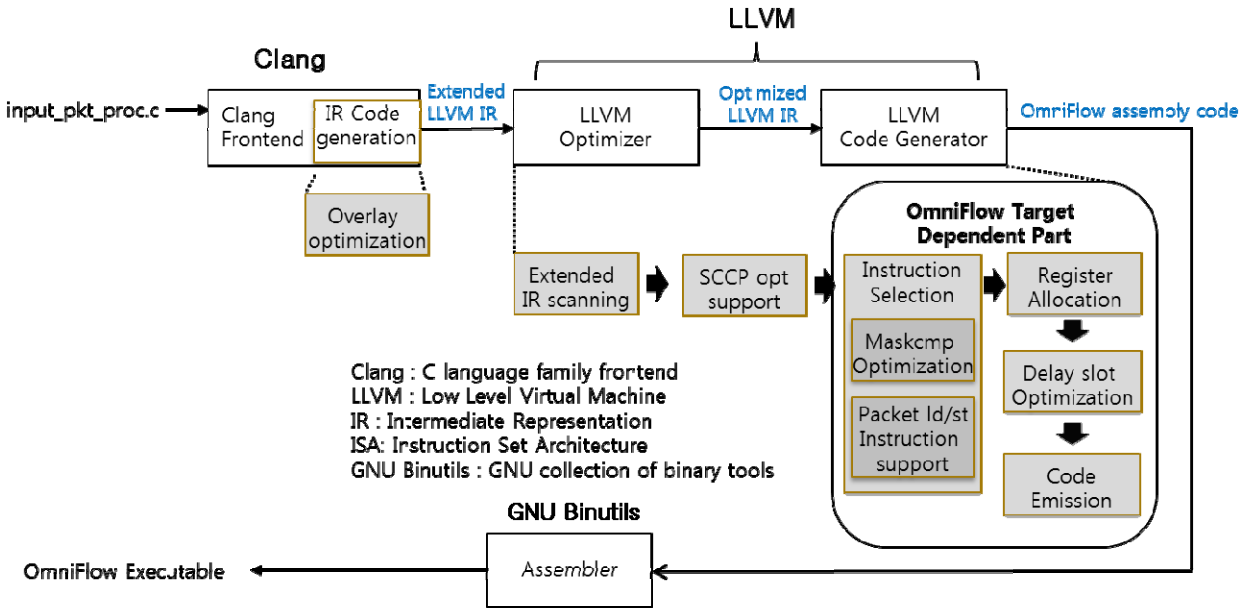

Figure 1. Overview of the OmniFlow processor compiler framework. The gray colored components were modified and extended for the OmniFlow processor compiler.

Instruction Format								Operation
Overlayr (MOVR)	Rd	Rd mask Hi bit	Rd mask Lo bit	Rs	Rs mask Hi bit	Rs mask Lo bit	Number (Shifts)	Shift Right and overlay A = (REG[Rd] & ~(Rd::MASK); B = (REG[Rs] & (Rs::MASK) >> Number; REG[Rd] = A | B;
Overlayl (MOVL)	Rd	Rd mask Hi bit	Rd mask Lo bit	Rs	Rs mask Hi bit	Rs mask Lo bit	Number (Shifts)	Shift left and overlay A = (REG[Rd] & ~(Rd::MASK); B = (REG[Rs] & (Rs::MASK) << Number; REG[Rd] = A | B;

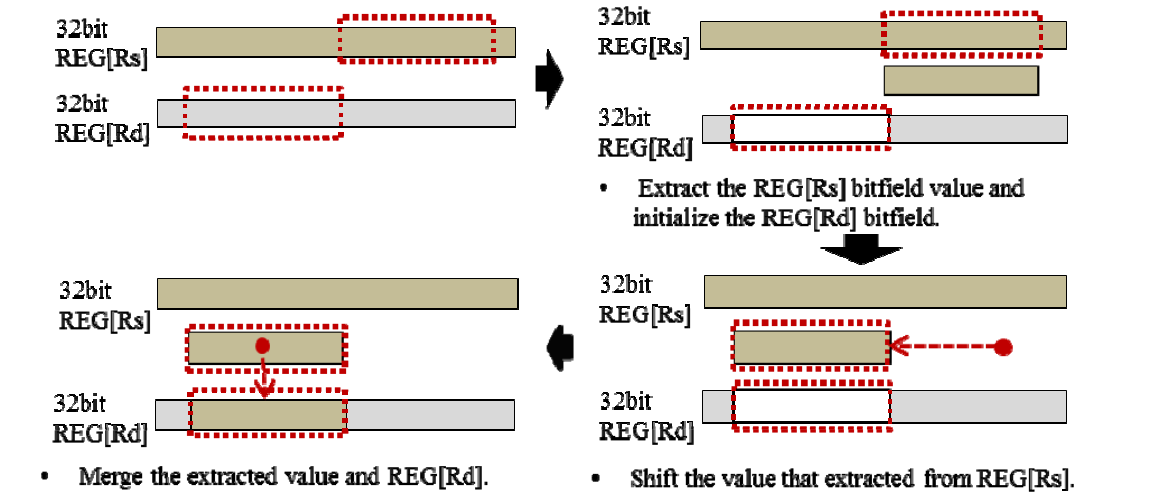

Figure 2. Overlay instruction format and OVERLAYL operation. OVERLAYR and OVERLAYL perform these complex operations at once.

1. Bit field assignment: This AST pattern performs bit field data movement from a source bit field value to a destination bit field value.
2. Bit field assignment after constant shift: This pattern is basically the same with bit field assignment, but it performs shift operation before bit field value movement.
3. Bit field use/def patterns: Other patterns, which are not included in the first and second patterns.

As a result of this optimization, bit field manipulation code size can be reduced to 1/4.

At the first time, we could not generate the OVERLAY instruction when the size of bit field is larger than 32-bit, because there is no algorithm for finding accurate data location in the large and complex bit field structure. It is a critical problem, when managing large and complex bit field structures. So, we added algorithms for solving this problem, by considering a field index and a target offset.

3.3.2 Maskcmp Instruction

MASKCMP and MASKCMPI are bit field comparison and conditional branch instructions. These instructions can compare bit field values without help from other instructions. MASKCMPI has one target register to be compared with zero in a bit mask range. MASKCMP instruction is similar with MASKCMPI instruction. This has one more register, and it compares two registers in a bit mask range, instead of zero. These instructions have 3 address values that are jumping targets. These conditional branch instructions select one of three jump addresses after bit field comparison.

Without support of MASKCMP instructions, target bit field values have to be extracted and then compared for the bit field comparison. Therefore, the use of these maskcmp instructions can reduce register usages and number of executed instructions.

3.3.3 Delay Slot Optimization

We performed the delay slot optimization for RET and unconditional jump operations. There are two delay slots after these instructions, and we inserted two NOPs at the slots. We made delay slot optimization pass, where we replaced NOPs with independent instructions. This optimization can reduce the code size significantly.

4. Evaluation

4.1 Verification

We evaluated our compiler implementation in a testbed environment which consists of the S20 chassis and the IxN2X network testing system. The S20 chassis is a commercial service controller equipped with two OmniFlow processors, one for the input and the other for the output packet processing, and the chassis has been developed in collaboration of ETRI and Sable networks. Three gigabit Ethernet ports of S20 were connected to the three ports of IxN2X with optical fibers. Given the egress flow of GE-0/0/1,

each ingress flow of GE-0/0/2 was configured to receive 700kbps guaranteed rate service whereas the ingress flows of GE-0/0/0 did not receive any guarantee, i.e., the best effort service.

The output queue, where packet were waiting to be sent out to the egress port, of the system was implemented as a calendar queue which had a cylindrical shape with a time slot duration of 16us by default. The QoS binary determines how often packets of the given flow are needed to be scheduled to meet the service requirement, e.g., 700kbs guaranteed rate service, by accessing the flow block data structures in DRAM and computing time slot intervals with the floating point arithmetic operations given the observed flow arrival statistics.

For compiler evaluation, we translated a hand-written original assembly code into C code manually, and we used our compiler for assembly code generation. Then we compared the original code with the compiler generated codes.

4.2 Code Size

Figure 3 shows the code size overhead of different code generations with respect to the hand-written original codes. Overall, the size of fully optimized code by our compiler was increased only 9.36% against the original code. We can see that MASKCMP and OVERLAY instructions greatly impact on the performance, but the delay optimization does not.

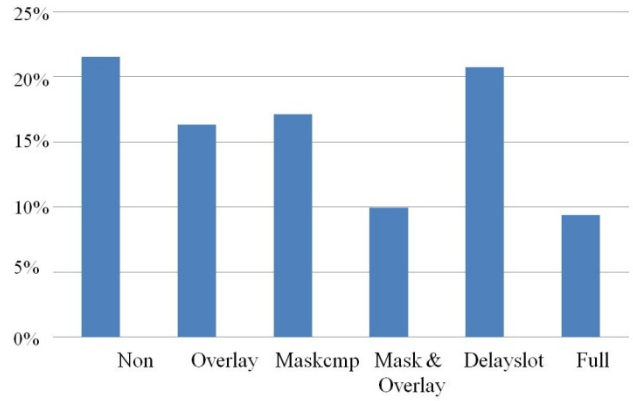

Figure 3. Code size overhead with respect to the hand written codes.

5. Conclusion

In this paper, we introduced compiler construction based on the LLVM 3.0 compiler infrastructure for the OmniFlow network processor. Different from a general processor, NP has many hardware constraints and supports special instructions for QoS packet flow processing. Therefore, most NP codes are hand-written. This paper describes in detail about compiler construction issues and their solutions with optimizations for NP.

For performance evaluation, we tested QoS control code on a real router, S20 chassis. Our compiler incurs only 9.36% code size overhead over hand-generated code while satisfying QoS.

참고문헌

[1] IXIA. IxN2X. http://www.ixiacom.com/products/ixn2x/index.php.

[2] Sable Networks. S20 chassis. http://www.sablenetworks.com/idex.php/en/products.

[3] S. Meijer, J. Walters, D. Snuijf, B. Kienhis, "Automatic partitioning and mapping of stream-based applications onto the Intel IXP Network Processor," In *Proceeding of the10th International Worshop on Software & Compilers for Embedded Systems*, pages 23-30, Nice, France, April 2007.

[4] EZCHIPS. NP-4. 100-Gigabit Network Processor for Carrier Ethernet Applications. http://www.ezchip.com/Images/pdf/NP-4_Short_Brief_online.pdf.

[5] M. Adiletta, M. Bluth, D. Bernstein, G. Wolrich, and H Wilkinson, "The Next Generation of Intel IXP Network Processor," *Intel Technology Journal*, vol. 6, no. 3 pages 6-18, August 2002

[6] J. S. Marcus. "IP interconnection, traffic trends, and wholesale and retail prices," *BEREC/OECD expert workshop on IP Interconnection, Brussels*, Belgium, November 2011.

[7] J. Jones. "Abstract Syntax Tree Implementation Idioms," *The 10th Conference on Pattern Languages of Programs 2003*, Illinois, September 2003.

[8] C. Lattner. “LLVM and Clang: Next generation compiler technology,” *In BSDCan 2008: The BSD Conference*, Ottawa, Canada, May 2008.

[9] M. N. Wegman, and F. K. Zadeck. "Constant propagation with conditional branch," In *Proceeding ACM Transactions on Programming Languages and Systems*, vol. 13, issue 2, pages 181-210, April 1991.

[10] T. Sassen, N. Ventura, and S.Shepstone. "The Implementation of a Differentiated Services Architecture on Network Processors," *Southern African Telecommunication Networks and Applications Conference*, Spier Wine Estate, Western Cape, South Africa, September 2004.

[11] Binutils. http://www.gnu.org/software/binutils/.

[12] LLVM. http://www.llvm.org.

[13] C. Lattner, and V. Adve. “LLVM: A Compilation Framework for Life-long Program Analysis & Transformation,” In *Proceeding of the International Symposium on Code Generation and Optimization*, pages 75-86, San Jose, CA, USA, March 2004.

[14] B. Y. Yoon, B. C. Lee, and S. S. Lee. “Scalable Flow-based Network Processor for Premium Network Service,” *2011 International Conference on ICT Convergence*, pages 436-440, Seoul, Korea, September 2011.

[15] M. Ahmadi, and S Wong. “Network Processors: Challenges and Trends,” In *Proceeding of the 17th Annual Workshop on Circuits, Systems and Signal Processing*, pages 223-232, Veldhoven, Netherland, November 2006.