Retargetable C Compiler for Network Processors

Jun Li [*], François-R Boyer [**], El Mostapha Aboulhamid [*]

[*] DIRO, Université de Montréal
2920 Chemin de la Tour
C.P. 6128 Centre-Ville
Montréal, Québec, Canada H3C 3J7
{ljun, aboulham}@iro.umontreal.ca

[**] DGI, École Polytechnique de Montréal
5255 Av. Decelles
C.P. 6079 Centre-Ville
Montréal, Québec, Canada H3C 3A7
Francois-R.Boyer@polymtl.ca

Abstract

Application-specific instruction-set processors (ASIP) are widely used in network processors. With a high demand from the market for faster new product development, retargetable compilers, and the associated knowledge, become essential for development. Based on the LCC retagetable C compiler, we added an ASIP target derived from the DLX instruction set, which was successfully used in a network platform. Therefore, network processor application programs are now written using C instead of assembler code.

Keywords

Retargetable compilers, ASIP, network processor, code generation.

1. INTRODUCTION

With the new medias, as high quality radio over IP and TV on demand, the need for higher bandwidth has become actually huge; our live trends demand more and more of this bandwidth. Hence, the market will require wider bandwidth links but also faster and higher level analysis of packets in routers and front-ends to server arrays. High performance network processors are needed. There are several solutions that meet the requirements of network processing demand:

- Application Specific Integrated Circuits (ASIC)
- Application Specific Instruction-Set Processors (ASIP)
- Field Programmable Gate Arrays (FPGA)
- Co-processors (can be any of the above types)
- General Purpose Processors (GPP)

Application Specific Instruction-Set Processors (ASIP) – an instruction set processor typical for a particular application domain becomes the most popular solution for network processing because of its special characteristics. ASIP sits in between the highest efficiency of ASIC and the lowest development cost of GPP. ASIP provides good balance of hardware and software to meet all requirements, such as: performance, flexibility, fast time to market, power consumption, etc. As a result, networking application domain is specified as a software programmable device with architectural features and/or special circuitry for packet processing; therefore a network processor can be seen as an ASIP for the networking application domain [1].

Compiler support for network processors is very important and with great demand. The market demands fast response time to requests, and high reliability, for network processing systems. In order to meet the demand, embedded software requires compilers to avoid slow and error-prone development in assembly language. As a result, for the particular architecture of network processors, classical compiler technology is not sufficient. Moreover, to completely exploit the processor capabilities, more specialized code generation must be used. For that reason, a retargetable compiler is required [2].

The purpose of this paper is to show how to implement an efficient retargetable C compiler for a network processor architecture. In addition, we will present some functions used for basic input/output, mathematics and other purposes, without using standard C libraries.

The remaining part of this paper is organized as follows. In section 2, we make a brief description of the network platform that we have worked on. In section 3 we explain how we implemented a retargetable C compiler based on LCC. Section 4 describes the result of retargetable C compiler implementation and the functions we used to eliminate the need of the standard C libraries. Section 5 contains conclusions and future works.

2. WORK PLATFORM

SystemC is an open source class library in C++. It permits to develop cycle-accurate or more abstract models of software algorithms, hardware architectures and system level designs. SystemC is an interoperable modeling platform, which allows seamless tool integration [3].

We modeled a system on chip (SoC) platform, including multiple processors. It was developed using SystemC on Linux PCs. Detailed descriptions are given in reference [4]. The modeled processors are DLX or ARM architectures (still in development), and both can work together. Each processor is an addressable device,

and are connected via an interconnect device like *ring_device* [4]. Figure 1 shows the architecture of a DLX processor and its adjacent nodes.

The current interconnect device is very simple, but a model for the AMBA bus is in development. The memory modules are located next to the processors, but we plan to have both distributed and shared memory mechanisms in the future version of the network platform.

Our colleagues worked on the platform using handwritten assembler code, but this was demanding a lot of time in writing and debugging DLX code. For this reason, a suitable compiler was a necessity. In this paper, our target processor is based on the DLX instruction set; we added some special instructions to fit the network processing demand. We call it DLXpro in the rest of this paper.

In most network formats, bit-fields are packed together to save bandwidth, but these fields (or bit-packets) are not handled efficiently by normal instructions on GPP. Bit-packet-oriented instructions [2] are needed to accelerate network packets processing, but are not currently implemented on our DLXpro. From a compiler point of view, it is hard to add these instructions, since it is difficult, in the general case, to find which sequence of instructions could map to a bit-packet instruction. Another important aspect of network processors is parallelism. Currently no automatic partitioning, or multi-threading, is done, the C code of each processor is written separately; a memory-mapped I/O mechanism is used to communicate between processors.

We may also use other network processors as the target when necessary. Figure 2 shows a high level diagram of the idealize platform.

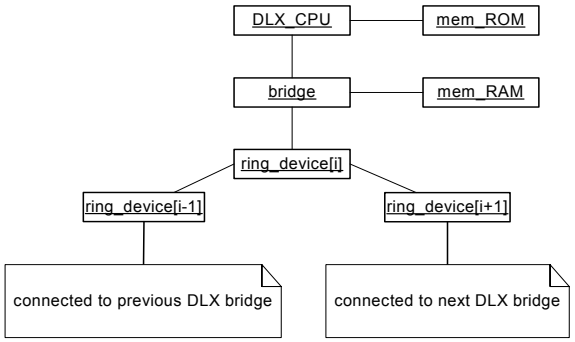

Figure 1: The architecture of nodes [4]

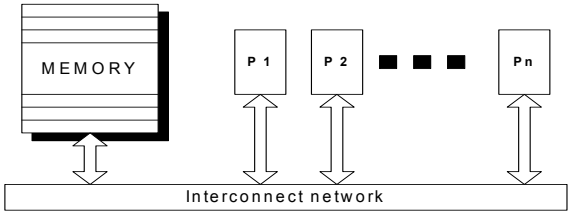

Figure 2: The idealize platform

(P is processor)

The DLX instruction set architecture includes five pipeline stages:

· Instruction Fetch (IF), the first stage, is responsible for getting the instructions from memory.

· Instruction Decode (ID) stage is in charge of selecting the operand registers, decoding the instruction and evaluating the branch condition.

· Execution (EX) stage is for arithmetic and logical operations, as well as memory address calculation.

· Memory (MEM) stage is for data memory access (read or write).

· Write Back (WB), the final stage, writes the results calculated by EX, or read by MEM, to the destination register when needed.

3. RETARGETABLE COMPILER DESIGN

LCC is a retargetable compiler for ANSI C. It has been ported to the VAX, SPARC, MIPS, X86 and other target processors. LCC is a small, fast C compiler now available on most popular operating system [5].

Similar to most other compilers, the LCC compiler is subdivided into two parts: a frontend and a backend part. The frontend is in charge of source code analysis, generation of an Intermediate Representation (IR), and machine-independent optimizations. The backend maps the machine-independent IR into machine-dependent assembly code. It is said retargetable since we can easily add one or more different target code generators to the backend.

3.1. Target architecture

First of all, let us take a look at the target architecture: the DLXpro. It is almost the same as DLX instruction set, except that we added some instructions. The DLX processor comes from a combination of ideas from other load/store RISC architectures [6].

The DLX Instruction Set Architecture (ISA) contains 32 (R0-R31) 32-bit general-purpose registers. Registers R1-R30 are real general-purpose registers. Registers R0 always contains zero. Register R31 is used for saving the return address for the Jump And Link (JAL) instructions [6].

The DLX ISA also has 32 single-precision floating-point (32-bit) registers (F0-F31). These registers can also be addressed as pairs (two consecutive registers, the first one being even-numbered) to form 16 double-precision floating-point (64-bit) registers.

DLX ISA has three specific registers: Program Counter (PC), the Interrupt Address Register (IAR), and the Floating-Point Status Register (FPSR).

In DLX ISA, a word is defined as 32 bits and a byte is 8 bits. Memory is byte addressable and word storage adheres to the big endian byte ordering [7].

3.2. Implementation of DLXpro code generator

The LCC compiler backend can be divided into two parts: code selection and register allocation. The code selector maps the intermediate representation (trees or directed acyclic graphs), generated by the front-end, using the backend interface, into DLXpro instructions. The register allocator maps all the virtual registers to physical registers. We used the MIPS [8] code generator as a model, and made some changes to suit DLXpro instruction set.

A. Instructions selection

The instruction selectors in LCC are automatically generated from compact specifications by the program *lburg*. That is, you give a grammar to *lburg* to partition the IR tree, and it generates the C code for the backend. A tree parser accepts a subject tree of intermediate code and partitions it into chunks that correspond to DLXpro assembly instructions.

Tree grammar is the core in *lburg*. Tree grammar is a list of rules, which have four parts. First is a non-terminal that replaces the part of the tree if the rule is applied. Then, a tree-matching expression (non-terminals and IR nodes) specifies where the rule can be applied. And finally, what assembler instructions must be added to make the transformation and their cost (what the compiler tries to minimize; size or number of cycles).

DLXpro non-terminals list as below:

acon: address constants
addr: address calculations from registers
addrr: address calculations from immediate values
con: constants
reg: computations that result to a register
stmt: computations done for side effects

The above non-terminals give a high level overview of the tree grammar used for mapping to DLXpro assembler instructions. Here are some actual rules, as example:

```
reg:  BCOMI4(reg)    " xori r%c,-1\n"          1
reg:  BCOMU4(reg)    " xori r%c,-1\n"          1
reg:  NEGI4(reg)     " sub r%c,r0,r%0\n"       1
.
.
stmt: EQI4(reg,reg)  " seq r3,r%0,r%1\n  bnez r3, %a\n"   2
stmt: GEI4(reg,reg)  " sge r3,r%0,r%1\n  bnez r3, %a\n"   2
stmt: GTI4(reg,reg)  " sgt r3,r%0,r%1\n  bnez r3, %a\n"   2
```

The elements in the first column, like *reg* and *stmt*, are non-terminals. In the second column are tree nodes that name in uppercase and operands types in the parentheses (here, operands are non-terminal *reg*). In the third column, inside double quotation marks are the assembler code templates. The final column's numbers are the optional cost.

B. Registers allocation

The DLXpro instruction set only gives a few constraints: R0 is always 0 and R31 only contains jump and link instructions' return address. The assembler reserves R1 for pseudo-instructions. R2, R3 are reserved by convention for return values. R26, R27 are reserved for Operating System (OS). R4-R7 are for procedure arguments. R8-R15, R24 and R25 are scratch registers. R16-R23 are for register variables. R28 is used as global pointer. R29 is the stack pointer and R30 is for compiler temporary values.

The register allocator is a small part written in C, using predefined LCC register management functions.

4. APPLICATION RESULTS

The LCC compiler for the DLXpro target described in the previous section is fully functional. The performance of the generated code has been tested, on the platform described in section 2, giving the expected results. For testing purposes, we added two addressable devices to the platform: an *output device* (address 0x4001) for outputting characters to the screen, and a *data interface*, (address 0x5001). The *output device* is direct: writing to 0x4001 a character writes it to the screen. For the *data interface*, we use two First In First Out (FIFO) files as data buffers to exchange data with other devices and a more complicated DMA interface is used. As mentioned before, SystemC is an object-oriented modeling language based on C++ [3]. It was easy to implement these two modules with the methods inherited from class *Addressable_Device* that is derived from *SC_METHOD* of SystemC.

The entire idealize platform structure is shown on figure 3.

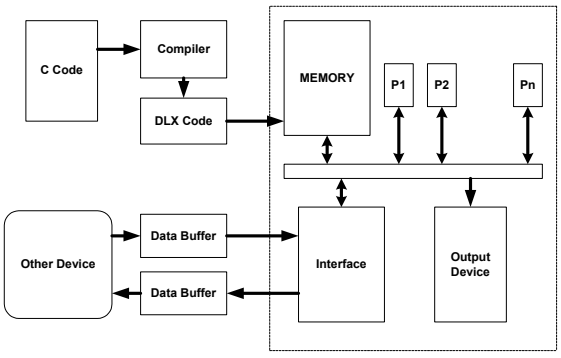

Figure 3: Overview of entire platform

We also wrote some special pure C functions to implement basic I/O functions, mathematical functions and conversion functions without C libraries. These functions include *getchar, string _to_integer, printString, sqrt, etc*. All these functions are small and easily ported.

By using these functions and modules, we can conveniently implement some library functions such as

printf, but often we don't need a printing function as complex and big as *printf*.

The following example shows how the above-mentioned functions and modules work:
Suppose we need to print "Hello" to the screen. By using the *printString* function, we could write the C code as follows:

```
 void printString(char *p)
{
  while(*p)  *((unsigned int*)0x4001) = (*(unsigned char*)p++);
}
void main()
{
  printString("Hello");
}
```

Compiling the above code, we got the corresponding DLX assembler code:

```
        addi   r29, r0, 1000
        jal    main
        trap   0
        nop
        nop
printString:
        j      L.3
L.2:    addi   r24, r4, 0
        addi   r4, r24, 1
        addi   r15, r0, 0x4001
        lbu    r24, (r24)
        sw     (r15), r24          ; access to a addressable device using
                                   ; memory mapped mechanism
L.3:    lb     r24, (r4)
        sne    r3, r24, r0
        bnez   r3, L.2             ; need optimisation
        addi   r24, r0, 0x4001
        sw     (r24), r0
L.1:    jr     r31
main:
        addi   r29, r29, -24
        sw     16(r29), r31
        addi   r24, r0, L.6
        sw     -4+24(r29), r24
        lw     r4, -4+24(r29)
        jal    printString
L.5:    lw     r31, 16(r29)
        addi   r29, r29, 24
        jr     r31
L.6:
        .byte  72, 101, 108, 108, 111, 0
```

We ran the above assembler code on the network platform model using DLXpro processors and got the expected results, but the code is not optimal. The "sne" in *printString* could be removed changing the condition on the next line.

5. CONCLUSION AND FUTURE WORK

In this paper, we made a brief description of how a retargetable C compiler like LCC can be used to target to an ASIP like a modified DLX architecture. We could easily add other network processors as targets since DLX instruction set comes from many real world RISC processors. The compiler base on LCC, with its fast, small and convenient characteristics, will significantly improve development and debug time. We could foresee the bright future of the usage for network processor in the real world. We also presented our tested compiler on our network platform with expected results. Examples are illustrated to show how the specific functions and modules are written to act as some of standard C libraries. These functions and modules will play an important role in testing and code writing in terms of time saving and simplified work.

The quality of compiler-generated code is not as good as handwritten DLX assembly code. In the future, we will add some bit-packet-oriented instructions to DLXpro, in order to speedup network processor applications, which does a lot of bit extraction and manipulation. However, we will encounter a difficulty in the network compiler development to generate bit-packet-oriented instructions from a high-level language. There are several approaches that have already been explored, for instance, Compiler-Known Functions (CKFs) [2]. We will optimize the quality of this retargetable C compiler especially in bit-packet-oriented instruction generation. We also plan to explore other retagetable compilers, like SUIF, trying to solve the multiple-thread problem in network processors.

Acknowledgements: We gratefully acknowledge Luc Charest providing a lot of information on the initial platform.

6. References

[1] N. Shah, *Understanding Network Processors.* Dept. EECS, UC, Berkeley. September 2001

[2] J. Wagner and R. Leupers, "C Compiler Design for a Network Processor." *IEEE transaction on computer-aided design of integrated circuits and systems* VOL.20, NO. 11,November 2001

[3] Open SystemC Initiative (OSCI), Functional Specification for SystemC 2.0, http://www.systemc.org, 2001

[4] L. Charest, E. M. Aboulhamid, C. Pilkington, P. Paulin. "SystemC Performance Evaluation using a Pipelined DLX Multiprocessor," *DATE*, 2002

[5] C. W. Fraser, D. Hanson, *A Retargetable C Compiler: Design and Implementation*. The Benjamin/Cummings Publishing Company, Inc. 1994

[6] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Second ed: Morgan Kaufmann Publishers, 1995

[7] P. M. Sailer and D. R. Kaeli, *The DLX Instruction Set Architecture Handbook*. Morgan Kaufmann Publishers, Inc. 1996

[8] Gerry Kane, *MIPS RISC Architecture.* Prentice Hall, Englewood Cliffs, NJ, 1989