Review of “C Compiler Design for a Network Processor”

Tomas Henriksson
Department of Electrical Engineering, Linköpings universitet
SE-581 83 Linköping
tomhe@isy.liu.se

1 Introduction

This review describes the approach and results from the paper by Wagner and Leupers [1]. The methods they have developed for compilers for application specific instruction set processors (ASIPs) and specifically network processors (NPs) with bit-packet addressing are mainly concentrated to extensions to the traditional register allocation by graph coloring.

ASIPs are used in areas where general-purpose processors do not provide enough performance and application specific integrated circuits (ASICs) do not provide enough flexibility. The most successful area so far has been digital signal processors (DSPs). ASIPs have an instruction set that is specifically targeted at a group of applications and normally also a non-homogeneous register file. This complicates the task for the compiler. Since traditional code selection and register allocation algorithms cannot be used.

Often the special instructions are not easily described in the C language, which forces us to extend the source language. This may however not be the case for all special instructions.

2 Network Processors

The network processors are according to the authors characterized by the need for bit-packet instructions. By this is meant that operations take place on parts of a register or memory location. I see this as only one particular feature of network processors, which may be very diverse. The bit-packet instructions are important especially for packet manipulation operations.

2.1 The Infineon NP

The target architecture in the paper is the Infineon network processor. It is based on a standard RISC architecture with 16 bit data paths and a register file of 12 general purpose registers. The specialities of the architecture lie in the bit-packet addressing. An instruction does not necessarily operate on the whole 16 bits of a register and may even operate on a field which crosses a register boundary.

To be able to specify such operations, the assembly instructions use the following format:

CMD reg1.off1, reg2.off2, width

CMD specifies the assembly instruction, reg1 is the register and off1 is the offset (starting bit in that register) for the first operand. Reg2 and off2 work similarly for the second operand. Width specifies how many bits to include in the computation. If the offset + the width exceeds the register width (16) the operand will continue into the next register.

This bit-packet addressing can also be done by using special pointers, called bit-packet pointers, which point to a register with an offset. There are special instructions for incrementing such a pointer with a predefined number of bits. In this way it is possible to have
arrays of non-aligned variables and this is what complicates the compiler design as will be seen later on.

3 Source Code Bit-Packet Processing

The usage of these special features is explicitly stated in the source C code. The authors call this compiler-known functions (CKFs). This allows ease of expression for bit-packet operations. For example if we want to add the constant 2 to a variable, which resides on bit 3-9 in variable a. In C this would look like:

```
    a = (a & 0xFE03) | ((a + (2 << 2)) & 0x01FC);
```

With the new extensions, which generally can be described by as the packet access (PA) CKF:

```
    PA(int op, int var1, int off1, int var2, int off2, int width);
```

the same expression can be written as:

```
    int a, b;
    b = 2;
    PA(PA_ADD, a, 3, b, 0, 7);
```

This makes the source code dependent on the target architecture and the authors admit later in the paper that they currently work on extracting the bit-packet instructions from standard C language. In any case the CKF provides a convenient way to make use of the special instructions, which can easily be combined with standard C control structures, such as if statements, for loops, etc.

It gets even nicer, when considering an example where bit-packet pointers are used. By using the CKF INIT a bit-packet pointer is specified with starting point and width. After the initialization the CKF INC can be used to increment it with the specified width.

4 Compiler Design

The compiler is divided into a frontend and a backend. The frontend is taken from the LANCE compiler system from University of Dortmund. It parses C code and generates an IR, which can easily be transformed into data flow tree format.

The backend consists of two parts, a code selection part and a register allocation part. The code selector assumes an infinite number of virtual registers. It uses dynamic programming for mapping the data flow tree into assembly instructions. The instructions set was specified in the OLIVE tool which generated the implementation. During the code selection the CKFs are identified by their name and converted into the corresponding assembly instruction. This is a simple step, since the CKFs exactly correspond to assembly instructions.

The register allocation is responsible of mapping the virtual registers into the physical registers. Here the novelty of the compiler is needed. The traditional method that uses an interference graph and graph coloring does not suffice, since virtual registers, that contain fragments of register arrays must be mapped to consecutive physical registers. Therefore a new register allocation method was invented, starting from the traditional method. First, the lifetime analysis is done conservatively since bit-packet pointers cannot be known at compile time. So they extend lifetimes of all possible registers. Second, supernodes are created in the interference graph. A supernode contains all registers that belong to the same register array. Each supernode is given a weight, corresponding to the size of the register array. Then
the heaviest supernode is allocated first and so on until all supernodes have been allocated. After that the normal nodes, that do not belong to any register array, are allocated. As usual, if there is not enough registers, spill code is generated. If a register array is bigger than the physical register file, that cannot be handled by the compiler and a partitioning has to be done in the source code.

5 Results and Conclusions

The authors claim to have a fully operational compiler. The quality of the generated code very much depends on the correct use of CKFs. By careful use of CKFs, which requires detailed application knowledge, code can be generated, that is only 10% less effective than manually optimized assembly code.

The gain from using CKFs is in average 28% over a certain set of applications. Which I do not consider overwhelming, since their compiler cannot make use of the special features of the processor without the CKFs and they basically compare to very poor code. Anyway, the authors conclude that their new register allocation technique works and can be used in industry. The compiler is machine dependent, but is believed to be easily portable to for example Intel’s i960 network processor.

Future work is in two directions. First the authors want to improve the code selection so that the CKFs are not needed. Second, they want to optimize the spill code so that the register memory traffic can be minimized.

The work that Wagner and Leupers have done deals with a very small part of the network processor compilation problem. What they done however, solves a particular problem. It would be very interesting to see how industrial compilers for NPs work, because they do exist. Especially interesting is the Agere approach, which uses a special source language, called functional programming language (FPL) and uses a three chip solution with several programmable cores which operate in parallel. There totally new compilation problems arise, since the processors in the system are heterogeneous ASIPs.

References