A C Compiler for a Processor with a Reconfigurable Functional Unit

Zhi Alex Ye   Nagaraj Shenoy   Prithviraj Banerjee

Department of Electrical and Computer Engineering,

Northwestern University

Evanston, IL 60201, USA

{ye, nagaraj, banerjee}@ece.nwu.edu

ABSTRACT

This paper describes a C compiler for a mixed Processor/FPGA architecture where the FPGA is a Reconfigurable Functional Unit (RFU). It presents three compilation techniques that can extract computations from applications to put into the RFU. The results show that large instruction sequences can be created and extracted by these techniques. An average speedup of 2.6 is achieved over a set of benchmarks.

1. INTRODUCTION

With the flexibility of the FPGA, reconfigurable systems are able to get significant speedups for some applications. As the general purpose processor and the FPGA each has its own suitable area of applications, several architectures are proposed to integrate a processor with an FPGA in the same chip.

In this paper, we talk about a C compiler for a Processor/FPGA system. The target architecture is Chimaera, which is a RISC processor with a Reconfigurable Functional Unit (RFU). We describe how the compiler identifies sequences of statements in a C program and changes them into RFU operations (RFUOPs). We show the performance benefits that can be achieved by such optimizations over a set of benchmarks.

The rest of the paper is organized into five sections. Section 2 discusses related work. In Section 3, we give an overview of the Chimaera architecture. Section 4 discusses the compiler organization and implementation in detail. In this section, we first discuss a technique to enhance the size of the instruction sequence: control localization. Next, we describe the application of the RFU to SIMD Within A Register (SWAR) operations. Lastly, we introduce an algorithm to identify RFUOPs in a basic block. Section 5 demonstrates some experimental results. We summarize this paper in Section 6.

2. RELATED WORK

Several architectures have been proposed to integrate a processor with an FPGA [6,7,8,9,13,14,15]. The usage of the FPGA can be divided into two categories: FPGA as a coprocessor or FPGA as a functional unit.

In the coprocessor schemes such as Garp[9], Napa[6], DISC[14], and PipeRench[7], the host processor is coupled with an FPGA based reconfigurable coprocessor. The coprocessor usually has the ability of accessing memory and performing control flow operations. There is a communication cost between the coprocessor and the host processor, which is several cycles or more. Therefore, these architectures tend to map a large portion of the application, e.g. a loop, into the FPGA. One calculation in the FPGA usually corresponds to a task that takes several hundred cycles or more.

In the functional unit schemes such as Chimaera[8], OneChip[15], and PRISC[13], the host processor is integrated with an FPGA based Reconfigurable Functional Unit (RFU). One RFU Operation (RFUOP) can take on a task that usually requires several instructions on the host processor. As the functional unit is interfaced only with the register file, it cannot perform memory operations or control flow operations. The communication is faster than the coprocessor scheme. For example, in the Chimaera architecture, after an RFUOP's configuration is loaded, an invocation of it has no overhead in communication. This gives such architecture a larger range of application. Even in cases where only a few instructions can be combined into one RFUOP, we could still apply the optimization if the execution frequency is high enough.

3. CHIMAERA ARCHITECTURE

In this section, we review the Chimaera architecture to provide adequate background information for explaining the compiler support for this architecture. More information about Chimaera can be found in [8].

The overall Chimaera architecture is shown in Figure 1. The main component of the system is the Reconfigurable Functional Unit (RFU), which consists of FPGA-like logic designed to support high-performance computations. It gets inputs from the host processor's register file, or a shadow register file which duplicates a subset of the values in the host's register file. The RFU is capable of computing data-dependent operations (e.g., tmp=r2-r3, r5=tmp+r1), conditional evaluations (e.g., "if (b>0) a=0; else a=1;"), and multiple sub-word operations (e.g., four instances of 8-bit addition).

The RFU contains several configurations at the same time. An RFUOP instruction will activate the corresponding configuration in the RFU. An RFU configuration itself determines from which
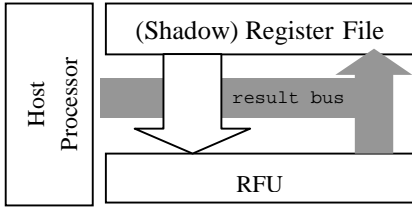
Figure 1. The overall Chimaera architecture

registers it reads its operands. A single RFUOP can read from all the registers connected to the RFU and then put the result on the result bus. The maximum number of input registers is 9 in Chimaera. Each RFUOP instruction is associated with a configuration and an ID. For example, an execution sequence "r2=r3<<2; r4=r2+r5; r6=lw 0(r4)" can be optimized to "r4=RFUOP #1; r6=lw 0(r4)". Here #1 is the ID of this RFUOP and "r5+r3<<2" is the operation of the corresponding configuration. After an RFUOP instruction is fetched and decoded, the Chimaera processor checks the RFU for the configuration corresponding to the instruction ID. If the configuration is currently loaded in the RFU, the corresponding output is written to the destination register during the instruction writeback cycle. Otherwise, the processor stalls when the RFU loads the configuration.

4. COMPILER IMPLEMENTATION

We have developed a C compiler for Chimaera, which automatically maps some operations into RFUOPs. The generated code is currently run on a Chimaera simulator to gather performance information. A future version of the compiler will be integrated with a synthesis tool.

The compiler is built using the widely available GCC framework. Figure 2 depicts the phase ordering of the implementation. The C code is parsed into the intermediate language of GCC: Register Transfer Language (RTL), which is then enhanced by several early optimizations such as common expression elimination, flow analysis, etc. The partially optimized RTL is passed through the Chimaera optimization phase, as will be explained below. The Chimaera optimized RTL is then processed by later optimization phases such as instruction scheduling, registers allocation, etc. Finally, the code for the target architecture is generated along with RFUOP configuration information.

From the compiler's perspective, we can consider an RFUOP as an operation with multiple register inputs and a single register output. The goal of the compiler is to identify the suitable multiple-input-single-output sequences in the programs and change them into RFUOPs.

Chimaera Optimization consists of three steps: Control Localization, SWAR optimization and Instruction Combination. Due to the configuration loading time, these optimizations can be applied only in the kernels of the programs. Currently, we only optimize the innermost loop in the programs.

The first step of Chimaera optimization is control localization. It will transform some branches into one macroinstruction to form a larger basic block. The second step is the *SIMD Within A Register* (SWAR) Optimization. This step searches the loop body for subword operations and unrolls the loop when appropriate.
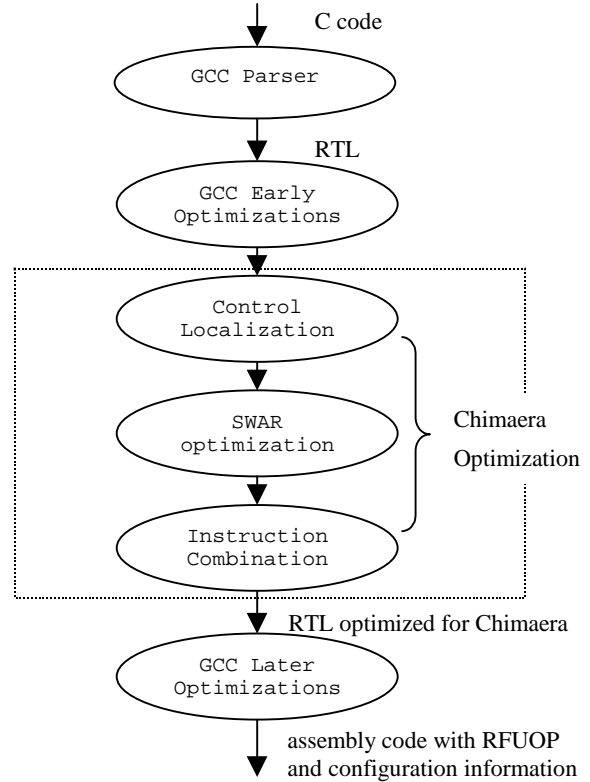

Figure 2: **Phase ordering of the C compiler for Chimaera**

The third step is instruction combination. It takes a basic block as input and extracts the multiple-input-single-output patterns from the data flow graph. These patterns are changed into RFUOPs if they can be implemented in RFU. The following subsections discuss the three steps in detail.

4.1 Control Localization

In order to get more speedup, we want to find larger and more RFUOPs. Intuitively, a larger basic block contains more instructions, thus has more chances of finding larger and more RFUOPs. We find that control localization technique [11][13] is
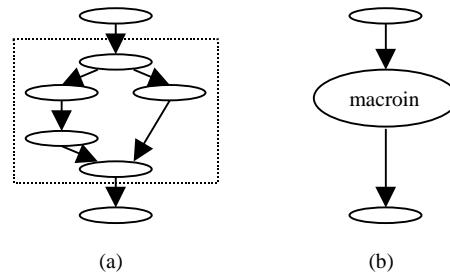

(a)   (b)

Figure 3: Control Localization
(a) control flow graph before control localization. Each oval is an instruction, and the dashed box marks the code sequence to be control localized.
(b) control flow graph after control localization

useful in increasing the size of basic blocks. Figure 3 shows an example of it. After control localization, several branches are combined into one macroinstruction, with multiple output and multiple input. In addition to enlarging the basic block, the control localization sometimes finds RFUOPs directly. When a macroinstruction has only one output, and all the operations in it can be implemented in the RFU, this macroinstruction can be mapped into an RFUOP. This RFUOP can speculatively compute all operations on different branch paths. The result on the correct path where the condition evaluates to true is selected to put into the result bus. This macro instruction is called as "CI macroin" and can be optimized by Instruction Combination.

4.2 SWAR Optimization

As a method to exploit medium-grain data parallelism, SIMD (single instruction, multiple data) has been used in parallel computers for many years. Extending this idea to general purpose processors has led to a new version of SIMD, namely *SIMD Within A Register* (SWAR)[4]. The SWAR model partitions each register into fields that can be operated on in parallel. The ALUs are set up to perform multiple field-by-field operations. SWAR has been successful in improving the multimedia performance. Most of the implementations of this concept are called multimedia extensions, such as Intel MMX, HP MAX, SUN SPARC VIS, etc. For example, "PADDB A, B" is an instruction from Intel MMX. Both operands A and B are 64-bit and are divided into eight 8-bit fields. The instruction performs eight additions in parallel and stores the eight results to A.

However, current implementations of SWAR do not support a general SWAR model. Some of their limitations are:

- The input data must be packed and aligned correctly, causing packing and unpacking penalties sometimes.

- Most of current hardware implementations support 8, 16 and 32-bit field size only. Other important sizes such as 2-bit and 10-bit are not supported.

- Only a few operations are supported. When the operation for one item becomes complex, SIMD is impossible. For example, the following code does not map well to a simple sequence of SIMD operations:

```
char out[100],in1[100],in2[100];
for(i=0;i<100;i++) {
if ((in1[i]-in2[i])>10)
  out[i]=in1[i]-in2[i];
else
  out[i]=10;}
```

With the flexibility of the FPGA, the RFU can support a more general SWAR model without the above disadvantages. The only requirement is that the output fields should fit within a single register. The inputs don't need to be stored in packed format, nor is there limitation on the alignment. In addition, complex operations can be performed. For example, the former example can be implemented in one RFUOP.

Our compiler currently supports 8-bit field size, which is the size of "char" in C. In current implementation, the compiler looks for the opportunity to pack several 8-bit outputs into a word. In most cases, this kind of pattern exists in the loop with stride one. Therefore, the compiler searches for the pattern such that the memory store size is a byte and the address changes by one for

```
ALGORITHM: FindSequences
INPUT:DFG G, NCI, CI, Live-on-exit registers R
OUTPUT: A set of RFU sequences S
begin
    S=∅
    for each node n∈G
        Color(n) ←WHITE
    end
    for each node n∈NCI
        Color(n) ←BLACK
        for each node p∈Pred(n)
            if  p∈CI then
                Color(p) ←BROWN
            endif
        end
    end
    for each register r∈R
        n← the last node that updates r in G
        if n∈CI then
            Color(n) ←BROWN
        endif
    end
    for each node n∈G
        if Color(n)=BROWN then
            sequence=∅
            AddSequence(n, sequence)
            if sizeof(sequence)>1 then
                S=S∪{sequence}
            endif
        endif
    end
end

AddSequence( n, sequence)
begin
    if Color(n)=(BROWN or WHITE) then
        sequence←sequence∪{n}
        for each p∈Pred(n)
            AddSequence(p, sequence)
        end
    endif
end
```

Figure 4: Algorithm to find RFU sequences

each loop iteration. When such patterns are found, the loop is unrolled four times. In the loop unrolling, conventional optimizations such as local register renaming and strength reduction are performed. In addition, the four memory stores are changed to four sub-register movements. For example,

```
"store_byte r1,address;
 store_byte r2,address+1;
 store_byte r3,address+2;
 store_byte r4,address+3;"
```
are changed into

```
"(r5,0)=r1; (r5,1)=r2;
 (r5,2)=r3; (r5,3)=r4;".
```

The notation (r, n) refers to the nth byte of register r. We generate a pseudo instruction "collective-move" that moves the four sub-registers into a word register, e.g. "r5=(r5,0) (r5,1) (r5,2) (r5,3)". In the data flow graph, the four outputs merge through this "collective-move" into one. Thus a multiple-input-single-output subgraph is formed. The next step, Instruction Combination, can

recognize this subgraph and change it to an RFUOP when appropriate. Finally, a memory store instruction is generated to store the word register. The compiler then passes the unrolled copy to the instruction combination step.

4.3 Instruction Combination

The instruction combination step analyzes a basic block and changes the RFU sequences into RFUOPs. It first finds out what instructions can be implemented in the RFU. It then identifies the RFU sequences. At last, it selects the appropriate RFU sequences and changes them into RFUOPs.

We categorize instructions into Chimaera Instruction (CI) and Non-Chimaera Instruction (NCI). Currently CI includes logic operation, constant shift and integer add/subtract. The "collective_move", "subregister movement" and "CI macroin" are also considered as CI. NCI includes other instructions such as multiplication/division, memory load/store, floating-point operation, etc.

The algorithm FindSequences in Figure 4 finds all the maximum instruction sequences for the RFU. It colors each node in the data flow graph(DFG). The NCI instructions are marked as BLACK. A CI instruction is marked as BROWN when its output must be put into a register, that is, the output is live-on-exit or is the input of some NCI instructions. Other CI instructions are marked as WHITE. The RFU sequences are the subgraphs in the DFG that consists of BROWN nodes and WHITE nodes.

The compiler currently changes all the identified sequences into RFUOPs. Under the assumption that every RFUOP takes one cycle and the configuration loading time can be amortized over several executions, this gives an upper bound of the speedup we could expect from Chimaera. In the future, we will take into account other factors such as the FPGA size, configuration loading time, actual RFUOP execution time, etc.

5. EXPERIMENTAL RESULTS

We have tested the compiler's output through a set of benchmarks on the Chimaera simulator. The simulator is a modification of SimpleScalar Simulator[3]. The simulated architecture has 32 general purpose 32-bit registers and 32 floating point registers. The instruction set is a superset of MIPS-IV ISA. Presently, the simulator executes the programs sequentially and gathers the

	insn1	insn2	Speedup	IC	CL	SWAR
dcmp	37.4M	32.7M	1.14	100%	0	0
cmp	34.2M	28.6M	1.20	80%	20%	0
jcbi	2.1M	1.3M	1.63	94%	6%	0
life	6.7M	5.0M	1.34	100%	0	0
dct	78.4M	52.6M	1.49	10%	82%	8%
rcn	11M	2.6M	4.23	6%	10%	84%
rst	22.3M	3.1M	7.19	16%	0	84%
avg	-	-	2.60	22%	7%	61%

Table 1: Performance results over some benchmarks. The "avg" row is the average of all benchmarks.
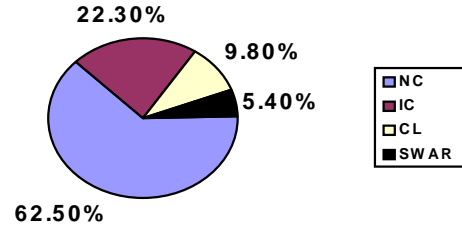

Figure 5: Distribution of the kernel instructions
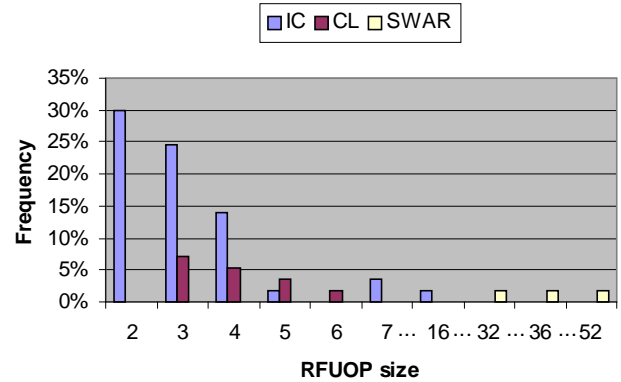

Figure 6: The frequency of RFUOP sizes. IC, CL and SWAR are the RFUOPs generate by Instruction Combination, Control Localization and SWAR respectively.

instruction count.

Early results on some benchmarks are presented in this section. Each benchmark is compiled in two ways: one is using "gcc -O2", the other is using our Chimaera compiler. We studied the differences between the two versions of assembly codes as well as the simulation results. In the benchmarks, decompress.c and compress.c are from Honeywell benchmark[10], jacobi and life are from Raw benchmark[2], image reconstruction[12] and dct[1] are implementations of two program kernels of MPEG, image restoration is an image processing program. They are noted as dcmp, cmp, life, jcbi, dct, rcn and rst in the following figure.

Table 1 shows the simulation results of the RFU optimizations. Insn1 and insn2 are the instruction counts without and with RFU optimization. The speedup is calculated as insn1/insn2. The following three columns IC, CL and SWAR stand for the portion of performance gain from Instruction Combination, Control Localization and SWAR respectively. The three optimizations give an average speedup of 2.60. The best speedup is up to 7.19.

To illustrate the impact of each optimization on the kernel sizes, we categorize instructions into four types: *NC*, *IC*, *CL* and *SWAR*. *NC* is the part of instructions that cannot be optimized for Chimaera. NCI instructions and some non-combinable integer operations fall in this category. *IC*, *CL* and *SWAR* stand for the instructions that can be optimized by Instruction Combination, Control Localization and SWAR optimization respectively. Figure 5 shows the distribution of these four types of instructions in the program kernels. After the three optimizations, the kernel size can be reduced by an average of 37.5%. Of this amount, 22.3% is from Instruction Combination, 9.8% from Control Localization and 5.4% from SWAR.

Further analysis shows that 58.4% of the *IC* portion comes from address calculation. For example, the following C code "int a[10], ...=a[i]" is translated to "r3=r2<<2, r4=r3+r1, r5=lw 0(r4)" in assembly. The first two instructions can be combined in Chimaera. The large portion of address calculation indicates that our optimizations can be applied to a wide range of applications, as long as they have complex address calculations in the kernel. Furthermore, as the address calculation is basically sequential, existing ILP architectures like superscalar and VLIW cannot take advantage of it. This suggests that we may expect speedup if we integrate a RFU into an advanced ILP architecture.

Figure 6 illustrates the frequencies of different RFUOP sizes. For Instruction Combination and Control Localization, most of the sizes are from 2 to 6. These small sizes indicate that these techniques are benefiting from the fast communication of the functional unit scheme. In the coprocessor scheme, the communication overhead would make them prohibitive to apply. The SWAR optimization generally identifies much larger RFUOPs. The largest one comes from the image reconstruction benchmark, whose kernel is shown in Figure 7. In this case, a total of 52 instructions are combined in the RFU, which results in a speedup of 4.2.

```
(1) char x[64], y[64], idct[64];
(2) ...
(3)  for (i=0;i<64;i++) {
(4) ...
(5) temp=(x[i]+y[i]+1)>>1+idct[i];
(6) if (temp<0) temp=0;
(7) if (temp>255) temp=255;
(8)...
```
Figure 7: kernel of the "rcn" benchmark. The compiler unrolled the loop 4 times and mapped the operations in (5), (6), and (7) to an RFUOP.

The presented results are based on an in-order execution model. We have also simulated the architecture in an out-of-order execution environment. We considered a superscalar host processor, different latencies of RFUOPs, and configuration loading time. These results are reported in [16].

In summary, the results show that the compilation techniques are able to create and find many instruction sequences for the RFU. Most of their sizes are several instructions, which demonstrate that the fast communication is necessary. The system gives an average speedup of 2.6.

6. CONCLUSION

This paper describes a C compiler for the Processor/FPGA architecture when the FPGA is served as a Reconfigurable Functional Unit (RFU).

We have introduced an instruction combination algorithm to identify RFU sequences of instructions in a basic block. We have also shown that the control localization technique can effectively enlarge the size of the basic blocks and find some more sequences. In addition, we have illustrated the RFU support for SWAR. By introducing "sub-register movement" and "collective-move", the instruction combination algorithm is able to identify complex SIMD instructions for the RFU.

Finally, we have presented the experimental results, which demonstrate that these techniques can effectively create and identify larger and more RFU sequences. With the fast communication between RFU and the processor, the system can achieve considerable speedups.

8. REFERENCES

[1] K. Atsuta, DCT implementation, http://marine.et.u-tokai.ac.jp/database/koichi.html.

[2] J.Babb, M.Frank, et al. The RAW benchmark Suite: Computation Structures for General Purpose Computing. FCCM, Napa Vally, CA, Apr.1997

[3] D. Burger, and T. Austin, The Simplescalar Tool Kit, University of Wisconsin-Madison Computer Sciences Department Technical Report #1342, June, 1997

[4] P. Faraboschi, et al. The Latest Word in Digital and Media Processing, IEEE signal processing magazine, Mar 1998

[5] R. J. Fisher, and H. G. Dietz, Compiling For SIMD Within A Register, 1998 Workshop on Languages and Compilers for Parallel Computing, North Carolina, Aug 1998

[6] M.B. Gokhale, et al. Napa C: Compiling for a Hybrid RISC/FPGA Architecture, FCCM 98, CA, USA

[7] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer. PipeRench: A Coprocessor for Streaming Multimedia Acceleration, ISCA'99, May 1999, Atlanta, Georgia

[8] S. Hauck, T. W. Fry, M. M. Hosler, J. P. Ka, The Chimaera Reconfigurable Functional Unit, IEEE Symposium on FPGAs for Custom Computing Machines, 1997

[9] J. R. Hauser and J. Wawrzynek. GARP: A MIPS processor with a reconfigurable coprocessor. Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines (FCCM), Napa, CA, April 1997.

[10] Honeywell Inc, Adaptive Computing Systems Benchmarking, http://www.htc.honeywell.com/projects/acsbench/

[11] W. Lee, R. Barua, and et al. Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine, MIT. ASPLOS VIII 10/98, CA, USA

[12] S. Rathnam, et al. Processing the New World of Interactive Media, IEEE signal processing magazine March 1998

[13] R. Razdan, PRISC: Programmable Reduced Instruction Set Computers, Ph.D. Thesis, Harvard University, Division of Applied Sciences,1994

[14] M. J. Wirthlin, and B. L. Hutchings. A Dynamic Instruction Set Computer, FCCM, Napa Vally, CA, April, 1995

[15] R. D. Wittig and P. Chow. OneChip: An FPGA Processor with Reconfigurable Logic, FCCM, Napa Vally, CA, April, 1996

[16] Z. A. Ye, A. Moshovos, P. Banerjee, and S. Hauck, "Chimaera, a high performance architecture with a tightly-coupled reconfigurable functional unit", submitted to the 27th International Symposium on Computer Architecture (ISCA-2000).