Tool Chain Support with Dynamic Profiling for RISP

Chao Wang ¹+ Huizhen Zhang¹ Xuehai Zhou² Jinsong Ji¹ Aili Wang²

¹ School of Computer Science
University of Science and Technology of China
Hefei, Anhui, China.
{saintwc,zhanghz,jsji}@mail.ustc.edu.cn

² Suzhou Institute of USTC
University of Science and Technology of China
Suzhou, Jiangsu, China
{xhzhou,wangal}@ustc.edu.cn

Abstract

This article proposes a concept of dynamic profiling reconfigurable instruction set processor (RISP) and related retargetable tool chain support. The tool chain consists of a profiler, a code mapper, and a retargetable compiler. Firstly dynamic profiler is employed to obtain hot path for applications. Then hot block is implemented in reconfiguration logic units. After newly designed hardware block is integrated into system, mapper supplies a mechanism to map hot blocks to hardware implementations. Retargetable compiler is used for recompilation and regenerating executable binary code. The three modules have been demonstrated on simulation platform separately. Experimental result in previous work has already demonstrated the profiler can reach 97% of accuracy. A prototype code mapper shows the feasibility of the mapping mechanism. The simulation results of retargetable compiler shows with the decrease of code size and reconfiguration time, application can still be largely accelerated by RISP processor.

Keywords: reconfigurable instruction set processor; dynamic profiling; code mapping; retargetable compilation

1. Introduction

Processor architecture design is shifting towards reconfigurable instruction set processor (RISP). As RISP combines general purpose processor and reconfigurable logic unit, it’s possible to adapt instruction set to a specific application. Compared to other reconfigurable systems, RISP benefits from application specific instruction processor (ASIP) design method to focus on the implementation of reconfigurable instruction set. In RISP, instruction set is normally divided into two subsets: 1) fixed instructions, which cannot be modified after fabrication; 2) reconfigurable instructions, which can be modified according to distinct applications characteristics. Moreover, reconfigurable instructions can be changed dynamically in reconfigurable logic unit during execution.

Meanwhile, since instruction set of RISP can be reconfigured after fabrication, it makes comprehensive differences with traditional processor. As a result, new design methods must be considered to construct tool chain.

Generally, RISP are implemented in single chip with reconfigurable logic units, such as FPGA and CPLD. Since FPGA can only provide limited hardware resources, it is essential to choose the hot spots of applications and put them into hardware implementations. In order to get the hot spot, profiling technologies are involved. Profiler analyzes the frequencies of different function or codes in an application, which can help developers to find out which part cost most time during execution. Traditional approaches usually use software instrumentation as mainstream profiling method, which slows down the execution speed and causes considerable costs.

To solve the problems mentioned above, this paper proposes a retargetable software tool chain approach with hardware dynamic profiling. The approach is based on the parts of previous work. At first, Profiler is integrated into the tool chain to locate the hot sport of application. Additionally our profiler is implemented in hardware to improve performance. After new logic array is implemented in hardware, application are recompiled and mapped to hardware for execution. The approach can guarantee reuse of application after hardware reconfiguration. Therefore it can significantly shorten the time to market (TTM) of processor, reduce the burden of developers and make great contribution to the design and evaluation of embedded processor.

The rest of this paper is organized as follows. In Section 2 we present related work on RISP and tool chain design methodologies. We outline the overview of the dynamic profiling RISP architecture in section 3. Section 4 presents the retargetable software tool chain, consists of profiler, mapper and compiler. In Section 5, we measure the three components of software tools separately. Section 6 concludes the paper and discusses about further studies.

2. Related work

RISP plays a vital role in microprocessor design methodologies. Among current studies, VLIW [1, 2, 3, 4] has been widely used in RISP processor design. Beeck [4] refers a model through Design Space Exploration (DSE) that can meet different levels of constraints. XiRisc [2] states a load-store structure, in which all data access simply use two kinds of instructions, sharing register files. Iqbal [5] introduces RT-RISP, which defines single instruction is loaded or unloaded as separate modules, through partial reconfiguration. The modules are scheduled dynamically.
on the principle of requirements of application to achieve better performances. Molen [6, 7] integrates hardware reconfiguration as a part of processor design. By using reconfigurable micro-code, hardware tasks are considered as atomic operation.

Moreover, performance analysis aiming to RISP is also carried out. Barat [8] established a Wattch [9]-like power model to evaluate coarse-grained RISP, which can reduce 18% power cost compared to conventional RISC processors.

More than RISP itself, software tools have been highly valued by academic and industrial researchers. There are plenty of design tools based on RISP architecture. Most closely related to our work are AMBER [10], ARISE [11, 12] and EnCore [13].

AMBER’s tool chain is based on SimpleScalar simulation environment. It firstly gets the start address of hot spot basic block by profiling, and then reads the code and generates data flow graph (DFG) and related instruction list. Afterwards instruction generators receive the data, construct the instructions and optimize for the object code, then reconstruct reconfigurable logic unit (RLU) based on the instructions. Due to real-time execution information, this method can obtain a fixed relationship between customized instructions and RLU. However, since the profiling mainly depends on basic blocks rather than the overall situation, the improvements of performance is limited.

ARISE [11] treats software and hardware as modules, it can detect customized instructions and support hybrid calculation model software development environment. Nevertheless, the back-end of the tools does not describe the instructions in detail, and also to the representation of ARISE block annotations. In addition, this approach uses static profiling of the program, which results in low accuracy in hot spot detection and recognition.

EnCore [13] processor enhances GCC with end to end software tool chain design methodology. It can deal with the search and selection for multi input and output instruction template. But this concept involves inaccurate DFG with the real results due to static profiling, so the capability and performance are restricted.

As current existing tool chains including three above approaches have the limitation of hot spot recognition and low efficiency, we propose a new environment that combines dynamic profiling RISP and retargetable software tools.

3. Concepts and Implementation of DP-RISP

Before the retargetable software tool chain is proposed, we need to describe the processor architecture and execution mode of dynamic profiling RISP (DP-RISP) in this section.

3.1 Dynamic Profiling RISP Architecture

The architecture of DP-RISP is shown in Figure 1. The whole processor is implemented in FPGA, including embedded processor, reconfigurable logic unit, cache, and other modules such as I/O and memory blocks.

Embedded processor is composed of ALU, decoder and register files. Like traditional ASIP, instruction and data are stored in cache and transferred to processing elements during instruction fetch stage. Decoder obtains the instruction during application execution and subsequently sends instruction to ALU and profiler.

Reconfigurable logic unit consists of profiler, configuration controller, configuration memory, wrapper interfaces and reconfiguration array. Profiler collects instruction historical information and locates the hot spot of application. Configuration memory is used to store different configuration contexts. Reconfiguration controller can reconstruct array into hardware implementation with the hot spot information. Reconfiguration array is employed to execute reconfigurable instructions and can be configured by controller.

Additionally, I/O interface supplies communication interfaces with main memory and local caches. Cache stores instruction and data for RISP.

![Figure 1. DP-RISP Organization and Architecture](image1)

3.2 DP-RISP Execution Flow

DP-RISP utilizes embedded processor core to run ordinary instructions, and reconfiguration logic unit to implement reconfigurable instructions. There are two working modes for DP-RISP.

![Figure 2. DP-RISP Execution Modes](image2)

1) Normal Instruction Execution Mode
Figure 2 shows the 5 stages of RISP pipeline: IF, ID, EX, MEM and WB. Instructions compiled from application are decoded and sent to either processor or reconfiguration array for execution. Processor is responsible for normal instruction execution; while reconfigurable instructions are transferred to array wrapper, waiting to be handled by the reconﬁgurable computing resources.

Before reconﬁgurable instruction run on RLU, FPGA array must be reconstructed to ﬁt the instructions. If the target array is already implemented in hardware, instructions can run directly. Or else, conﬁguration controller requires the corresponding conﬁguration context from conﬁguration memory and then reconfi gures the array through wrapper.

2) Profiling Execution Mode

In proﬁling mode DP-RISP provides hot spot information through sampling mechanism-based proﬁler. In proﬁling mode, operation information is received from decoder; data and address information are transferred from memory unit, and then proﬁler stores hot-path and relevant messages into conﬁguration memory. Data in conﬁguration memory will be transmitted into data path of processor to regenerate the reconﬁgurable instruction.

Normally DP-RISP works in ﬁrst mode. During the execution, the execution track information is collected and recorded by proﬁler. When a new block needs to be implemented, DP-RISP switches to second mode to get hot spot and reconfigure hardware resources. After new hardware is organized, it turns back to ﬁrst mode for execution.

4. Retargetable Software Tool Chain

Based on DP-RISP proposed in Section 3, we present a concept of retargetable software tool chain. The retargetable software tool chain is called during the reconﬁguration period. The main architecture of tool chain is shown in Figure.3. It is composed of hot-spot proﬁler, retargetable compiler, simulator, and other related tools (assembler, linker, loader, etc.).

As the traditional RISP tool chain development, the work ﬂow of our proposed software tool chain can still be divided to three stages,

First, source code is compiled into binary code and then executed in simulator. Similarly to ordinary compiler, front-end and back-end tools are both employed to generate binary code. Afterwards, executable codes run on simulator, e.g. simplescalar. At the same time, proﬁler gets runtime information using path based detection method. Runtime information will be transferred to proﬁler and custom instructions are provided.

Then, after custom instructions are obtained, due to limited reconﬁgurable hardware resources, a selection and generation algorithm [14] is called to choose the speciﬁc instruction with highest priority which needs to be implemented in hardware. After the target instruction is ﬁxed, logic array is conﬁgured for new functionality. Since the hardware platform is changed, original instructions are mapped to new designed functional units. In our concept, sources codes are annotated with labels after mapping rule is changed.

Finally, the annotated sources code must be recompiled. To make sure the new code is recognizable by compiler and other tools such as assembler and linker, back-end of compiler need to be re-designed. As a result, the annotated codes will be dispatched to conﬁgurable logic unit for execution.

The above three components will be described in following sections.

4.1 Dynamic Proﬁler

The ﬁrst tool introduced is dynamic proﬁler. The proﬁler is used to keep the execution track and provide hot spot information which helps to generate custom instruction in speciﬁc hardware blocks.

Our proﬁler uses path-detection method, which considers a block which consists of a certain path of running instructions. Proﬁler in this paper is derived from previous work, based on the approach of sampling and path proﬁling [15]. The proﬁler gets the frequently executed instruction blocks through the execution paths.

In contrast with the software proﬁler, we tentatively put forward the hardware implementation in proﬁler design. Traditional proﬁler normally uses software instrumentation, which slows down the execution speed of application. On the contrary, our hardware proﬁler is loose coupled with processor. It can work as an individual slave module of
instruction bus. Consequently processor performs read and write operation as usual, meanwhile profiler collects the instruction information from memory bus. From the bus detector can analyze profiling information at run time and stores hot spots into hot path table for candidate instructions.

4.2 Custom Instructions Generation and Mapping Method

The second issue is the custom instruction generation and mapping method. After we get the hot path using the profiler, we need to choose the specific instructions implemented on hardware logic blocks.

The traditional custom instructions are usually generated from data flow chart in front-end analyzing tools. However due to limited RISP computation resources, we need to design a simple analyzer to generate data flow chart.

In our approach, we propose an algorithm integrating greedy strategy and differential evolution algorithm. The main contribution of the concept is not shown in this paper, you can reach [16] for more information.

4.3 Retargetable Compiler

The third aspect deals with retargetable compilation of the software program. In our concepts, back-end compiler can be divided into three stages: instruction selection, register allocation and instruction scheduling. Based on the three-stage process, we reconstruct the process with mixed code-generation algorithm. The process of algorithm is described in [17].

5. Experiments and Results

In this section, we propose the experiment for the RISP tool chain. As the three parts of profiler, mapper and compiler have not been integrated in one flow, in this paper the above three parts are measured separately.

5.1 Accuracy and Cost of Dynamic Profiler

For the accuracy of profiler analysis, this paper employs Wall weight-matching scheme [17] to design measurement system. We run CRC and MD5 test cases on the profiler to measure the accuracy of profiler. This part of work is already proposed in [15].

In profiling, we define a criterion of profiling accuracy, frequency path size $S(p)$. The parameter relates to the hot spot block size and the execution frequency. $S(p)$ is treated as a major parameter of the accuracy analyzed.

$$S(p) = freq(p) \times Len(p)$$

As shown in (1), $P$ indicates a certain target path, $freq(p)$ and Len $(p)$ are the frequency and length of the path. As profiler get a set of hot spot, we can get a total frequency path size, as the sum of frequency path is of $S(P)$, in (2).

$$S(P) = \sum(freq(p) \times Len(p)) \quad p \in P$$

Figure 4 shows the hot path under the ideal situation of unlimited storage resources. According to the experimental results, when the path selection threshold in the hot spot is less than 0.15%, the coverage of CHDPP is more than 95%.

$$R_N = \frac{|P_{est} \cap P_{act}|}{|P_{act}|} \quad R_S = \frac{S(P_{est} \cap P_{act})}{S(P_{act})}$$

In (4), $R_N$ indicates the ratio of effective hot path number to the ideal number of hot paths during the path analysis. $R_S$ is the ratio of the size of frequent path analysis to the ideal situation. Figure 5 shows the accuracy of CHDPP respectively. On the average, $R_N = 94.1\%$, $R_S = 97.7\%$.

In order to assess CHDPP implementation costs, we use Verilog language to implement CHDPP, and through the ModelSim software for simulation. Synplify Pro 8.0 synthesis tool is employed to complete the first two modules of the integrated. Selected FPGA device is Xilinx Virtex2 series XC2V500. The resources consumed in FPGA are shown in Table 1.
### Table 1. Resource Consumed by Profiler

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Total LUTs</td>
<td>86</td>
<td>Global Clock Buffers</td>
<td>1</td>
</tr>
<tr>
<td>I/O Register bits</td>
<td>37</td>
<td>I/O primitives</td>
<td>92</td>
</tr>
<tr>
<td>BUFGP</td>
<td>1</td>
<td>Register bits (no I/Os)</td>
<td>56</td>
</tr>
</tbody>
</table>

#### 5.2 Custom Instruction Generation and Mapping

To identify and demonstrate the correctness of the instruction set extension, we worked on custom instruction generation and mapping with NetBench [18] and MiBench [19]. Figure 6 shows the number of blocks with different size of instructions.

Figure 6 shows the basic blocks of most applications normally consist of 0–10 instructions, which means the size is quite small. Furthermore, through analysis towards the extended instructions, the basic blocks normally contain less than ten custom instructions, shown in Figure 7.

Table 2 shows the extended instructions of CRC32 application. The instructions ID was named with basic BlockID.InstructionID.InstructionName. As basic blocks in most applications have less than 7 instructions, we design an example with 6 instructions for demonstration. The functionality of instruction in first column represents the combination of instructions in the whole table line. For example, instruction 0.6: add1 is combined by 6 different instructions, from 0.0: xor to 0.6: add.

### Table 2. Extended Instructions (CRC32)

<table>
<thead>
<tr>
<th>ID</th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
<th>4th</th>
<th>5th</th>
<th>6th</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.6:</td>
<td>xor</td>
<td>andi</td>
<td>slli</td>
<td>movhi</td>
<td>addi</td>
<td>add</td>
</tr>
<tr>
<td>0.6:</td>
<td>xor</td>
<td>add</td>
<td>0.6:</td>
<td>slli</td>
<td>movhi</td>
<td>add</td>
</tr>
<tr>
<td>0.6:</td>
<td>xor</td>
<td>andi</td>
<td>slli</td>
<td>movhi</td>
<td>addi</td>
<td>add</td>
</tr>
<tr>
<td>0.6:</td>
<td>xor</td>
<td>add</td>
<td>0.6:</td>
<td>add</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.6:</td>
<td>xor</td>
<td>add</td>
<td>0.6:</td>
<td>add</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.6:</td>
<td>xor</td>
<td>add</td>
<td>0.6:</td>
<td>add</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.6:</td>
<td>xor</td>
<td>add</td>
<td>0.6:</td>
<td>add</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.6:</td>
<td>xor</td>
<td>add</td>
<td>0.6:</td>
<td>add</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.6:</td>
<td>xor</td>
<td>add</td>
<td>0.6:</td>
<td>add</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2 shows the results of instruction generation and mapping method. From this approach, we can derive a new combination instruction from original instruction sets.

#### 5.3 Retargetable Compiler Experiment

The retargetable compiler is implemented from gcc-3.1. We used Nios II instruction set for template, and combined GCC with uclibc-0.9.29 to implement the cross-compiler. SimpleScalar is ported to NiosII platform to run CRC and MD5 test cases. Reconfigurable instructions are treated specially through the modification of back-end modules.

Figure 7 shows the sizes of extended instructions.

Figure 8 shows the experiment results in different resources limitations.
Experiment results in different resources limitations are shown in Figure 8[17]. As the reconfigurable resources increase from 0.3 to 1.0, the CRC and MD5 application code size and reconfiguration time decreases, meanwhile, the speedup of application increases from 1.7 to 5.1 with the reconfigurable resources. From the results we can tell, with the support of retargetable compiler, application can be compiled into smaller code sizes while obtaining a considerable speedup.

6. Summary
New design methodologies need to be concerned for dynamic instructions in RISP. This paper is based on our previous parts of work and draws a conclusion of RISP tool chain support. In this paper we introduce a dynamic profiling RISP as the fundamental processor, and propose a software tool chain. Firstly the proposed software tools use dynamic profiling to get the hot spot of an application, and then implement it in hardware logic blocks. After function block is configured and deployed, application need to be recompiled and mapped to new hardware for execution. The proposed tool chain can guarantee the reuse and coherence of application after hardware reconfiguration. Experimental results demonstrate that profiler can achieve high accuracy of 97.1% with limited hardware cost. A mapper shows the feasibility of instruction generation and mapping mechanism. Retargetable compiler can reduces code size and reconfigurable time while accelerating application execution.

ACKNOWLEDGMENT
This work was supported by a grant from the National High Technology Research and Development Program of China (863 Program, No. 2008AA01Z101), and we owe many thanks to anonymous reviewers and editors.

REFERENCES