The Journal of Systems and Software 125 (2017) 427–438

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Hot spots profiling and dataflow analysis in custom dataflow computing SoftProcessors

Chao Wang[a], Xi Li[a,*], Huizhen Zhang[c], Aili Wang[b], Xuehai Zhou[a]

[a] Department of Computer Science, University of Science and Technology of China, China
[b] School of Software Engineering, University of Science and Technology of China, China
[c] Department of Computer Science, Huaqiao University, China

ARTICLE INFO

Article history:
Received 15 May 2015
Revised 8 July 2016
Accepted 9 July 2016
Available online 15 July 2016

Keywords:
Adaptive processor
Dynamic profiling
Hot spot
Instruction set extension

ABSTRACT

In the past decades, instruction set extension problem has been a key research area for state-of-the-art design automation of Very Large Scale Integration (VLSI) systems. Meanwhile, recently there is a renewed interest for hot spot profiling and dataflow analysis in custom instruction set processors. This paper proposes HOTISE, an architecture framework for adaptive reconfigurable instruction set processors (RISP) with dynamic profiling and dataflow analysis. A dynamic profiler is employed to obtain hot spots for each application at run-time. Then the selected hot spots will be considered as custom instructions and implemented in reconfigurable logic arrays. An instruction generator based on dataflow generation provides a mapping scheme from each selected instruction to hardware processing element in the array. To demonstrate the accuracy and feasibility of HOTISE, we have implemented a profiler prototype using simulator-based RTL codes. Experimental results show that the profiling results can cover more than 97% hot spots of MiBench and NetBench applications. In particular, the custom instruction of CRC and MD5 application proves the effectiveness of the mapping mechanism, the code sizes of CRC and MD5 could decrease to 32.5% and 37%, while achieving the speedup at 4.7x and 5.1x, respectively.

© 2016 Elsevier Inc. All rights reserved.

1. Introduction

Adaptive and reconfigurable computing technology are becoming one of the dominant technologies in design automation of electronic systems, not only in the traditional domains as multi-agent systems, self-organizing computing systems, peer-to-peer networks, etc. but also expand to the hardware and processor architectures. Past decades have witnessed the tremendous invasion of heterogeneous processor architectures in the field of reconfigurable computing (Compton and Hauck, 2002). As we expand the use of computing devices, capabilities beyond raw performance such as flexibility, scalability, and programmability are becoming increasingly important. To achieve multiple design objectives, reconfigurable computing platforms on Field Programmable Gate Array (FPGA) and Graphic Processing Unit (GPU) approaches have been considered as the promising future microprocessor design paradigms (Borkar and Chien, 2011). Compared to GPU methods, reconfigurable computing can achieve more flexibility to run different programs, either on single core processor or on multiprocessor system-on-chip (MPSoC) platforms (Singh, 2011; Wang and Li, 2014). Both approaches are widely used in the state-of-art processor design literature, among which the single processor solution is recognized as customizable processors or adaptive processors (Ienne and Leupers, 2006). In this paper, we introduce an architecture framework of adaptive processors with dynamic profiling techniques, in which reconfigurable technologies are integrated into Application-Specific Instruction-Set Processors (ASIPs).

For the sake of hardware implementation, RISP is generally fabricated on a single chip with reconfigurable logic units, such as FPGA and Complex Programmable Logic Device (CPLD). After several years of primarily being a research area, there are now several credible commercial reconfigurable processors.

Throughout this paper, the terms GPP, ASIP, RISP refer to following definitions:

- **GPP** stands for General Purpose Processor, in which the instruction set is fixed and cannot be modified after fabrication.
- **ASIP** refers to Application Specific Instruction Processor that can provide dedicated ISA for specialized applications, but the ISA cannot change after fabrication either.
- **RISP** represents Reconfigurable Instruction Set Processor. RISP is extended from ASIP with a reconfigurable logic block, in

* Corresponding author.
E-mail addresses: cswang@ustc.edu.cn, saintwc@mail.ustc.edu.cn (C. Wang), llxx@ustc.edu.cn (X. Li), zhanghz@hqu.edu.cn (H. Zhang), wangal@ustc.edu.cn (A. Wang), xhzhou@ustc.edu.cn (X. Zhou).

http://dx.doi.org/10.1016/j.jss.2016.07.025
0164-1212/© 2016 Elsevier Inc. All rights reserved.

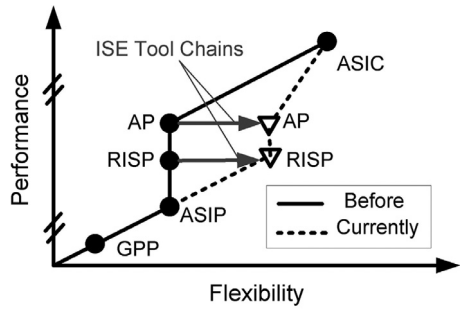

Fig. 1. Comparison between different processors, ISE toolchains should generally make AP and RISP more flexible as the generation of new instructions.

which the extended instructions can be implemented either statically or dynamically.

- **AP** refers to Adaptive Processors, in which ISA can be reconfigured automatically all by itself without user intervention. In particular, AP is a specific kind of RISP with adaptive and dynamic reconfigurable conditions.
- **ASIC** refers to the Application Specific Integrated Circuit, which has dedicated functionalities and cannot be modified after fabrication.
- **Hot Spot** is an instruction sequence that executes with highest frequencies. Because a branch instruction could result in unexpected results, we only consider the sequences within one basic block.

Fig. 1 illustrates the comparisons of the performance and flexibility between GPP, ASIP, RISP, AP and ASIC. Out of these approaches, GPP gains the lowest performance with the least flexibility. Due to that ASIP is specially designed for some critical applications. Therefore it achieves higher performance and flexibility than GPP. Then AP and RISP are particular instances of ASIP. Therefore it needs additional programming support for custom instructions, i.e. static for RISP while dynamic for AP. Finally, ASIC is a solid circuit that cannot be programmed with the highest performance. Another overlooked aspect is that with the support of instruction set extension (ISE) tool chains, the flexibility of AP and RISP can be significantly improved. As a consequence, AP can efficiently leverage performances and programming with ISE toolchain support, which could further result in the shorter time to market and lower design complexity.

However, it still poses a significant challenge on several key issues of AP design. To select from the candidate hot spots, profiling technologies are primarily considered. A profiler is employed to analyze the frequencies of different functions or codes in an application, by which means to find out which part costs most time during execution. During the instruction extension process, the identification, definition and implementation of those selected operations providing the largest performance improvement should be hardwired (Galuzzi and Bertels, 2011).

To tackle the above problem, we propose HOTISE in this paper; a custom instruction set extension and generation method with dynamic profiling techniques for hot spots. The HOTISE toolchain is based on the previous work on Wang and Zhang, (2011). Like the general ISE solutions, HOTISE consists of a dynamic profiler and an instruction mapping mechanism. The main contributions of this article can be summarized as follows:

- First, we propose a configurable dynamic profiler to identify the hot spots which are instruction sequences within a basic block in a particular application. The profiler works as a slave module attached to the instruction bus, which consequently will not drag down the overall execution performances.
- Second, the identified hot spots can be considered as single custom instructions, after the instructions have been implemeted as the accelerator. We present a custom instruction mapping method based on the reconfigurable arrays.
- Finally, we use CRC and MD5 test cases to demonstrate the accuracy of the profiler and effectiveness of the mapping scheme, as well as the resources utilization.

The remainder of this paper is organized as below. In Section 2 we present related work on adaptive processors and toolchain design methodologies. After that in Section 3 we detail the custom instruction set execution with profiling techniques. In Section 4, we measure the accuracy of the profiler, the optimized results of the profiler, and the transferring time from the host processor to the accelerator. Section 5 provides a feasible custom instruction example for MiBench and NetBench. Finally, Section 6 concludes the paper.

2. Background and related work

Academic and industrial researchers have highly valued the instruction set extension. There are plenty of design tools based on RISP architecture. Besides the above RISP toolchain literature, there are also conducted researches summarized in Galuzzi and Bertels, (2011). In this section, we first present some related work on the general toolchain framework for the adaptive processors, then illustrate dynamic profiling techniques, and instruction generation respectively.

2.1. General framework and adaptive processors

Adaptive processors play a vital role in microprocessor design methods. Among current studies, Very Long Instruction Word (VLIW) has been widely used in modern RISP processor design. XiRisc (Lodi and Toma, 2003) states a load-store structure, in which all data access simply use two kinds of instructions, sharing register files. Iqbal (Iqbal and Awan, 2009) introduces RT-RISP, which defines single instruction implemented as separate modules, through partial reconfiguration. The modules are scheduled dynamically on the principle of requirements of the application to achieve better performances. MOLEN (Vassiliadis and Wong, 2004; Kuzmanov and Gaydadjiev, 2004) integrates hardware reconfiguration as a part of processor design. By using reconfigurable microcode, hardware tasks are considered as atomic operations.

On the general framework, most closely related to our work are i-Core (Henkel and Bauer, 2011), MOLEN (Panainte and Bertels, 2007), AMBER (Noori and Mehdipour, 2008), ARISE (Vassiliadis and Theodoridis, 2009).

AMBER toolchain (Noori and Mehdipour, 2008) is based on the classic SimpleScalar simulation environment. It firstly gets the start address of hot spot basic block by profiling, then reads the codes, generates dataflow graph (DFG) and related instruction list. However, since the profiling mainly depends on basic blocks instead of the overall execution paths, the performance improvement is limited by the basic block classification.

ARISE architecture (Vassiliadis and Theodoridis, 2009) treats both software and hardware as separate modules, through which it can detect customized instructions automatically and support hybrid calculation model within hardware/software development.

EnCore (Almer and Bennett, 2009) processor enhances GNU Compiler Collection (GCC) with an end-to-end software tool chain design methodology. It can deal with the search and selection for multi-input and output instruction templates. But due to static profiling approach, the generated DFG of EnCore is inaccurate with the real results; therefore the capability and performance are restricted.

Furthermore, there are also both software and hardware frameworks those can be adaptive to changing hardware architecture. Dynamo (Vasanth and Evelyn, 2011) and software prediction methods (Meeuws and Yankova, 2007) are the demonstrative works in software, and there are also some credible approaches in hardware (Ganesh and Mark, 2010). For example, Leupers and Karuri, (2006) propose a design flow for configurable embedded processors based on optimized instruction set extension synthesis. Bonzini and Pozzi, (2007) present a retargetable framework for automated discovery of custom instructions. However, as these state-of-the-art methods can be used for leverage the software/hardware partition and mapping plans when the custom instructions are mapped to the hardware logic arrays, current existing tool chains have the limitations of low efficiency during hot spot recognition stage. We thereby employ a dynamic hardware path profiler into HOTISE design, which can not only increase the accuracy of hot spot location effectively but also reduces the profiling overheads to improve the custom instruction performances.

2.2. Dynamic profiling techniques

One main contribution of this paper is to introduce dynamic profiling into AP architecture framework with toolchains. Currently, there are several credible approaches with profiling technologies.

In profiling methods, the notion of program paths has always attracted significant academic interest. Ball and Larus (Ball and Larus, 1996) demonstrated the feasibility of obtaining path profiles automatically and efficiently. The proposed instrumentation based profiling algorithm splits dynamic instruction streams into acyclic, intra-procedural sequences and tracks their occurrences. But such paths do not provide adequate information about a program's control flow across procedure and loop boundaries. Moreover, the algorithm tends to have high overheads, contributing 31% and 16% on average for path and edge profiling.

Subsequent efforts have focused on overcoming the information limitation (Larus, 1999). They can profile the programs during execution across procedure and loop boundaries and reach high accuracy consequently. But the overhead is still regarded as the major bottleneck, which is one of the most critical factors in cost-sensitive dynamic optimization systems.

For the overhead issues, there are many approaches conducted targeting the high-efficiency solutions to reduce costs. Some online profiling techniques collect a one-time profile for edges or paths for applications. For example, structural path profiling (Toshiaki and Toshio, 2003) receives a unique path with dynamic instrumentation which is removed later when the paths are recorded. But this method may not capture the whole-program behavior and performance will suffer consequently. Based on the idea of profile-guided optimization, Bond et al. (Bond and McKinley, 2005) proposed a practical path profiling scheme that simplifies path profiling instrumentation using an edge profile. Apparently, the profiling accuracy and performance depends on the characteristic of the edge profile. In contrast, sample-based approaches can avoid high run-time overheads of software instrumentation. Anderson (Anderson and Berc, 1997) described a digital continuous profiling infrastructure that collects a basic block profile by occasionally sampling the program counter. Bond and Kathryn (Bond and McKinley, 2005) optimized B-L algorithm by a sampling mechanism. They assigned each path with a unique number according to profiling information, but the approach requires additional architecture support of switching mechanisms and operation systems. Arnold and Grove (Arnold and Grove, 2005) also proposed a sampling-based profiling technique to obtain accurate information efficiently, and Bond (Bond and McKinley, 2005) improved it with a simple mechanism to reduce the overhead.

Furthermore, the importance of profiling in improving performance has been recognized by computer architects. Most processors, like SPARC and Pentium-IV, provide some event counters for performance monitoring. To meet the requirement of profile-guided optimizations, hardware profilers have been proposed (Toshiaki and Toshio, 2003), which has constructed approximate point profiles using information from processors. K. Vaswani (Vaswani and Thazhuthaveetil, 2005) implemented a hardware path profiler based on stacks and hash tables. The profiler could collect various types of path profiles and associate architectural metrics with program paths.

2.3. Instruction generation and mapping

In general, custom instruction generation methods can be classified into three groups: instruction decomposition, instruction reunion, and instruction composition. For the decomposition method, in the beginning, one original instruction is divided into multiple microinstructions, and then all the selective micro instructions are combined into a custom instruction (Leupers and Karuri, 2006). Meanwhile, the most commonly used approaches are based on instruction composition.

Then a significant step is to select the candidate instructions which are to be mapped from DFG to the FPGA arrays. The procedure includes two phases: candidate instruction selection and the final instruction preparation. Once the hot basic blocks (HBB) are derived, one way to generate custom instructions is to utilize Multiple Input Single Output (MISO) cluster algorithms on the DFG. Kastner proposed a profiling based pattern to create an iterative custom instruction template (Kastner and Kaplan, 2002), but it only applied to the situation when the instruction set is at a small scale. ISEGEN (Biswas and Banerjee, 2006) follows the Kernighan-Lin min-cut heuristic algorithms for custom instruction generation. Atasu and Pozzi, (2003) also illustrate an alternative paradigm with automatic application specific instruction-set extensions, which supports multi-output and can be applied to direct acyclic graph model under microarchitectural constraints situation. CRISP (de Beeck and Barat, 2001) refers a model through Design Space Exploration (DSE) that can meet different levels of restrictions.

After the candidate custom instructions are identified, the next step is to illustrate the selection among the candidate custom instructions. Of current approaches, Arnold (Marnix and Henk, 2001) presents a dynamic programming based approach to derive the instructions without user constraints. However, in the real applications, most instructions will be confined to various run-time situations, such as performance and area. Cheung (Newton and Jorg, 2003) raised a greedy strategy which first considers the area constraints for candidate custom instructions and then selects the instruction with the max speedup from simulation results. Lee and Choi, (2002) and Pan and Tulika, (2004) both propose heuristic algorithms are accounting for execution time, speedup, area consumption, and other parameters, and analyze the tradeoffs between heuristic and linear programming approaches.

These state-of-the-art methods can be used for leverage or guidance during the mapping methods design to generate the custom instructions to the hardware logic arrays. In Table 1 we list some example references for each type of research, with their strengths and weaknesses.

3. HOTISE framework for coarse-grained adaptive processors

In this section, we present the HOTISE architecture framework for adaptive processors. In particular, the HOTISE working process is divided into three steps, as shown in Fig. 2.

Table 1
Summary of adaptive processor and framework.

Research	Example references	Strengths	Weakness
General framework	i-Core, AMBER, ARISE, ENCORE, dynamo	Automatic generated retargetable toolchain	Capability and performance restricted
Adaptive processors	XiRisc, MOLEN, RT-RISP, domain-specific processors	Reconfigurable and custom instructions	Tightly coupled with systems
Dynamic profiling techniques	Call graph profiles, efficient path profiling	Achieve hot spot most with software profiling	Dependent on processor architectures
Instruction generation and mapping	ISEGEN, CRISP	Mapping instruction to hardware logic	High complexity

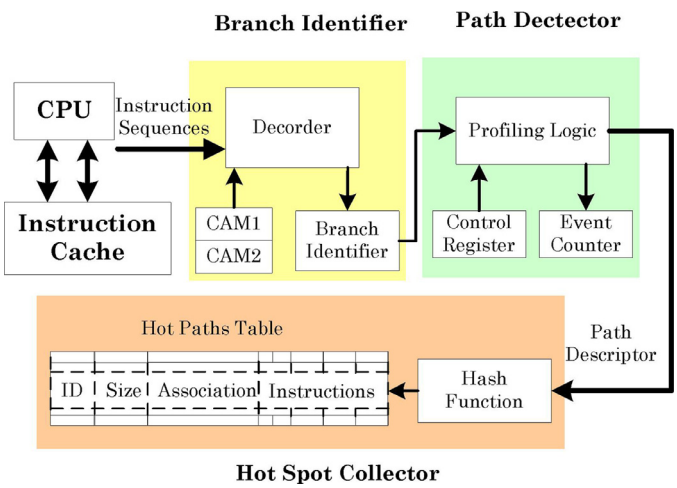

Fig. 2. The profiler framework.

(1) The profiling and path collection step

First, source applications are compiled into binary codes and then executed on the microprocessor. At the same time of the application execution, a profiler detects and collects the execution paths of the application. In this paper, HOTISE is based on a coarse-grained FPGA, in which each execution path is an instruction sequence. Then all the reserved paths are classified, and the most frequently executed paths are stored in the Hot Path Table (HPT).

(2) The instruction selection step

Second, after custom instructions are obtained, an instruction selection algorithm chooses the specific instructions with the highest priority to be implemented in hardware. After the target instruction is implemented, the logic array is configured for new functionality. Then the processor will include both the reconfigurable logic array and the microprocessor. The custom instructions are executed on the logic array, with the data transferred from the microprocessor via on-chip interconnection, e.g. Xilinx FSL links. Since the ISA is already changed, custom instructions should be remapped to new designed functional units.

(3) The recompilation step

Finally, the source program must be recompiled to make the compiler recognize the new codes. In the back-end of the compiler, the program will undergo a process including following three stages: instruction selection, register allocation and instruction dispatch. As a result, the original hot spot region in the application will be transmitted to the configurable logic unit for execution.

3.1. Configurable dynamic path profiler

To identify the hot spot, a dynamic profiler (HAProf) which is enhanced from Zhang and Ji, (2009) is introduced at first. HAProf records the execution paths at run-time and identifies hot spots which are potential candidates for custom instructions. HAProf is based on path sampling and profiling mechanisms. As standard jump and branch instructions lead to different execution flows at run-time, associating one single macro instruction with these instructions is difficult. Consequently, in this paper, we only take into account the instruction sequences residing in one basic code block (A basic code block refers to the code block between two neighbor branch instructions).

Rather than implementing the profiler in software, we implemented it as a component attached to the instruction bus. Consequently the microprocessor is capable of performing read and writing operation as usual, meanwhile the profiler collects the instruction from instruction bus. HAProf can be integrated with processors across a wide range of instruction set architectures. The diagram in Fig. 2 illustrates the general framework of HAProf, including three modules named branch identifier, path detector, and hot spot collector. First branch identifier listens to the instruction buses and fetches the instruction sequences. Then path detector records the frequency of the executing paths during the execution and generates path descriptors. Finally, the hot spot collector stores the descriptors and stores the paths into the Hot Path Table.

(1) Branch identifier

Branch identifier consists of three components: CAMs (Content Addressable Memories), decoder and branch identifier. The contents in CAMs include all opcodes of flag instructions (like branches) and masks for extracting opcodes from instructions. Then decoder fetches the instruction from buses and looks up in CAMs to determine whether the current instruction is a branch. If yes, decoder passes it to branch identifier, where the branch instructions are analyzed based on the addresses of instructions.

(2) Path detector

Path detector tracks the program execution by the classified branches and generates path descriptors. The detailed path detecting algorithm is presented in Algorithm 1.

The path detector receives branch type, address and current directions as input parameters. After the execution, path descriptors for the hot spot are identified. Within the algorithm, a control loop keeps the hot spot record dynamically. The parameter of loop count and length count are used to mark the number of loops and hot spot size respectively. Moreover, the loop count is treated differently for the forward (line 5∼9), backward (line 10∼15) and CALL/return (16∼17) branch types. If the loop count reaches the preset value or the length exceeds the max length that can be stored into the table (line 6 and 12), then the current path can accept no more instructions for a current hot spot.

(3) Hot spot collector

The data paths of hot spot applications are stored in the info collector. Hot spot collector is based on a data structure called Hot Path Table (HPT), which holds multiple entries to store path descriptors and the frequency of the path. Info collector uses a novel collecting mechanism based on a hash function and replacement policies to store the significant paths. The path collecting mechanism determines the efficiency of the hot path collection, which

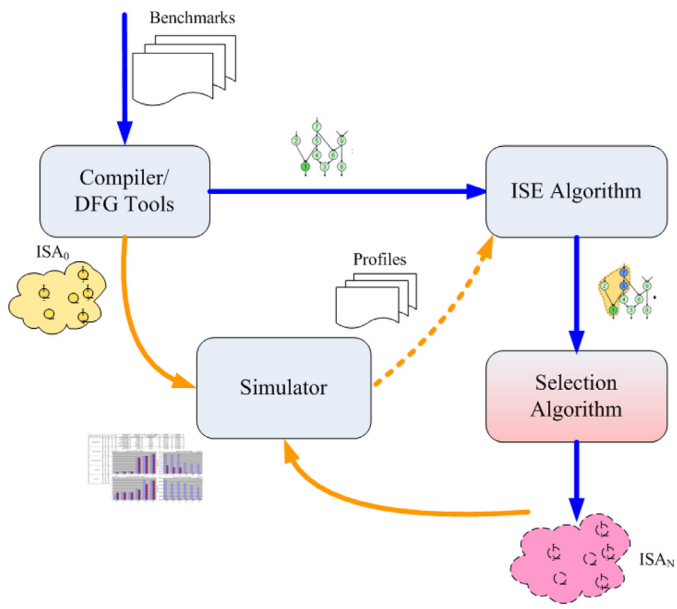

Fig. 3. Instruction set extension process.

includes the basic parameters about HPT structure, hash function and replacement policies.

a. **HPT structure**. The attributes of the HPT structure include the size, association and the instructions. The size of HPT depends on the number of significant paths in programs, and it can be as large as it is suffered in hardware to reach high accuracy. To obtain better storage utilization, the association mechanism in cache design is envisioned.
b. **Hash function.** The structure of the path descriptor illuminates the fact that the number of potential paths is so large that HPT is too small to hold them; therefore the indices for HPT cannot be the descriptors though they are the best choices. To store paths, it needs a hash function to generate the indices that come from path descriptors. The hash function should be simple for hardware and cause conflicts as few as possible.
c. **Replacement policy.** As our purpose is to store high-frequency paths in HPT, the Least Frequently Used (LFU) algorithm is the obvious choice for replacement. Info Collector receives a path descriptor and calls the hash function to compute an index into HPT. If a corresponding entry is found or unused yet, the path descriptor is stored, and the related information is updated. If the lookup fails, the first unused entry of the indexed entry set is selected and initialized with information about the incoming path descriptor. If all entries in the collection have been used, LFU algorithm will be employed for replacement.

3.2. Custom instruction generation with dataflow analysis

Given the target functions as hotspots, custom instructions should be selected and considered to be implemented via hardware logic blocks. Traditionally custom instructions are usually generated from dataflow chart in front-end analyzing tools. In this paper, the custom instructions are generated statically after hotspot information is obtained, and the generation is composed of three phases: dataflow analysis, custom instruction extension, and custom instruction generation. The instruction extension process is shown in Fig. 3.

(1) Dataflow analysis

Dataflow analysis is employed to analyze the generated instruction sequence and choose the custom instructions. Since all the instructions are regarded as the nodes in a dataflow graph (DFG), we first designed an algorithm named BuildDFG to generate the DFG. The BuidDFG algorithm is shown in Algorithm 2.

In this paper, the codes between two branch instructions are regarded as a basic block (denoted as bb in the algorithm). To keep the record of the producers of dataflow, BuildDFG maintains a register table. Once a new instruction arrives, the register table should be checked whether data already exists in the table. If not, then the associated registers are marked as LiveIn registers; once the basic block is finished, all the non-empty registers are marked as LiveOut registers.

This algorithm outlines how the dataflow is analyzed by building a dataflow graph. The input parameters of BuildDFG consist of the basic block, source and destination registers, while the output parameters are LiveIn and LiveOut registers. The algorithm looks up for every basic block to mark the registers to source registers (Sreg) or destination registers (Dreg). The Sreg and Dreg identify the input and output parameters for custom instructions. The variable *sindex1* indicates the flag of the LiveIn registers, and is initialized to -1. If the *sindex1* is still -1 after the *getproducer* operation is processed, then the current register could be the LiveIn register.

(2) Custom instruction selection

After the dataflow graph is generated, the custom instruction can be selected. Some instructions, such as memory access (load & store) instructions, are not cost effective and are not included in custom instructions. Moreover, there are limitations that should be considered during instruction-set extension:

– Number of operands. Considering the limitation of the register ports, the operand number must not overpass the number of registers.
– A number of custom instructions. The size of instruction code limits the number of generated custom instructions.
– Area constraints. All the custom instructions are implemented in hardware (e.g., FPGA, CPLD). Thus their hardware implementations area must not exceed the space limitations.

The instruction set extension problem can be defined as a graph problem which is described as follows:

Definition 1. Construct a directed acyclic graph *G(V, E)* which represents the data dependencies between different instructions, in which *V* represents instructions, while *E* represents data relations. In the graph, each vertex *v* represents nodes included in custom instructions. Assume *P* is the node set under processing. All nodes which require external data are in *IN(P)*, and those producing external data are in *OUT(P)*. The limitations of the number of inputs and outputs are M_{in} and M_{out}, respectively. *PATH (u, v)* is the path from node *u* to node *v*. The custom instruction problem is to find a suitable node set *P* to meet the following constraints: first the node number in *IN(P)* and *OUT(P)* should not exceed the limitation of M_{in} and M_{out},. Then for each pair of nodes v_i and v_j in *P* and v_k not in *P*, the path between v_i and v_k and v_j and v_k should not share any common nodes.

Since in the instruction sequences not all the instructions have data hazards, the sequences can be divided into several clusters. All the clusters have no data dependence between each other, so we only need to check the constraints inside each cluster and then combine the clusters.

The number assigned to each node indicates the execution order. In one cluster, each instruction has data paths to every other instruction. Different clusters need to be marked with individual

colors corresponding to different categories. The algorithm is described in Algorithm 3.

The marking algorithm checks each node in the graph. First, it checks whether the current node is valid. The term *regmark* refers to the number of successor nodes already marked. The function ***getminregmark()*** in line 7 returns the minimum ID from the current node and all its successor nodes. If all the successor nodes and the current node have not been marked, then a new cluster is created including all these nodes. Otherwise, the marks of the successor nodes should be updated (lines 12-17).

(3) Custom instruction generation

After the second step, we can get all the instruction clusters. For each cluster, if all the constraints could be met, the whole cluster can be regarded as a candidate custom instruction. However, in most situations, to get fine-grained parallelization, the cluster has to be divided into several smaller groups, each of which is a standalone custom instruction.

We design an algorithm to list the instruction groups, as shown in Algorithm 4. Each group is regarded as a subgraph. At first, we exploit the entire connected sub graph for each node, then extend and merge the subgraph in predecessor and successor: Algorithm 4 outlines the subgraph enumerating process that enumerates the subgraphs. For each cluster, at first, all the nodes in the cluster are marked. Since each vertex refers to one instruction, and the vertex is inserted into the target set. Then all the predecessor nodes are marked. After that, the connected graph for each vertex is merged to graphs. Furthermore, if different merged graphs have certain violations (e.g. two graphs share one vertex), then only one graph can remain. Besides the merging operation for subgraphs, the merging process is also responsible for the combination of the subgraphs in predecessors and successor directions.

3.3. Interconnection

Since the accelerator is deployed with the processor in a single chip, the interconnect structure needs to be carefully considered in order to maintain a high speed of data transmission. There are plenty of capable state-of-art networks on chip solutions in the market. For small scale platforms (less than 10 accelerators), the system employs pairs of fast simplex link (FSL) buses between the scheduler and accelerator, which is a one-way FIFO-based peer-to-peer link utilized between hardware modules for fast communication. Furthermore, in the Xilinx FPGA prototyping system, we integrate Microblaze microprocessors that can perform read and write FSL bus transactions using specific instructions directly. Each accelerator is connected with the main scheduler and all task requests and source data are sent from the scheduler. Moreover, as all the tasks are assigned by the scheduler, there are no direct data paths among the accelerators. If one task requires the results from some other task, these tasks must be issued in sequence, otherwise, the tasks can be dispatched to different accelerators at the same time and run in parallel.

In general, tasks can be classified into three categories for the overall system: hardware tasks which can only be run on hardware computing servants to meet the constraints; software tasks that can only run on software servants, e.g. I/O functions and debugging operations; and software/hardware (S/H) tasks that can either run on software or hardware, depending on the scheduling decision made by the scheduler. In particular, the scheduling and mapping plan for the third type has a significant impact on the final system speedup and throughput. Each S/H task is bound to a type of software function or hardware module. The scheduler runs an online scheduling algorithm to detect data dependences automatically. If the tasks are independent of each other, that is, the inputs of one do not depend upon the outputs of another, then these tasks can be spawned to heterogeneous computing servants in parallel. However, if different tasks that are destined for different accelerators require the outputs of one to be provided as inputs to the other, then these tasks are prevented from being executed out-of-order. In this case, these tasks must be issued in sequence. The output consuming task has to wait until the output producing task is finished and releases the required (dependent) data.

Table 2
Percentage of different custom instructions.

OpCode	Opcode semantics	Rate (%)
add	Addition	18.72
addi	Addition imm	12.13
andi	Logical AND	11.47
xor	Logical exclusive XOR	6.71
slli	Logical left shift	8.97
srli	Logical right shift	1.38
movhi	Move high bytes	11.66
cmpne	Comparison	0.55
sub	Subtraction	3.39
subi	Subtraction imm	1.04

4. Experiments and results

In this section, we propose the test of the HOTISE toolchain. We first measure the accuracy of the dynamic profiler and then give an example of the custom instruction generation. Before the profiler and instruction mapping is tested, Fig. 4 shows the basic block size from profiling for Netbench and Mibench. The x-axis is the size of the basic block, and the z-axis is the number of such blocks in a program. In most applications, the basic block size is less than 10, which means the common basic block only contains less than ten instructions. Furthermore, Fig. 4 gives a summary of the biggest block size for each application. Most basic blocks contain less than 100 instructions, except for the crypt applications includes more than 500 instructions.

On the frequency of candidate instructions, Table 2 summarizes a frequency statistics of different arithmetic instructions. ADD, ADDI, ANDI, and MOVE are most used instructions based on the profiling. Meanwhile, the data transfer instructions are cost effective (as load/store). These results could guide the custom instruction selection during the extension procedure.

4.1. Accuracy of the profiler

To evaluate the performance and the overheads (power, area, etc.), we measured the accuracy of the dynamic profiler. The accuracy metric reflects the difference between the hot spots explored by the profiler and the hot spots that reside in the original application (Arnold and Grove, 2005). To evaluate the accuracy of HAProf, we employ the classic Wall weight-matching scheme (Wall, 1991) in measurement methodologies. We run CRC in MiBench and MD5 in NetBench as the test applications.

In particular, we define a criterion of profiling accuracy using frequency path size S(p). The parameter depends on both the hot spot block size and the execution frequency. S(p) is treated as a significant parameter of the accuracy analyzed.

$$S(p) = freq(p) \times Len(p) \tag{1}$$

As shown in (1), p indicates a certain target path, while freq (p) and Len (p) refer to the frequency and length of the path respectively. Assume P is the total path set for the whole application, and the sum of frequency path is S (P). As profiler gets a subset of paths belong to hot spot applications, we can get a total frequency of all the paths in the hot spot (2).

$$S(P) = \Sigma(freq(p) \times Len(p))\ p \in P \tag{2}$$

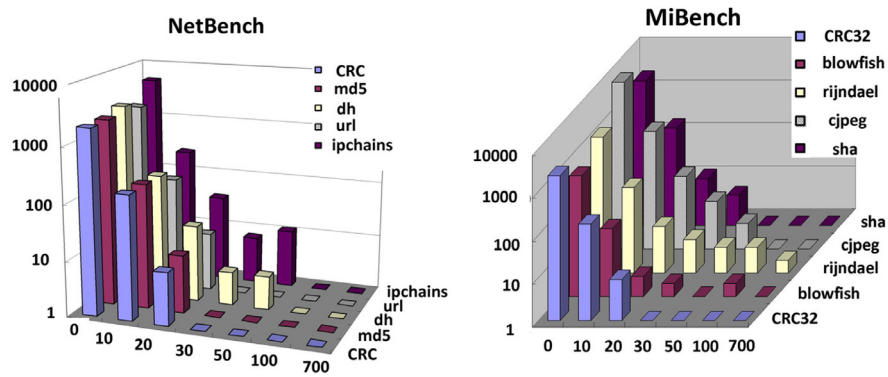

Fig. 4. Basic block size for NetBench and MiBench.

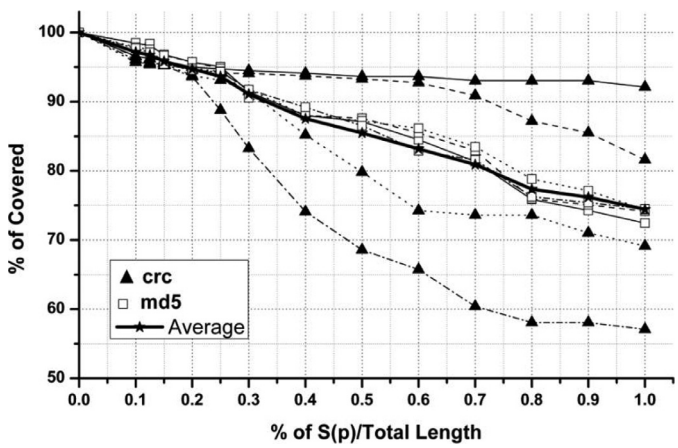

Fig. 5. Identification of hot paths.

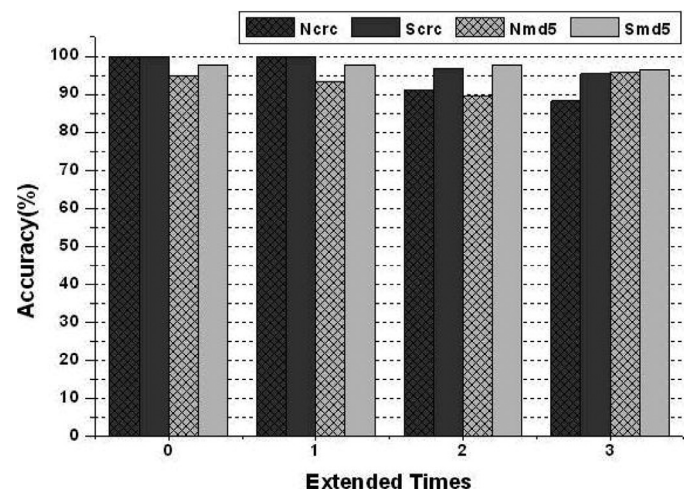

Fig. 6. Accuracy of HAProf.

Fig. 5 presents the coverage of the hot path with ideal storage resources. The X-axis is the percent of S(p) by the total length of an application, while the y-axis is the coverage. According to the experimental results, when the path length threshold in the hot spot is lower than 0.15%, the coverage of the hotspot is higher than 95%. The coverage is dramatically reduced with the increase of the path length. Due to that CRC benchmark includes more loop based basic blocks; therefore more executed paths will be generated at runtime, which results in a bigger variation.

Let P_{ideal} be the collection of the ideal hot paths and P_{all} represents all the paths. Furthermore, the ideal hot path set is presented in (3).

$$P_{ideal} = [p|S(p) \geq S(P_{all}) \times 0.15\%] \tag{3}$$

According to the number of paths in the P_{ideal} collection, we classify the S(p) values in sequence, paths with the same number of sets P_{real} are selected. From P_{est} and P_{ideal}, we define the term accuracy of R_N and R_S.

$$R_N = |P_{real} \cap P_{ideal}|/|P_{ideal}| R_S = S(P_{real} \cap P_{ideal})/S(P_{ideal}) \tag{4}$$

In (4), R_N indicates the ratio of useful 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. Fig. 6 shows the measured R_N and R_S accuracy parameters of HAProf respectively. On the average, $R_N = 94.1\%$, $R_S = 97.7\%$, which demonstrates HAProf can achieve satisfying accuracy to identify the hot spots. Table 3 summarizes the power consumption and hardware cost in different modules. From the table, we can get that large proportional power consumption is on the block RAM, which reaches $25/35.7 \approx 70.03\%$. In particular, the power consumption of the profiler is quite low that it can be ignored compared to the host processor and Block RAM modules.

Table 3
Power consumption and hardware cost of the entire system.

IP cores	LUTs	Area (uM^2)	FFs	BRAMs	Power (mW)
Host processor	1650	1331,550	1489	0	9.7
Profiler	86	69,402	56	0	0.1
Block RAM	26	20,982	24	32	25.0
Peripherals	853	688,371	773	0	0.9
Total	2618	2110,305	2342	32	35.7

4.2. Optimization with sampling

To alleviate the overheads of the profiler, we optimized the original profiler with the Arnold-Grove sampling mechanisms (Arnold and Grove, 2005). Fig. 7 illustrates the comparison results between the no sampling, Arnold-Grove sampling (noted as AG) and simplified Arnold-Grove sampling (noted as SAG). The frequency of the timer interrupt used in experiments is 1/1000 of the processor clock; and the notation (STRIDE, SAMPLE) is utilized on the label.

For the programs of nat, drr, route and tl in the MiBench/NetBench applications, the overlaps by sampling are almost equal to the ones without sampling. For the other programs, simplified Arnold-Grove sampling gets better results than Arnold-Grove sampling. The profiling results depend on the value of STRIDE and SAMPLE. When STRIDE is 32 and SAMPLE are 64, simplified

Table 4
Custom instructions example for CRC32.

ID	1st	2nd	3rd	4th	5th	6th
0.6:add1	0.0:xor	0.1:andi	0.2:slli	0.3:movhi	0.4:addi	0.6:add
0.6:add2	0.1:andi	0.2:slli	0.3:movhi	0.4:addi	0.6:add	
0.6:add3	0.1:andi	0.2:slli	0.4:addi	0.6:add		
0.6:add4	0.1:andi	0.2:slli	0.6:add			
0.6:add5	0.2:slli	0.3:movhi	0.4:add	0.6:add		
0.6:add6	0.2:slli	0.4:addi	0.6:add			
0.6:add7	0.2:slli	0.6:add				
0.6:add8	0.3:movhi	0.4:addi	0.6:add			
0.6:add9	0.4:addi	0.6:add				
0.4:addi1	0.3:movhi	0.4:addi				
0.2:slli1	0.0:xor	0.1:andi	0.2:slli			
0.2:slli2	0.1:andi	0.2:slli				
0.1:andi1	0.0:xor	0.1:andi				
0.8:xor1	0.5:srli	0.8:xor				

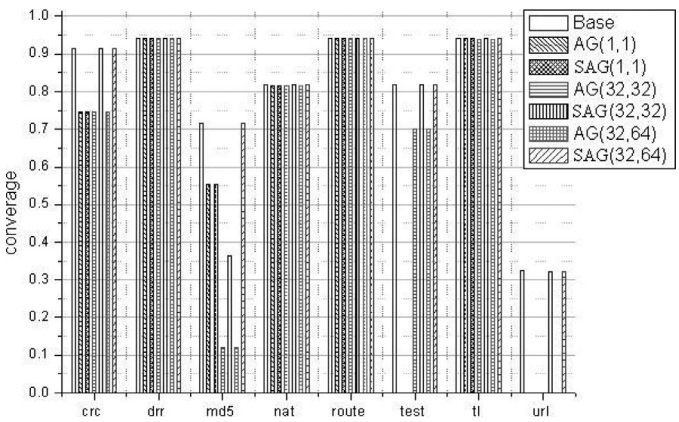

Fig. 7. Results of profiling with different sampling mechanisms.

Arnold-Grove sampling obtains the same results as no sampling. The experiments show that profiling with sampling policies can get the accurate hot information, which achieves up to 95% for typical benchmark applications.

5. Results on instruction generation

5.1. Results on custom instruction generation

After the hot spot is obtained by the HAProf, we can implement the hot spots as custom instructions extensions. To demonstrate the correctness and feasibility of the instruction set extension, we worked on custom instruction generation and mapping using CRC and md5 benchmarks. Taking identified hot spot using profiling into account, Table 4 lists an example of custom instructions of the CRC32 application. The instruction ID in the demonstration was renamed with the manner of **BlockID.InstructionID: InstructionName**. The **BlockID** represents the basic block where the extended instruction is stored; **InstructionID** refers to the specific ID for extended instructions, and **InstructionName** indicates the operand of the instruction. As basic blocks in most applications have less than 7 instructions, we design an example with up to 6 instructions. The functionality of instruction in the first column represents the combination of custom instructions in each table row. For example, 0.6:add1 is the first extended instruction, which is combined with 6 different instructions in the first row, from 0.0: xor to 0.6: add.

5.2. Mapping instructions to reconfigurable logic

In this section, we illustrate an example of mapping instructions to reconfigurable logic. The hardware architecture is constructed on the Xilinx FPGA Virtex 5 development board. To facilitate the instruction level parallelism, we apply the state-of-the-art MP-Tomasulo algorithm (Wang and Li, 2013) for out-of-order instruction execution. Custom instructions are implemented as the standalone unit in the reconfigurable logic array architecture; here we need to discuss the constraints of Opcode and operands for the custom instructions:

Algorithm 1. Path detecting algorithm.

```
Input: Branch type, address and current Direction
Output: The path descriptor for hot spot.
Repeat
lengthCnt++;
directBits=(directBits <<1)| curren Direction
case(branchType)
  FORWARD:
    if (lengthCnt== maxLength)
     then
    output path descriptor and clear the counter and registers;
    else
      break;
  BACKWARD:
    loopCnt++;
      if (loopCnt==maxLoop||lengthCnt== maxLength)
    output path descriptor and clear the counter and registers;
    else
      break;
  CALL or RETURN:
    output path descriptor and clear the counter and registers;
end
```

Algorithm 2. BuildDFG algorithm.

```
Input: Basic Block, SrcReg, DestReg
Output: LiveIn Registers and LiveOut Registers
for i=0 to num of insts in basic block do
dreg=getdestreg(bb [i]);
sreg1=getsrcreg1(bb [i]);
sreg2=getsrcreg1(bb [i]);
sindex1=getproducer(sreg1);
if sindex1== -1 then
  marklivein(sreg);
end
sindex2=getproducer(sreg2);
if sindex2== -1 then
  marklivein(sreg2);
end
setproducer (dreg,i);
setrefindex (sreg1,i);
setrefindex (sreg2,i);
end
```

Algorithm 3. Cluster marking algorithm.

```
Input: G (V, E)
Output: Mark for G (V, E)
for i=0 to num_nodes do
  n=g.vGraph [i];
  if isValidNode(n) != OK then
    continue;
  end
  regmark=(mark[i]==-1)?regionnum+1 : mark [i];
  regmark=getminregmark();
  if regmark==regionnum+1 then
    regmark=regionnum++;
  else
    collectremarknodes(remarklist);
    mark [i]=regmark;
    for j=0 to succ_index do
      mark [successors [j]]=regmark;
    end
    mark(remarklist,regmark);
  end
end
```

Algorithm 4. Subgraph enumerating.

```
Input: All the Clusters
Output: All the Subgraphs
Mark all the nodes in cluster;
for i=0 to num_nodes do
Add v [i] into target set;
for j=0 to succ_index do
  Mark [predecessor [j]]=regmark;
end
Merge the connected graph for each vertex;
Delete the cluster violating constrains; //exceeding area limitation.
end
```

— **Arithmetic and logic instructions selection.**
To achieve considerable speedup, arithmetic and logic instructions are our primary options. For arithmetic instructions, we select ADD and SUB for demonstration, and for logic instructions, we choose SLI (Shift Left) and SRL (Shift Right) for demonstration.

— **Number of input/output operands.**
As usual instructions have less than three operands, in this paper, we generate the custom instructions with four input operands and two output operands. It has been well explored that four-input-two-output architecture could cover most cases (Bonzini and Pozzi, 2007), which could also bring adequate speedup.

— **Depth and width of the array organization.**
The depth and the width of the reconfigurable logic array have a significant impact on the final performance. Taking the area limitations into consideration, we use 5 function units in the first row, 4 units for both the second and third rows and finally two units in the fourth row.

— **Data paths between modules**
The function units are connected to organize a logic array. As is introduced above, the reconfigurable logic array is composed of three levels, which has four input operands and two output operands. The input parameters can be connected to each function unit. The output operands from each level can be directly forwarded to the final output.

Fig. 8 illustrates the array logic organization. Based on this hardware prototype, the mapping scheme should be capable of dispatching the custom instructions onto the new arrays. In particular, for each instruction, the automatic mapping procedure first looks up the producers of sources registers. If an individual register has no producers, then it is marked as LiveIn registers, and the instruction is mapped to current level; otherwise, which means the register is produced by other instructions, the instruction can be only mapped to the next level. All the subsequent instructions cannot be mapped higher level than current instruction. To support the mapping method, three tables are integrated, one is used to store all the producers, and other two tables keep the records of LiveIn and LiveOut registers. The numbers of LiveIn and LiveOut registers refer to the input and output operands of custom instructions, respectively. Fig. 8 shows an example of the automatic mapping scheme.

Assume we have four custom instructions: ANDI, ADDI, OR, and SLLI. The mapping process is described as follows:

— **ANDI R3, R1, #4.**
After checking the producers of the instruction, R1 is marked to LiveIn registers since it does not have any producer. Meanwhile, R3 is characterized to LiveOut registers. The ANDI instruction is mapped to module A, which is also the producer of R3.

— **ADDI R3, R3, #1.**
The process is similar to (1). Since the R3 has a producer A, it should be mapped to the F block in level 2. The producer of R3 has updated to F accordingly.

— **OR R4, R2, R4.**
There are no producers for both R2 and R4, hence the OR instruction is mapped to G. As no instruction is producing R2 and R4, therefore they are marked as LiveIn registers, R4 is also the LiveOut register, and G is the producer.

— **SLLI R4, R4, #2.**
Similarly, the SLLI instruction can only be mapped to next level since the R4 has a producer J.

After the mapping procedure is finished, the reconfiguration controller needs to examine for the constraints, including whether the number of LiveIn and LiveOut registers exceed the table size, or if they could not be deployed simultaneously due to device limitations. If a certain custom instruction fails the checking examination, the instruction should be removed from the table.

5.3. Speedup, resources utilization, and code size

Experiment results in different resources limitations are described in Fig. 9. The x-axis refers to the available resource percentage in the FPGA, and the y-axis indicates the experimental results of the code size, reconfiguration times and speedup. We can get following conclusions from the figure:

(1) When more hardware resources are available, the hardware accelerators will become more powerful, which results in a higher speedup. As a consequence, the cSpeedup and mSpeedup increase when the x-axis grows.
(2) When more hardware resources are available, more accelerators can be integrated at the same time, which means the hardware does not need to be frequently reconfigured. As a result, the cReTimes and mReTimes decrease when the x-axis grows.
(3) When more hardware resources are available, more custom instructions are implemented, which means the new binary code will include less number of instructions. As a result, the cCodeSize and mCodeSize reduce when the x-axis grows.

In particular, with respect to the area-limited FPGA hardware, when the available reconfigurable resources increase from 0.3 to 1.0 (30% to 100%), the CRC application code size decreases from 3.1 to 1.0 (which is 32.5%), while the MD5 application code size reduces from 2.7 to 1.0 (37%). At the same time, the reconfiguration times of CRC decreases from 7 to 0, and the MD5 falls from 6 to 0, respectively.

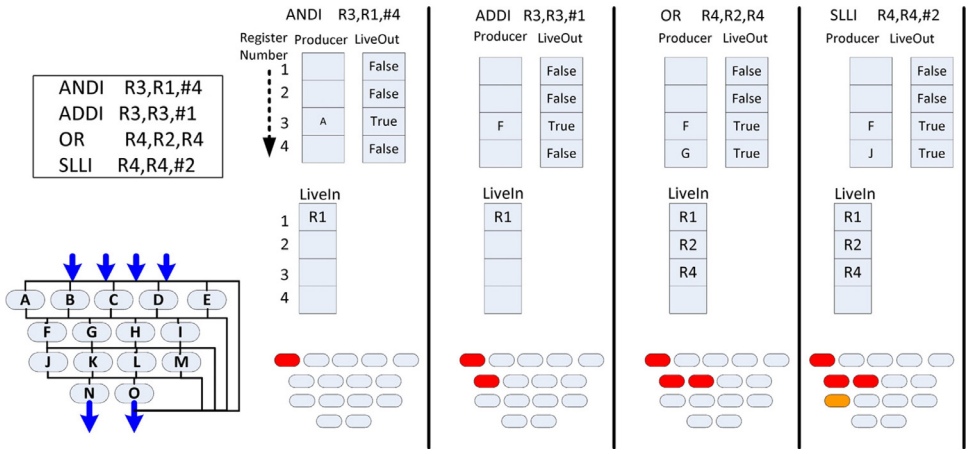

Fig. 8. An example of instruction mapping.

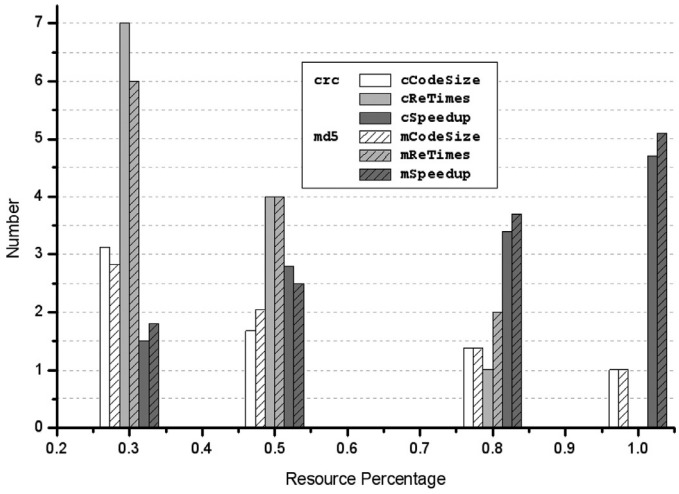

Fig. 9. Experiment results in different resources limitations.

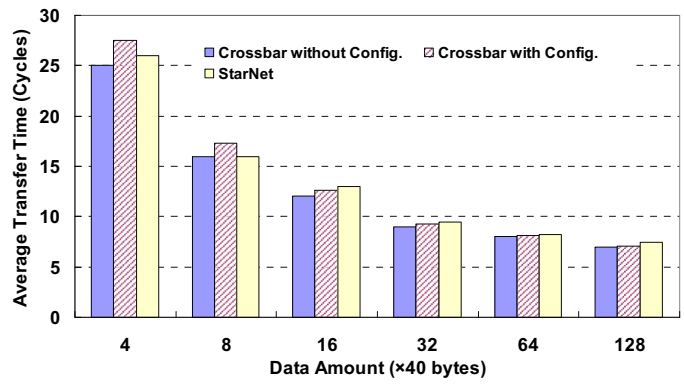

Fig. 10. Average transfer time of crossbar and star network.

In contrast, the speedup of CRC application increases from 1.5x to 4.7x, compared with the performance using FPGA hardware on traditional VLIW architectures. Meanwhile, the MD5 application could also achieve 1.7x to 5.1x speedup. Above all, with the support of hot spot profiler and our instruction extension, the application can be compiled into smaller code sizes while obtaining a considerable speedup.

5.4. Accuracy of the profiler

To evaluate the performance of the interconnection topology, in this paper we used the star network and crossbar-based interconnection as a demonstration. Both topologies are based on the FSL based channels. The communication time for each word is illustrated in the Fig. 10.

(1) First, for the star network, we have built a prototype on the Xilinx FPGA with Fast Simplex Link (FSL) channels. Growing with data number, the average transfer time for each word reduces from 26 cycles to 7 cycles.
(2) Besides the star network based topology, we also examined a crossbar-based interconnection. We constructed a hardware prototype on Xilinx FPGA platform that included 3 Microblaze processors and 2 accelerators (CRC and MD5). Every MB could communicate with either accelerator through the crossbar. Furthermore, the data path between each module and the crossbar was built on a Xilinx FSL link.

Since the data paths through the crossbar had to be configured prior to data communication, we evaluated the average transfer time both with and without the configuration overheads. We have drawn the following conclusions from Fig. 10:

(a) It can be observed that the configuration time takes 2%∼10% of the raw data transfer time for crossbar based topology.
(b) When data scale is small, the communication overhead of star network lies between the crossbar with configuration and raw crossbar communication. Meanwhile, when much more data needs to be transferred, star network will take 5% more time than the crossbar-based scheme.

From the above discussion, we conclude that our architecture and concept could be implemented with different interconnection schemes, and different topologies do not make much difference to the performance of the system.

6. Conclusions

In this paper, we have presented HOTISE, a custom instruction set extension method using dynamic profiling technique and dataflow analysis. Firstly, HOTISE uses a dynamic profiler to obtain the hot spot of an application, which includes several instructions. Then the selected instructions are mapped to logic arrays and combined into a single custom instruction. The proposed HOTISE toolchain can guarantee the reuse and coherence of applica-

tion after hardware reconfiguration. Experimental results demonstrate that profiler can achieve high accuracy of 97.1% with limited hardware cost. With the help of the custom instruction mapping, the code sizes of CRC could decrease to 32.5% while achieving the speedup as high as 4.7x. Another case study on MD5 demonstrates the code size could be reduced to 37%, while achieving the speedup at 5.1x, respectively.

Although the experimental results are promising, there are still many aspects worth pursuing. First, we need to evaluate how different out-of-order schemes could affect the performance, area and power utilization; second, a high efficient programming model should be provided to alleviate the burden of architects and code monkeys.

Acknowledgments

This work was supported by the National Science Foundation of China under grants (No. 61379040, No. 61272131), Jiangsu Provincial Natural Science Foundation (No. SBK201240198), Anhui Provincial Natural Science Foundation, CCF-Tencent Open Research Fund, Natural Science Foundation of Fujian Province of China (No. 2014J05075), CSC Fellowship, Open Project of State Key Laboratory of Computer Architecture, ICT-CAS (No. CARCH201407), and Fundamental Research Funds for the Central Universities (WK2150110003). The authors sincerely appreciate Dr. Jinsong Ji, and many reviewers for their insightful comments and suggestions.

References

Almer, Oscar, Bennett, Richard, et al., 2009. An end-to-end design flow for automated instruction set extension and complex instruction selection based on GCC. In: 1st International Workshop on GCC Research Opportunities. Paphos, Cyprus, p. 12.

Anderson, JenniferM., Berc, LanceM., et al., 1997. Continuous profiling: where have all the cycles gone? ACM Trans. Comput. Syst. (TOCS) 15 (4), 357–390.

Arnold, Matthew, Grove, David, 2005. Collecting and exploiting high-accuracy call graph profiles in virtual machines. In: Proceedings of the International Symposium on Code Generation and Optimization. IEEE Computer Society, pp. 51–62.

Atasu, Kubilay, Pozzi, Laura, et al., 2003. Automatic application-specific instruction-set extensions under microarchitectural constraints. In: Proceedings of the 40th Annual Design Automation Conference. Anaheim, CA, USA. ACM.

Ball, T., Larus, J.R., 1996. Efficient path profiling. In: Microarchitecture, 1996. MICRO-29. Proceedings of the 29th Annual IEEE/ACM International Symposium on, pp. 46–57.

Biswas, Partha, Banerjee, Sudarshan, et al., 2006. ISEGEN: an iterative improvement-based ISE generation technique for fast customization of processors. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 14 (7), 754–762.

Bond, MichaelD., McKinley, KathrynS., 2005. Practical path profiling for dynamic optimizers. In: Proceedings of the International Symposium on Code Generation and Optimization. IEEE Computer Society.

Bond, MichaelD., McKinley, KathrynS., 2005. Continuous path and edge profiling. In: Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture. Barcelona, Spain. IEEE Computer Society.

Bonzini, Paolo, Pozzi, Laura, 2007. A retargetable framework for automated discovery of custom instructions. In: Application-Specific Systems, Architectures, and Processors – ASAP, pp. 334–341.

Borkar, Shekhar, Chien, Andrew A., 2011. The future of microprocessors. Commun. ACM 54 (5), 67–77.

Compton, Katherine, Hauck, Scott, 2002. Reconfigurable computing: a survey of systems and software. ACM Comput. Surv. 34 (2), 171–210.

de Beeck, PieterOp, Barat, Francisco, et al., 2001. CRISP: a template for reconfigurable instruction set processors. In: Proceedings of the 11th International Conference on Field-Programmable Logic and Applications. Springer-Verlag.

Galuzzi, Carlo, Bertels, Koen, 2011. The instruction-set extension problem: a survey. ACM Trans. Reconfigurable Technol. Syst. 4 (2), 18.

Ganesh, Dasika, Mark, Woh, et al., 2010. Mighty-morphing power-SIMD. Proceedings of the 2010 International Conference on Compilers, Architectures and Synthesis for Embedded Systems. ACM.

Henkel, Jörg, Bauer, Lars, et al., 2011. i-Core: a run-time adaptive processor for embedded multi-core systems. International Conference on Engineering of Reconfigurable Systems and Algorithms.

Paolo Ienne and Rainer Leupers, Customizable *E*mbedded *P*rocessors, 2006.

Iqbal, M.A., Awan, U.S., 2009. Run-time reconfigurable instruction set processor design: RT-RISP. In: Computer, Control and Communication, 2009. IC4 2009. 2nd International Conference on, pp. 1–6.

Kastner, R., Kaplan, A., et al., 2002. Instruction generation for hybrid reconfigurable systems. ACM Trans. Des. Autom. Electron. Syst. 7 (4), 605–627.

Kuzmanov, G., Gaydadjiev, G., et al., 2004. The MOLEN processor prototype. In: Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on, pp. 296–299.

Larus, JamesR., 1999. Whole program paths. In: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation. Atlanta, Georgia, United States. ACM.

Lee, Jong-eun, Choi, Kiyoung, et al., 2002. Efficient instruction encoding for automatic instruction set design of configurable ASIPs. In: Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design. San Jose, California. ACM.

Leupers, R., Karuri, K., et al., 2006. A design flow for configurable embedded processors based on optimized instruction set extension synthesis. Proceedings of the Conference on Design, Automation and Test in Europe: Proceedings. European Design and Automation Association.

Lodi, A., Toma, M., et al., 2003. A VLIW processor with reconfigurable instruction set for embedded applications. Solid-State Circuits IEEE J. 38 (11), 1876–1886.

Marnix, Arnold, Henk, Corporaal, 2001. Designing domain-specific processors. CODES. ACM, Copenhagen, Denmark.

Meeuws, R., Yankova, Y., et al., 2007. A quantitative prediction model for hardware/software partitioning. In: Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on, pp. 735–739.

Newton, Cheung, Jorg, Henkel, et al., 2003. Rapid configuration and instruction selection for an ASIP: a case study. In: Proceedings of the Conference on Design, Automation and Test in Europe – Volume 1. IEEE Computer Society.

Noori, Hamid, Mehdipour, Farhad, et al., 2008. An architecture framework for an adaptive extensible processor. J. Supercomput. 45 (3), 313–340.

Pan, Yu, Tulika, Mitra, 2004. Scalable custom instructions identification for instruction-set extensible processors. CASES. ACM, Washington DC, USA.

Panainte, ElenaMoscu, Bertels, Koen, et al., 2007. The MOLEN compiler for reconfigurable processors. ACM Trans. Embed. Comput. Syst. 6 (1), 6.

Singh, Satnam, 2011. Computing without processors. Commun. ACM 54 (8), 46–54.

Toshiaki, Yasue, Toshio, Suganuma, et al., 2003. An efficient online path profiling framework for Java just-in-time compilers. In: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques. IEEE Computer Society.

Vasanth, Bala, Evelyn, Duesterwald, et al., 2011. Dynamo: A Transparent Dynamic Optimization System. ACM, pp. 41–52.

Vassiliadis, Nikolaos, Theodoridis, George, et al., 2009. An application development framework for ARISE reconfigurable processors. ACM Trans. Reconfigurable Technol. Syst. 2 (4), 1–30.

Vassiliadis, S., Wong, S., et al., 2004. The MOLEN polymorphic processor. Comput. IEEE Trans. 53 (11), 1363–1375.

Vaswani, Kapil, Thazhuthaveetil, MatthewJ., et al., 2005. A programmable hardware path profiler. In: Proceedings of the International Symposium on Code Generation and Optimization. IEEE Computer Society.

Wall, DavidW., 1991. Predicting program behavior using real or estimated profiles. SIGPLAN Not. 26 (6), 59–70.

Wang, Chao, Li, Xi, et al., 2014. Architecture support for task out-of-order execution in MPSoCs. IEEE Trans. Comput. 64 (5), 1296–1310.

Wang, Chao, Zhang, Huizhen, et al., 2011. Tool chain support with dynamic profiling for RISP. In: Proceedings of the 2011 IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications. IEEE Computer Society, pp. 155–160.

Zhang, Huizhen, Ji, Jinsong, et al., 2009. Design and implementation of a configurable hardware profiler supporting path profiling and sampling. In: International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery, pp. 325–330.

Wang, Chao, Li, Xi, et al., 2013. MP-Tomasulo: a dependency-aware automatic parallel execution engine for sequential programs. ACM Trans. Archit. Code Optim. 10 (2).

Chao Wang received B.S. and Ph.D. degree from University of Science and Technology of China, in 2006 and 2011 respectively, both in of computer science. He is an associate professor in School of Computer Science, University of Science and Technology of China, Suzhou, China. His research interests focus on multicore and reconfigurable computing. He is now the associate editor of MICPRO, IET CDT and IJHPSA. He is a member of the IEEE, ACM and CCF.

Xi Li is a Professor and vice dean in the School of Software Engineering, University of Science and Technology of China. There he directs the research programs in Embedded System Lab, examining various aspects of embedded system with the focus on performance, availability, flexibility and energy efficiency. He has lead several national key projects of CHINA, 863 projects and NSFC projects. Prof. Li is a member of ACM and IEEE, a senior member of CCF. He is the corresponding author of this paper.

Huizhen Zhang is an assistant Professor in the School of Computer Science, Huaqiao University. He received B.S. and Ph.D. degree from University of Science and Technology of China, in 2005 and 2010 respectively, both in of computer science. His research interests focus on Multicore and reconfigurable computing.

Aili Wang is a lecture of School of Software Engineering, University of Science and Technology of China. She serves as the guest editor of Applied Soft Computing, and International Journal of Parallel Programming. Meanwhile she is a reviewer for International Journal of Electronics. She has published over 10 International journal and conference articles in the areas of software engineering, operating systems, and distributed computing systems.

Xuehai Zhou is a Professor in the School of Computer Science, and the executive dean of School of Software Engineering, University of Science and Technology of China. He serves as general secretary of steering committee of computer College fundamental Lessons, and technical committee of Open Systems, CCF. His research interests include various aspects of multicore and distributing systems. He has lead several national 863 projects and NSFC projects.