Hardware Implementation on FPGA for Task-Level Parallel Dataflow Execution Engine

Chao Wang, *Member, IEEE*, Junneng Zhang, *Student Member, IEEE*, Xi Li, *Member, IEEE*, Aili Wang, and Xuehai Zhou, *Member, IEEE*

Abstract—Heterogeneous multicore platform has been widely used in various areas to achieve both power efficiency and high performance. However, it poses significant challenges to researchers to uncover more coarse-grained task level parallelization. In order to support automatic task parallel execution, this paper proposes a FPGA implementation of a hardware out-of-order scheduler on heterogeneous multicore platform. The scheduler is capable of exploring potential inter-task dependency, leading to a significant acceleration of dependence-aware applications. With the help of renaming scheme, the task dependencies are detected automatically during execution, and then task-level Write-After-Write (WAW) and Write-After-Read (WAR) dependencies can be eliminated dynamically. We extended the instruction level renaming techniques to perform task-level out-of-order execution, and implemented a prototype on a state-of-art Xilinx Virtex-5 FPGA device. Given the reconfigurable characteristic of FPGA, our scheduler supports changing accelerators at runtime to improve the flexibility. Experimental results demonstrate that our scheduler is efficient at both performance and resources usage.

Index Terms—FPGA, dataflow, dependency analysis, out-of-order, parallel architecture

1 Introduction

Task-level parallelism (TLP) has motivated research into simplified parallel programming models [1]. However, a drawback of common task-based models such as Cilk [2], OpenMP [3], MPI, Intel TBB, CUDA and OpenCL [4] is burdening the programmer with the non-trivial assignment of resolving inter-task data dependencies. To resolve this problem, automatic parallelization has been widely researched at different levels, e.g., programming model [5], compiler and runtime library [6], and microarchitecture [7]. Most of the programming models perform well for regular operations (e.g., loop structures [8]). However, the results may be unacceptable satisfying for most irregular operations. At compiler level, it poses significant challenges to detect dependencies statically. Alternatively, using special architecture to support task-level Out-of-Order (OoO) execution is becoming effective and efficient on performance [9]. However, how to make the architecture flexible to suite for various systems remains unresolved, especially for heterogeneous systems with different types of processing elements (PEs).

Meanwhile, with diverse types of accelerators integrated, heterogeneous multicore platform is able to achieve high performance for a large variety of applications. However, the software for such heterogeneous systems can be quite complex due to the management of low-level aspects of the computation. To maintain the correctness, the software must decompose an application into parallel tasks and synchronize the tasks automatically. As a consequence, data dependences within an application may seriously affect the overall performance. Generally data dependences are classified into three categories: read-after-write (RAW) dependence, write-after-write (WAW) dependence and write-after-read (WAR) dependence, of which WAW and WAR can be eliminated using the renaming techniques.

To tackle this problem, in this paper we propose a hardware scheduler that supports task-level OoO parallel execution. The fundamental system is an FPGA based platform that contains different types of PEs: one or several general purpose processor (GPPs) and a variety of Intellectual Property (IP) cores. A earlier version of this paper is presented at [10]. We have extended and implemented a demonstrating hardware scheduler for heterogeneous platforms to support OoO task execution. The scheduler detects task-level data dependencies and eliminates WAW and WAR dependencies automatically at runtime using renaming scheme. We claim following contributions and highlights:

1) This work implements a hardware scheduler on FPGA to support parallel dataflow execution and dynamic accelerator reconfiguration. By eliminating the WAW and WAR dependences among dataflow, applications will get as much parallelism as possible. This is especially important for dependence-aware applications in real-time system.
2) Reconfigurability is supported so as to improve the flexibility of heterogeneous multicore platform. Our scheduler is as efficient as Task Superscalar [7] at the ability to eliminate WAW and WAR dependence while has much lower scheduling overhead due to the FPGA efficiency. Furthermore, as our scheduler is implemented on a practical FPGA board instead of the simulation, therefore it is more practical compared to the Task Superscalar scheme.

• *The authors are with the University of Science and Technology of China, China. E-mail: {saintwc, zjneng}@mail.ustc.edu.cn, {llxx, wangal, xhzhou}@ustc.edu.cn.*

Manuscript received 12 Apr. 2015; revised 31 July 2015; accepted 14 Sept. 2015. Date of publication 5 Oct. 2015; date of current version 20 July 2016. Recommended for acceptance by M. Huebner.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2015.2487346

1045-9219 © 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

The structure of the paper is organized as follows: Section 2 summaries the related work and the motivation. Section 3 shows a high level view of the proposed framework, which includes the problem description and definition of tasks. Section 4 details the programming model layer. Thereafter in Section 5 we illustrate the composition of the scheduler, and Section 6 gives the hardware implementation for the scheduler. Experiments method and results are presented in Section 7. Finally section 8 concludes the paper and outlines some future works.

2 Related Works

Data dependencies and synchronization problem has already posed a significant challenge in parallelism. Traditional algorithms, such as Scoreboarding and Tomasulo [11], explore instruction level parallelism (ILP) with multiple arithmetic units, which can dynamically schedule the instructions for out-of-order execution. Much effort has been made to extract task-level parallelism from programs in the last decade. Schemes on different levels have been proposed to solve TLP problems.

With the increasing popularity of MPSoC platform, parallelism is shifting from instruction level to task level. There are already some creditable FPGA based research platforms, such as RAMP [12], Platune [13] and MOLEN [14]. These studies focus on providing reconfigurable FPGA based environments and related tools that can be utilized to construct application specific MPSoC. Alternatively, products and prototypes of processor are designed to increase TLP with coarser grained parallelism, such as RAMPSoC [15], MLCA [16], Multiscalar [17], Trace Processors [18], IBM CELL [19], RAW Processor [20], Intel Terascale and Hydra [21]. These design paradigms present thread-level or individual cores which can split a group of applications into small speculatively independent threads. Some other works like TRIPS [22] and WaveScalar [23] combine both static and dynamic dataflow analysis in order to exploit more parallelism. Yoga [24] is a hybrid dynamic processor with out-of-order execution and VLIW instructions sets. The concept of Yoga is to leverage the tradeoff between high performance OoO and low power VLIW, therefore it can not be applied to general heterogeneous frameworks. Furthermore, a common concept of these literatures is to split a large task window into small threads that can be executed in parallel. However, the performance is seriously constrained by inter-task data dependencies.

At programming model level, MPI, Pthreads, OpenMP, Cilk, and TBB are the state-of-the-art parallel programming frameworks. It is now common knowledge that these non-speculative models rely upon developer experiences to parallelize programs. When an irregular program written in these programming models is executed, the decomposition of parallel tasks could be difficult. During dynamic execution of the statically-parallel program, a multitude of complexities may arise that make program development onerous. Deterministic, speculative models offer simple programming models, but incur potentially significant dynamic overheads to support recovery from mis-speculation and to dynamically check for potential conflicts. Of the cutting-edge programming paradigms, CellSs [5] and StarSs [25] are programming models that uncover TLP in particular supporting irregular tasks. Besides, dependency detecting and eliminating are left to the hardware scheduler, which greatly simplifies the design of the programming model and runtime library. Harmony [26] is also an execution model and runtime for heterogeneous manycore systems. However, it only considers the tasks that run on GPPs, while tasks that run on accelerators (e.g., DSP, IP) are not taken into account.

Along with the prototype platforms targeting specific hardware, there are some well-known reconfigurable hardware infrastructures: ReconOS [27], for instance, demonstrates hardware/software multithreading methodology on a host OS running on the PowerPC core using modern FPGA platforms. Hthreads [28], RecoBus [29] and [30] are also state-of-the-art FPGA-based reconfigurable platforms. Besides FPGA-based research platforms, FlexCore [9] proposes a heterogeneous platform which supports run-time monitoring and incorporates bookkeeping techniques.

At architectural level, Task Superscalar [7], Multiscalar [17], Stanford Hydra [21] and Program Demultiplexing [31] are hardware schedulers for multi-processor systems based on renaming techniques. McFarlin [32] presents a speculative out-of-order execution architecture support for heterogeneous multicore architectures. Similarly, Sridharan [33] presents an efficient parallel execution of parallel programs, which can be adaptive to different running applications. But the scheduling overhead of the software speculation is significantly higher than hardware schedulers. In summary, none of them has been implemented as hardware circuits.

It is a trend that in foreseeable future most computing systems will contain a considerable number of heterogeneous computing resources. So far in industry, GPU has been widely used to support graphic processing and thereby achieve speedup of performance. It will be highly possible that more heterogeneous accelerators based on FPGA will be integrated to the computer architecture. So we focus on the OoO execution that supports heterogeneous systems, trying to find an efficient way to design the programming model, compiler and scheduler. In order to summarize the differences between related works and our approach, we list the strength and weakness of the typical parallel scheduling algorithms and engines. The differences are illustrated in Table 1, which includes general parallel programming models, out-of-order programming models and task level scheduling mechanisms.

3 High Level View

3.1 Problem Description

It has been widely acknowledged that traditional programming models for TLP such as OpenMP and MPI perform well for regular programs, especially for loop based codes. However, the parallelism degree of irregular programs of OpenMP and MPI depends on the programmer, which requires too much burden to the programmers and the performance may be dramatically unsatisfied. Consequently the motivation of this paper is to pursue the automatic OoO task parallel execution method, especially for heterogeneous systems.

In order to compare the differences of three programming models: serial model, OpenMP [3] and CellSs [5], we illustrate the example code snippet respectively.

TABLE 1
Summary for State-of-the-Art Parallel Execution Engines on FPGA

Types	Typical	Strength	Weakness
General parallel programming models	OpenMP [3] Intel's TBB, Ct, CnC MapReduce OpenCL [34] Cilk [35]	General model for CMP processors	Bring burden to programmers
Out-of-order programming models	StarSs [25], CellSs [36], Oscar [37]	Support automatic OoO execution	Applied only to CellBE architecture and SMP servers
Task Level Out-of-order Scheduling	CEDAR [38], MLCA [16], Multiscalar [17], Trace [18], IBM CELL [19], RAW [20], Hydra [39], Wave Scalar [23], TRIPS [22]	Perform well for traditional superscalar machines, run tasks in parallel on different processor cores	OoO not supported, programmers handle the task data dependency manually
	TaskSuperscalar[40], Yoga[24]	Support OoO automatic parallel execution	Not applicable to FPGA with reconfiguration
Dataflow Based OoO Execution Model	DSP [41]	With register renaming technologies of Tomasulo	Limited to DSP architectures
	OoOJava [42], Dataflow [1]	An OoO compiler for Java runtime with determinate parallel execution	Not applicable to hardware execution engines

Fig. 1 illustrates the three code segments of the same functionality. Task 1 takes *'a'* as input parameter, and write results to *'b'*, while task 2 takes *'c'* as input, and also write results to *'b'*. Fig. 1a presents the normal serialized codes, in which task 2 cannot be issued until task 1 is done. Assuming that task 1 and task 2 can run in parallel using different computing resources. Fig. 1b shows the sample codes in OpenMP pattern. Because both task 1 and task 2 write *'b'*, the programmer must rename *'b'* of task 2 manually to support parallel execution. Consequently all tasks after task 2 that use *'b'* also need to rename their parameter *'b'* to *'c'*, which greatly limits the programmability. If *'b'* is to be written to non-volatile storage devices (e.g., disk), then the task 1 and task 2 cannot be executed in parallel by OpenMP. In order to release programmers from such burden, CellSs was proposed, as is illustrated in Fig. 1c. The first two annotated lines of Fig. 1c inform the compiler that task 1 and task 2 can be executed simultaneously, enabling task 1 and task 2 to be sent to different computing resources for parallel execution. The programmers are only required to add the simple annotations of the tasks, and then the parallelization is automated by compiler and hardware support.

As mentioned above, both CellSs and Task Superscalar use renaming scheme to detect and eliminate WAW and WAR dependencies. However, they are both designed for CMP architecture, in which all processors are able to run all kinds of tasks. This characteristic brings higher flexibility and scalability, but also causes significant difficulties to task-level OoO execution for heterogeneous systems. Besides, Task Superscalar detects inter-task dependencies at a fine-grained level, which makes it too sophisticated for resources constrained systems. In order to make task-level parallelization efficient for heterogeneous systems, in this paper we propose a variable-based OoO framework for heterogeneous system.

(a) Serial Code	(b) OpemMP Code	(c) CellSs
	#pragma omp section	**#pragma .....**
Task_1(b,a);	Task_1(b,c);	**#pragma .....**
....	**#pragma omp section**	Task_1(b,a);
Task_2(b,c);	Task_2(e,c);	Task_2(b,c);

Fig. 1. Sample code snippet of different programming models.

3.2 Definition of Tasks and Task-Level Dependencies

Throughout this paper, *tasks* refer to the execution of actors, and actors can be GPPs or IP cores. Each task has a set of input parameters and a set of output parameters, and is defined as ***task_name ({the set of output parameters}, {the set of input parameters}).***

Exploiting task-level concurrency is especially important in a heterogeneous environment due to the requirement to match the execution requirements of different parts in the program with computational capabilities of the different platforms. Some tasks may require special-purpose hardware, either because the hardware can execute that task more efficiently or because the hardware has some unique functionality that the task requires. Consequently the motivation of this paper is to pursue the automatic OoO task parallel execution method especially for heterogeneous systems.

In Table 2 we define three types of task-level dependencies. The bold parameters indicate the parameters those lead to data dependencies. Tasks have RAW dependency should always run in order, meanwhile, WAW and WAR dependencies can be eliminated using renaming scheme.

TABLE 2
Definitions of Task-Level Dependencies

Type	Definition	Example
RAW	When an issuing task reads a parameter in the output set of an issued task, a RAW dependency occurs.	task_1({*b*}{a}); task_2({c},{*b*});
WAW	When an issuing task writes a parameter in the output set of an issued task, a WAW dependency occurs.	task_1({*c*},{a}); task_2({*c*},{b});
WAR	When an issuing task writes a parameter in the input set of an issued task, a WAR dependency occurs.	task_1({b},{*a*}); task_2({*a*},{c});

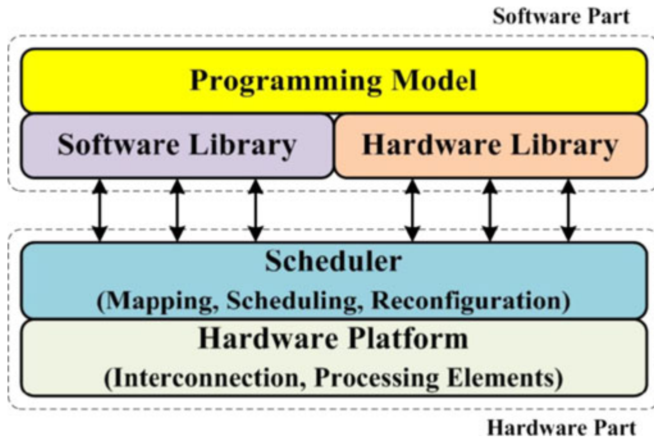


Fig. 2. Task-level OoO framework. The framework is divided into software part and hardware part, and both parts have two individual layers.

```
1  //------declaration segment------//
2  #include <Hardware.h>
3  #Pragma input (int a[8]) output (int b[8])
4  void IDCT(b,a){... ... }
5  #Pragma input (int a[16], int b[16]) output (int c[16])
6  void AES_ENC(c,a,b){... ... }
7  //------normal segment------//
8  void main()
9  {
10         Hd_IDCT(... ... );
11         Hd_AES_ENC(... ... );//call hardware computing resources
12         IDCT(... ... );
13         AES_ENC(... ... );//call software computing resources
14         return;
15 }
```

Fig. 3. Codes snippet in our programming models.

Fig. 2 presents the framework that is composed of four layers. The top layer is the programming model layer which makes the details of the scheduler and the hardware platform transparent to the programmer. The second layer is the library layer that consists of the software library and the hardware library. The software library includes all the tasks interface definitions, while the hardware library contains the bit files of the IP cores. The third layer is the scheduler which is in charge of task mapping, task-level OoO scheduling and reconfiguration. The bottom layer is the hardware platform layer which is composed of interconnections and processing elements. The PEs includes GPPs and IP cores, and each IP core can do only a specific type of task.

4 Programming Model Layer

In this section, we first introduce the programming model layer, which is able to facilitate software developers for high level programming. The programming model uses annotations to define tasks. In our framework, computing resources are divided into two categories: hardware computing resources and software computing resources. Each type hardware computing resource can only do a specific kind of task, while software computing resources have the capability to do all kinds of tasks. In our framework, the interfaces to call hardware computing resources are regulated in runtime libraries. To use software computing resources, programmers need to define the tasks in prior to the hardware execution.

The programming model layer makes all the details of the platform transparent to the programmers. In order to simplify the programming, programmers only have a sequential view of the systems, in other words, the programmers do not need to know what the underlying platform is composed of or how tasks execute in parallel.

In order to reduce the burden of the scheduler, tasks are divided into two categories: those that pass through the scheduler and those that execute on the local GPP. Thus the program is divided into two parts: kernel part and non-kernel part. A kernel is defined as a sequence of tasks that all pass through the scheduler while the non-kernel part is defined as a sequence of tasks that all run locally.

When compiling the program, the compiler identifies the kernel parts at first. Then synchronizations are inserted to the head and the end of each kernel part. The synchronizations make sure that the data used by tasks has the latest value. The synchronizations are inserted to the code by a source-to-source compiler automatically, and thereby programmers do not have the burden to do such things by themselves.

After the source-to-source compilation, the program then is compiled into executable binaries. Tasks in the kernel parts are sent to the scheduler for OoO execution. Fig. 3 illustrates sample code snippet in our programming model. The codes are divided into two segments: declaration segment and normal segment. The declaration segment (Line 2 - Line 6) contains header files inclusion (Line 2) and task definitions (Line 3 - Line 6). All interfaces to call hardware computing resources are included in *Hardware.h*. When hardware computing resources are designed, the corresponding interfaces should be added to *Hardware.h*. Tasks defined by programmers are started with annotations, as Line 3 shows. Each annotation includes three aspects of parameters: direction, type and size. The direction can be chosen from *input*, *output*, and *inout*, which is detailed as follows:

Input: The parameter will be read by the task, and when the task is completed, the parameter will not be updated.

Output: The task starts without considering the value of the parameter, and when the task is completed, the parameter will be updated.

Inout: The task will use the data stored in the parameter, when the task is finished, the parameter will be updated. Line 4 and Line 6 are normal task definitions written in C language. Line 8 to Line 15 presents sample codes of main function. *Hd_idct* and *Hd_aes* are tasks defined in *Hardware.h*, while *Idct* and *Aes* are defined by the programmer.

In this paper, we bring the instruction-level dependency analysis concepts to task-level. Each task is treated as an abstract instruction, and the parameters of a task is equivalent to operands at instruction level, with the expansion that each task can have more than three parameters (In a standard x86 instruction set, an instruction has three operands at most). All the parameters must be defined as variables. Fig. 4a describes a sample code pattern, in which *wr_set* and *rd_set* stand for write set and read set of the function, respectively. Each function in the loop is viewed as a task, i.e., *wr_set* contains variables defined as *Input* and *Inout*, while

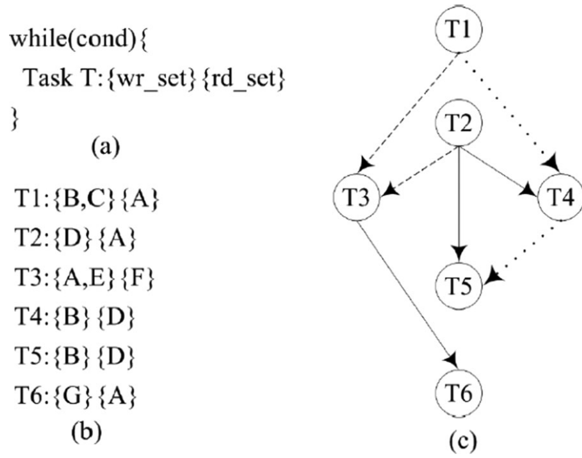


Fig. 4. Task-level data dependencies.

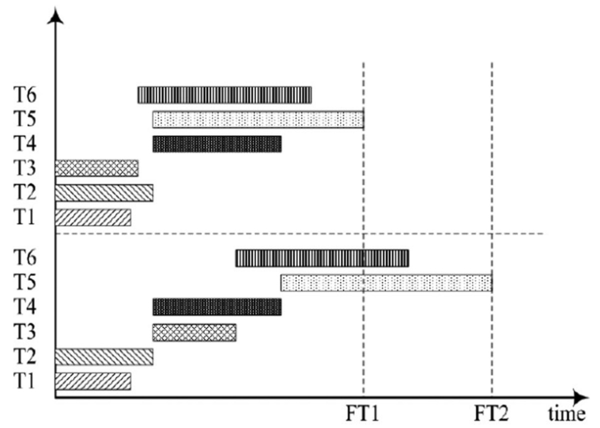


Fig. 5. Execution comparison. The bottom part is the execution time chart in [1], which finishes at time FT2, while the top part is the OoO mode presented in our scheduler, which finishes at time FT1.

rd_set contains variables defined as *Output* and *Inout*. Fig. 4b is a task sequence when the codes in Fig. 4a actually execute. Fig. 4c shows all inter-task dependencies, in which different shapes of edges stand for different types of dependencies, in particular direction of an edge is from the task that brings about the dependency to the task that is affected. In particular, inter-task dependencies are divided into three categories as follows:

RAW: The solid edges indicate RAW dependencies. All solid edges compose a normal Directed Acyclic Graph (DAG). When an issuing task reads a variable in the *wr set* of an issued task, a RAW dependency occurs, for example, task 4 takes variable D as input while task 2 writes D, so there is a solid edge from task 2 to task 4.

WAW: The dashed edges indicate WAW dependencies. When an issuing task writes a variable in the *wr set* of an issued task, a WAW dependency occurs. Task 4 and task 1 both write variable B so that there is a dashed edge.

WAR: The dotted edges indicate WAR dependencies. When an issuing task writes a variable in the *rd set* of an issued task, there is a WAR dependency. Task 3 writes variable A while task 1 reads A, therefore a dotted edge is from task 3 to task 1.

It must be noted that tasks that have RAW dependency should always run in order. Meanwhile, WAW and WAR dependencies can be eliminated using special techniques. Fig. 5 illustrates two different executing models for the tasks in Fig. 4b. The bottom refers to the mode used in [1], where WAW and WAR dependencies are detected but not eliminated, while the top stands for the ideal OoO mode presented in our scheduler, in which only RAW dependency will affect the parallelism. It can be obviously derived that the OoO mode will gain great performance improvement (the time between FT1 and FT2). In this paper, we are to propose a scheduling algorithm that automatically eliminates task-level WAW and WAR dependencies, especially for heterogeneous platforms.

5 Scheduler Layer

The scheduler layer is in charge of scheduling tasks to exploit the potential parallelism. In this paper, OoO task scheduler is implemented as a hardware layer to uncover TLP. For demonstration we have implemented MP-Tomasulo algorithm in hardware, which extends the instruction level Tomasulo algorithm to task level for parallel execution. The MP-Tomasulo algorithm is able to dynamically detect and eliminate inter-task WAW and WAR dependencies, and thus speeds up the execution of the whole program. With the help of MP-Tomasulo algorithm, programmers need not to take care of the inter-task data dependencies.

5.1 Hardware Scheduler Structure

At instruction level, each instruction has one destination operand and two source operands at most. However, a task may have more than one output parameter, or more than two input parameters. We extend the traditional Tomasulo algorithm to support multi outputs and more than two inputs. MP-Tomasulo algorithm is designed as a hardware module so that the maximum of outputs and inputs are limited to a certain number. Assuming that *wr_set* refers to the output set, *rd_set* represents the input set, while *num_wr_set* and *num_rd_set* stand for the maximum of output and input, respectively. Fig. 6 illustrates MP-Tomasulo module that is composed of the following components:

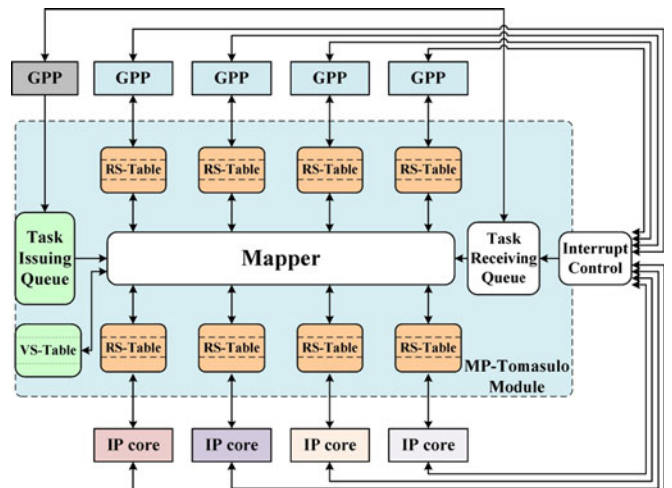


Fig. 6. High level structures of MP-Tomasulo module. The part surrounded by the dashed line is named MP-Tomasulo module, which is the critical part of the framework.

Task issuing queue. In our prototype, each task has *num_wr_set* outputs and *num_ rd_set* inputs, so each task in Task Issue Queue can be formally regulated as $< Type; wr_set [num_wr_set]; rd_set[num_rd_set] >$, in which Type identifies the type of the task, *wr_set* and *rd_set* stand for the set of output parameters and the set of input parameters, respectively.

Mapper. The Mapper sends tasks to different RS-Tables according to *Type* of each tuple stored in Task Issuing Queue. The Mapper works as a black box, and therefore system designers can design other alternative mapping algorithms. If the RS-Tables do not have enough entries to store the task, the system will be blocked until some RS-Table entry is free.

RS-Table. RS-Table refers to reservation station table, which contains an implicit dependency graph of tasks and uses automatic renaming scheme to eliminate WAW and WAR dependencies. In order to send different tasks simultaneously, each PE has a private RS-Table. Once all the input parameters of a task are prepared, the task is offloaded to the associated PE for execution. Each entry of RS-Table is a tuple of $< ID; Busy; V [num_rd_set]; Q[num_rd_set] >$, in which *ID* is used to identify different entries, *Busy* indicates whether the entry is in use or not, *V* is used to temporarily store input parameters of a task, *Q* indicates if the value in *V* is valid.

Task receiving queue. The Task Receiving Queue buffers the finished tasks.

VS-Table. VS-Table is regarded as register status table that holds the status of variables used in applications. Each variable is mapped to an entry of VS-Table. Each entry of VS-Vable is a tuple of $< ID; PE_ID; Value >$. ID stands for the primary unique key. *PE_ID* represents the PE that will produce the latest value of the variable. Value storages the actual value.

MP-Tomasulo algorithm is divided into three stages as follows:

Issue. The head task of Task Issuing Queue is in Issue stage if a RS-table entry is available.

Execute. If all the input parameters of a task are prepared, then the task can be distributed to the associated PE for execution immediately. Otherwise the RS-Table builds an implicit data dependency graph indicating which task will produce needed parameters. Once all the input parameters of a task are ready, the task is spawned to the corresponding PE.

Write results. When a PE completes a task, it sends results back to the Task Receiving Queue of MP-Tomasulo module, and updates RS-Table. For each task in RS-table, if the input parameters are produced by the completed task, the associated RS-Table entry is updated with the results.

Compared to the software scheduler in presented in [43], we remove the reorder buffer (ROB) in this hardware implementation. We have following concerns:

1) The ROB structure is designed for speculation. In fact, due to the scheduler only deals with task sequences that always should execute, thus there is no need to make any speculation.
2) The ROB structure will make a significant contribution to the hardware utilization. Therefore we save the ROB area considering the area and cost of the FPGA chip.

Furthermore, in order to make the scheduler flexible, the size of the VS-Table, RS-Table and the number of PEs can be configured according the characteristics of applications. These configurations can be changed when the system starts and remain fixed during execution. To make efficient use of the on-chip resources, the IP cores are allowed to be dynamically reconfigured at run time.

5.2 Adaptive Task Mapping

The scheduling module decides when the task can be executed due to data dependencies, furthermore, when a task is ready, only one target function units is selected from multiple options. The task mapping scheme should decide the target for each task, as is described in Algorithm 1.

Algorithm 1. Task Scheduling and Mapping Algorithm

Input: Generated Task Set T
Output: IP Core ID set for each task S

```
1  Foreach t ∈ T do
2  snum = ValidIPCoreNumber(t.opcode)
             // get candidate IP Cores
3        switch (snum)
4        case: snum = 0 //no available IP Cores
5              return null;
6        case: snum = 1 //only one available
7              add s to the IP Core ID set
8              return s;
9        default:
10             for j = 0 to snum do
11             T_finish = T_waiting+T_execution+T_transfer // Calculate execution time
12             end
13   TNum = number of minimum T_overall
14             If TNum>1
15       Select the s using Round-Robin
16   Else
17       Select the s with minimum T_overall
           // Select a target IP Core
17           add s to the IP Core ID set
18     end
```

5.3 FCFS Queue Based Scheduling

The task scheduler architecture is shown in Fig. 7. We implement a queue-based task scheduling algorithm using first come first serve (FCFS) policy. Scheduler is responsible for tasks partition, scheduling and distributing to different computing resources. The major work includes the following subjects:

1. Receive task requests. Task requests are transferred to scheduler during execution. Furthermore, the current status of each task is rendered by scheduler to programming models. Status checking interface can obtain the status of each IP core of the system.
2. Task partitioning and allocation among hardware and software: scheduler classified tasks into two types: kernel part and non-kernel part. As to the sub tasks requesting for the kernel part, scheduler needs to distribute data to related hardware IP cores through on chip interconnection. Otherwise, the control tasks and glue functions are executed on local scheduler.

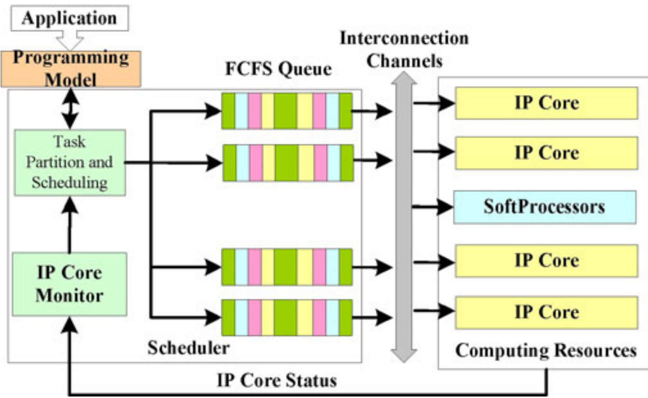


Fig. 7. Queue based task scheduling.

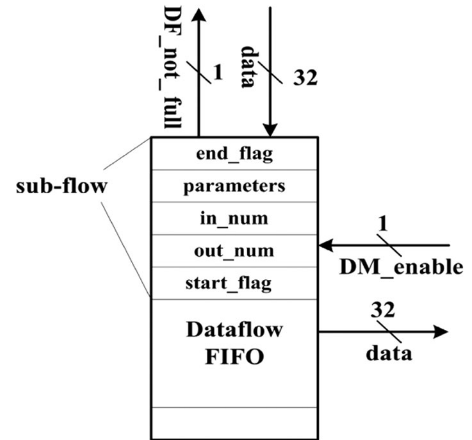


Fig. 9. Dataflow FIFO hardware implementation.

6 Hardware Scheduler Details

In order to make dataflow run in parallel, renaming method is introduced to eliminate WAW and WAR dependences. The main reason for causing WAW and WAR dependences is that a sub-flow may update its write set before a previous sub-flow reads or writes part of it. The critical idea of renaming method is to make a copy of the data that may cause dependences.

Fig. 8 illustrates the hardware implementation of the scheduler, which is redesigned from the high level structure in Fig. 6. Applications always start from GPP that is in gray color, and the dataflow of execution is sent to Dataflow FIFO (DF). The critical part of the hardware scheduler is the Dataflow Mapper (DM). The DM receives dataflow from DF, identifies the write set and read set of each sub-flow, searches for a proper PE to execute the sub-flow and updates Variable Set (VS) and the found PE's Map Table (MT). A MT contains the parameters of the sub-flow, or which PE will produce the needed parameters. When a PE finishes a sub-flow, an interrupt occurs and the Interrupt Controller (ICtr) reports it to DM. The DM updates MTs and writes results to Ordering Unit (OU). OU writes results to the memory in the order as sub-flows' order in DF. These modules will be detailed respectively.

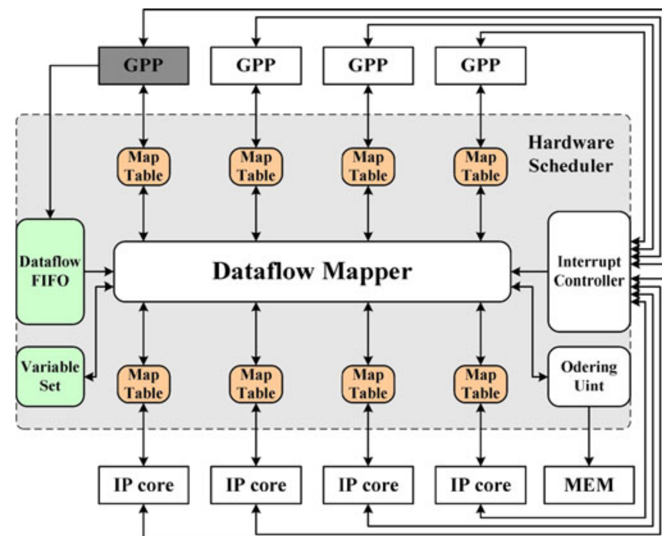


Fig. 8. Hardware implementation of the scheduler.

6.1 Dataflow FIFO (DF)

The DF stores the dataflow which consists of a series of sub-flows. The detailed structure of DF is illustrated in Fig. 9. The communication interfaces between DF and GPP contains a 1-bit *DF_not_full* indicating the status of DF and a 32-bit *data* for transmission. The communication interface between DF and DM contains a 1-bit *DM_enable* indicating the status of DM and a 32-bit *data*. Each sub-flow starts with *start_flag* and ends with *end_flag*. *Out_num* and *in_num* refer to the number of write set and read set, respectively, both measured by 32-bit. Between *in_num* and *end_flag* are actual parameters.

6.2 Variable Set and Ordering Unit

Variable Set keeps the latest value of a certain parameter that the sub-flow in OU produces. When a sub-flow passes through DM, the information of its parameters in write set will be updated in VS, indicating a new producer of these parameters. Assuming that each entry of VS has N bits, and OU has 2^N entries.

OU is a FIFO stores all the executing sub-flows as the order they pass through DM. As long as the head sub-flow in OU finishes, the entry is released and VS is updated. The results of the head entry of OU are written to the memory. Communication between VS and OU is controlled by DM.

Fig. 10a describes the detailed view of VS and OU. VS and OU are both composed of Block RAM (BRAM) and BRAM controllers. Assuming that VS has 256 entries and OU has 64 entries (throughout this paper we all use this configuration), so the widths of *Addr* wires of VS and OU are 8 and 6 bits, respectively. As the content of each entry of VS is associated with a certain position of OU, so the content with of VS is 6 bits. Supposing that each sub-flow has no more than 16 parameters (parameters in write set and read set are both limited to no more than 8), each of which is no more than 32 bits (usually used as 32-, 16- or 8-bit), then the width of a OU entry is 256 bits because only the write set need to be stored while the read set is ignored. Fig. 10b presents how VS and OU are updated when the head sub-flow of OU finishes. In the left part of Fig. 9b, the head sub-flow of OU is the latest producer of No.1 entry of VS. After communication through DM, both VS and OU are updated. The content of No.1 entry of VS is changed to *N*, meaning that the entry is free. The content of No.0 entry of OU is also changed to *N*. Besides, the head of OU is back to No.1 entry.

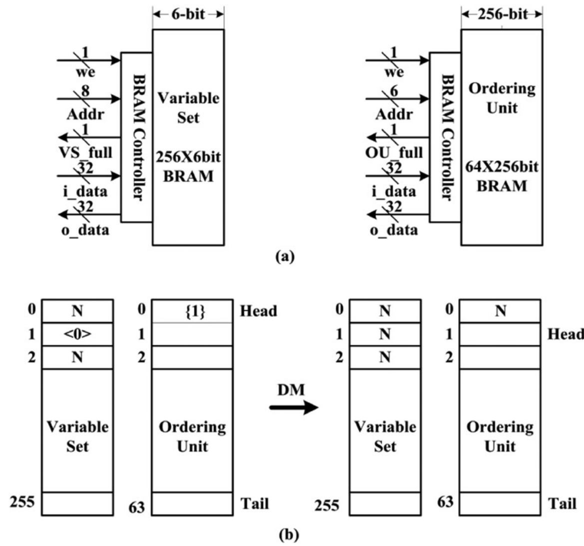


Fig. 10. Circuit Implementation of variable set and ordering unit.

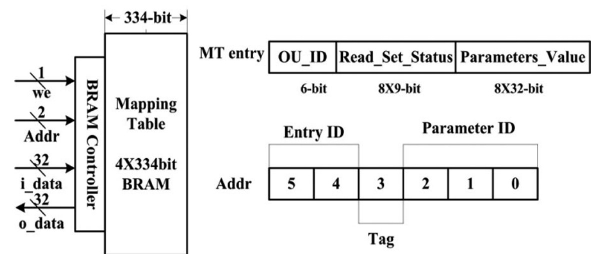


Fig. 11. Circuit implementation of mapping table.

6.3 Map Table (MT)

The MTs hold an implicit DAG of all the executing sub-flows and eliminate WAW and WAR dependences through copies of sub-flows' parameters. Fig. 11 shows the detailed view of a MT. In the proposed design, each PE has a separate MT and there are four entries in each MT. An entry of a MT consists of a 6-bit *OU_ID*, a 72-bit *Read_Set_Status* and a 256-bit *Parameters_Value*. *OU_ID* indicates the location of sub-flow in OU. The parameters in the read set of a sub-flow are recorded in the MT. Each parameter has 9-bit *Read_Set_Status*, of which the first 8 bits represent the parameter ID and the last bit indicates whether the parameter is prepared. If the last bit is 1, then the associated *Parameters_Value* contains the exact value of the parameter. Otherwise, *Parameters_Value* contains the OU entry ID which will produce the actual value.

The address of a MT has three components. The first 2 bits index the entry, the flowing bit is a tag indicating write *Read_Set_Status* or *Parameters_Value,* and the last three bits shows which part of *Parameters_Value* is to be updated.

6.4 Interrupt Controller (ICtr)

When a PE finishes the execution of a sub-flow, an interrupt occurs and the ICtr handles the interrupt. The interrupt results will be transferred to DM. Each PE is indexed by a unique *PE_ID* and has a fixed priority in ICtr. A PE with a smaller *PE_ID* has a higher interrupt priority. In order to get better overall performance, IP cores always have higher interrupt priority than GPPs, meaning that when occurring simultaneously, IP interrupt request (IPR) are always handled in prior to GPP interrupt request (GPPR). When all interrupt requests pass through ICtr, only one output wire is set to 1, indicating that the associated interrupt has the highest priority and results can be received while others are pending. Interrupt nesting is not supported due to that interrupt nesting needs extra hardware resources to store the interrupt stack. An entry of MT or the OU is released upon finished for the DM to receive sub-flows.

6.5 Dataflow Mapper (DM)

DM is the critical part of the hardware scheduler connecting DF, VS, MTs, ICtr and OU. The DM is called (x,y)-channel if it supports x IP cores and y GPPs at most. Fig. 11 illustrates the details of DM. On receiving sub-flows from DF, the DM first generates the address of VS, OU and MT. *VS_addr* is included in the sub-flow so that VS can be easily updated. In order to accelerate the generation of *OU_addr*, a register named *Tail Reg* is utilized to keep the address of the first free entry of OU. *Head Reg* indicates the first used entry of OU, and together with *Tail Reg* it is easy to check whether OU is full. DM is responsible for mapping sub-flows to different PEs. The strategy for mapping is summarized in Algorithm 2. Please note that the Mapper works as a black box, and the system designer can design their own mapping algorithm.

Algorithm 2. Mapping Algorithm

input: a sub-flow
output: a PE ID
1: **if** (a IP core p can execute the sub-flow)
2: **if** (the MT of p has a free entry)
3: update MT,OU,VS;
4: **return** p;
5: **else** select the GPP g that has the most free entries N
6: **if** (N is larger than 0)
7: update MT,OU,VS;
8: **return** g;
9: **else** pending the sub-flow;

It can be seen that IP cores have higher priority than GPPs for executing sub-flows. In order to record the number of free entries in MTs, each MT has an associated 3-bit counter. When DM sends a sub-flow to a PE the counter increments by 1, and when an interrupt from a PE is accepted the associated counter decrements by 1. A decoder is used to identify if a sub-flow can be executed by some IP cores of the platform. To avoid dead locks, a sub-flow can be sent to DM only when MTs, VS and OU all have a free entry, as presented in the bottom part of Fig. 12. The MT_full is set to 1 if none of PEs that can execute the sub-flow has a free entry.

The multi-channel design of DM can easily support the reconfigurability of FPGA platform using Xilinx Early Access Partial Reconfiguration (EAPR) technology. For example a (4, 4)-channel DM, when replacing an IP core, the associated counter is set to the number of entries of a MT, meaning that it can accommodate no more sub-flows. The only logic needs to be changed is the decoder. Fig. 13 illustrates the reconfigurable flow in which bitstream0 is replaced by other bitstreams (a bitstream is routed circuits for a PE).

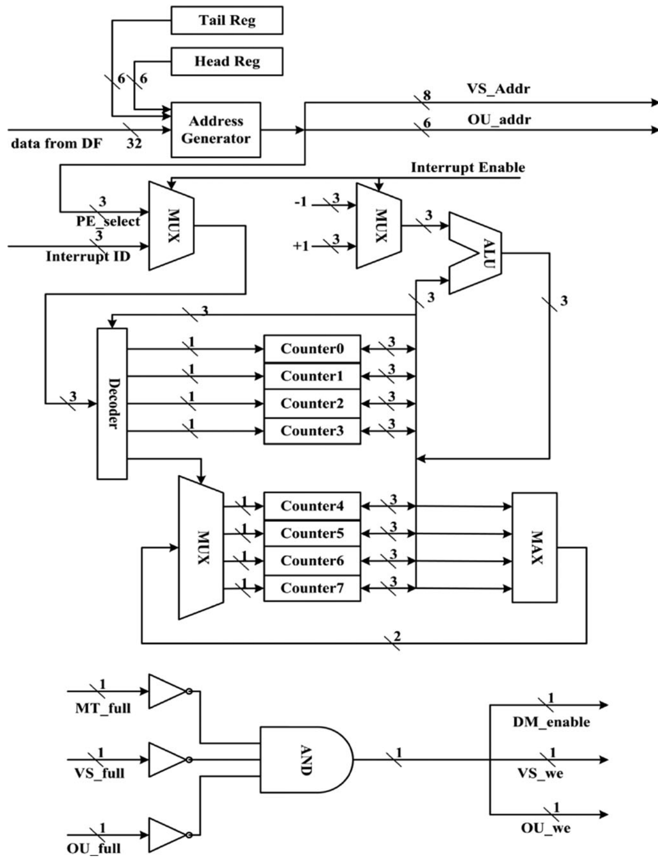


Fig. 12. Hardware implementation of dataflow mapper.

7 Experiments and Results

Our experimental platform is built on Xilinx Virtex-5 XC5VLX110T FPGA. The structure of the platform is presented in Fig. 14, including four PEs, one scheduler and peripherals. The PEs can be classified into three types: (1) the upper left part is a MicroBlaze processor that is responsible for issuing tasks and receiving data. Besides, it is also used as computing resources. (2) The upper right part is a MicroBlaze processor only used for computing. (3) The lower middle part contains two IP cores used as accelerators, which are used to accelerate the execution of the CC/DCT/Quant of JPEG application. The upper middle part is a hardware scheduler that detects RAW/WAW/WAR dependencies and eliminates WAW/WAR dependencies using renaming scheme. The lower left part is peripherals including timer, interrupt controller (intc), etc.

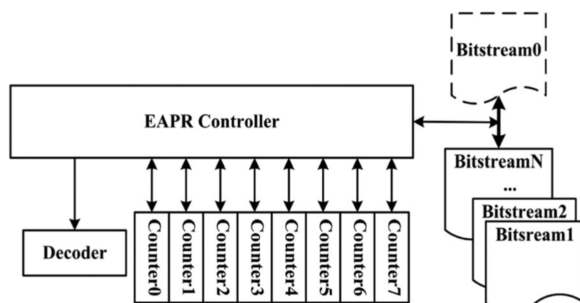


Fig. 13. Reconfigurable flow.

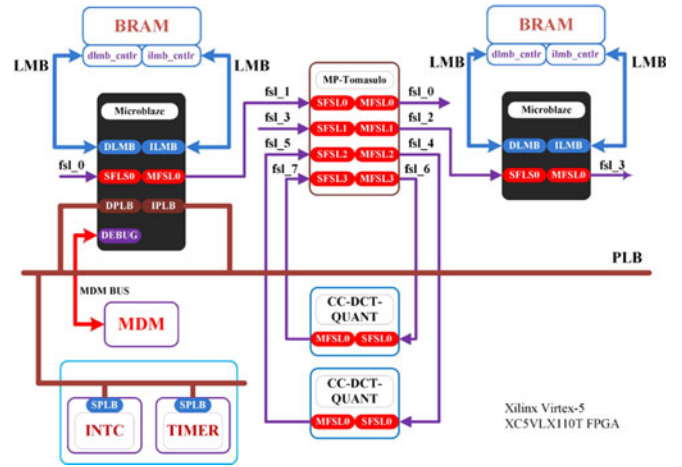


Fig. 14. Experimental platform built on Xilinx Virtex-5 FPGA.

7.1 JPEG Case Study

We first use JPEG to test the efficiency of our framework. JPEG can be divided into four stages: Color space Convert (CC), two-dimensional Discrete Cosine Transform (DCT), Quantization (Quant) and Entropy coding (Huffman). The first three stages have fixed amount input parameters and output parameters; therefore they are appropriate to be implemented as a hardware IP core. In contrast, the Huffman stage runs on MB because it does not have fixed amount output parameters and only takes 1.94 percent of the total execution time of the whole program on average. There are two IP cores on our platform, both are named CC-DCT-Quant and used to accelerate the execution the first three stages of JPEG application. The JPEG algorithm takes a BMP picture as input, and output a picture in JPEG format. The whole picture is divided into 8*8 bits blocks, each time only one block is processed. The configuration of the scheduler can be configured according to different applications.

With the help of MP-Tomasulo module, the sequential execution mode of JPEG program is converted to an OoO mode, only tasks that have RAW dependency with others need to run in order, while the WAW dependency will be dynamically eliminated by MP-Tomasulo module. We select 30 pictures of 6 different sizes for experiments, for each size we randomly picked five pictures in BMP format. Fig. 15 presents the experimental results [43]. We use the sequential execution time on a GPP as the basis, and

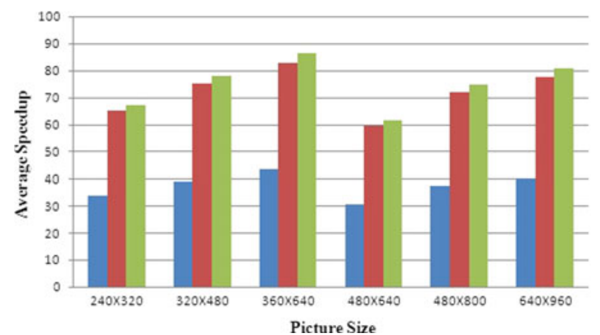


Fig. 15. Experimental results. The *X*-axis refers to the picture size, and the *Y*-axis is average speedup. The blue (first) bar is the average speedup if the WAW dependency is not eliminated. The red (second) bar is the actual speedup using our proposed method. The green (third) bar is the ideal speedup ignoring the scheduling time.

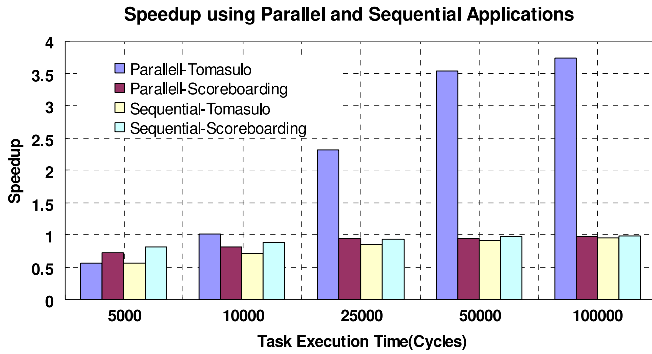


Fig. 16. Speedup using parallel and sequential applications.

compute the speedup using different strategies. It can be easily concluded that our algorithm actually eliminates the WAW/WAR dependencies, and achieves more than 95 percent of the ideal speedup.

7.2 Comparison Using Parallel and Sequential Applications

By reviewing the data dependencies problems (RAW, WAW and WAR) at instruction level, Scoreboarding and Tomasulo are both effective methods used in superscalar pipeline machines for out-of-order instruction execution. Of these two methods, Tomasulo is more widely used because it can eliminate WAW and WAR dependencies, while Scoreboarding only solves it by run tasks in sequence. Therefore, in this paper we choose Tomasulo instead of Scoreboarding at task level. In particular, we have compared our approach with the Scoreboarding techniques that has been applied in the reference [44].

We evaluated the speedup of Tomasulo algorithm and compared with the Scoreboarding techniques proposed in [43]. Both sequential and parallel applications in [43] are used as benchmarks, and the execution of the applications is configured from 5,000 to 100,000 cycles. It is clearly illustrated from Fig. 16 that the peak speedup of the Tomasulo algorithm on *parallel applications* achieves 3.74X, which significantly higher than that of the Scoreboarding (0.97X). The reason is that due to the renaming techniques employed by the Tomasulo, the WAW and WAR dependencies can be automatically eliminated. In contrast, Scoreboarding scheme is only able to detect the dependency.

Regarding the speedup of sequential applications, Tomasulo and Scoreboarding has similar performance as the data dependency cannot be eliminated, which are 0.95X and 0.98X respectively.

7.3 Comparison on Regular Tasks

We measured speedups of the typical task sequences through partitioning the test tasks in [2]. In this test case, we select regular test with 11 different random tasks. The sample task sequence is presented in Table 3.

TABLE 3
Sample Task Sequence

Task number	Task type	Source variables	Destination variable
T_1	JPEG	a	c
T_2	IDCT	b	a
T_3	JPEG	J	i
T_4	AES_ENC	d, e	F
T_5	AES_ENC	h,e	d
T_6	DES_DEC	E,e,	g
T_7	JPEG	a	c
T_8	DES_DEC	H,e	g
T_9	DES_DEC	H,e	a
T_{10}	JPEG	f	e
T_{11}	IDCT	b	a

Fig. 17 depicts the comparison between theoretical and experimental results of the task sequence. The length of the sequence increases from 1 to 11, and the curve of experimental value is consistent with theoretical value, but slightly larger. This is because theoretical value has not considered of scheduling and communication overheads. The average of overheads is less than 4.9 percent of task running time itself. The execution time remains flat when the length of the queue increases from 2 to 6. The result is caused by out-of-order execution and completion. Taking task No.2 and No.4 for instances, No.2 takes more clock cycles to finish than No.4. As there are not data dependencies among these tasks, they can be issued at the same time. After task No.4 is finished, it must wait until all the predecessor tasks are finished.

Compared to Scoreboarding, Tomasulo has higher scheduling overheads, which leads to a bigger gap between experimental and theoretical value. However, since MP-Tomasulo can not only detect WAW and WAR hazards but also eliminate them by register renaming, the overall speedup is significantly larger than Scoreboarding.

7.4 Discussion

In this section we use the timing diagram to show the inter-task data dependencies, and how it is solved.

Fig. 18 illustrates the timing diagraph for both with Tomasulo scheduling and without Tomasulo scheduling algorithms. [a] presents the timing graph with Tomasulo schedulers. At the model start-up, the token representing the task T_1 is generated and then can be dispatched. The transition checks the state of computational kernels in the modeled system, and assigns the JPEG IP core to the task T_1. In the second time unit, the task T_2 is generated. At that time, the sources variable “a” is not ready due to T_1 is not finished, thus a read-after-write structural dependence is identified and the task T_2 stalls. In the third time unit, task T_3 is generated. As T_3 has no data dependencies with T_1,

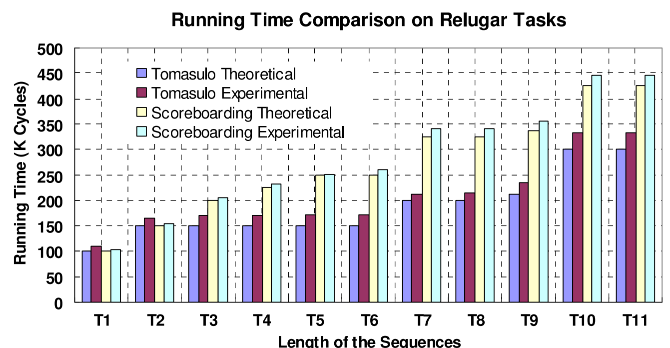


Fig. 17. Experimental results of regular tasks.

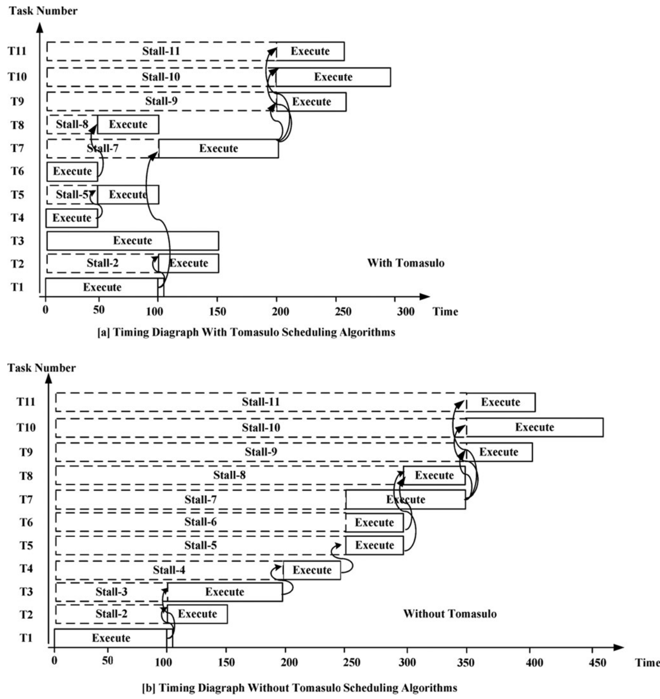


Fig. 18. Timing Diagraph. The horizontal axis denotes the model time, while the vertical axis denotes the execution flow of tasks. The events executing and writing result of each task are represented by bars outlined with solid lines, while the stalls of tasks are represented by the bars outlined with dashed lines. The length of a bar represents the duration time of each event. Besides, the arrowed lines are used to represent different types of inter-task dependences.

therefore it could be assigned to the GPP for computational kernel immediately. Similarly, task T_4 and T_6 can also be issued immediately due to there are no inter-task dependencies while other tasks should be stalled until the dependencies are eliminated. Finally, all the tasks are finished at the time 300k cycles.

By contrast, [b] illustrates the timing graph without the Tomasulo scheduling algorithm. Due to there is no OoO scheduling methods, all the tasks have to be issued in sequential. As is clearly described in Fig. 19 [b], all the tasks are finished at 450k cycles, which means the Tomasulo algorithm can efficiently reduce the execution time of the task sequencing by about 33 percent.

By investigating the timing diagram, we gain a deep insight on how tasks interact with each other and maintain the data dependences in our scheme. Furthermore, we can verify the correctness of simulation result through

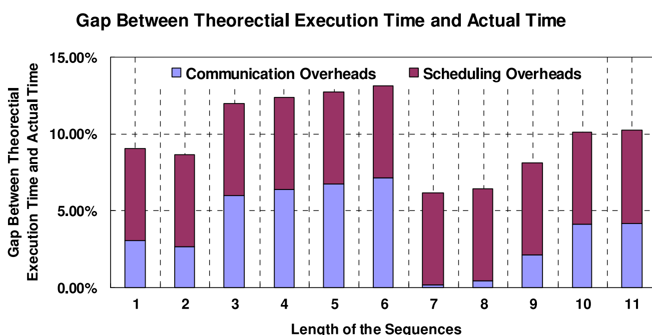


Fig. 19. Gap between theoretical execution time and actual execution time.

TABLE 4
Scheduler Configuration

	Number of Units	Number of Entries	Entry Width (bits)
DF	1	64	32
VS	1	256	6
OU	1	64	256
MT	8	4	334

comparing the timing diagram with task execution flow on a prototype system. We have studied simulation results in a number of cases. The simulation results are exactly consistent with the actual results obtained from prototype systems. To this end, we confirm that our Out-of-order scheduling scheme can correctly schedule tasks with various dependences to exploit parallelism.

Fig. 19 depicts the gap between theoretical and experimental results of a regular task sequence. In this case we use the same benchmark as the regular tasks used in Section 7.3. When the length of the sequence increases from 1 to 11, the gap between the experimental and actual results keeps at 10 percent. The gap includes both communication overheads and scheduling overheads. In particular, the communication overheads keep stable approximately at 6 percent, while the scheduling overheads fluctuate from 0.15 to 7.13 percent, respectively.

7.5 Hardware Utilization

Our experimental platform is constructed on Xilinx Virtex-5 XC5VLX110T FPGA. A (4,4)-channel hardware scheduler is prototyped, the results are shown in the following tables.

Table 4 illustrates the scheduler configuration. In the JPEG case, we integrate 1 DF, 1 VS, 1 OU and 8 MT modules. Each DF has 64 entries and data width is configured at 32 bits. In order to improve the task level parallelism, we use eight MT in total, each of which contains four entries at 334-bit width.

Table 5 illustrates the resources consumption for our hardware scheduler and the MicroBlaze processor. Our scheduler takes about the same Look-Up Tables (LUTs) as a MicroBlaze while significantly less number of less registers. Most of the LUTs in a MicroBlaze are used as logic, while about one third are used as RAM in our scheduler. Furthermore, to calculate the area cost, we use the area utilization model that is presented by Jonathan Rose in [45]. The area cost of the hardware scheduler is similar to the Microblaze processor based on the LUTs consumption.

TABLE 5
Resources Usage

	Hardware Scheduler	MicroBlaze
Registers	295(0.4%)	1,445(2.1%)
LUTs used as Logic	1,158(1.7%)	1,434(2.1%)
LUTs used as RAM	510(0.7%)	84(0.1%)
Total LUTs	1,668(2.4%)	1,518(2.2%)
Area Utilization	1.35 MM^2	1.23 MM^2

(%): Resources used/Total on-chip resources.*

TABLE 6
Scheduling Overhead Comparison

	Average Scheduling Overhead (cycles)	Frequency	Scheduling Time
SW Scheduler on Microblaze	>10,000	100 MHz	>100 us
SW Scheduler on Intel Core i5 4460	≈10,000	3.2 GHz	≈3.13 us
HW Scheduler	~20	100 MHz	0.2 us

In order to show the performance of our scheduler, Table 6 compares our work with software MP-Tomasulo scheduler, which runs at the Microblaze microprocessor at 100 MHz. On average, the software scheduler takes more than 10,000 cycles for scheduling while our hardware scheduler only takes about 20 cycles, which means the speedup of the scheduling achieves up to 500X. Furthermore, we have evaluated the scheduling time at Intel Core i5 4460 processors at 3.2 GHz. The speedup of the HW scheduler against Intel SW scheduler on Core i5 is 15.65X approximately.

8 Conclusions

Out-of-order scheduling is playing a key role in exploit task level parallelization. In this paper, we focused on the hardware implementation of automatic out-of-order execution engines on a FPGA based heterogeneous platforms. We have proposed a flexible hardware scheduler to fit reconfigurable heterogeneous systems, and then we detail one of the OoO task execution methods. WAW and WAR dependences can be dynamically eliminated so as to greatly accelerate the execution of applications on heterogeneous platforms. Experimental results demonstrate that our framework is efficient and the average performance achieves 95 percent of theoretical speedup. Furthermore, experimental results also show that our design is efficient on both performance and resources usage. The performance is 500X faster than the software scheduler, while the resources usage in not more than a MicroBlaze. Besides, our hardware scheduler supports the dynamic reconfiguration of the FPGA platform by changing a small part of logic.

Although the experimental results are promising, there are a few future directions worth pursuing. First, improved task partitioning and further adaptive mapping schemes are essential to support automatic task-level parallelization. Second, we also plan to study the out-of-order task execution paradigm to explore the potential parallelism in data-intensive applications.

Acknowledgments

This work was supported by the National Science Foundation of China under grants (No. 61379040, No. 61272131, No. 61202053, No. 61222204, No. 61221062), Jiangsu Provincial Natural Science Foundation (No. SBK201240198), Open Project of State Key Laboratory of Computer Architecture, ICT, CAS, and CCF-Tencent Open Research Fund. The authors deeply appreciate many reviewers for their insightful comments and suggestions. Professor Xi Li is the corresponding author.

References

[1] G. Gupta and G. S. Sohi, "Dataflow execution of sequential imperative programs on multicore architectures," in *Proc. 44th Annual IEEE/ACM Int. Symp. Microarchit.*, 2011, pp. 59–70.
[2] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Proc. 5th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 1995, pp. 207–216.
[3] O. A. R. Board. (1998). OpenMP C and C++ application program interface version 1.0 [Online]. Available: http://www.openmp.org
[4] OpenCL [Online]. Available: http://www.khronos.org/opencl/, Jan. 2015.
[5] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta, "CellSs: A programming model for the cell BE architecture," in *Proc. ACM/IEEE Conf. Supercomput.*, 2006, p. 86.
[6] C. Wang, J. Zhang, X Zhou, X. Feng, and X. Nie, "SOMP: Service-Oriented Multi Processors," in *Proc. IEEE Int. Conf. Services Comput.*, 2011, pp. 709–716.
[7] Y. Etsion, F. Cabarcas, A Rico, A. Ramirez, R. M. Badia, E. Ayguade, J. Labarta, and M Valero, "Task superscalar: An out-of-order task pipeline," in *Proc. 43rd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2010, pp. 89–100.
[8] Y. Kim, J. Lee, T. X. Mai, and Y Paek, "Improving performance of nested loops on reconfigurable array processors," *ACM Trans. Archit. Code Optimization*, vol. 8, no. 4, pp. 1–23, 2012.
[9] M. Thuresson, M. Själander, M. Björk, L. J. Svensson, "Larsson-edefors per, and stenstrom per," in *FlexCore: Utilizing Exposed Datapath Control for Efficient Computing*. Norwell, MA, USA: Kluwer, 2009, pp. 5–19.
[10] J. Zhang, C. Wang, X. Li, and X. Zhou, "FPGA implementation of a scheduler supporting parallel dataflow execution," in *Proc. IEEE Int. Symp. Circuits Syst.*, 2013, pp. 1216–1219.
[11] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM J. Res. Develop.*, vol. 11, no. 1, pp. 25–33, 1967.
[12] J. Wawrzynek, D. Patterson, M. Oskin, S.-L. Lu, C. Kozyrakis, J.C. Hoe, D. Chiou, and K. Asanovic, "RAMP: Research accelerator for multiple processors," *IEEE Micro*, vol. 27, no. 2, pp. 46–57, Mar./Apr. 2007.
[13] T. Givargis and F. Vahid, "Platune: A tuning framework for system-on-a-chip platforms," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 21, no. 11, pp. 1317–1327, Nov. 2002.
[14] G. Kuzmanov, G. Gaydadjiev, and S. Vassiliadis, "The MOLEN processor prototype," in *Proc. 12th Annu. IEEE Symp. Field-Programmable Custom Comput. Mach.*, 2004, pp. 296–299
[15] D. Gohringer, M. Hubner, V. Schatz, and J. Becker, "Runtime adaptive multi-processor system-on-chip: RAMPSoC," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, 2008, pp. 1–7.
[16] K Faraydon, M. Alain, N. Anh, A. Utku, and A. Tarek, "A multilevel computing architecture for embedded multimedia applications," *IEEE Micro*, vol. 24, no. 3, pp. 56–66, May/Jun. 2004.
[17] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar processors," in *Proc. Int. Symp. Comput. Archit.*, 1995, pp. 414–425.
[18] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith, "Trace processors," in *Proc. Int. Symp. Microarchit*, 1997, pp. 138–148.
[19] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the cell multiprocessor," *IBM J. Res. Develop.*, vol. 49, pp. 589–604, 2005.
[20] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, L. Jae-Wook, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "The raw microprocessor: A computational fabric for software circuits and general-purpose programs," *IEEE Micro*, vol. 22, no. 2, pp. 25–35, Mar./Apr. 2002.
[21] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukolun, "The Stanford Hydra CMP," *IEEE Micro*, vol. 20, no. 2, pp. 71–84, Mar. 2000.
[22] K. Sankaralingam, R. Nagarajan, R. McDonald, R. Desikan, S. Drolia, M. S. Govindan, P. Gratz, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, S. W. Keckler, and D. Burger, "Distributed microarchitectural protocols in the TRIPS prototype processor," in *Proc. Int. Symp. Microarchit.*, 2006, pp. 480–491.
[23] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "WaveScalar," in *Proc. Int. Symp. Microarchit.*, 2003, p. 291.

[24] C. Villavieja, J. A. Joao, R. Miftakhutdinov, and Y. N. Patt, "Yoga: A hybrid dynamic vliw/ooo processor," High Performance Systems Group, Dept. Elect. Comput. Eng., The University of Texas at Austin, Austin, TX, USA, 2014.

[25] J Planas, R. M. Badia, E. Ayguad, and J. Labarta, "Hierarchical task-based programming with starss," *Int. J. High Perform. Comput. Appl.*, vol. 23, no. 3, pp. 284–299, 2009.

[26] G F. Diamos and S. Yalamanchili, "Harmony: An execution model and runtime for heterogeneous many core systems," in *Proc. 17th Int. Symp. High Perform. Distrib. Comput.*, 2008, pp. 197–200.

[27] E. Lubbers and M. Platzner, "ReconOS: Multithreaded programming for reconfigurable computers," *ACM Trans. Embedded Comput. Syst.*, vol. 9, no. 1, pp. 1–33, 2009.

[28] W. Peck, E. Anderson, J. Agron, J. Stevens, F. Baijot, and D. Andrews, "Hthreads: A computational model for reconfigurable devices in international conference on field programmable logic and applications," in *Proc. Int. Conf. Field Programmable Logic Appl.*, Madrid, Spain, 2006, pp. 1–4.

[29] D. Koch, C. Beckhoff, and J. Teich. "A communication architecture for complex runtime reconfigurable systems and its implementation on spartan-3 FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, 2009, pp. 253–256.

[30] K. Rupnow, K. D. Underwood, and K. Compton, "Scientific application demands on a reconfigurable functional unit interface," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 4, no. 2, pp. 1–30, 2011.

[31] S. Balakrishnan and G. Sohi. "Program demultiplexing: Dataflow based speculative parallelization of methods in sequential programs," in *33rd Int. Symp. Comput. Archit.*, 2006, pp. 302–313.

[32] D. S. McFarlin, C. Tucker, and C. Zilles, "Discerning the dominant out-of-order performance advantage: Is it speculation or dynamism," in *Proc. Int. Conf. Architectural Support Program. Languages Operating Syst.*, 2013, pp. 241–252.

[33] S. Sridharan, G. Gupta, and G. S. Sohi, "Adaptive, efficient, parallel execution of parallel programs," in *Proc. 35th ACM SIGPLAN Conf. Program. Language Des. Implementation*, 2014, pp. 169–180.

[34] KHRONOS Group. (2010). OpenCL [Online]. Available: http://www.khronos.org/opencl/

[35] P. Bellens, J. M. Perez, F. Cabarcas, A. Ramirez, R. M. Badia, and J. Labarta, "CellSs: Scheduling techniques to better exploit memory hierarchy," *Sci. Program. - High Perform. Comput. Cell Broadband Engine*, vol. 17, nos. 1/2, pp. 77–95, 2009.

[36] A. Hayashi, Y. Wada, T. Watanabe, T. Sekiguchi, M. Mase, J. Shirako, K. Kimura, and H. Kasahara, "Parallelizing compiler framework and API for power reduction and software productivity of real-time heterogeneous multicores," in *Proc. 23rd Int. Conf. Languages Compilers Parallel Comput.*, 2010, pp. 184–198.

[37] D. Kuck, E. Davidson, D. Lawrie, A. Sameh, C. Q. Zhu, A. Veidenbaum, J. Konicek, P. Yew, K. Gallivan, W. Jalby, H. Wijshoff, R. Bramley, U. M. Yang, P. Emrath, D. Padua, R. Eigenmann, J. Hoeflinger, G. Jayson, Z. Li, T. Murphy, and J. Andrews, "The cedar system and an initial performance study," in *Proc. 25 years Int. Symp. Comput. Archit. (Selected Papers)*, 1998, pp. 462–472.

[38] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukotun, "The Stanford Hydra CMP," *IEEE Micro*, vol, 20, no. 2, pp. 71–84, Mar./Apr. 2000.

[39] Y. Etsion, F. Cabarcas, A Rico, A. Ramirez, R. M. Badia, E. Ayguade, J. Labarta, and M. Valero, "Task Superscalar: An Out-of-Order Task Pipeline," in *Proc. 43rd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2010, p. 89–100.

[40] B. Mladen and N. Tim, "A distributed, simultaneously multithreaded (SMT) processor with clustered scheduling windows for scalable DSP performance," *J. Signal Process. Syst.*, vol. 50, pp. 201–229, 2008.

[41] J. C. Jenista, Y. H. Eom, and B. C. Demsky, "OoOJava: Software out-of-order execution," in *Proc. 16th ACM Symp. Principles Practice Parallel Program.*, 2011, pp. 57–68.

[42] F. Liu, H. Ahn, S. R. Beard, T. Oh, and D. I. August, "DynaSpAM: Dynamic spatial architecture mapping using out of order instruction schedules," in *Proc. 42nd Int. Symp. Comput. Archit.*, 2015, pp. 541–553.

[43] C. Wang, X. Li, J. Zhang, X. Zhou, and X. Nie, "MP-Tomasulo: A Dependency-aware automatic parallel execution engine for sequential programs," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 2, pp. 1–26, 2013.

[44] C. Wang, X. Li, J. Zhang, P. Chen, Y. Chen, X. Zhou, and R. C. C. Cheung, "Architecture support for task out-of-order execution in MPSoCs," *IEEE Trans. Comput.*, vol. 64, no. 5, pp. 1296–1310, May 1, 2015.

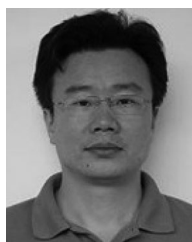
[45] K. Ian and R. Jonathan, "Area and delay trade-offs in the circuit and architecture design of FPGAs," in *Proc. 16th Int. ACM/SIGDA Symp. Field Programmable Gate Arrays*, 2008, pp. 149–158.

Chao Wang (M'11) received the BS and PhD degrees from the University of Science and Technology of China, in 2006 and 2011, respectively, both in of computer science. He is a faculty member in Embedded System Lab, Suzhou Institute of University of Science and Technology of China, Suzhou, China. His research interests focus on multicore and reconfigurable computing. He serves as the editor board member for MICPRO, IET CDT, and IJHPSA. He is also the publicity chair of HiPEAC 2015 and ISPA 2014. He is a member of the IEEE.

Junneng Zhang received the BS degree in computer science from the University of Science and Technology of China in 2009. He is currently working toward the PhD degree in the Embedded System Lab, Suzhou Institute of University of Science and Technology of China, Suzhou, China. His research interests focus on multiprocessor system on chip, reconfigurable computing, and operating system. He is a student member of the IEEE.

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, several national 863 projects, and NSFC projects. He is a member of the IEEE.

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



Xuehai Zhou is the executive dean in the School of Software Engineering, University of Science and Technology of China, and a professor in the School of Computer Science. He serves as a general secretary of steering committee of computer College fundamental Lessons, and technical committee of Open Systems, China Computer Federation. His research interests include various aspects of multicore and distributing systems. He has led many national 863 projects and NSFC projects. He has published more than 100 International journal and conference articles in the areas of software engineering, operating systems, and distributed computing systems. He is a member of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.