Architecture Support for Task Out-of-Order Execution in MPSoCs

Chao Wang, *Member, IEEE*, Xi Li, Junneng Zhang, Peng Chen, Yunji Chen, Xuehai Zhou, *Member, IEEE*, and Ray C.C. Cheung, *Member, IEEE*

Abstract—Multi-processor system on chip (MPSoC) has been widely applied in embedded systems in the past decades. However, it has posed great challenges to efficiently design and implement a rapid prototype for diverse applications due to heterogeneous instruction set architectures (ISA), programming interfaces and software tool chains. In order to solve the problem, this paper proposes a novel high level architecture support for automatic out-of-order (OoO) task execution on FPGA based heterogeneous MPSoCs. The architecture support is composed of a hierarchical middleware with an automatic task level OoO parallel execution engine. Incorporated with a hierarchical OoO layer model, the middleware is able to identify the parallel regions and generate the sources codes automatically. Besides, a runtime middleware Task-Scoreboarding analyzes the inter-task data dependencies and automatically schedules and dispatches the tasks with parameter renaming techniques. The middleware has been verified by the prototype built on FPGA platform. Examples and a JPEG case study demonstrate that our model can largely ease the burden of programmers as well as uncover the task level parallelism.

Index Terms—Middleware, architecture support, MPSoC, data dependencies, FPGA, out-of-order execution

1 Introduction

High performance reconfigurable computing technology has emerged and widely applied during the past decades. The remarkable evolution of heterogeneous multicore research paradigms and the invasion of reconfigurable hardware accelerators have made it possible to integrate hundreds of cores into the current petaflop supercomputing machines. The combination of reconfigurable computing and multi-core technologies has been regarded as one of the most promising future processor architectures [1], [2]. However, critical issues beyond raw computational capabilities are becoming increasingly important, such as programmability, flexibility, scalability, power consumption and so on.

This scenario and raising demands has led to the emergence of FPGA based multiprocessor system on chip (MPSoC) composed of a variety of heterogeneous computational units. It has been widely acknowledged that the roadmap to the next generation of exascale computers will bring a tremendous speed up in various applications through integration of multi-core processors and hardware accelerators, like graphic processing units (GPUs) or FPGAs [2].

Of the cutting-edge GPU and FPGA based research approaches, reconfigurable heterogeneous hardware accelerators can achieve both high performance and promising flexibility. On one hand, since numerous processors are being integrated into single chip, reconfigurable multiprocessor system-on-chip can provide increasingly speedups to diverse embedded systems and applications. On the other hand, the involvement of reconfigurable hardware platform like FPGA and CPLD could efficiently facilitate researchers to decrease the embedded system design time and space costs, as well as to shorten the time-to-market (TTM) simultaneously.

However, the side effect of FPGA based MPSoC has been exposed like a double-edged sword. Programmability for MPSoC is still posing serious challenges in particular. Since the hardware is adapted to fit in the applications, programming models and middleware architecture support should be invisibly filling the gaps between different architectures.

Up to now, considerable amount of literatures on hybrid programming models have been conducted at task level. For instances, OpenMP [3], MPI, Intel's TBB, OpenCL [4] and Cilk [5] are very successful programming paradigms. Recently, CUDA is becoming very popular in GPU based programming models. However, a major weakness of these approaches is the inadequate automatic parallelization degree. In particular, task mapping and scheduling plans are operated manually, which means the achieved speedup is largely dependent on the experiences of programmers. Meanwhile, most of these works focus on either the symmetric multiprocessors or GPU based acceleration engines, which cannot be applied to reconfigurable heterogeneous MPSoC scenarios directly.

Alternatively, there have been creditable MPSoC programming models devoted to specific hardware architectures,

- *C. Wang, X. Li, and P. Chen are with the Department of Computer Science, University of Science and Technology of China, Hefei, Anhui, China. E-mail: saintwc@mail.ustc.edu.cn, llxx@ustc.edu.cn, blueardour@gmail.com.*
- *J. Zhang is with the University of Science and Technology of China, Hefei 230027, Anhui, China. E-mail: zjneng@mail.ustc.edu.cn.*
- *Y. Chen is with the State Key Lab of CARCH, CAS, Beijing, 100190, China. E-mail: cyj@ict.ac.cn.*
- *X. Zhou is with the Suzhou Institute of University of Science and Technology of China, Suzhou 215123, China. E-mail: xhzhou@ustc.edu.cn.*
- *R.C.C. Cheung is with the Electrical Engineering Department, City University of Hong Kong, Kowloon, Hong Kong. E-mail: r.cheung@cityu.edu.hk.*

Manuscript received 30 Dec. 2012; revised 31 Jan. 2014; accepted 16 Mar. 2014. Date of publication 8 Apr. 2014; date of current version 8 Apr. 2015. Recommended for acceptance by Y. Pan.
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/TC.2014.2315628

0018-9340 © 2014 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

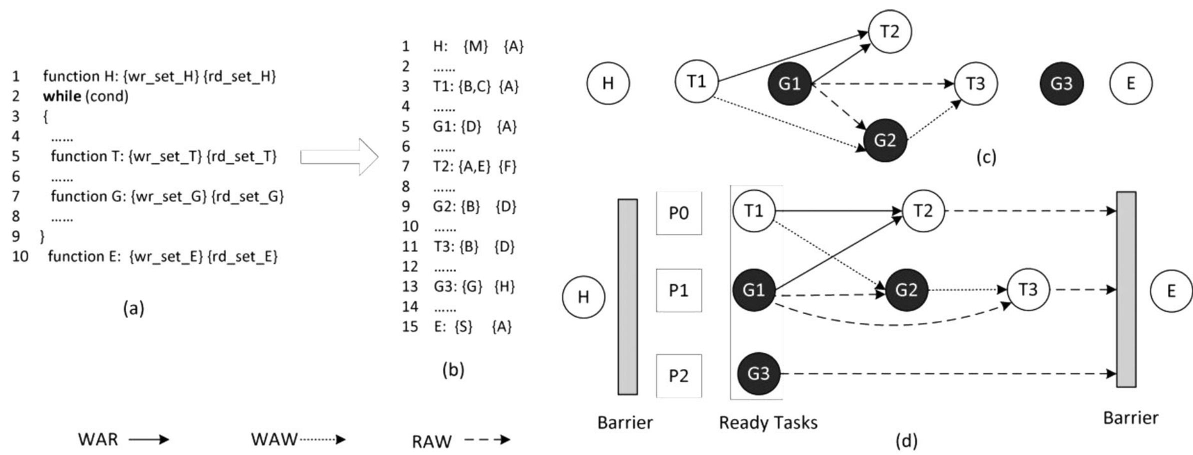


Fig. 1. (a) Example Pseudocode that invokes functions H, T, G and E, e.g., the task T:{wr_set_T} {rd_set_T} reads {rd_set_T} tokens and generates {wr_set_T} tokens during execution. (b) Dynamic loop unrolling of the functions H, T, G and E in the program order, with the read/write data set of each invocation. (c) Dataflow graph of the dynamic function stream, in which RAW, WAW and WAR inter-task dependencies are presented with heterogeneous processors. (d) Dataflow execution schedule of the function stream, under the architecture scenario of three processors.

such as StarSs [6] and CellSs [7]. Although both approaches provide superscalar or renaming techniques allowing tasks out-of-order (OoO) execution, the constrained architecture avoids them to be directly applied to the reconfigurable MPSoC architectures. As the system complexity grows, the problem of how to design a flexible programming model is becoming increasing challenging. Until now the problem has not been completely figured out yet.

To address the above problems, in this paper we present an architecture support for heterogeneous multiprocessors with hierarchical middleware for OoO execution. We intend to integrate a sound framework that is composed of a hierarchical layer model, an execution flow, an OoO scheduler with a mapping scheme. We claim following contributions:

1) We present a hierarchical middleware on reconfigurable MPSoCs, from which programmers are no longer required to gain hardware implementation and task partitioning plans. In order to analyze the annotation based codes, we propose an execution model that translates sources programs to internal functions for parallel execution.
2) We propose an OoO scheduling mechanism that checks the data dependencies, renames the parameters, and issues the tasks automatically when the tasks are ready. The renaming technique is applied from traditional instruction level to task level, regarding each processor (IP core) as a function unit.
3) We introduce an adaptive mapping scheme to map the tasks to target function units at runtime. When hardware architecture is reconfigured, tasks can be remapped automatically without the rebooting.
4) We implement the MPSoC prototype on Xilinx FPGA. EEMBC benchmarks (e.g., IDCT, AES, DES and JPEG) are implemented in both software and hardware. Multiple Microblaze processors are integrated as scheduler and computing processors. The prototype platform can be used for evaluation and verification for task partitioning scheme, scheduling, interconnect, etc.

The structure of the paper is decomposed as the following. In Section 2 we outline the OoO motivation and review the literatures related to this work. Section 3 describes a typical reconfigurable MPSoC architecture and the hierarchical middleware. Section 4 presents the features implemented in the runtime library, including detailed OoO task scheduling, mapping scheme and synchronization. Section 5 illustrates the FPGA prototype with experimental results. Finally, we conclude the paper in Section 6.

2 Related Work

2.1 OoO Task Execution

In order to illustrate the motivation of OoO task execution, we first present how the tasks are running on the MPSoC hardware architectures in parallel. We inherit the token based description of dataflow execution model in [8], which is extended to a general heterogeneous multicore computing scenario in this paper. Generally, dataflow execution model handles inter-task dependences using tokens to manifest production and availability of I/O parameters. Based on the token based technique, we make two essential enhancements. First we map tokens with parameters instead of memory addresses, to match the abstraction for functional source and destination. Second, we assign each function with multiple read tokens and a single write token, to manage both production and consumption of parameters.

When the execution encounters a *task* to be considered for dataflow execution, it requests read (write) tokens for exclusive guarantee in the function read (write) set; it is ready for execution only after it has acquired all its requisite tokens. Similarly, upon completion, it releases the tokens which are then spawned to the waiting function(s) if necessary. When a pending function has acquired its requisite tokens, it can be offloaded and submitted for execution immediately.

We illustrate the execution model with the simple sequential program example of Fig. 1a, which invokes the two functions T and G within a loop, as well as a head task H in prior to the loop and a tail task E executed after the loop. Fig. 1b describes an example dynamic sequence of

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

Types	Typical	Strength	Weakness
General parallel programming model **(No OoO)**	OpenMP [3] CnC MapReduce OpenCL [4] Cilk [5]	General model for CMP processors	Bring burden to programmers
Specific parallel programming model **(With OoO)**	StarSs [6], CellSs [7], Oscar [9]	Support automatic OoO execution	Applied only to CellBE architecture and SMP servers
Coarse Grained Task Level Parallelism **(No OoO)**	CEDAR [10], MLCA [11], Multiscalar [12], Trace [13],	Perform well for traditional superscalar machines, run	OoO not supported, programmers handle the task
Dataflow Based OoO Execution Model **(No OoO)**	FlexCore [21]	Tasks running as instructions	OoO not supported
Dataflow Based OoO Execution Model **(with OoO)**	TaskSuperscalar[19]	Support OoO automatic parallel execution	Not applicable to FPGA with reconfiguration
	DSP [20]	With register renaming technologies of Tomasulo	Limited to DSP architectures
	OoOJava [22]	An OoO compiler for Java runtime	Not applicable to hardware execution engines
	Dataflow [8]	Race-free and determinate parallel execution	Not applicable to FPGA with reconfiguration

invocations of T and G after the loop is unrolled during execution, along with their dynamically computed write and read token sets generated in the program. Fig. 1c presents the data dependence between the functions. For example, T2 writes objects A and E, and thus it has a WAR dependence (solid arrows) on T1 and G1, which reads object A as an input. Likewise G2 has a RAW dependence (dashed arrows) on G1, and T3 has a WAW (dotted arrows) dependence on G2. These dependences must be preserved if the dynamic dataflow is to maintain the sequential execution of the static program, otherwise the correct results of the OoO execution cannot be guaranteed. Fig. 1d introduces the parallel execution model of the sequential code on different processors. The task H and E standard for the synchronization barriers before and after the loop iteration.

The above example illustrates how the loop based tasks should be analyzed and mapped to general heterogeneous multicore situations for OoO execution, while in the following sections, we will describe how the OoO mechanism is guaranteed in the FPGA based MPSoC scenario, especially for massive parallel computing machines.

Finally, the summary of state-of-the-art parallel execution engines is listed in Table 1. Although these approaches provide OoO engine by superscalar or renaming engines, they do not focus on the adaptive mapping for general FPGA platform with reconfigurable IP cores, therefore the flexibility across different architectures is still worth pursing.

2.2 Related Study

Parallel task execution models have been studied for parallel computing machines during past decades. First of all, task-based parallel programming model are quite popular to enhance ILP to TLP, such as OpenMP [3], MPI, Intel's TBB, OpenCL [4] and Cilk [5]. Most of these state-of-the-art programming paradigms focus on symmetric multiprocessors to significantly reduce the workload of programmers. One of the major drawbacks of these approaches is that automatic parallelization is not fully supported, which means programmers are required to handle the task mapping and scheduling schemes manually, therefore the speedup achieved is largely confined by the inadequate experiences of programmers. As a side effect, this could also increase the burden of programmers with synchronization and task scheduling on the symmetric multiprocessor architectures. For example, OpenMP [3] mainly depends on mutex lock mechanism to achieve synchronization between threads, but mutex locks are managed by the programmer. Other parallel programming paradigms such as MPI also needs programmer to find the potential task parallelism and synchronization explicitly, which not only increases the difficulty of multi-core programming complexity, and also led to unsatisfactory task parallelism due to limited experience of programmers. In addition, MapReduce, Intel Ct [23], and Intel CnC [24] are approaches facilitating high-level programming paradigms through its inherent support of task-based dataflow execution.

Meanwhile, compared to symmetric processors, heterogeneous processors are becoming increasingly dominating in embedded and high performance computing domains. Of the cutting-edge researches, FPGA based MPSoC is regarded as on of the most promising heterogeneous processor architectures [1]. With the increasing popularity of reconfigurable computing technology and MPSoC platform, parallelism is shifting from instruction level to task level. One approach is to utilize reconfigurable FPGA platform and integrate acceleration engines, such as Chimaera [25], Garp [26], OneChip [27]and other function units [28], [29], [30]. Moreover, there are several creditable general FPGA research platforms, such as RAMP [31], Platune [32] and MOLEN [33]. These studies focus on providing reconfigurable FPGA based environments with software tool chains to construct application specific MPSoC.

Alternatively, products and prototypes of processor are designed to increases TLP with coarser grained parallelism, such as CEDAR [10], MLCA [11], Multiscalar [12], Trace Processors [13], IBM CELL [14], RAW Processor [15], Intel Terascale and Hydra CMP [16]. These designs present thread-level or individual cores which can split a group of applications into small speculatively independent threads. Some other works like TRIPS [18] and Wave Scalar [17] combine both static and dynamic dataflow analysis in order to exploit more parallelism.

To attack the programming wall problem with the tremendous invasion of chip integration, MPSoC programming models are taken into consideration, such as Oscar [9], StarSs [6] and CellSs [7]. With the help of the MPSoC programming paradigms, automatic parallelization with task level scheduling methods are also been motivated to operate high level parallelism such as [34], [35], [36], and [37]. In order to run tasks OoO, inter-task data dependency analysis and synchronization problem has posed a significant challenge for coarse-grained parallelization. Traditional algorithms, such as Scoreboarding and Tomasulo [38], explore instruction level parallelism (ILP) with multiple arithmetic units, which can dynamically schedule the instructions for OoO execution. Task Superscalar [19] proposes abstraction of OoO superscalar pipelines regarding processor as function units. [20] provides a modified version of Tomasulo [38] scheme to DSP based processor architectures to perform OoO task execution. FlexCore [21] presents a hybrid process architecture using an on-chip reconfigurable fabric (FPGA) to support runtime monitoring and bookkeeping techniques. Hyperprocessor [39] manages global dependencies using a universal register file. OoOJava [22] is a compiler-assisted approach that leverages developer annotations along with static analysis to provide an easy-to-use deterministic parallel programming model. The method is based on task annotations that instruct the compiler to consider a code block for OoO execution. [8] is a dataflow execution model that achieves parallel execution of statically-sequential programs. It dynamically parallelizes the execution of suitably-written sequential programs, in a dataflow fashion on multiple processing cores.

Finally, the summary of state-of-the-art parallel execution engines is listed in Table 1. Although these approaches provide OoO engine by superscalar or renaming engines, they do not focus on the adaptive mapping for general FPGA platform with reconfigurable IP cores, therefore the flexibility across different architectures is still worth pursing.

3 Architecture and Execution Model

The OoO middleware proposed in this paper is intended to provide an efficient middleware support between hardware and high level user applications, taking the benefit of the general MPSoC hardware platform with reconfigurable abilities. In this section we propose the hardware architecture and the execution model for the middleware support.

3.1 Hardware Platform

For most heterogeneous MPSoC architectures, application specific instruction processors (ASIP), digital signal processor (DSP), and intellectual property (IP) cores are introduced to uncover task level parallelism (TLP). Through the common feature among those platforms, Fig. 2 illustrates a heterogeneous MPSoC hardware platform constructed in FPGA, which consists of multiple general purpose processors (GPPs), DSP/ASIP processors, and a variety of heterogeneous IP cores. The responsibilities of the components are as follows:

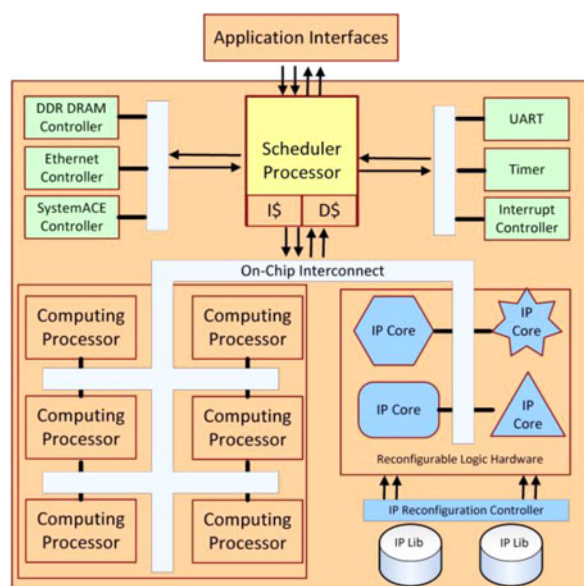


Fig. 2. The experimental MPSoC hardware platform constructed on Xilinx FPGA, the scheduler is connected to multiple computing processors, IP cores, and peripherals via on-chip interconnection.

1) Scheduler processor is employed to operate task scheduling and provides programming interface to users. At runtime, each task is mapped and then distributed to certain processor or IP core for execution. Scheduler also keeps the running status of processors and IP cores.
2) Computing processors provide a runtime environment for software tasks. In general, computing processors can execute different types of software tasks. Each processor provides software runtime function library for different applications.
3) Hardware IP cores are responsible for a specific kind of tasks to achieve acceleration. In addition, IP cores can be reconfigured and customized according to application demands. Specific tasks can be also spawned to DSP/ASIP or IP cores for hardware acceleration. Each hardware module can execute only limited types of tasks for accelerations, due to the RTL functional implementations.
4) Interconnect modules between schedulers, computing processors and hardware blocks are in charge of data communication. In this paper, we setup our experiments and simulation on a star network based on Xilinx peer to peer FSL [40] channels. Scheduler is connected to every processor or IP core with a pair of FSL bus links. All the interfaces are packaged into unified FSL manner for data communication. Note the interconnect structure can be replaced by other

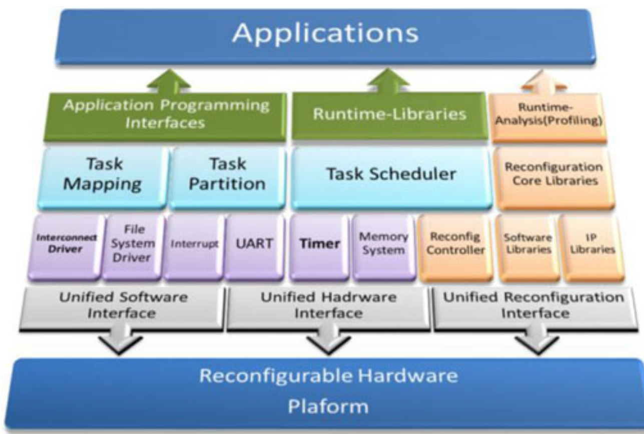


Fig. 3. High-level hierarchical model is composed of four layers: application layer, task scheduling layer, driver layer and communication layer.

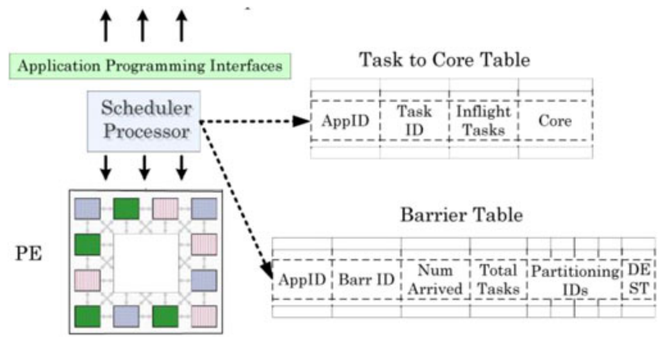


Fig. 4. Task to Core Table and Barrier Table, the former table keeps the record of which task is mapping to which core, while the later operates the synchronization between different computational tasks.

topologic modules, such as crossbar, mesh, hierarchical bus or ring architectures.

5) Memory and peripherals are integrated to maintain local data storage and peripherals, such as DDR DRAM controller, Ethernet controller, system ACE controller, UART, timer and interrupt controller. All these modules are connected to scheduler processor with bus-based interconnects, such as CoreConnect Processor Local Bus (PLB).

3.2 OoO Middleware Hierarchical Model

Throughout this paper, *tasks* refer to dynamic instances created when scheduler spawns a piece of code to computing GPPs or IP cores. Moreover, tasks are regarded as abstract instructions, and each IP core is treated as a dedicated functional unit for a specific hardware task. Fig. 3 illustrates the middleware hierarchical architectural model, which consists of four layers in general.

3.2.1 Application Management Layer

The middleware employs user runtime libraries to provide an execution environment for tasks. Application programming interfaces (APIs) show a high-level view of the internal implementations. After definition, the interfaces should be kept consistent after hardware reconfiguration.

Moreover, a runtime analysis module is integrated to support application monitoring and bookkeeping techniques. Also, the hotspot obtained by profiling indicates the parts executed for high frequencies and can be used to guide IP configurations.

3.2.2 Task Partitioning and Scheduling Layer

Task partitioning and scheduling methods play a vital role in architectural supports. Before tasks are offloaded to IP cores, OoO middleware should identify the target processor to run current task, and also decide when the task can be issued.

(1) Task to core mapping. Compared to state-of-the-art middleware and operation systems, this paper utilizes state-of-the-art dynamic partial reconfiguration support at runtime. Static core modules and reconfiguration modules (RMs) are implemented separately, of which only RMs are reconfigured at runtime to reduce the bitstream downloading overheads. In task partition and scheduling layer, reconfiguration core libraries are integrated. After IP cores are reconfigured, tasks mapping and scheduling strategies need to be reconsidered. Therefore a task-to-core table is employed to identify the target IP core, as is described in Fig. 4. The table maintains a mapping of tasks to cores to virtualize the selection of the destination core. Each table entry contains the task ID currently running on that core as well as a count of the number of issued tasks destined for that core. When new an IP core is deployed, the table elements will be flushed and updated.

When a task is issued, it obtains the core currently assigned to its destination core from the table and stores its results to the appropriate output queue upon completion. A side effect of this table based approach is that instructions will not issue to the fabric if the destination core is not available. This prevents the producing task from filling up the fabric if the consumer is not present. Even with the table, however, spawned tasks could accumulate in the fabric if the current task forces are switched out while data is in flight to it, which would require the consumer to be switched back into the same core to receive the values. To prevent this situation, the task-to-core mapping table maintains a count of the number of in-flight tasks destined for each core. On a request to switch out, the scheduler checks the number of in-flight tasks bound for its core. If this is greater than zero, the fabric is blocked from accepting any new tasks destined for that core and the core continues to execute until the in-flight counter reaches zero. At this point the application can be stalled and the fabric unblocked.

For each IP core, the specific task execution time, speedup, area cost and power consumption information are also maintained by scheduler. The information will assist scheduler to make task partition decisions and to achieve better load-balancing status and higher throughputs. Since FPGA is an area-constrained platform, different IP cores are competing for the limited hardware resources. For task scheduling, tasks are also considered to be arranged in sequences, which should improve the throughput as well as FPGA area efficiency.

(2) Barrier synchronization. Barriers are one of the most common synchronization operations. However, with a typical memory-based implementation, the overhead of executing a barrier can be significant, especially as the number of

cores increases. This overhead prevents the use of barriers at fine granularities. In cases where a barrier is followed by a serial function that is performed by one of the tasks and the output communicated to all participating tasks, the scheduler may directly synthesize the function into the fabric with the output communicated to the participants' output parameters.

To implement barriers for synchronization, a barrier table is integrated to ensure that all the returning tasks must not be allowed to issue to the fabric until all participating cores have arrived at the barrier, as is presented in Fig. 4. To achieve this, each core participating in the barrier loads some value(s) into its input queue. Once the loads from all of the cores have reached the head of their respective input queues and all tasks have indicated arrival at the barrier. The Barrier Table also determine that all tasks have arrived at the barrier, with information related to each active barrier. Each table contains as many entries as cores attached to a PE cluster, which includes both general processors (denoted in central white block), and heterogeneous accelerators (described in coloured blocks). The table keeps track of the total number of tasks, the number of arrived tasks, and the cores that are participating in the barrier. The number of arrived tasks and participating cores are updated whenever a task arrives, meanwhile the total and arrived task counts are compared to determine when to issue a task. In a system with multiple PE clusters, a dedicated bus communicates barrier updates among clusters. The bus transmits the barrier ID as well as the associated application ID. All tasks participating in a barrier must be actively running in order for all input data to be available. Each table entry maintains a list of the IDs of the local tasks that are participating in the barrier as well as a bit indicating if they are actively running. If a barrier is ready to be released but not all participating tasks are active, the scheduler controller triggers an exception to switch the missing tasks back in. Once all tasks are available, the barrier can proceed.

3.2.3 Driver and I/O Layer

In order to utilize the hardware resources integrated in FPGA platform, drivers for peripherals and memory systems are implemented. Similar to devices drivers of a general operating system, we introduce following modules for prototype demonstration: Interconnect driver is used to allow applications transfer data and control messages between microprocessor and IP cores, through buses or network-on-chip (NoC) infrastructures. File system driver is utilized to provide I/O access to the files. Local static and partial configuration bitstreams are also stored into file systems. When task execution is finished, the results are returned with interrupt signals. Peripherals such as UART and timer are integrated for user debugging. Reconfiguration controller manipulates the software and IP libraries for function units' replacement.

Corresponding to reconfiguration features, software and IP libraries are introduced at this level. Software libraries include functions to be executed locally, while IP libraries consist of back-up IP cores ready to be integrated.

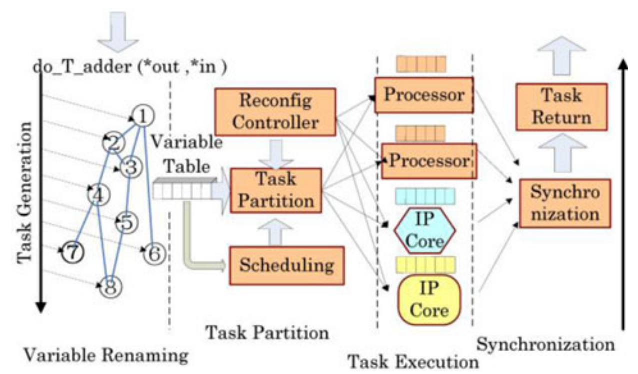


Fig. 5. High-level view of execution flow includes four stages: variable allocation and renaming, task partition, execution and synchronization.

3.2.4 Communication Interfaces Layer

The communication interfaces layer is in charge of data transmission between middleware and reconfigurable hardware platform. Generally there are three kinds of primitives: the unified software interface (USI), unified hardware interface (UHI), and unified reconfiguration interface (URI): USI primitive is employed when the information in transferred between two microprocessors. USI is composed of a series of function in libraries. UHI primitive is introduced to model the communication between microprocessor and hardware IP cores. Interrupt controller is employed in UHI to detect interrupt requests from interconnections. URI primitive is used only for IP reconfiguration. Reconfiguration controller in driver layer is utilized by URI to switch the partial bitstream at runtime.

3.3 General Execution Flow

Fig. 5 illustrates the task execution flow model. The entire task execution flow is divided into following four stages:

3.3.1 Variable Allocation and Renaming Stage

Similar to task superscalar [41], a task-generating thread sends tasks to the variable renaming stage for dependency decoding. Tasks are represented as do_T_adder (*out,*in), in which *in is the start address of input array where stores all the input operands, while *out indicates for outputs array. In order to detect data hazards automatically, the scheduler needs to collect all the operands for each issued task. However, we can only keep the limited operands instead of infinite user-defined variables (In most programming models, users are allowed to use any operands as they want). Therefore each task operand requires for a table entry allocation. If the table is not full yet, the variable will be renamed to an internal variable implicitly. The internal variable will live for the whole lifecycle of the task execution. Furthermore, changing numbers of operands is supported in our programming model. As *in and *out indicate the start of the operands array, each operand inside the array will be renamed to a fixed variable implicitly.

3.3.2 Task Partitioning Stage

Before tasks are sent to computing processors, the scheduler must decide which task runs on which function unit, and

also when the task is be issued. These two questions are solved by task partition methods and scheduling algorithms.

On one hand, for task partition, a task-to-processor table is employed to identify the target processor. The table maintains a mapping of tasks to cores to virtualize the selection of the destination core. Each table entry contains the task ID currently running on that core as well as a count of the number of issued tasks destined for that core. In most cases, IP core can accelerate task executions by specialized hardware logic or circuit design. So this paper utilizes a greedy strategy: if a vacant idle hardware function unit exists, the task will be sent to hardware; or else, task will be sent to software processor. However, in some cases, this method cannot provide an optimal partition result, therefore the method can be easily replaced by other algorithms.

On the other hand, potential inter-task dependencies need to be exploited to avoid data hazards from dramatically reduces the task level parallelism. Task scheduling mechanisms are employed to detect data hazards (RAW, WAW and WAR), and further for out-of-order task execution. In this framework, we tentatively apply both Scoreboarding and Tomasulo algorithms to task level. Of the two approaches, Scoreboarding can obtain shorter scheduling overheads, but the WAW tasks can only run in sequences. Meanwhile, Tomasulo algorithm can eliminate WAW data hazards by register renaming.

Additionally, benefiting from current dynamic partial reconfiguration supports, IP cores can be dynamically reconfigured at runtime. After IP cores are reconfigured, the task-to-processor table structures will be updated.

3.3.3 Task Execution Stage

The computing processor begins execution automatically when all the operands are received. Based on the hardware interconnect, results are returned through interrupts. One interrupt controller is integrated to detect results interrupt request and update the task variables. Since results from different tasks may be returned at the same time, a first-come-first-serve (FCFS) policy is used to deal with interrupts, and no interrupt preempt is supported.

3.3.4 Synchronization

The synchronization checks for inter-task data dependencies, and make sure all tasks are returned in-order. The cases of normal result-writing occur when there are no WAW or WAR hazards between current task and its predecessor tasks. Therefore, the scheduler consists of a design flow including in-order issue and out-of-order completion.

4 Runtime OoO Scheduling

Based on the hierarchical and execution flow, the most significant part is the scheduling scheme that to spawn the tasks out-of-order by analyzing the inter-task dependencies. At runtime, each function call of do_T_* is responsible for the intended behavior of the main program in the scheduler processor. At each call to these functions, the runtime will do the following actions:

1) Analyze data dependency including Read after Write (RAW), Write after Read (WAR) and Write after Write (WAW) [41]. The data dependency analysis is based on the parameters used by tasks.
2) Eliminate the WAW and WAR dependencies by parameter renaming techniques.
3) Identify the target function unit to run current task by a task mapping method.

4.1 Data Structures for Task Level Scoreboarding

If current task has no data dependency with previous issued tasks, then it can be spawned immediately. However, dependencies need to be checked and eliminated before they can be spawned.

By reviewing the data dependencies problems at instruction level, register renaming techniques are effective methods for OoO execution. Traditional approaches, like Scoreboarding and Tomasulo, are quite successful in the processor architecture designs. One major contribution of Task-Scoreboarding is an algorithm for OoO task execution.

In the description of programming model, each task is composed of task name, source and destination operands. As the tasks are treated as macro instructions, then the data dependencies problem can also happen for task level. By reviewing the data dependencies problems (RAW, WAW and WAR) at instruction level, Scoreboarding and Tomasulo are both effective methods for OoO instruction execution. The reasons for which we choose Scoreboarding instead of Tomasulo are the following:

1) Scoreboarding can provide a light-weight task hazards engine for OoO execution. The architecture is simpler, which brings smaller scheduling overheads.
2) For task level parallelization, WAW and WAR happens not as much as at instruction level. Most programmers are intended to use different parameters in case of WAR and WAW hazards. Therefore introducing a mechanism as complexes as Tomasulo is not just fair enough.

Similar to instruction level, we use parameter renaming techniques to uncover task level parallelism. Tomasulo is applied to task level instead of Scoreboarding which could do no more than stall when dependency occurs. There are five components of the scheduler:

1) *Functional unit status*—indicates the state of the functional unit (FU).

   Table 2 lists the function unit status. Whenever there is a task whose input operands are ready, the task will be dispatched to function units for execution. There are eight fields for each item:

   Busy—Indicates whether the unit is busy or not.
   Fi—Destination variables.
   Fj, *Fk*—Source-variables.
   Qj, *Qk*—Functional units producing sources *Fj*, *Fk*.
   Rj, *Rk*—Flags indicating when *Fj*, *Fk* are ready and not yet read. Set to No after operands are read.
2) *Variable result status*—indicates which functional unit will store values for each variable, if an active task has the variable as its destination. This field is set as blank whenever there are no pending tasks that will write that variable, as shown in Table 3, each variable is assigned with a unique ID.

TABLE 2
Function Unit Status in Task-Scoreboarding

Name	Function Unit Status							
	Busy	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Adder								
IDCT								
AES_ENC								
AES_DEC								

TABLE 3
Variable Table in Task-Scoreboarding

	Variable Results Table						
	0	1	[illegible]	3	.	30	3 1
F U							

3) *Variable Table*—stores the value for each variable. After each task is finished, the results are sent back to variable table through interconnect directly.
4) *Task Partition Module*—in charge of task partition and mapping. Since each task can either run on processor or IP core, thus a good partition method will largely increase the system throughput. For demonstration, we employ a greedy strategy: If there are idle IP cores, the task will be sent to a specific IP; or else, task will be sent to a computing processor. If all the available function units are busy, task must wait until a specific unit is released.
5) *Function Unit Monitor*—monitors and collects the running information of all the function units. The running information helps task partition module map task to certain unit, and also, can achieve load-balance of the hardware.

4.2 Processing Flow for OoO Scheduling

Now let's see how the task sequence is issued and executed. The algorithm is divided into five stages: issue stage, read op stage, partition stage, execution stage, and write result stage. Each task undergoes five steps in executing, as is shown in Table 4.

The whole five stages are similar to instruction level but adding a task partition stage as stage 3. From Table 4, we can examine the steps informally and then see in detail how the scoreboard keeps the necessary information determining when to progress from one step to the next. The five steps are as follows:

1) *Issue*—if a functional unit for the task is free and no other active task has the same destination variable, the scoreboard issues the task to the functional unit and updates its internal data structure. By ensuring that no other active functional unit wants to write its result into the destination variable, we guarantee that WAW hazards cannot be present. If a structural or WAW hazard exists, the task issue will stall, and no further tasks can issue until these hazards are cleared.
2) *Read operands*—the scoreboard monitors the availability of the source operands. If one or more of the operands is not yet available, the scoreboard monitor will wait for the results. A source operand is available if no earlier issued active task is going to write it. When both operands are available, task will be dispatched to certain function units with task partition. The scoreboard resolves RAW hazards dynamically in this step, and tasks may be sent into execution out of order.
3) *Task Partition*—when in the issue stage, the decided function unit is to make sure that there are no structural hazards. However, after read op stage is finished, maybe there are other available function units, which may provides shorter task execution time. Therefore, in this stage a partition strategy is called for task reallocation. If there are other function units available, the task execution time on each function unit will be compared. After the comparison finished, scoreboard will choose the function unit on which current task can finish as early as possible.
4) *Execution*—the functional unit begins execution once receiving operands. When the result is ready, it notifies the scoreboard that it has completed execution. Task distribution and data transfer are both performed through on-chip interconnect.

TABLE 4
Processing Flow of Task-Scoreboarding

Task Status	Wait Until	Action or Bookkeeping
Issue	Not Busy [FU] and not Results [D]	Busy [FU] ←yes; Op [FU]←op; Fi [FU] ←D; Fj [FU] ←S1; Fk [FU] ←S2; Qj←Result [S1]; Qk←Result [S2]; Rj←not Qj; Rk←not Qk; Result [D] ←FU
Read Operands	Rj and Rk	Rj ←No; Rk←No;
Task Partition	! ∀ Busy [FU]	Compare the task execution time and choose FU with min task execution time, then replace the table entries.
Execution Complete	Function unit done	Distribute tasks to function units
Write Results	∀f((Fj [f]≠Fi [FU] or Rj [f] = No) & (Fk [f] ≠Fi [FU] or Rk [f] =No)	∀f(if Qj [f]=FU then Rj [f] ←Yes); ∀f(if Qk [f]=FU then Rk [f] ←Yes); Result [Fi [FU]] ←0;Busy [FU] ←No

5) Based on the hardware interconnect, the execution results are returned through interrupts. One interrupt controller is integrated to detect interrupt request signals from all the interconnect channels. The interrupt handler assigns the variables with results. In our proposed architecture, since results from different tasks may be transferred back at the same time, a first-come-first-serve policy is used to deal with interrupts, and no interrupt preempt is supported.
6) *Write result*—this is the final stage of completing tasks. Once the scoreboard is aware that the functional unit has completed execution, it checks for WAR hazards. The cases of normal result-writing occur when there are no WAR hazards between current task and its predecessor tasks.

4.3 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 service, as is described in Algorithm 1.

Algorithm 1. Task scheduling and mapping algorithm
Input: Generated Task Set T
Output: Servant ID set for each task S

```
1  Foreach t ∈ T do
2  snum= ValidServiceNumber(t.opcode)
                         // get candidate servants
3      switch (snum)
4      case: snum = 0   //no available servants
5              return null;
6      case: snum = 1   //only one available
7              add s to the Servant 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             Select the s with minimum T_overall
                         // Select a target servant
14             add s to the Servant ID set
15     end
```

Each function unit has its own private memory not sharing with other units. The tasks with necessary I/O data are spawned by the messages through hardware interconnects, for example, FSL in the demonstration of this paper. Since the tasks throughout this paper are pure functional, the parameters from multiple tasks are synchronized in the synchronization methods.

Our proposed middleware provides implicit synchronization methods. An implicit synchronization barrier is setup at the end of automatic parallel regions, before the output codes. For example, if the output functions in the main program want to print out the results (e.g., Printf) at the end of an automatic parallel region, it will automatically wait until all the annotated functions are finished. At the time of writing this paper, OoO middleware is allowed to deal with synchronizations among all the functions defined in the function library.

Moreover, when the results are returned through the FSL bus, scheduler processor is running the subsequent tasks, therefore an interrupt signal is raised to stall the main program. For hardware support to the interrupt mechanisms, an interrupt controller is integrated into the hardware platform, which traces the interrupt events for all the FSL links.

5 Prototype and Results

To evaluate the OoO middleware in real hardware, we built a hardware prototype on a state-of-the-art Xilinx Virtex-5 and Zynq FPGA. Due to page limitations here we present the Virtex-5 platform in detail. We use microblaze version 7.20.a (with the clock frequency 125 MHz, local memory of 8 KB, no configurable task or data cache) as scheduling processor. The experimental platform is extended from our previous work in [42]. The prototype system is composed of following components:

1) One scheduling Microblaze processor is integrated as the scheduler to run main program.
2) One Microblaze is employed as computing processor. Software task functions are implemented and packaged in standard C libraries.
3) Hardware IP cores are implemented in HDL (Verilog) and packaged into Xilinx FSL manners. Parts of the EEMBC DENBench benchmarks are used to measure the performance and cost. For each test case, we have transplanted the software benchmark program to FPGA and also implemented a related IP core with RTL implementations. At the time of writing this paper, we have designed five types of hardware IP cores: Adder, IDCT, AES (ENC and DEC), DES (ENC and DEC), and JPEG. For each integrated IP core, software task runtime library is also implemented on microblaze at the same time.
4) Peripherals including UART, Timer and interrupt controller are integrated for debugging through processor Local bus.

5.1 OoO Execution Results

Based on the prototype system, we designed several sample applications to measure the performance, scheduling overheads and hardware cost of the MPSoC system [43]. In this case, we measured the speedup under four situations: RAW, WAW, no hazards, and WAR. In order to evaluate the peak speedup, we define two parameters:

First, the task execution time denotes the entire execution time using in different types of data hazards. In the circumstances of No dependencies, WAW and RAW, the task execution time is configured to the same value (varied from 5k to 100k cycles), while in WAR and WAW, the execution time is configured to different values for heterogeneous computational tasks.

Second, the task scale refers to the total amount of different tasks. In particular, as we use multiple loop iterations to construct the intra-loop and inter-loop data hazards between tasks, the task scale indicates the number of loop iterations. In demonstration, we set the task scale less than 4,096 in all the test cases.

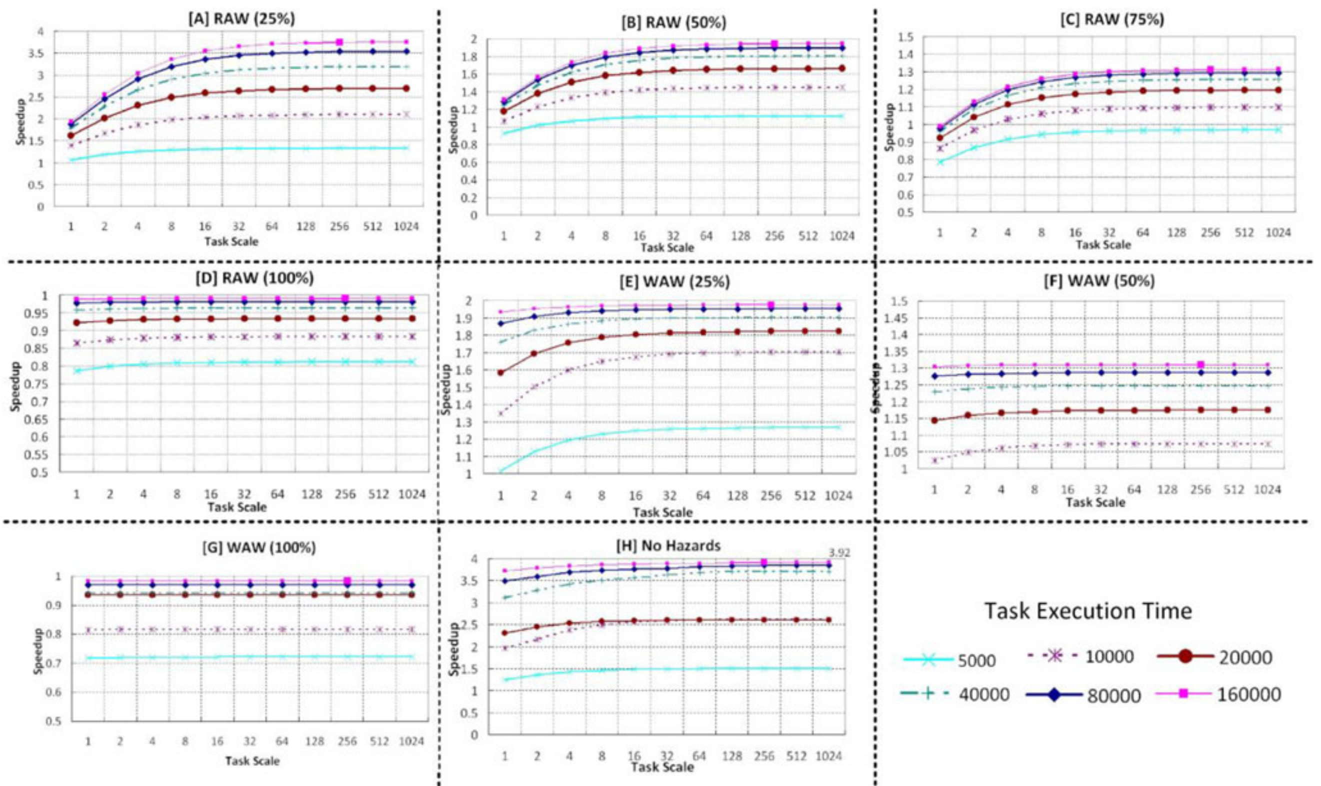


Fig. 6. Experimental results for RAW-WAW-No Hazards, [A] to [D] refer to different rate of RAW hazards, [E]-[G] reflects the WAW hazards, and [H] presents the No hazards application.

5.1.1 RAW Data Hazards

In order to evaluate the speedup in RAW case, execution time of different tasks is configured to the same value (from 5k to 16k cycles). Table 5 lists the task sequences designed to measure the performance for RAW data hazards. The four test cases have different percents of RAW hazards.

Increased from 25 to 100 percent, the theoretical speedup of the task sequences are 4.0×, 2.0×, 1.33× and 1.0× respectively.

For RAW (25 percent) case, the theoretical speedup is the same as No-Hazards. This is because tasks are executed in continuous loops. After loop is unrolled, the first task do_T_adder (a, e) in later loop will run in the meantime of the task do_T_aes_dec(d, h, f) in prior loop, which means the execution time of the adder task will be hidden in the final time. Therefore, the theoretical speedups are the same as the test case with only structural hazards.

The experimental results for RAW hazards are shown in Fig. 6[A] $\sim$ [D]. The Task Execution Time in the legend illustrates the average execution time for the tasks belong to different series. The experimental results show that the maximum speedup for each task sequence is 3.75×, 1.95×, 1.31×, and 0.99× respectively. This means that the Scoreboarding on RAW hazards can achieve 93.81, 97.31, 98.45 and 99.05 percent of the theoretical speedups.

TABLE 5
Task Sequence to Test RAW Data Hazards

RAW (25%)	RAW (50%)
do_T_adder(a,e)	do_T_adder(a,e)
do_T_idct(b,a)	do_T_idct(b,a)
do_T_aes_enc(c,g,f)	do_T_aes_enc(c,g,b)
do_T_aes_dec(d,h,f)	do_T_aes_dec(d,h,f)
RAW (75%)	RAW(100%)
do_T_adder(5,e)	do_T_adder(a,e)
do_T_idct(b,a)	do_T_idct(b,a)
do_T_aes_enc(c,g,b)	do_T_aes_enc(c,g,b)
do_T_aes_dec(d,h,c)	do_T_aes_dec(e,h,c)

5.1.2 WAW Data Hazards

Table 6 lists the task sequences designed to measure the performances for WAW hazards. Increased from 25 to 100percent, the theoretical speedup of the three task sequences are calculated in (1) to (3), respectively:

TABLE 6
Task Sequence to Test WAW Data Hazards

WAW (25%)	WAW (50%)	WAW (100%)
do_T_adder(a,c)	do_T_adder(a,c)	do_T_adder(a,c)
do_T_idct(a,d)	do_T_idct(a,d)	do_T_idct(a,d)
do_T_aes_enc(g,e,f)	do_T_aes_enc(a,e,f)	do_T_aes_enc(a,e,f)
do_T_aes_dec(b,d,f)	do_T_aes_dec(g,h,b)	do_T_aes_dec(a,e,g)

TABLE 7
Task Sequence to Test No Data Hazards

No Data Hazards
do_T_adder(a,e)
do_T_idct(b,f)
do_T_aes_enc(c,g,h)
do_T_aes_dec(d,g,i)

$$S1 = (T_{Adder} + T_{IDCT} + T_{enc} + T_{dec})/(T_{Adder} + T_{IDCT}) = 2.0, \quad (1)$$

$$S2 = (T_{Adder} + T_{IDCT} + T_{enc} + T_{dec})/(T_{Adder} + T_{IDCT} + T_{enc}) = 1.33, \quad (2)$$

$$S3 = (T_{Adder} + T_{IDCT} + T_{enc} + T_{dec})/(T_{Adder} + T_{IDCT} + T_{enc} + T_{dec}) = 1.0. \quad (3)$$

The experimental results for WAW data hazards are shown in Fig. 6[E]∼[G]. The maximum experimental speedups for each task sequence are 1.97×, 1.31×, and 0.98×, respectively. It means the Scoreboarding on WAW hazards can reach 98.70, 98.25 and 98.46 percent of the theoretical peak speedups.

5.1.3 No Data Hazards

The no-hazard situation is more straightforward than other types of data hazards. Therefore we choose one test case, as is shown in Table 7.

Fig. 6[H] shows the experimental speedup for No-hazard case. Since the four IP cores have same execution time, the theoretical maximum speedup is 4.0×. However, because of the software scheduling and communication overheads between different processors, the experimental results cannot reach the maximum speedup, especially when with smaller and less tasks. From the figure, we can see that when the task scale and running time are large enough, the experimental maximum speedup can reach Speedup = 3.922×, which is 98.04 percent of ideal value.

TABLE 8
An Example Task Sequence

ID	Request	Type	Start	Finish
1	do_T_adder(a,c)	H	0	100000
2	do_T_idct(b,a)	H	100000	150000
3	do_T_adder(j,i)	S	0	150000
4	do_T_aes (d,e,f)	H	0	25000
5	do_T_aes (h,e,d)	H	25000	50000
6	do_T_des (e,e,g)	H	0	12500
7	do_T_adder(a,c)	H	100000	200000
8	do_T_des(h,e,g)	H	12500	25000
9	do_T_des(h,e,a)	H	200000	212500
10	do_T_adder(f,e)	S	150000	300000
11	do_T_idct(b,a)	H	200000	250000

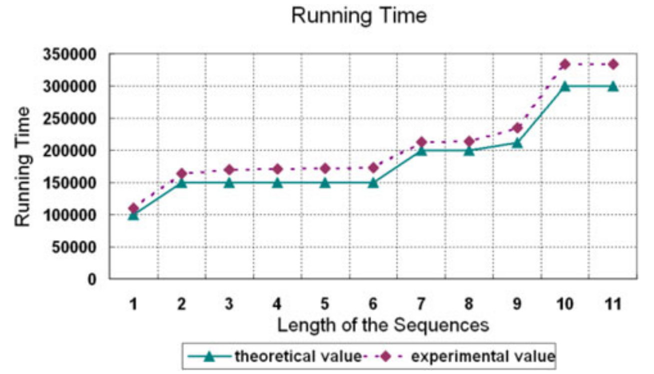


Fig. 7. Experimental results of the Example Task Sequence, the x-axis refers to the length of sequences listed in Table 10, while the y-axis is the running time of the application.

5.2 Example Results and Discussion

Based on the prototype, we designed several example test cases using the specific functions of the implemented IP cores. Generally, since the platform integrates both computing processor and hardware IP cores, each application can either run on microblaze processor with software or in specific IP in hardware. In this section, we measure speedups of two typical task sequences through partitioning the test tasks.

A task sequence with 11 tasks is listed in Table 8. Type indicates whether this task runs on GPP (S) or IP core (H). The last two columns refer to the ideal start and finish time for current task, taking no account of the scheduling and transfer costs.

Fig. 7 depicts the comparison between theoretical and experimental results of the task sequence. The curve of experimental value is consistent with theoretical value, but slightly larger. The gap between them stands for the scheduling and communication overheads. The average of overheads is less than 4.9 percent of task running time itself, which proves that the scheduling overhead of the scheduling scheme is quite low.

Fig. 7 also depicts the OoO task execution architecture support. Tasks No. 2∼6 complete OoO, which leads to the running time of the task sequences (No.1∼2) and (No.1∼6) are the same. Similarly, task No.7 and 8, No.10 and 11 also finish OoO.

5.3 Results of Adaptive Task Mapping

In order to verify the adaptive hardware/software task mapping scheme, we design a task sequence in Table 9. In this case, we introduce task mapping mechanism to operate hardware/software collaboration. For these 11 tasks, task No.4 and No.9 are dispatched to computing processor for execution. This is because that when the two tasks are issued, target IP cores are busy. Thus, scoreboard chooses software computing processors as target function units.

Fig. 8 gives the comparison between theoretical and experimental results of the task sequences in Table 9. Similarly, the scheduling and communication overheads are less than 3.3 percent of task running time. Moreover, in this case, tasks have been partitioned between hardware and software. A computing processor has been integrated to

TABLE 9
Task Sequence to Test Adaptive Mapping Scheme

ID	Request	Type	Start Time	Finish Time
1	do_Task_adder(a,b);	H	0	100000
2	do_Task_idct(c,a);	H	100000	150000
3	do_Task_adder(d,e);	H	100000	200000
4	do_Task_adder(f,c);	S	150000	300000
5	do_Task_idct(g,a);	H	150000	200000
6	do_Task_aes_enc(h,d,g)	H	200000	225000
7	do_Task_adder(f,c);	H	300000	400000
8	do_Task_aes_dec(d,d,f);	H	400000	412500
9	do_Task_aes_dec(h,d,a)	S	400000	1000000
10	do_Task_adder(i,d);	H	412500	512500
11	do_Task_adder(i,d);	H	521500	612500

accelerate tasks execution when all the hardware IP cores are busy. Task No.4 and No.9 are distributed to computing processor for execution, which also demonstrated that our HW/SW platform can be used to verify alternative scheduling and mapping methods.

Meanwhile, as in this test case, task No.9 is distributed to computing processor for execution, which demonstrates that the current partition method is not an optimal choice for the entire task sequence. Alternately, if task No.9 waits until task No.8 finish on AES_DEC core, instead of run immediately on computing processor, the entire task running time can be further reduced.

5.4 Performance versus Task SuperScalar

One of the most related works of our Task-Scoreboarding scheme is the Task Superscalar, therefore, we also applied renaming schemes employed in Task Superscalar to FPGA for comparison, which is illustrated in Fig. 9. We use the generated 11 applications in Table 8, and then evaluated both TaskSs and Task-Scoreboarding algorithm respectively. Due to the scheduling overheads, the performance of both approaches gets larger than the ideal scenario, which is normalized as 1.0×. It can be derived that the TaskSs always achieves higher scheduling overheads than our Task-Scoreboarding approach, from 3 (T7) to 14 percent (T5). What's more, our approach can get an average overhead less than 5 percent. Therefore our approach obtains much smaller overheads than TaskSs.

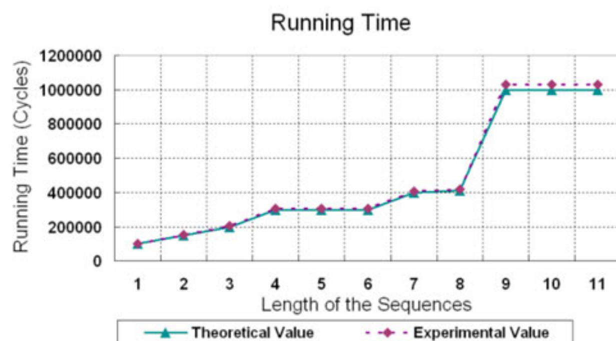


Fig. 8. Experimental results of Adaptive Task Mapping.

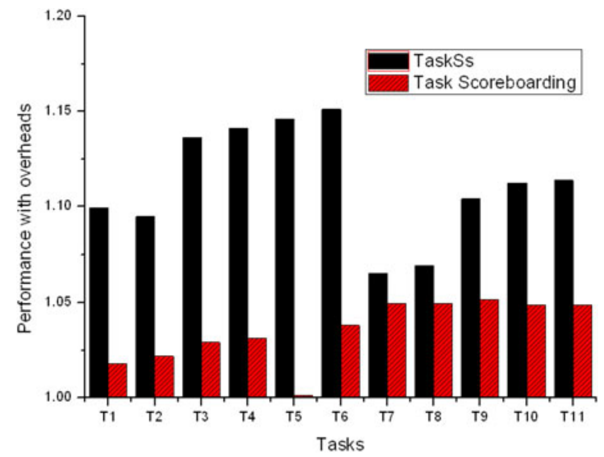


Fig. 9. Performance comparison between TaskSs and Task-Scoreboarding, we use the same benchmarks in Table 8.

5.5 JPEG Case Study

In order to evaluate the Task-Level Scoreboarding mechanism in real applications, we have implemented the Task Scoreboarding algorithm with the JPEG encoding applications, as is illustrated in Fig. 10. We use Lena picture at different pixel sizes for JPEG encoding procedure. The picture is decomposed to plenty of 8×8 blocks, which are compressed by the main body of the algorithm as a normalized unit. An 8×8 R-G-B block is read from the origin bmp file first and begins to be converted to the Y-Cr-Cb colour space (CC). Then, each vector in the <Y, Cr, Cb> tuple is handed over for a two dimensional discrete cosine transform (DCT-2D). Next, all the data in the unit are normalized (Quant) for further encoding. At last, Zig-Zag and Huffman (ZZ/Huffman) encoding algorithm is used to compress the block data into the final bit stream in the end of the iteration.

We first profiled the JPEG applications to identify the parts for different phases, as illustrated in the Ratio term in the legend of Fig. 11. Due to that the ZZ/Huffman phase takes only 2.6 percent of the total execution time, thereby we have not implemented this part as hardware yet. Meanwhile, the DCT-2D phase is regarded as the major bottleneck of all the four phases, as it takes 73.8 percent of the entire execution time. In contrast, the other two CC and Quant steps take 20.8 and 2.8 percent of the total execution time respectively.

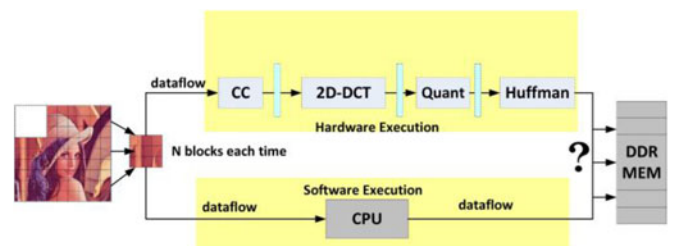


Fig. 10. Reconfigurability study. Using JPEG applications to leverage hardware execution with software execution.

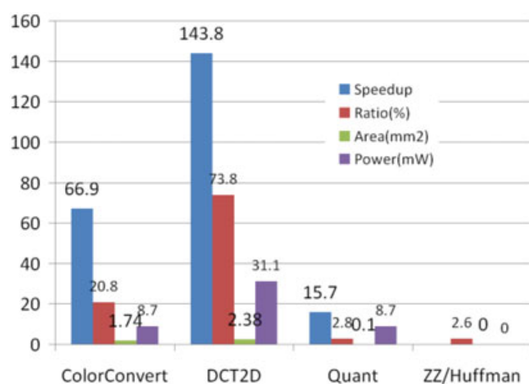


Fig. 11. Profiling results of the four stages in JPEG application.

Corresponding to the percentage of execution time for different phases, the speedup of the hardware implementation for different functionality modules are illustrated as the Speedup term in the legend of Fig. 11. The DCT-2D hardware module has been optimized to achieve as high 143.8×, while the CC and Quant modules are 66.9× and 15.7× respectively.

Along with the achieved speedup, the area and power consumption are considered as the two most significant metrics for evaluation. The DCT-2D occupies most area and power of the four parts, which are 2.38 mm^2 and 31.1 mW, using Xilinx ISE synthesis tools and XPower Analyzer. In Comparison, the hardware CC module takes 1.74 mm^2 and 8.7 mW, while Quant module takes 0.1 mm^2 and 8.7 mW respectively. Fig. 12 illustrates the differences between the experimental and the estimated metrics. The x-axis in Fig. 11 indicates the specific hardware configuration, while the y-axis represents the differential percentage of speedup, power, and resource cost and core efficiency. We can learn from Fig. 12 that the actual architectures come up to a speedup of 26.01× for hybrid systems and a speedup of 3.83× for homogeneous systems.

For homogeneous architectures, the difference in resource cost is less than 5.6 percent and that in power consumption is less than 6.8 percent. Besides, both speedup and core efficiency difference are less than 2.9 percent. For hybrid architectures, all items have insignificant difference except for the performance metrics with the hardware configuration at 1 MB + 1 CC + 1 DCT and 1 MB + 1 CC + 1 DCT + 1 Quant. Both of them have a difference up to 14.2 percent. Considering that Quant and ZZ/Huffman steps in the JPEG 8 × 8 block compression only take a small ratio (2.6 percent depicted in Fig. 10), therefore any bus delay or communication overheads will have a scaled influence on the system speedup.

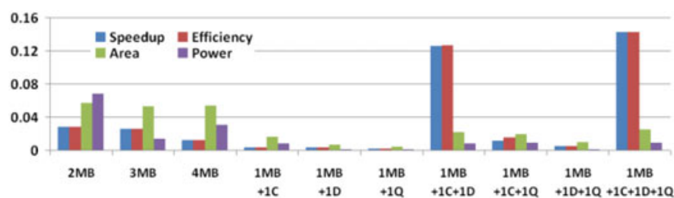


Fig. 12. Difference between experimental and estimated metrics on speedup, area, efficiency, and power.

TABLE 10
Hardware Cost of the Entire Hardware System

Resource	Used/Available	Utilization
Number of Slice Registers	5238/28800	18%
Number of Slice LUTS	19209/28800	66%
LUTS Used as Memory:	534 out of 7680	6%
Number of External IOBs	4 out of 480	1%
Number of BUFGs	3 out of 32	9%
Number of DSP48Es	31 out of 48	64%

5.6 Hardware Costs

The whole system is implemented in Xilinx Virtex5 LX50T FPGA, including one Microblaze processor, one computing processor, following different IP blocks: Adder, IDCT, AES (ENC and DEC), DES (ENC and DEC), JPEG, memory and peripheral modules.

Table 10 summarizes the hardware cost within single FPGA. The whole system takes 5,238 slice Registers and 19,209 Slice LUTs. Considering the resources supplied in FPGA, the hardware cost is acceptable for the proposed architecture. By looking further into the synthesis report, most of the resources are occupied by microblaze processors and hardware IP cores, we can get that the scheduling Microblaze processor takes 1,650 LUTs and 1,489 FFs, while the hardware costs of the IP cores distinguish from each other. For example, JPEG function including four phases: Color Conversion (CC), 2D-DCT, Zig-Zig reordering/Quantization (ZZ/Q) and Huffman encoding, in which the 2D-DCT was identified as the critical section.

6 Conclusions and Future Work

In this paper, we have proposed an architecture support for OoO task execution with middleware on FPGA based MPSoCs. The middleware can automatically identify the parallel region and eliminate the data dependencies with renaming techniques. An adaptive mapping scheme allows the scheduler spawn tasks to a suitable function unit even when the hardware reconfigured. In order to allow OoO task execution, we have proposed Task-Scoreboarding, a data hazards detecting engine for OoO task execution. Regarding processors and IP cores as function units, Task-Scoreboarding treats tasks as abstract instructions. It can analyze inter-task data dependencies at runtime and issue tasks to heterogeneous function units automatically. Experimental results on the state-of-art FPGA platform demonstrate that our middleware is flexible to support different IP cores with acceptable hardware costs. The Task-Scoreboarding can achieve more than 95 percent of the ideal peak speedup. In particular, the Task-Scoreboarding algorithm costs more than 10 percent overheads than state-of-the-art approaches.

The experimental results are inspiring but there is a lot of work left in the future. First, we plan to extend the renaming technologies to dynamic partial reconfigurable situations where processors and IP cores can be adaptive to fit in different applications at runtime. Second, we are implementing the module in RTL which is more capable to handle smaller tasks. In the meantime, new annotations in the

programming paradigms are taken into consideration by the source to source compiler to identify both the library tasks and the general purpose software tasks. Finally, the improvement of the synchronization mechanism using real-time operating system supports is another promising direction for heterogeneous MPSoC research paradigms.

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), and the Strategic Priority Research Program of CAS (No. XDA06010403). The authors deeply appreciate many reviewers for their insightful comments and suggestions. Professor Xi Li is the corresponding author.

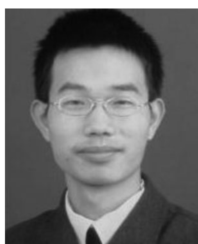
REFERENCES

[1] S. Singh, "Computing without processors," *Commun. ACM*, vol. 54, no. 8, pp. 46–54, 2011.
[2] S. Borkar and A. A. Chien, "The future of microprocessors," *Commun. ACM*, vol. 54, no. 5, pp. 67–77, 2011.
[3] O. A. R. Board, OpenMP C and C++ application program interface, version 1.0. 1998 [Online]. Available: http://www.openmp.org.
[4] K. Group. (2010). OpenCL. [Online]. Available: http://www.khronos.org/opencl.
[5] D. B. Robert, F. J. Christopher, C. K. Bradley, E. L. Charles, H. R. Keith, and Z. Yuli, "Cilk: An efficient multithreaded runtime system," in *Proc. 5th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 1995, pp. 207–216.
[6] 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.
[7] 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 Perf. Comput. Cell Broadband Engine*, vol. 17, nos. 1/2, pp. 77–95, 2009.
[8] G. Gupta and G. S. Sohi, "Dataflow execution of sequential imperative programs on multicore architectures," in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2011, pp. 59–70.
[9] 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.
[10] 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. (Sel. Papers)*, 1998, pp. 213–223.
[11] 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.
[12] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar processors," in *Proc. Int. Symp. Comput. Archit*, 1995, pp. 414–425.
[13] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith, "Trace processors," in *Proc. Int. Symp. Microarchit.*, 1997, pp. 138–148.
[14] 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.
[15] 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.
[16] 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.
[17] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "WaveScalar," in *Proc. Int. Symp. Microarchit.* 2003, p. 291.
[18] 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.
[19] D. Y. Deng, D. Lo, G. Malysa, S. Schneider, and G. E. Suh, "Task superscalar: An out-of-order task pipeline," in *Proc. 43rd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2010, pp. 89–100.
[20] B. Mladen and N. Tim, "A distributed, simultaneously multithreaded (SMT) processor with clustered scheduling windows for scalable DSP performance," *J. Signal Process. Syst. - Special Issue: Embedded Comput. Syst. DSP*, vol. 50, pp. 201–229, 2008.
[21] B. Mladen and N. Tim, "Flexible and efficient instruction-grained run-time monitoring using on-chip reconfigurable fabric," in *Proc. 43rd Annu. IEEE/ACM Int. Symp. Microarchit.* 2010, pp. 137–148.
[22] 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.
[23] A. Ghuloum, T. Smith, G. Wu, X. Zhou, J. Fang, P. Guo, B. So, M. Rajagopalan, Y. Chen, and C. B., "Future-proof data parallel algorithms and software on intel multi-core architecture," *Intel Technol. J.*, vol. 11, no. 4, pp. 333–348, 2007.
[24] K. Knobe, "Ease of use with concurrent collections (CnC)," in *Proc. First USENIX Conf. Hot Topics Parallelism*, 2009, p. 17.
[25] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao, "The Chimaera reconfigurable functional unit," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 12, no. 2, pp. 206–217, 2004.
[26] J. R. Hauser and J. Wawrzynek, "Garp: A MIPS processor with a reconfigurable coprocessor," in *Proc. 5th Annu. IEEE Symp. FPGAs Custom Comput. Mach.* 1997, pp. 12–27.
[27] R. D. Wittig and P. Chow, "OneChip: An FPGA processor with reconfigurable logic," in *Proc. IEEE Symp. FPGAs for Custom Comput. Mach.*, 1995, pp. 126–135.
[28] R. Razdan and M. D. Smith, "A high-performance microarchitecture with hardware-programmable functional units," in *Proc. 27th Annu. Int. Symp. Microarchit.*, 1994, pp. 172–180.
[29] H.-S. Kim, A. K. Somani, and A. Tyagi, "A reconfigurable multi-function computing cache architecture," in *Proc. 8th ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, 2000, pp. 85–94.
[30] M. A. Watkins and D. H. Albonesi, "ReMAP: A Reconfigurable Heterogeneous Multicore Architecture," in *Proc. 43rd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2010, pp. 497–508.
[31] 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.
[32] T. Givargis and F. Vahid, "Platune: A tuning framework for system-on-a-chip platforms," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 21, no. 11, pp. 1317–1327, Nov. 2002.
[33] 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.
[34] N. E. J. Sheng Ma and Z. Wang, "DBAR: An efficient routing algorithm to support multiple concurrent applications in networks-on-chip," in *Proc. 38th Annu. Int. Symp. Comput. Archit.*, 2011, pp. 413–424.
[35] M. Mehrara, P.-C. Hsu, M. Samadi, and S. Mahlke, "Dynamic parallelization of JavaScript applications using an ultra-lightweight speculation mechanism," in *Proc. IEEE 17th Int. Symp. High Perform. Comput. Archit.*, 2011, pp. 87–98.
[36] A. Kotha, K. Anand, M. Smithson, G. Yellareddy, and R. Barua, "Automatic parallelization in a binary rewriter," in *Proc. 43rd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2010, pp. 547–557.
[37] J. Suh and M. Dubois, "Dynamic MIPS rate stabilization in out-of-order processors[J]," *SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 46–56, 2009.
[38] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM J. Res. Develop.*, vol. 11, no. 1, pp. 25–33, 1967.
[39] F. Karim, A. Mellan, B. Stramm, A. Nguyen, T. Abdelrahman, and U. Aydonat, "The hyperprocessor: A template system-on-chip architecture for embedded multimedia applications," in *Proc. Workshop Appl. Specific Processors*, 2003, pp. 66–73.

[40] Xilinx. Inc. (2009). Fast simplex link (FSL) specification. [Online] Available: http://www.xilinx.com/products/ipcenter/FSL.htm.
[41] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 4th ed. San Mateo, CA, USA: Morgan Kaufman, 2007.
[42] 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.
[43] C. Wang, P. Chen, X. Li, X. Feng, J. Zhang, and X. Zhou. "Detecting Data Hazards in Multi-Processor System-on-Chips on FPGA," *IPDPS Workshops*, pp. 282–287, 2012.

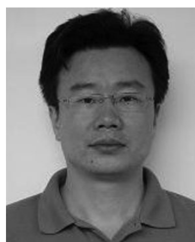
Chao Wang received the BS and PhD degrees from the University of Science and Technology of China, in 2006 and 2011, respectively, both in computer science. He is a postdoc researcher in Embedded System Lab of the University of Science and Technology of China. His research interests focus on multicore and reconfigurable computing. He has authored more than 30 publications and patents, including ACM TACO and FPGA conferences. He serves as a TPC member and a reviewer for more than 20 conferences/journals. He is a member of the IEEE, ACM, and CCF.

Junneng Zhang received the BS degree of 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 in Suzhou Institute of University of Science and Technology of China, Suzhou, China. His research interests focus on multiprocessor system on chip, reconfigurable computing and big data oriented heterogeneous platforms. He is a student member of the IEEE and China Computer Federation (CCF).



Peng Chen received the BS degree in computer science from the University of Science and Technology of China, in 2010. He is currently working toward the PhD degree in the Embedded System Lab in Suzhou Institute of University of Science and Technology of China, Suzhou, China. His research interests focus on multiprocessor system on chip, reconfigurable computing, and big data oriented heterogeneous platforms. He is a student member of the IEEE and China Computer Federation (CCF).

Yunji Chen graduated from the Special Class for the Gifted Young, University of Science and Technology of China, Hefei, China, in 2002. He received the PhD degree in computer science from the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, China, in 2007. He is currently a professor at ICT. His research interests include computer architecture and machine learning. He has authored or coauthored one book and more than 50 papers in these areas.



Xuehai Zhou is the executive dean of 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. 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 ACM and IEEE, a senior member of the CCF.

Ray C.C. Cheung (M'07) received the BEng and MPhil degrees in computer engineering and computer science and engineering from the Chinese University of Hong Kong (CUHK), Hong Kong, in 1999 and 2001, respectively, and the PhD degree and DIC in computing from Imperial College London, London, United Kingdom, in 2007. After completing the PhD degree, he received the Hong Kong Croucher Foundation Fellowship for his postdoctoral study in the Electrical Engineering Department, University of California, Los Angeles (UCLA). In 2009, he was a visiting research fellow in the Department of Electrical Engineering, Princeton University, Princeton, NJ. He is currently an assistant professor in the Department of Electronic Engineering, City University of Hong Kong (CityU). He is the author of more than 30 journal papers and more than 40 conference papers. His research team, CityU Architecture Lab for Arithmetic and Security (CALAS), focuses on the following research topics: reconfigurable trusted computing, applied cryptography, and high-performance biomedical VLSI designs.

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