# MP-Tomasulo: A Dependency-Aware Automatic Parallel Execution Engine for Sequential Programs

CHAO WANG, University of Science and Technology of China
XI LI and JUNNENG ZHANG, Suzhou Institute for University of Science and Technology of China
XUEHAI ZHOU, University of Science and Technology of China
XIAONING NIE, Intel

This article presents MP-Tomasulo, a dependency-aware automatic parallel task execution engine for sequential programs. Applying the instruction-level Tomasulo algorithm to MPSoC environments, MP-Tomasulo detects and eliminates Write-After-Write (WAW) and Write-After-Read (WAR) inter-task dependencies in the dataflow execution, therefore to operate out-of-order task execution on heterogeneous units. We implemented the prototype system within a single FPGA. Experimental results on EEMBC applications demonstrate that MP-Tomasulo can execute the tasks out-of-order to achieve as high as 93.6% to 97.6% of ideal peak speedup. A comparative study against a state-of-the-art dataflow execution scheme is illustrated with a classic JPEG application. The promising results show MP-Tomasulo enables programmers to uncover more task-level parallelism on heterogeneous systems, as well as to ease the burden of programmers.

Categories and Subject Descriptors: C.1.4 [**Processor Architecture**]: Parallel Architectures; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*

General Terms: Performance, Design

Additional Key Words and Phrases: Automatic parallelization, data dependency, out-of-order execution

**ACM Reference Format:**
Wang, C., Li, X., Zhang, J., Zhou, X., and Nie, X. 2013. MP-Tomasulo: A dependency-aware automatic parallel execution engine for sequential programs. *ACM Trans. Archit. Code Optim.* 10, 2, Article 9 (May 2013), 26 pages.
DOI: http://dx.doi.org/10.1145/2459316.2459320



## 1. INTRODUCTION

The past decades have witnessed a tremendous invasion of MultiProcessor System on Chip (MPSoC), especially in high-performance parallel computing domains. As more processors are being increasingly integrated into a single chip, it is possible to bring higher computation abilities to heterogeneous platforms for various applications. In particular, the Field Programming Gate Array (FPGA)-based MPSoC and Graphic Processing Unit (GPU)-based heterogeneous architectures have been regarded as the promising future microprocessor design paradigms [Borkar and Chien 2011]. Compared to GPU architectures, FPGA can provide a more flexible framework to construct prototypes for different applications efficiently.

This work was supported by the National Science Foundation of China under grants 61272131 and 61202053, China Postdoctoral Science Foundation grant BH0110000014, Fundamental Research Funds for the Central Universities, and Jiangsu Provincial Natural Science Foundation grant SBK201240198.
Authors' addresses: C. Wang, X. Li (corresponding author), J. Zhang, and X. Zhou, School of Computer Science, University of Science and Technology of China; X. Nie, Intel, Munich, Germany.

DOI: http://dx.doi.org/10.1145/2459316.2459320

However, task partitioning and scheduling approaches on heterogeneous MPSoC platforms have encountered serious challenges, especially with inter-task dependencies, including structural dependencies, Read-After-Write, Write-After-Write, and Write-After-Read (denoted as RAW, WAW and WAR) data dependencies. Different tasks using same source or destination parameters may cause tasks to run in sequence, which largely confines the task-level parallelism.

To address the data dependencies (or named data hazards) problem at instruction level, there are several traditional hazards detection and elimination solutions [Patt et al. 1985], such as Scoreboarding and Tomasulo [Tomasulo 1967]. Both algorithms provide Out-of-Order (OoO) instruction execution engines when there are sufficient computing resources.

Meanwhile, parallel programming models have brought an alternative venue to enhance task-level parallelism for MPSoC platforms. However, a major drawback of the state-of-the-art programming models like OpenMP and CUDA is that programmers are required to handle the task assignments with data dependencies manually. In contrast, an adaptation of Instruction-Level Parallelism (ILP) scheduling algorithm to Task-Level Parallelism (TLP) provides a new insight to utilize the MPSoC platform effectively. Current cutting-edge studies such as Task Superscalar [Etsion et al. 2010] and Gupta and Sohi [2011] explore new research directions into managing heterogeneous CMPs at a higher abstraction level. However, both approaches address the out-of-order task execution on a dedicated architecture without considerations for FPGA-based MPSoC platforms. Therefore, in this article, we focus on offering a middleware support on FPGA, which treats tasks as abstract instructions.

This article proposes MP-Tomasulo, a task-level out-of-order execution model for sequential programs on FPGA-based MPSoC. The major contributions are listed as follows.

(1) We bring an out-of-order scheduling scheme to a system-on-chip architecture, and build a prototype MPSoC computing platform in the FPGA architecture with General-Purpose Processors (GPP) and Intellectual Property (IP) cores. Based on the platform, a software/hardware codesign flow and a flexible programming model with compiler support is presented.
(2) We propose an MP-Tomasulo engine which applies the instruction-level Tomasulo algorithm to an MPSoC scenario. MP-Tomasulo can eliminate WAW and WAR data dependencies automatically to operate out-of-order execution at task level.
(3) We built a prototype system on a real FPGA hardware platform. Speedups of different types of inter-task data dependencies are studied. A comparative study against state-of-the-art dataflow execution using JPEG applications is also presented.

This article is decomposed as follows: Section 2 starts with the scope and motivation of this article, and then introduces the state-of-the-art of out-of-order task execution methods with parallel programming models. We show their characteristics and their limitations with respect to the future high-end massively parallel embedded applications requirements. Section 3 presents the hardware architecture and software/hardware design flow. Then, in Section 4, we introduce the compiler-related support, including the programming models and structure APIs. Section 5 details the MP-Tomasulo architecture and execution model. We will illustrate its applicative system environment, its data structures, and processing flows. The functionalities and interpretabilities of the hardware components are presented in detail. We also summarize the essential differences and limitations between instruction-level Tomasulo and MP-Tomasulo. Section 6 proposes the implementation of the hardware prototype on FPGA. Test cases from EEMBC benchmarks are presented and experimental results are analyzed in that section. A JPEG application is employed to evaluate the

Table I. RAW, WAW and WAR Dependencies

| Dependencies | Code Snippets | Task | Output Parameter | Input Parameter(s) |
|---|---|---|---|---|
| RAW | do_T_adder (**a** , b ) | Adder | a | b |
| | do_T_idct( c , **a** ) | IDCT | c | a |
| WAW | do_T_aes_dec(**a**,b,c) | AES_DEC | a | b, c |
| | do_T_aes_enc(**a**,b,e) | AES_ENC | a | d, e |
| | do_T_idct(**a**,f) | IDCT | a | f |
| WAR | do_T_adder(a,b); | Adder | a | b |
| | do_T_adder(c,**d**,a); | Adder | d | d, a |
| | do_T_idct(**d**,e); | IDCT | d | e |

MP-Tomasulo and the state-of-the-art parallel dataflow-based approach. Finally, we conclude the article and explain future works in Section 7.

## 2. MOTIVATION AND RELATED WORK

### 2.1. Scope and Design Goals

As the major contribution of this article is to propose a scheduling engine for out-of-order task execution, in this section the data-dependency issues (including RAW, WAW, and WAR) are first observed.

Before the OoO perspective is presented, we define the following terms used throughout this article first.

*Tasks.* Tasks refer to dynamic instances created when invoking special application programming interfaces. Moreover, tasks are regarded as functional abstract instructions, and each IP core is treated as a dedicated functional unit for a specific hardware task.

*Function units.* All the computing processors and IP cores are considered as structural function units to run tasks in parallel. All these function units are connected to the scheduler processor through on-chip interconnect.

*Parameters.* Parameters refer to the input and output operands. Through looking into the parameters for different tasks, inter-task data dependencies among tasks can be uncovered.

To illustrate this problem, we assume each task is composed of task name, source, and destination parameters. As the tasks are treated as abstract instructions, the data dependencies problem can also happen for task level. Table I presents the three kinds of inter-task data dependencies with code snippets. For the RAW situation, two tasks are illustrated in the example, do_T_adder and do_T_idct. Of the two tasks, assume that the sources parameter of IDCT (a) is the same as the destination parameter of AES_DEC task (a), therefore the IDCT task shall wait for the parameter (a) until the adder task finishes its execution and releases (a). Likewise, for a WAW dependency, we model three tasks, the first two of which have three parameters. Note only the first parameter is the destination while the other parameters are marked as sources. Consequently WAW dependency appears since all the three tasks have the same output operands, making the task executed in order. Finally, WAR hazards must also be detected to avoid violating antidependencies.

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 purposes. Of these two methods, Tomasulo is more widely used because it can eliminate WAW and WAR

dependencies, while Scoreboarding only solves it by running tasks in sequence. Therefore, we choose Tomasulo instead of Scoreboarding at task level.

The basic concept of instruction-level Tomasulo is that whenever all input operands are prepared, the instruction will be directly offloaded to an arithmetic unit. It is not required to wait until all its previous instructions have finished execution, or are even issued. At the end of execution, results are committed in order for synchronization. Therefore oriented from this motivation, this article is to present an MP-Tomasulo middleware support on MPSoC targeting out-of-order task-level parallelization.

### 2.2. Related Work

Data dependencies and the synchronization problem has already posed a significant challenge in parallelism. Traditional algorithms, such as Scoreboarding and Tomasulo [Tomasulo 1967], explore ILP with multiple arithmetic units, which can dynamically schedule the instructions for out-of-order execution.

Meanwhile, with the increasing popularity of the MPSoC platform, parallelism is shifting from instruction level to task level. There are already some creditable FPGA-based research platforms, such as RAMP [Wawrzynek et al. 2007], Platune [Givargis and Vahid 2002], and MOLEN [Kuzmanov et al. 2004]. 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 the processor are designed to increase TLP with coarser-grained parallelism, such as MLCA [Faraydon et al. 2004], Multiscalar [Sohi et al. 1995], trace processors [Rotenberg et al. 1997], IBM CELL [Kahle et al. 2005], RAW processor [Taylor et al. 2002], Intel Terascale, and Hydra CMP [Hammond et al. 2000]. 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 [Sankaralingam et al. 2006] and WaveScalar [Swanson et al. 2003] combine both static and dynamic dataflow analysis in order to exploit more parallelism. 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.

In contrast, parallel programming models are quite popular to improve Task-Level Parallelism (TLP) in MPSoC, such as OpenMP, MPI, Intel's TBB, CUDA, OpenCL [KhronosGroup 2010] and Cilk [Blumofe et al. 1995]. However, a major drawback of these programming models is letting the programmer handle task assignments with data dependencies manually, which increases the programmer burden of synchronization and task scheduling.

Meanwhile, with the growing popularity of task-based programming models, task-level scheduling methods are also motivated to operate high-level parallelism, such as Kotha et al. [2010] and Suh and Dubois [2009]. Task Superscalar [Etsion et al. 2010] proposes an abstraction of out-of-order superscalar pipelines that use the processor as function units. Mladen and Niggemeier [2008] provide a modified version of the Tomasulo scheme for DSP-based processor architectures to perform out-of-order task execution. FlexCore [Deng et al. 2010] presents a hybrid process architecture using an on-chip reconfigurable fabric (FPGA) to support runtime monitoring and bookkeeping techniques. Hyperprocessor [Karim et al. 2003] manages global dependencies using a universal register file. However, most of the works need hardware support, which causes insufficient flexibility across different architectures. Recently, Sanchez et al. [2011] have presented a dynamic fine grained scheduling scheme of pipeline parallelism. The work is based on multi-core SMP to achieve load balancing. EnCore project [Encore 2012] provides a new insight in programmable manycore systems. Some other parallelization engines like Carbon [Kumar et al. 2007] and Limberg et al. [2009] have

Table II. Strengths and Weaknesses of MP-Tomasulo

| **Classifications** | **Typical Approaches** | **Our Strength** | **Our Weakness** |
|---|---|---|---|
| General parallel programming model **(No OoO)** | OpenMP, Intel's TBB, Ct, CnC MapReduce, OpenCL, Cilk | No burden to programmers | Not applicable to large scale CMP processors |
| Specific parallel programming model **(With OoO)** | StarSs & CellSs | Extend from simulated CellBE architecture to real FPGA based systems | Limited by the hardware resources provided by FPGA |
| Coarse Grained Task Level Parallelism **(With OoO)** | MLCA, Multiscalar, WaveScalar, Trace, CELLRAW, Hydra, TRIPS | Automatic OoO is supported on FPGA | Not applicable to large scale CMP processors |
| Dataflow Based OoO Execution Model **(With OoO)** | TaskSuperscalar & Dataflow | Extend from simulators to real FPGA based systems | Limited by the hardware resources provided by FPGA |
| | DSP-Tomasulo | Extend from DSP to real FPGA based systems | |
| | FlexCore | Automatic OoO is supported on FPGA | |

figured out the fine grained task-level parallelization in CMP and software-defined radio application fields, respectively.

Of the state-of-the-art research, Task Superscalar is a recent sequential program-based framework that achieves function-level parallel execution. It requires the user to identify function parameters on which dependences may occur, using pragma directives. Task Superscalar builds a dynamic task-flow graph in prior to parallel execution based on memory locations. Task Superscalar uses a master thread to farm out work to other threads. It renames data, potentially incurring high memory usage. Furthermore, it targets the CellBE architecture and is implemented in simulation instead of FPGA-based real hardware implementation.

Furthermore, Serialization Sets (SS) [Allen et al. 2009] is a sequential program-based, determinate model that dynamically maps dependent (independent) computations into a common (different) "serializer". Computations within a serializer are serialized while those from different serializers are parallelized. Furthermore, based on SS, Gupta and Sohi [2011] introduce an object-based dataflow execution with data dependencies analysis method that we build upon SS to achieve an even more dataflow-like execution and exploit higher degrees of concurrency. They employ a decentralized scheduler as well as use the token protocol to handle WAW dependences. As the program is sequenced, dependent functions are shelved, and they are introduced into the deques after their dependences have been resolved. However, the WAW and WAR data hazards in Gupta and Sohi [2011] cannot be solved by renaming techniques.

To sum up, Table II lists the strengths and weaknesses of the proposed method compared to other related work. MP-Tomasulo, proposed in this article, applies instruction-level Tomasulo algorithm to MPSoC, and provides an out-of-order task execution engine. MP-Tomasulo divides the task issue and execution process with five stages: issue stage, task partition stage, execution stage, write results stage, and commit stage. It can detect RAW, WAW, and WAR data dependencies automatically with the task-adaptive partitioning and scheduling schemes, therefore MP-Tomasulo can improve the task-level parallelism without burden to programmers.

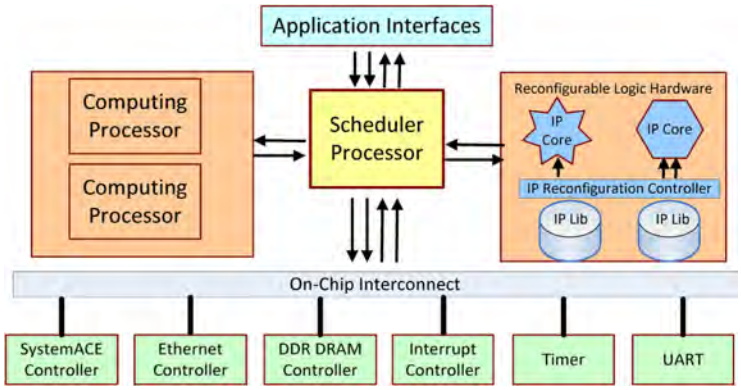


Fig. 1. Target MPSoC hardware platform.

## 3. HARDWARE PLATFORM AND DESIGN FLOW

### 3.1. Hardware Platform

Figure 1 illustrates the MPSoC hardware platform including the following components: one scheduler processor, multiple computing processors, and a variety of heterogeneous IP cores. Moreover, IP cores can be dynamically reconfigured from IP libraries to fit in different applications. In particular, the functionalities of different components in Figure 1 are described as follows.

(1) One scheduler processor (e.g., Microblaze, PowerPC, ARM) is employed to provide a programming interface to users. In particular, the MP-Tomasulo module is implemented as a software kernel running on the scheduler processor to schedule and distribute the tasks to function units at runtime.
(2) Computing processors provide runtime environments for software computational tasks. The tasks are offloaded from scheduler to computing processors for parallel execution directly.
(3) Each IP core is responsible for accelerating one specific kind of task in hardware. In addition, IP cores can be reconfigured and customized due to various application demands.
(4) Interconnect modules are in charge of inter-task communication between scheduler processor and diverse function units. For demonstration, the scheduler is connected to each processor or IP core with a pair of Xilinx Fast Simplex Link (FSL) bus channels [Xilinx 2009]. Moreover, the interconnect structure can be replaced by other topologic interconnection schemes, such as crossbar, mesh, hierarchical bus, or ring architectures.
(5) Memory blocks and peripherals are connected to the scheduler through the CoreConnect Processor Local Bus (PLB). These modules are integrated to maintain local data storage and debugging support, including DDR DRAM controller, Ethernet controller, systemACE controller, UART, timer and interrupt controller, etc.

### 3.2. Software/Hardware Codesign Flow

In this article, MP-Tomasulo is based on a flexible programming framework FPM [Wang et al. 2012], which is an extended version of CellSs [Bellens et al. 2006] from CellBE to general FPGA architecture. FPM is a software-hardware codesign paradigm that contains both C compilers for software applications and also hardware generation design flows for the IP-based accelerator on FPGA platforms.

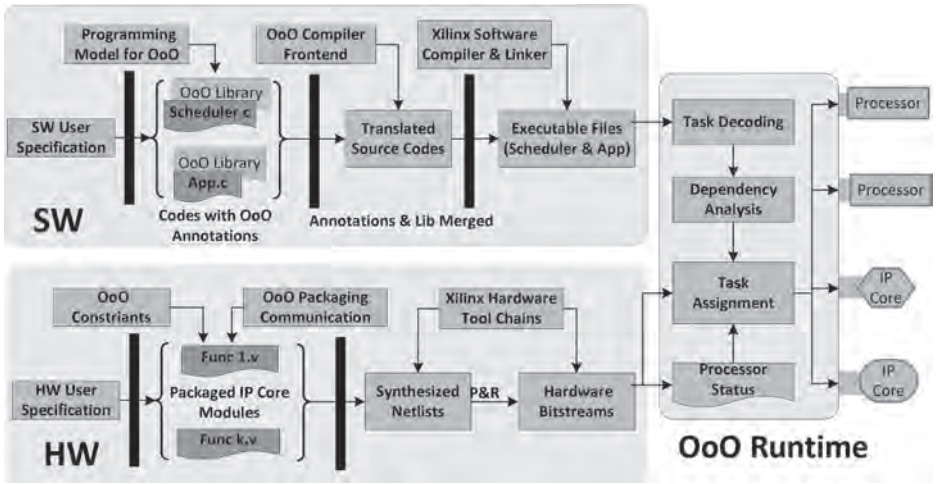


Fig. 2. Software/hardware codesign flow and runtime framework.

Taking both the integrated software and hardware modules into account, Figure 2 illustrates the software/hardware codesign flow, as well as the OoO runtime concept. The software executables and hardware bitstreams are generated separately.

*3.2.1. Software Design Flow.* With respect to the software generating thread, programming models and OoO libraries are provided to users to guide the implementation of source codes with OoO annotations. The first file (scheduler.c) refers to the main program of the application, and should be compiled with a Microblaze compiler to generate a scheduler object. Inside the program there are different code segments, including the functional tasks, I/O and debugging codes, etc. Except for the functional tasks, all the other codes are executed locally in sequence, while the computational-intensive tasks are offloaded to certain function units for parallel execution.

The second file (app.c) represents the application library on each computing Microblaze processor. Generally every computing processor contains a library with all kinds of functions. A certain function will be executed under the request from the main program. This file is also compiled with a Microblaze compiler, and a copy of the generated executable is located in each computing processor. In order to execute the target program, task requests mode by the scheduler processor are transferred through on-chip interconnects. For this reason, communication wrappers are embedded both in scheduler and computing processors.

Like CellSs, all the procedures (task mapping, dependence analysis, data transfer, and return) are transparent to the user code. What is required from programmers is to use simple annotations (# Pragma) in the sequential program that indicate which parts of the code will be run in parallel. Then FPM will map the tasks to target function units automatically, taking into account the ability of dynamic IP reconfiguration at runtime.

Afterwards codes with OoO annotations will be processed by the OoO compiler frontend, through which annotated codes and function libraries are merged into translated regular source codes. The regular source codes shall be further processed by the state-of-the-art Xilinx software tool chains (EDK, SDK, etc).

*3.2.2. Hardware Design Flow.* In contrast to the software compilation procedure, hardware design flow is presented in the bottom part of Figure 2. Besides the generated

executables for software processors, the third and fourth files (Func1.v and Funck.v) refer to RTL implementations for different functions in Hardware Description Languages (HDL). All the HDL source files should be designed under the user specification and the OoO constraints. Furthermore, in order to let the execution flow immigrate between software and hardware, a uniform communication interface should be employed to package the module into the IP core.

After the RTL codes are generated, they can be synthesized to netlists (e.g., *.ngc for FPGA devices), then placed and routed to bitstream objects.

*3.2.3. OoO Runtime Execution.* The main program executable is normally started in the scheduler processor. At the beginning of the program, the operation of each computing processor is initiated by loading the function library of each computing GPP. Meanwhile, each IP core is initiated with the downloaded hardware bitstreams. These computing processors and IP cores will stay in idle state until the main program starts spawning tasks to them. Whenever the main program runs into a parallel code region that can be spawned to a computing processor or IP core, the middleware runtime will check data dependencies with previously issued tasks. If the current task is ready for execution (no dependencies with previously issued tasks) and a selected target function unit is not in busy state, then the task can be spawned immediately. The task offloaded by FSL primitives is not blocking and, therefore, if the task is not finished, the system will continue with the subsequence of the main program.

When the tasks are finished, the results are returned through the FSL bus. At this time, as the scheduler processor is running the subsequent pieces of work, 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 from all the FSL links.

It is important to emphasize that all the procedures (task partitioning, data dependence analysis, data transfer, and results return) are transparent to the user code. These procedures are basically sequential applications using the provided API that indicates which parts of the code will be run in the parallel function units. Taking the task-level parallelization degree into account, the system can dynamically change the amount or the structure of IP cores for different applications.

## 4. COMPILER

In order to achieve the automatic OoO execution at task level, a programming model FPM is proposed to handle the annotations. FPM is composed of two key components: a source-to-source compiler and a runtime library, which includes an OoO scheduler and an adaptive mapping scheme. The current tool chain of FPM supports the following features.

(1) *translation from annotated source codes into codes recognizable to processors*: The source-to-source compiler locates whether the tasks are to be executed by computing processors or IP cores. From this feature, specific tasks can be accelerated by hardware execution engines, while general software tasks can be also executed simultaneously to uncover task-level parallelism.
(2) *integration of the hardware bitstream and software executable files*: From this feature, tasks can be decided to be spawned to either IP cores or GPP, as both of them are regarded as function units to run tasks in parallel.
(3) *specification of function parameter directions*: In order to allow the FPM runtime module to find the inter-task data dependencies, programmers should indicate the direction of the parameters explicitly (input or output).

```
/*-- Begin of FPMlib.h -- */
#pragma input(idct_in) output(idct_out)
void do_T_idct(int idct_out [N], idct_in [N]);
#pragma input (aes_in1,aes_in2) output(aes_out)
void do_T_aes(int aes_out [M], aes_in1 [M], aes_in2 [M]);
/*-- End of FPMlib.h -- */

/*-- Begin of the Main Program -- */

#include "FPMlib.h"
main( ){
      ......
      /*-- Begin of parameter declaration region-- */
      int idct_out[N], idct_in[N];
      int aes _out[M], aes _in1[M], aes _in2[M];
       /*-- End of parameter declaration region -- */
      ......
      /*-- Begin of Automatic Parallel region -- */
      ......
      do_T_idct(idct_out[N], idct _in[N]);
      do_T_aes_enc(aes _out[M], aes _in1[M], aes _in2[M]);
      /*-- End of Automatic Parallel region -- */
      ......
}
```

Fig. 3. Example of annotated codes in FPM.

### 4.1. Programmer's Perspective

Figure 3 outlines an example of annotated codes in FPM. Two files are listed: pragram definition in the FPMlib file, and the main program file.

(1) The upper part of Figure 3 gives an example of the FPM library that provides dedicated functions. The annotation indicates that the do_T_idct and do_T_aes functions can be executed on IP cores. Input and output parameters are defined in a declaration region before they are used in the annotated function. The sizes of the input and output arrays are specified in the definition as well.
(2) The bottom part of Figure 3 illustrates an example of a main program running on the scheduler processor. What is required by the programmer is to include the FPM library as head files, and define the necessary parameters before using them. Then the automatic parallel region maps the annotated functions to the target GPP or IP core automatically. The codes in the automatic parallel region work as normal sequential codes without annotations using the functions already defined in the FPM libraries.

The annotated regions will undergo a source-to-source compilation procedure at first, and then data dependencies are checked by middleware runtime support, and finally the codes generated from automatic parallel regions can be spawned either to

Table III. Example Programming Interfaces for Hardware

| Task | Programming Interfaces |
|---|---|
| IDCT | void do_T_idct(int * output , int * input )<br>void handler_T_idct() |
| AES_ENC | void do_T_aes_enc(int * output , int * input, int * input )<br>void handler_T_aes_enc() |
| AES_DEC | void do_T_aes_dec(int * output , int * input , int * input)<br>void handler_T_aes_dec() |

GPPs or IP cores. The task mapping decision is made by FPM at runtime. Although hardware execution is faster, the solution of waiting for hardware is still not a best choice when there are too many tasks already destined for the same hardware.

### 4.2. Code Translations of Different Regions

The FPM library includes the definition of parameters and annotated functions. Table III lists three examples for the annotated functions. For each annotated function, the compiler generates an adapter that can be called from the main program. In Table III IDCT stands for Inverse Discrete Cosine Transform and AES stands for Advanced Encryption Standard. Each function contains both one output and multiple input parameters. The parameters are annotated because they are also used to exploit the potential inter-task data dependencies. Some tasks may have multiple inputs or outputs, (e.g., AES_ENC needs plaintext and key as inputs, cipher as output). After a task is spawned, the scheduler can continue to run the subsequent tasks, and results will be returned through interrupts.

For each interface, a handler_T_xxx function is in charge of interrupt handling. The interrupt handler procedure is transparent to the user code, which means it is implicitly invoked automatically when the task results are returned.

*4.2.1. Translated Code of Specific Functions.* The main program running on the scheduler processor is a sequential application. Each annotated task calls an internal function to spawn the sequential tasks in parallel if they do not have inter-task data dependencies. Therefore all the annotated functions are translated into data transfer directives.

It is important to emphasize that the translated directives depend on the hardware interconnect between scheduler and function units. As a demonstration, we use directives with Xilinx Fast Simplex Link (FSL) peer-to-peer interconnection. All the computing GPP and IP cores are connected to the scheduler processor with a unique ID. For example, A do_T_IDCT (a,b) task indicates b is an input and a is the output. Therefore this task calls the directive of write_into_fsl (val, 2), where val refers to the input parameters (b), and "2" represents the ID of the IDCT IP core. Similarly, when the results are returned, the read_from_fsl (val, 2) function will be implicitly invoked by the interrupt handler procedure handler_T_IDCT(). The val parameter returned by the FSL bus will be assigned to the output parameters (a) inside the interrupt handler functions.

Furthermore, the write_into_fsl (val,2) directive invokes the putfsl (val, id) macro that is originally supplied by the FSL bus specification. On the contrary, the read_from_fsl (val,2) uses getfsl(val, id) as the basic data transfer macro. Related to Xilinx FSL bus channels, we also use Microblaze as our scheduler and computing GPP. By looking into the Microblaze instruction set architecture, PUT/GET/NPUT/NGET instructions are in charge of transfer data between the processor register files and external FSL bus channels.

*4.2.2. Code in Computing GPPs and IP Cores.* The computing processor can execute different types of applications. Figure 4 illustrates an example code with three functions

```
//Bus Specification Definitions:
#define write_into_fsl(val, id) putfsl(val, id)
#define read_from_fsl(val, id) getfsl(val, id)
#define WRITE_MicroBlaze_0(val) write_into_fsl(val, 0)
#define READ_MicroBlaze_0(val) read_from_fsl(val, 0)

// Internal Function Libraries
......
void idct_app() { /*–IDCT Functions--*/}
void aes_enc_app() { /*--AES_ENC Functions–*/}
void aes_dec_app() { /*– AES_DEC Functions–*/}
......
```

Fig. 4. Example code in computing GPPs.

integrated to run dedicated tasks. Corresponding to the translated interfaces in the scheduler processor side, FSL wrappers are provided in computing GPP as well to communicate with the main scheduler.

Besides the communication interfaces described in upper part of Figure 4, the lower part lists the internal function libraries implemented in computing GPP.

By contrast with the computing GPP, each IP core can run only one specific kind of functions, as is illustrated in the hardware/software design flow. All the IP cores are packaged in the same manner to communicate with the scheduler processor.

Taking FSL bus specification as an example, we utilized the signals transferred. FSL_Clk and FSL_Rst refer to input clock and system reset signals, respectively. FSL_S_* are control and data signals to drive the AES module as a slave module of the scheduler. The FSL_S_Exists signal will be activated when there is new data arriving from the scheduler. Similarly, when the tasks are finished, the results will be returned via the FSL_M_* signals. The FSL_M_Full signal indicates the full status of the FSL FIFO. In order to be compatible with the registers, the data width of the FIFO is set to 32 bits.

## 5. MP-TOMASULO EXECUTION MODEL

This section proposes the MP-Tomasulo architectural framework. The description focuses on how the MP-Tomasulo architecture is designed to dynamically detect inter-task data dependencies, uncover task-level parallelism, and enable out-of-order task execution. As the information of Tomasulo can be detailed in the referenced textbook and also is well-known to the research community, we summarize the essential differences between Tomasulo and MP-Tomasulo.

### 5.1. Architecture

Figure 5 illustrates the block diagram of the MP-Tomasulo architecture, which is composed of four hierarchical layers: preanalysis layer, scheduling layer, transfer layer, and computation layer.

First, the applications are divided into tasks in the preanalysis layer. The functionality of this layer is similar to the frontend analysis module in Task Superscalar [Etsion et al. 2010]. As the execution of the task-generating thread is decoupled from that of the tasks themselves, the inter-task control path is resolved by the preanalysis layer, and the tasks received by the pipeline are nonspeculative. The preanalysis layer also maintains a window of recently generated tasks, for which it generates

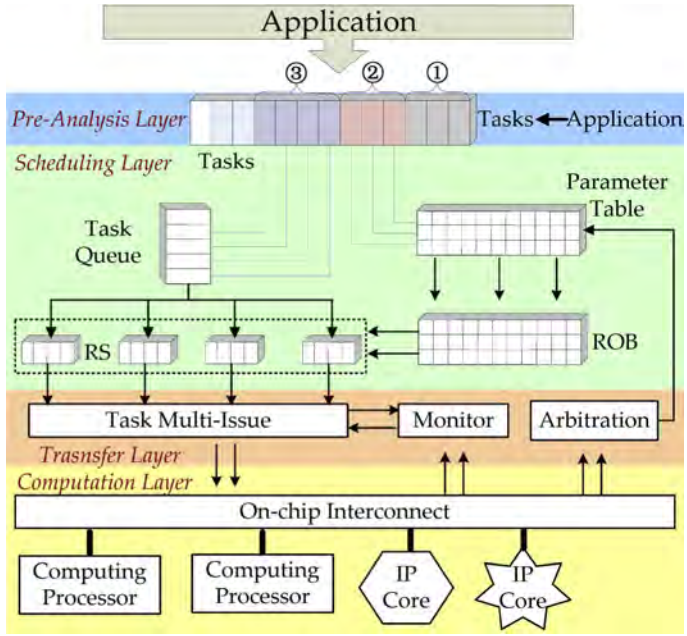


Fig. 5. MP-Tomasulo architecture.

the data-dependency graph, and uncovers task-level parallelism. All the tasks are issued in order, and then undergo a general process including three stages: label ① represents the finished tasks, while labels ② and ③ refer to the tasks under execution and waiting to be processed, respectively.

Second, the scheduling layer is responsible for data dependencies detection and out-of-order task scheduling. A task queue is employed to fetch tasks from the preanalysis layer. A ReOrder Buffer (ROB) is introduced to maintain in-order issue and commit for tasks. Reservation Stations (RS) are employed to analyze inter-task dependencies and store the intermediate values temporarily. A parameter table is utilized to record the mapping scheme between parameters and function units. All the ready tasks are issued from RS to transfer layer.

Third, the transfer layer is conducted to offload ready tasks to function units and synchronize them after they have finished execution. A task multiissue transmitter spawns multiple tasks to the related function units simultaneously. We integrate a monitor to keep track of all function units with respect to achieving system load balancing. Furthermore, one arbiter is essential when multiple tasks are returned simultaneously.

Finally, the computation layer consists of heterogeneous function units including both computing processors and IP cores. On-chip interconnections (buses, crossbar, mesh, ring, etc.) serve a group of functional units.

### 5.2. Data Structures of MP-Tomasulo

MP-Tomasulo algorithm runs as a software kernel on the scheduler processor. If the scheduler decides that the task cannot execute immediately, it will monitor any changes in the function units and then decide when the task can be issued. The scheduler also controls when the results will be stored into the local parameter table after

the task returns. In particular, there are five data structure components utilized by the MP-Tomasulo scheduler.

(1) *Reservation Stations (RS)* are employed to analyze inter-task dependencies. Whenever there is a ready task, it can be dispatched to function units immediately. There are nine fields for each reservation station: *Name* records the task name; *Busy* indicates whether the unit is busy or not; $V_j$ and $V_k$ are the source parameters; $Q_j$ and $Q_k$ are the flags indicating when $V_j, V_k$ are ready and not yet read. Finally *Dest* keeps the destination ROB entry.
(2) The *ReOrder Buffer (ROB)* utilized for in-order task issue and commitments. There exist two kinds of tasks: one consists the accelerated tasks running on the IP cores (e.g., FFT, JPEG) that are dispatched by MP-Tomasulo, and the other is comprised of software control tasks (e.g., I/O print, interrupt) running on the scheduler itself. However, due to the fact that MP-Tomasulo has no knowledge of software control tasks, when a specific parameter is being outputted by a Printf function, the producer task may still run in progress. In other words, no speculation and precise interrupts will be supported without ROB structure, which may cause dramatically wrong results of the I/O functions. As a consequence, ROB should be maintained to keep in-order commitment. In particular, ROB structure consists of the following elements: *Entry* refers to the functionality of current task; *Busy* indicates the task status; *Task* represents the task ID; *Dest* represents output operands; *Value* is the data value of output operands.
(3) The *parameter table* indicates which functional unit will store values for each parameter, if an active task has the parameter as its destination. Each table entry includes three fields: *Reorder*, *Busy*, and *Data*. *Reorder* reserves the entry of reorder to indicate which task to depend on; the *Busy* field keeps the reorder status of the parameter; and the parameter value is stored in the *Data* field.
(4) The *task partition module* is in charge of task partitioning and mapping. Since each task can either run on the processor or IP core, a fair partitioning method will largely increase the system throughput. For demonstration, we employ a greedy strategy: If there are idle IP cores, the task will be distributed to a specific IP core; otherwise, the task will be offloaded to a computing processor. If all the available function units are busy, the task must wait until certain hardware is released.
(5) *Function Unit Monitor and Arbitration* monitors and collects the running status of all function units. The status helps the task partitioning module to achieve load balance of the hardware.

### 5.3. Processing Flow

The MP-Tomasulo algorithm is divided into five stages: issue stage, partition stage, execution stage, write result stage, and commit stage. The five stages are similar to instruction level but adding a task partition stage. From Table IV, we can formally examine the steps and then see in detail how the algorithm keeps the necessary information determining when to progress from one step to the next. The five steps are as follows.

(1) *Issue*—Fetch a task from task queue. Issue the task if there is an empty reservation station and an empty slot in the ROB; send the operands to RS if they are available in ROB. Update the control entries to indicate which buffers are in use. The number of ROB entries allocated for the result is also sent to RS, so that the number can be used to tag the result when it is transferred back from processing elements. If either all the reservations are full or the ROB is full, the task issue is stalled until both have available entries.

Table IV. Processing Flow of MP-Tomasulo

| Task Status | | Wait until | Action or bookkeeping |
|---|---|---|---|
| Issue | Issue all tasks | Reservation station(r) and ROB(b) both available | if(VS[rs].Busy) /*in-flight task writes rs*/<br>{ h←VS[rs].Reorder;<br>if(ROB[h].Ready) /* task completed already*/<br>{RS[r].Vj←ROB[h].Value; RS[r].Qj←0;}<br>else {RS[r].Qj←h;} /*wait for task*/<br>} else { RS[r].Vj←VS[rs].Data; RS[r].Qj←0;}<br>RS[r].Busy←yes;RS[r].Dest←b;<br>ROB[b].Instruction←opcode; ROB[b].Destination←rd;<br>ROB[b].busy←yes;ROB[b].Ready←no;<br>VS[rd].Reorder←b; VS[rd].Busy←yes; |
| | Issue tasks with two inputs | | if(VS[rt].Busy) /*in-flight task writes rt*/<br>{ h←VS[rt].Reorder;<br>if(ROB[h].Ready) /* task completed already*/<br>{RS[r].Vk←ROB[h].Value; RS[r].Qk←0;}<br>else {RS[r].Qk←h;} /*wait for task*/<br>} else { RS[r].Vk←VS[rt].Data; RS[r].Qk←0;} |
| Task Partition | | Function unit done | Compare the task execution time and choose FU with minimum task execution time. Replaces the table entries. |
| Execute | tasks with one input | RS[r].Qj==0 | Distribute tasks to function units and compute results |
| | tasks with two inputs | (RS[r].Qj==0) and (RS[r].Qk==0) | |
| Write Result | | Execution done at r | b ←RS[r].Dest;RS[r].Busy ←no;<br>∀x(if (RS[x].Qj)==b {RS[x].Vj←result; RS[x].Qj←0;});<br>∀x(if(RS[x].Qk)==b{RS[x].Vk←result; RS[x].Qk←0;});<br>ROB[b].Value←result;ROB[b].Ready←yes; |
| Commit | | Task at the head of the ROB (entry h) and ROB[h].Ready ==yes | d←ROB[h].Destination; /*parameter dest, if exists*/<br>VS[d].Data←ROB[h].Value; /*update parameter*/<br>ROB[h].Busy←no; /*free up ROB entry*/<br>if(VS[d].Reorder==h) {VS[d].Busy←no;} |

(2) *Task Partition*—When all the operands are ready, MP-Tomasulo needs to decide which function unit to run the current task.

If there is only one function unit which can run the current task, then the task has no other options. However, in most situations, there are at least two available units (if both IP core and computing processor are available). Therefore in this situation, the task execution time on each function unit will be compared, and then a function unit with minimum execution time will be chosen. If a new function unit is chosen, the original function unit table entry will be replaced by the new one.

(3) *Execution*—The functional unit begins execution once operands are ready. When the current task is finished, it notifies the scheduler that it has completed execution. Task distribution and data transfer are both performed through on-chip interconnect. One interrupt controller is integrated to detect interrupt request signals from all the interconnect channels. The interrupt handler assigns the parameters with results. In our proposed architecture, since results from different tasks may be transferred back at the same time, a First-Come-First-Serve (FCFS) policy is used to deal with interrupts, and no interrupt preemption is supported.

(4) *Write Result*—After results are returned, they will be stored in ROB, as well as broadcasted to RS slots waiting for this result. If the value to be stored is available, it is written into the *Value* field of the ROB entry. If the value to be stored is not available yet, the interconnection must be monitored until that value is broadcasted, at which time the *Value* field of the ROB entry is updated.
(5) *Commit*—This is the final stage of completing a task, after which only its results remain. The normal commit cases occur when the task reaches the head of the task queue and its results are present in the buffer; at this point, MP-Tomasulo updates the parameters with results and removes the task from ROB.

### 5.4. Major Improvements of MP-Tomasulo

Compared to the traditional Tomasulo algorithm, the major improvements of MP-Tomasulo are summarized as follows.

(1) *Task mapping*. This article involves the situation when there are more than one function units available for each task. It mentions how to choose the suitable function unit for each task. Although a simple greedy strategy is presented, to our best knowledge, few literatures take task mapping together with out-of-order execution.
(2) *Monitoring and profiling*. MP-Tomasulo monitors and collects the running information of all the processors, so it can trace the whole execution process and locate the hotspot through profiling techniques. The runtime information can be used to guide the hardware reconfiguration.
(3) *IP core reconfiguration*. Benefiting from state-of-the-art dynamic partial reconfiguration technical supports (such as Xilinx EAPR), IP cores can be dynamically reconfigured at runtime. Since the FPGA provides an area-constrained platform, the reconfiguration technique largely improves hardware resource utilization.
(4) *On-chip interconnection*. This is utilized to distribute tasks and transfer results. In our proposed hardware architecture, we demonstrate a star network based on peer-to-peer links to model the on-chip interconnects. The star network can be replaced by other schemes, such as Network-on-Chip (NoC).

With the aforesaid improvements, MP-Tomasulo can efficiently increase TLP in MP-SoC by detecting data dependencies. Meanwhile, tasks are first buffered into ROB entries and then dispatched to different processors automatically when operands are ready. Therefore MP-Tomasulo uncovers task-level parallelism to minimize the number of stalls arising from the program's true data dependences.

### 5.5. Limitations

MP-Tomasulo supports the architecture in which IP cores are tightly coupled to the processor without shared memory access operations. In this situation, the IP core is more like a hardware accelerator for specific tasks. For more coarse-grained parallel architectures like SMP, there exists frequent memory access for each task. Even though MP-Tomasulo may be applied as well, programming models based on memory consistency will do a better job.

MP-Tomasulo is implemented as a software algorithm running on the central Microblaze scheduler. For this point, we have following concerns.

(1) The IP cores on FPGA can be dynamically reconfigured. Whenever IP cores are reconfigured (e.g., added, deleted, or replaced, etc.), the function unit table needs to be updated simultaneously, which will be difficult to implement on hardware.
(2) The area consumption of hardware MP-Tomasulo reduces the available resources for Microblaze and IP cores. Consequently, for now we use a software scheduler to involve more hardware IP cores in the limited FPGA chip area.

(3) Although some sort of memory management might be possible, allocation and management of arbitrary memory areas for such values might be a bit of a headache in real FPGA-based hardware implementations.

Besides the software implementation, MP-Tomasulo is also limited by several factors in eliminating program stalls.

(1) *The inter-task parallelism degrees* determine whether independent tasks can be found to execute at large. If each task relies on its predecessor, then our dynamic scheduling scheme can reduce no further stalls.
(2) *The size of ROB and RS entries* determines how far ahead MP-Tomasulo can find independent tasks. The sizes refer to the set of tasks examined as candidates for potential execution. Larger sizes mean that more tasks can be prefetched, however, bringing overheads when storing tasks and maintaining data concurrency among different entries.
(3) *The number and types of functional units* determine the impact of structural dependencies in the issue stage. If there are no more available function units, the tasks will stall, and no tasks can be issued until these dependencies are cleared.
(4) *Task partitioning plans* determine the target function unit for each task. The partition method has a significant impact on performances evaluation. Our demonstrated greedy strategy can only achieve a local optimum instead of global optimum for the whole task sequences. However, the current task partitioning plans can be switched to other schemes, for example, dynamic programming or heuristic methods.

## 6. PROTOTYPE IMPLEMENTATIONS

### 6.1. Platform Setup

To evaluate MP-Tomasulo in real hardware, we implemented a prototype on a state-of-art FPGA board, equipped with Xilinx Virtex-5 FPGA. We use Microblaze version 7.20.a (with the clock frequency 125 MHz, local memory of 8KB, no configurable task or data cache) as scheduling and computing processors.

Specifically, the prototype system is composed of the following components.

(1) One scheduling Microblaze processor is integrated to run the MP-Tomasulo software scheduling algorithm and provide API to programmers.
(2) One Microblaze is employed as computing processor. Software task functions are implemented and packaged in standard C libraries.
(3) Five hardware IP cores are implemented in HDL (Verilog) and packaged with uniform Xilinx FSL-based interfaces. The scheduler Microblaze is connected to the computing Microblaze and IP cores with pairs of Xilinx FSL channels. Tasks and results are transferred via FSL bus links.

Parts of the EEMBC-DENBench [EEMBC 2010] have been utilized for demonstration. For each following test case, we both transplanted the software benchmarks to Microblaze and implemented the IP core: Adder, IDCT, AES, DES, and JPEG. Since the IP cores can be configured with custom execution cycles in hardware, they can play the same role with the remaining benchmarks to demonstrate the effectiveness of the MP-Tomasulo approach. Furthermore, the hardware implementation of the platform is detailed in the referenced SOMP [Wang et al. 2011] architecture.

### 6.2. Results and Analysis

Based on the prototype system, we designed several applications to measure the performance and the scheduling overheads for the proposed MP-Tomasulo algorithm on

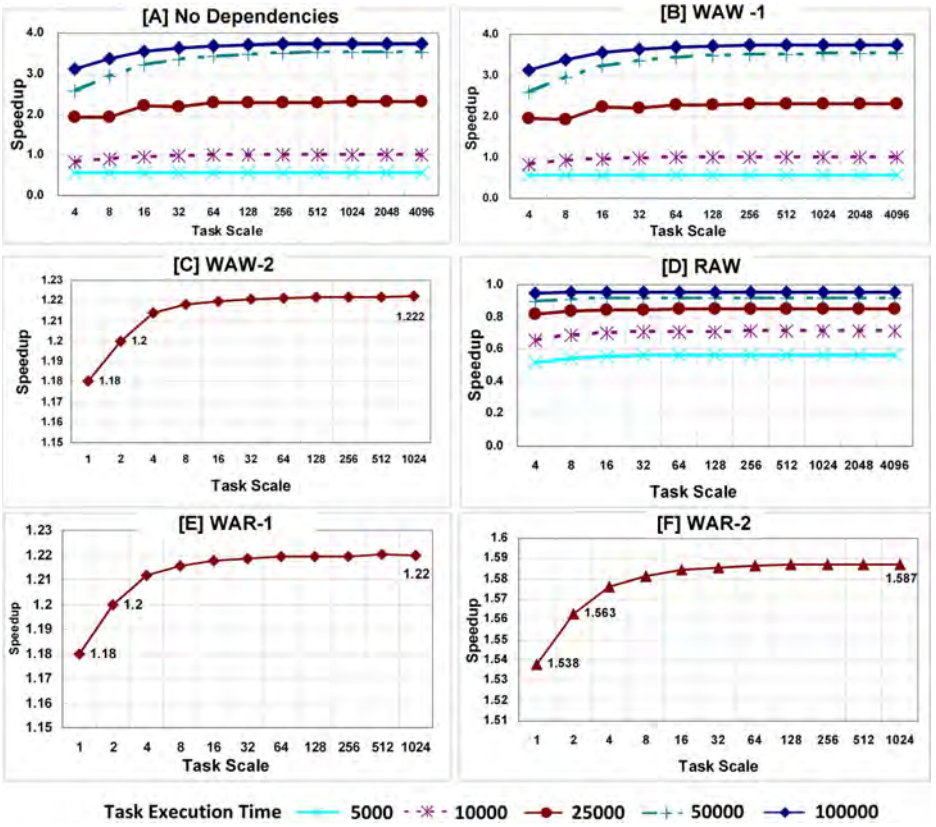


Fig. 6. Experimental results for no dependencies, RAW, WAW, and WAR.

the hardware prototype. Since the EEMBC benchmarks only provide individual applications (such as IDCT, AES, etc.), we combined different tasks with selected parameters to construct different types of inter-task data dependencies.

*6.2.1. Speedups of Different Types of Data Dependencies.* In this section, we measured the speedup under four situations: no dependencies, WAW, RAW, and WAR. In order to evaluate the peak speedup, we define two parameters.

First, the task execution time denotes the entire execution time used in different types of data hazards. In the circumstance of no dependencies, WAW, and RAW, the task execution time is configured to the same value (varying 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 to less than 4096 in all the test cases.

(1) *No Data Dependencies*. Figure 6(a) presents the speedup of the case with no dependencies. The horizontal coordinate refers to the task scale. When the four IP cores have the same running time, the theoretical maximum speedup is 4.0x. However, because of the software scheduling cost and communication overheads, the experimental results cannot reach the peak speedup, especially in the situation

Table V. Test Applications for WAW Data Dependencies

| WAW Test Sequence 2 | Execution Time(Cycles) |
|---|---|
| do_T_adder(a,c)<br>do_T_adder(b,c)<br>do_T_idct(a,c) | adder:100000<br>idct:50000 |

Table VI. Task Sequences to Test WAR Data Dependencies

| WAR Sequence 1 | WAR Sequence 2 | Execution Time |
|---|---|---|
| do_T_adder(a,b)<br>do_T_adder(c,a)<br>do_T_idct(a,b) | do_T_idct(a,b)<br>do_T_adder(c,a)<br>do_T_idct(a,b)<br>do_T_idct(d,a)<br>do_T_aes_enc(a,e,f) | adder:100000<br>idct:50000<br>enc:25000<br>dec:12500 |

with smaller and less tasks. From the figure, we can see that when the task scale and task running time are large enough, the experimental peak speedup can reach 3.744x, which is 93.6% of the ideal peak value.

(2) *WAW Data Dependencies*. The experimental results of WAW data dependencies are presented in Figure 6(b). In this case, all the tasks are configured as the same value, which means the ideal peak speedup is the same as the no dependency situation 3.744x, which is 93.6% of ideal speedup.
Another test sequence for WAW inter-task dependency is listed in Table V. Three tasks are designed in the sequence. In particular, task *do_T_adder(a,c)* and task *do_T_idct(a,c)* both use the parameter a as destination parameter (with c as the source). Since IDCT costs less time, it finishes before the second adder task, which causes a WAW dependency. The theoretical maximum speedup is calculated in Eq. (1).

$$Speedup = (T_{Adder} \times 2 + T_{IDCT})/(T_{Adder} \times 2) = 1.25 \quad (1)$$

The experimental result is described in Figure 6(c). It is easily observed that the experimental speedup grows with the task scale. The reason is that the scheduling phase and execution stage for each task run on different processors, which can be treated as two stages in the pipeline. For the first task, they can only run in sequence to fill the pipeline the. As the task scale grows, the influence of pipeline filling becomes smaller. The experimental speedup grows from 1.184x to 1.222x, which is from 93.13% to 97.74% of the theoretical value.

(3) *RAW Data Dependencies*. Figure 6(d) illustrates the experimental speedup of RAW. Since RAW dependency cannot be eliminated by any renaming technologies, tasks can only run in sequential, which means the ideal speedup is 1.0x. In this case, the maximum experimental speedup is 0.957, which is 95.7% of the ideal speedup.

(4) *WAR Data Dependencies*. In the WAR case, tasks with different execution time are configured in Table VI.

For the two sequences, the theoretical peak speedup is calculated in Eqs. (2) and (3).

$$Speedup_1 = (T_{Adder} \times 2 + T_{IDCT})/(T_{Adder} \times 2) = 1.25 \quad (2)$$

$$Speedup_2 = (T_{Adder} \times 2 + T_{IDCT} \times 2 + T_{aes\_enc})/(T_{Adder} \times 2) = 1.625 \quad (3)$$

The experimental result is shown in Figure 6(e) and (f) accordingly. The curve is similar to the WAW situation. The experimental peak speedup for the first task sequences grows from 1.183x to 1.220x, which is from 94.58% to 97.63% of the theoretical value. And for the second sequences, speedup increases from 1.538x to 1.587x, which is from 94.64% to 97.67%.

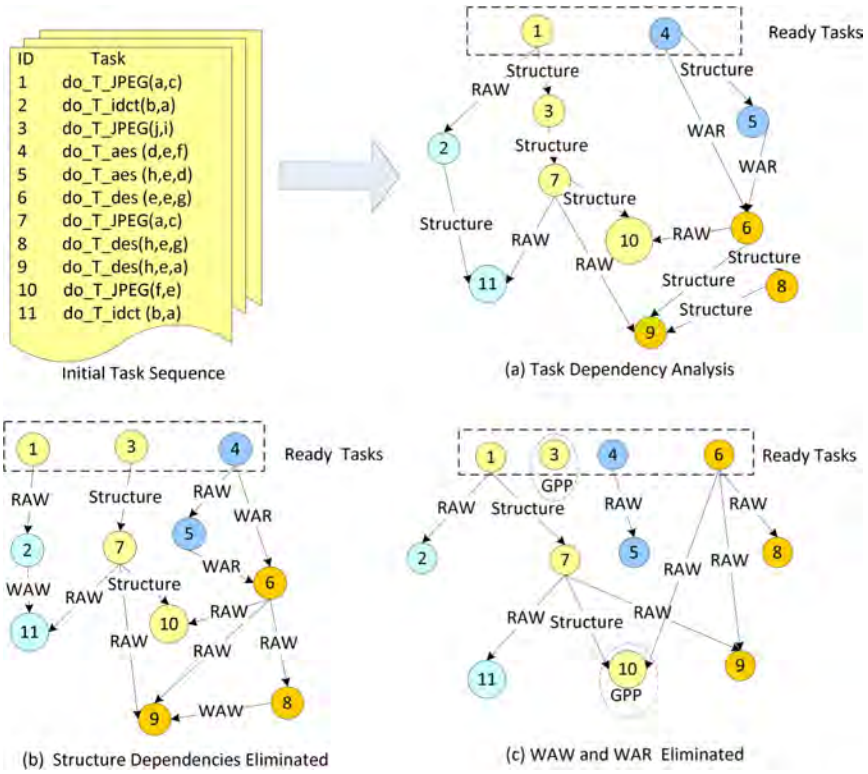


Fig. 7. Dependency-aware out-of-order task scheduling.

*6.2.2. Speedup of MP-Tomasulo vs. Scoreboarding.* We designed several example test cases using the specific functions of the implemented IP cores. A sample test case is illustrated in Figure 7. Each node stands for a task, and the edges refer to the dependencies. Different tasks are marked with colors. In the first step (a), the data and structure dependencies among all tasks are detected. Tasks that do not have data dependencies with other tasks are regarded as the ready tasks. In this stage, only task 1 and task 4 are in the ready task list.

In the second step (b), considering the multiple hardware resources, parts of structure dependencies are removed. Task 3 can be issued immediately as soon as it arrives.

Finally, all the WAW and WAR dependencies are eliminated by parameter renaming technologies, resulting in a simplified task-dependency graph in (c). Also task 6 is added into ready task list.

We measure speedups of one typical task sequence through partitioning the test tasks. In this test case, we select regular test with 11 different random tasks.

Figure 8 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 the theoretical value, but slightly larger. This is because the theoretical value has not considered scheduling and communication overheads. The average of these overheads is less than 4.9% 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 tasks 2 and 4 for instance, 2 takes more clock cycles to finish than 4. As there are not data


Fig. 8. Experimental results of regular tasks.

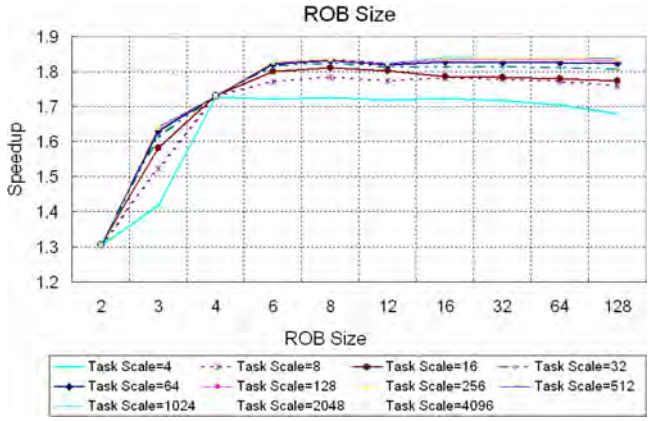


Fig. 9. Impact of ROB size on the speedup.

dependencies among these tasks, they can be issued at the same time. After task 4 is finished, it must wait until all the predecessor tasks are finished.

Compared to Scoreboarding [Wang et al. 2012], Tomasulo has higher scheduling overheads, which leads to a bigger gap between the experimental and theoretical value. However, since MP-Tomasulo cannot only detect WAW and WAR hazards but also eliminate them by renaming, the overall speedup is significantly larger than Scoreboarding.

*6.2.3. Impact of ROB Size on the Speedup.* Since the peak speedup for MP-Tomasulo is also dependent on the size of ROB structures, we designed three applications to measure the impact of ROB size.

Figure 9 illustrates the experimental result. As the ROB size grows bigger, the ROB has more slots for task allocation in the issue stage, which brings increasing speedup at first. When the ROB size is 8, the speedup achieves a saturation value at 1.8374x, which is 97.99% of the theoretical value. However, when the ROB is large enough, continuing to enlarge ROB size will cause additional overheads for update and synchronization operation, so the performances will be reduced.

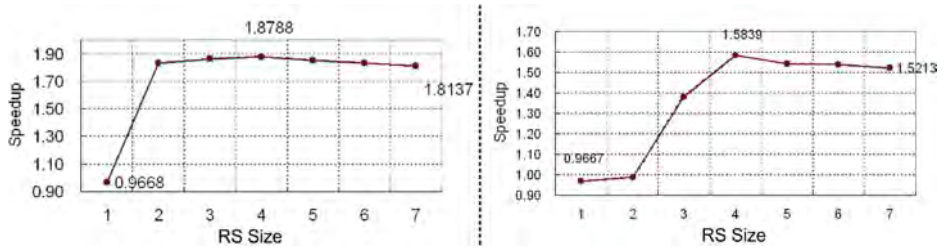


Fig. 10. Impact of RS size on the speedup.

*6.2.4. Impact of RS Size on the Speedup.* Besides ROB, RS size also has a significant impact on speedups. Based on the RS size and theoretical speedup, Figure 10 presents the impact of RS size on the speedup of all the three preceding task sequences. Growing with the RS size, there is a shape increase when it changes from 1 to 2, and then it saturates. When RS size is set to 1, it means there is only one reservation station integrated for each type of task. Therefore, after the first task is issued, all the other tasks must wait until the RS slot is free again, therefore the speedup is less than 1.0x.

However, when the RS changes to 2, up to 2 tasks in the same type are allowed to be issued to the RS at the same time. Respecting this, if the first task is blocked, there will also exist another available entry for the next task in the same type. Therefore the execution works in a dual buffer, and the speedup is significantly increased. For a different application, if there are more continuous tasks belonging to the same type in the application, the ideal RS size will be larger as well.

Then when the RS is larger than 2, the speedup is saturated. This is because in this test case only two of the same tasks are designed in sequence, which means two RS slots for each type of task are enough to achieve peak speedup. In fact, the overheads include two parts: task allocation cost for RS and data broadcast cost.

As the RS is utilized to store and transfer intermediate values, as the RS grows bigger, it gets easier to allocate RS to tasks, which reduces the allocation cost. Meanwhile, the growing RS size will also bring extra cost when data is broadcasted. The experimental results and theoretical values are 98.57%, 93.94%, and 95.03%.

### 6.3. A Comparative Study on JPEG Applications

*6.3.1. Performance Comparison.* Along with the software version of the MP-Tomasulo algorithm described before, it is necessary to show how MP-Tomasulo works if it is implemented on hardware. Therefore, we have also realized a hardware scheduling module according to the data structures and processing flow described in Section 5. Along with the Scoreboarding technique [Wang et al. 2012], there are also some creditable OoO execution engines, like [Gupta and Sohi 2011]. In order to evaluate its superiority, we use a JPEG application 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 variables and output variables, 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 variables and only takes 1.94% of the total execution time of the whole program on average.

In Figure 11 we show the analysis of the execution of the JPEG benchmark. The JPEG algorithm takes a BMP picture as input, and outputs a picture in JPEG format. The entire picture is divided into 8*8 bits blocks, and each time only one block is processed. Assuming that the benchmark is run on an FPGA platform and that the memory size is constrained so that the whole picture cannot be stored in the memory

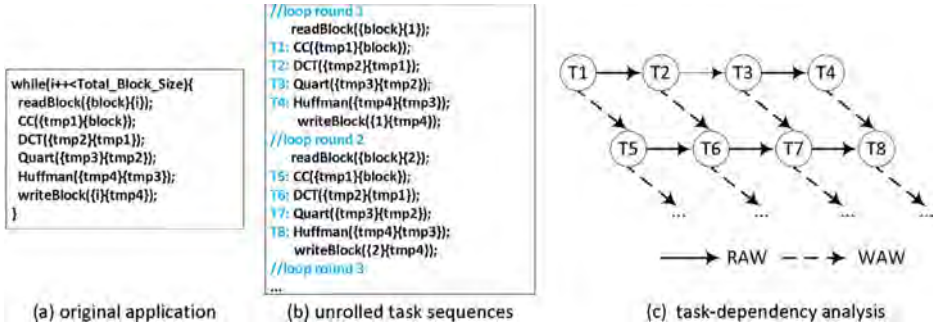


Fig. 11. JPEG application and task-dependency analysis.

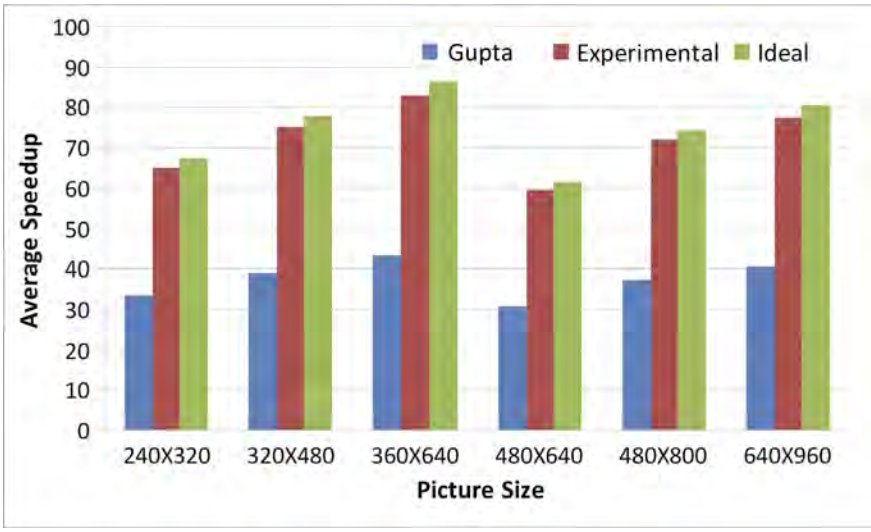


Fig. 12. Speedup comparison. In each group, the left bar is the speedup using the algorithm in Gupta and Sohi [2011], the middle bar is the practical speedup on our experimental platform, and the right bar is the ideal speedup using our platform.

at a time, a possible code as depicted in Figure 11(a) may be proposed. Figure 11(b) illustrates the task sequences when the first two loops are executed, and Figure 11(c) introduces the dependency graph of the tasks in Figure 11(b), in which the solid arrows stand for RAW and the dashed arrows stand for WAW.

For the specific JPEG application, a real hardware module has been integrated into the FPGA board as well. With the help of a MP-Tomasulo module, the sequential execution mode of the JPEG program is converted to an OoO mode, where only tasks that have RAW dependency with others need to run in order, while the WAW dependency will be dynamically eliminated by the MP-Tomasulo module. We select 30 pictures of 6 different sizes for experiments, and for each size we randomly picked 5 pictures in BMP format. In order to explain our framework does eliminate WAW/WAR dependency, we compare the experimental results with the method described in Gupta and Sohi [2011], as is illustrated in Figure 12. The method in Gupta and Sohi [2011] only detects dependency, however, WAW/WAR dependency is treated the same as RAW dependency when tasks are running. In Figure 12 we compare ideal speedup of Gupta and Sohi [2011], the practical, and ideal speedup of our framework. Note the speedup takes the

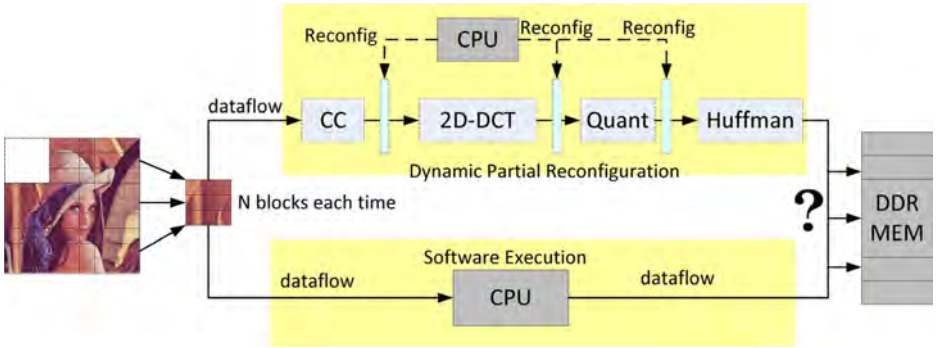


Fig. 13. Reconfigurability study comparing dynamic partial reconfiguration with software execution, using JPEG applications.

execution time of the software version running on an MB as the baseline, while ideal speedup is defined as the theoretical speedup ignoring the scheduling time. The ideal speedup of Gupta and Sohi [2011] is only half of the ideal speedup of our framework because, restricted by WAW dependency, only one MB and one CC-DCT-QUART IP core are in use, while the other PEs are all idle.

Experimental results depict that MP-Tomasulo can achieve an average of more than 95% of the ideal speedup, which greatly encourages our motivation and technical scope.

*6.3.2. Reconfigurability Study.* The aforesaid JPEG test case has demonstrated the superiority of MP-Tomasulo over the state-of-the-art OoO approach. As we use the real FPGA hardware platform, it is also interesting to explicitly exploit the reconfigurability of an FPGA-MPSoC to highlight the significance of MP-Tomasulo. It is common knowledge that due to the reconfiguration overheads, it is not always a fair way to get better results if we use dynamic reconfiguration methods rather than execute the application in software. We use the JPEG case to evaluate this point. As the entire picture for JPEG is divided into multiple blocks, the system may reconfigure when several blocks (let be N) are processed. If N is small, the reconfiguration overheads will drag down the parallel hardware speedup, the performance will be even worse than the sequential execution, therefore, we explore the trade-off in the revised version.

For the sake of area limitations for normal FPGA devices, in this study, we consider an area-optimal condition where only one IP core can be integrated at a time. As the four function modules are implemented in separate IP cores, the functionality will be reconfigured after certain blocks (let be N) are processed. Figure 13 illustrates the processing flow for both dynamic partial reconfigurable hardware and software. In the initialization step, only the CC IP core is integrated in the system. After the color space for N blocks is converted, the CC IP core will be reconfigured to 2D-DCT IP core. Consequently three reconfiguration operations happen for each N blocks, denoted as the three bars with dashed edges between the four phases. Due to the unavoidable overheads, the total execution time may be even longer than the software execution time in CPU, as is presented in the bottom part of Figure 13. Therefore we evaluate the trade-off using different N values for hardware reconfiguration and software.

For the JPEG test case, we run some preanalysis to identify that the threshold value of N lies between the span from 12 to 24. Then we evaluate the situation with different N, ranging from 13 to 23, as well as the software execution in CPU. We use the Lena picture in different sizes, from 64*64 to 512*512, to observe the trade-off between dynamic reconfiguration and software execution. Figure 14 presents experimental results of the execution time for each block. The x-axis refers to different size of pictures,

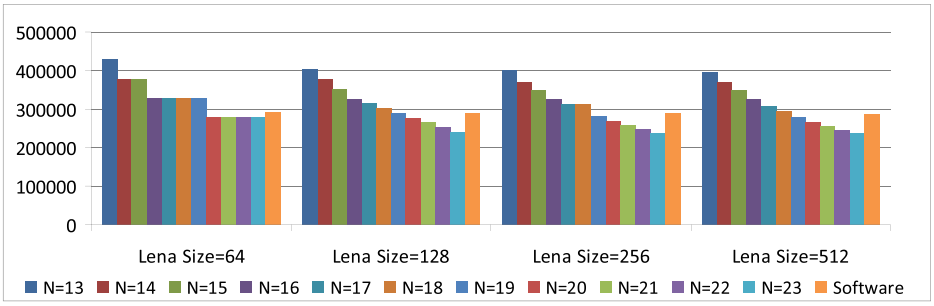


Fig. 14. Hardware execution time of different N vs. software execution.

Table VII. Hardware Cost of the Heterogeneous MPSoC System

| Resource | Used/Available | Utilization |
|---|---|---|
| Number of Slice Registers | 7536/28800 | 26.2% |
| Number of Slice LUTS | 19941/28800 | 69.2% |
| Number used as Memory: | 534/7680 | 7.0% |
| Number of External IOBs | 4/480 | 0.8% |
| Number of BUFGs | 3/32 | 0.9% |
| Number of DSP48Es | 31/48 | 64% |

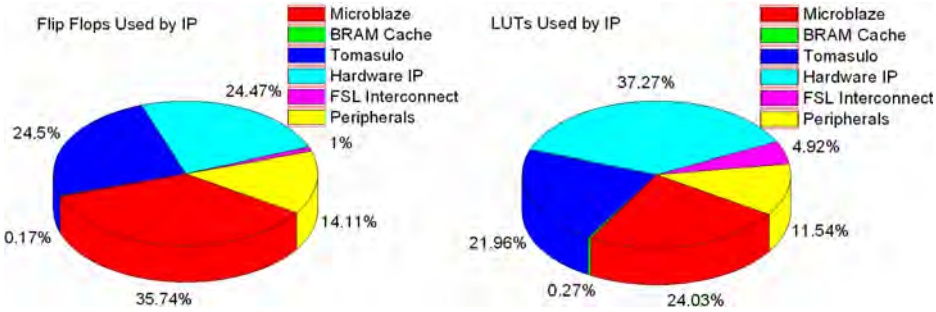


Fig. 15. Percent of hardware resources.

each with different N values and software execution, while the y-axis indicates the average execution time for each block. It can be observed that the threshold value of N in this case is approximately 18. When N is smaller than 18, the reconfigurable hardware execution time will be longer than software execution. Otherwise, as N grows bigger than 18, the dynamic partial reconfiguration shows its technical superiority. This experiment facilitates researchers to gain a quantitative analysis using reconfiguration techniques.

### 6.4. Hardware Cost

We have measured the hardware cost of the entire MPSoC system. Table VII summarizes the hardware cost within a single FPGA. The whole system takes 26.2% of slice registers and 69.2% of slice LUTs.

Furthermore, by looking further into the synthesis report, we see most of the resources are occupied by Microblaze processors and hardware IP cores, as is summarized in Figure 15. The basic system (including scheduler, the MP-Tomasulo module, peripherals, and interconnects) cost is acceptable, except that the hardware implementation of MP-Tomasulo takes 1714 of the flip-flops and 1963 LUTs.

Experimental results demonstrate that the hardware module is larger than one Microblaze processor in this JPEG application, but the hardware utilization could be adjusted in other application-specific computing systems. Taking the total amount of hardware resources into consideration, the entire hardware platform can be easily transplanted onto different boards and then used to test and verify for more scheduling algorithms, programming models, interconnect structure, and concepts.

## 7. CONCLUSIONS AND FUTURE WORK

We have proposed MP-Tomasulo, a dynamic scheduling algorithm for out-of-order task execution. Regarding processors and IP cores as function units, MP-Tomasulo processes tasks as abstract instructions. It can analyze inter-task dependencies at runtime and distribute tasks to heterogeneous function units automatically. The algorithm is carried out on a state-of-the-art FPGA platform. Test cases and experiments demonstrate the algorithm can achieve more than 95% of the theoretical peak speedup.

As future work, we plan to extend MP-Tomasulo to reconfigurable situations where processors and IP cores can be adaptive to fit in different applications at runtime. Also, we plan to study more hardware IP extensions to investigate how the approach can be applied to general multiple tasks in superscalar or thread-level parallelization in support of operating systems.

## ACKNOWLEDGMENTS

The authors deeply appreciate many reviewers for their insightful comments and suggestions.

## REFERENCES

Allen, M. D., Sridharan, S., and Sohi, G. S. 2009. Serialization sets: A dynamic dependence-based parallel execution model. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM Press, New York.

Bellens, P., Perez, J. M., Badia, R. M., and Labarta, J. 2006. CellSs: A programming model for the cell be architecture. In *Proceedings of the ACM/IEEE Conference on Supercomputing*. ACM Press, New York.

Berekovic, M. and Niggemeier, T. 2008. A distributed, simultaneously multithreaded (SMT) processor with clustered scheduling windows for scalable DSP performance. *J. Signal Process. Syst. 50*, 2, 201–229.

Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. 1995. Cilk: An efficient multithreaded runtime system. *ACM SIGPLAN Not. 30*, 8, 207–216.

Borkar, S. and Chien, A. A. 2011. The future of microprocessors. *Comm. ACM 54*, 5, 67–77.

Deng, D. Y., Lo, D., Malysa, G., Schneider, S., and Suh, G. E. 2010. Flexible and efficient instruction-grained run-rime monitoring using on-chip reconfigurable fabric. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 137–148.

EEMBC. 2010. The embedded microprocessor benchmark consortium. http://www.eembc.org/.

Etsion, Y., Cabarcas, F., Rico, A., Ramirez, A., Badia, R. M., et al. 2010. Task superscalar: An out-of-order task pipeline. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society.

Givargis, T. and Vahid, F. 2002. Platune: A tuning framework for system-on-a-chip platforms. *IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst. 21*, 11, 1317–1327.

Gupta, G. and Sohi, G. S. 2011. Dataflow execution of sequential imperative programs on multicore architectures. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM Press, New York.

Hammond, L., Hubbert, B. A., Siu, M., Prabhu, M. K., Chen, M., and Olukotun, K. 2000. The stanford hydra cmp. *IEEE Micro 20*, 2, 71–84.

Kahle, J. A., Day, M. N., Hofstee, H. P., Johns, C. R., Maeurer, T. R., and Shippy, D. 2005. Introduction to the cell multiprocessor. *IBM J. Res. Devel. 49*, 4–5, 589–604.

Karim, F., Mellan, A., Aydonat, U., Abdelrahman, T. S., Stramm, B., and Nguyen, A. 2003. The Hyperprocessor: A template system-on-chip architecture for embedded multimedia applications. In *Proceedings of the Workshop on Application Specific Processors*.

Karim, F., Mellan, A., Nguyen, A., Aydonat, U., and Abdelrahman, T. 2004. A multilevel computing architecture for embedded multimedia applications. *IEEE Micro 24*, 3, 56–66.

KhronosGroup. 2010. Automatic parallelization in a binary rewriter. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 547–557. http://www.khronos.org/opencl/.

Kumar, S., Hughes, C. J., and Nguyen, A. 2007. Carbon: Architectural support for fine-grained parallelism on chip multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*. ACM Press, New York.

Kuzmanov, G., Gaydadjiev, G., and Vassiliadis, S. 2004. The MOLEN processor prototype. In *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04)*.

Limberg, T., Winter, M., Bimberg, M., Klemm, R., and Fettweis, G. 2009. A heterogeneous MPSoC with hardware supported dynamic task scheduling for software defined radio. In *Proceedings of the 46th Design Automation Conference (DAC'09)*.

Patt, Y. N., Hwu, W. M., and Shebanow, M. 1985. HPS, a new microarchitecture: Rationale and introduction. In *Proceedings of the 18th Annual Workshop on Microprogramming*. ACM Press, New York.

Rotenberg, E., Jacobson, Q., Sazeides, Y., and Smith, J. E. 1997. Trace processors. In *Proceedings of the 30th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 138–148.

Sanchez, D., Lo, D., Yoo, R. M., Sugarman, J., and Kozyrakis, C. 2011. Dynamic fine-grain scheduling of pipeline parallelism. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society.

Sankaralingam, K., Nagarajan, R., Mcdonald, R., Desikan, R., Drolia, S., et al. 2006. Distributed microarchitectural protocols in the TRIPS prototype processor. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*.

Sohi, G. S., Breach, S. E., and Vijaykumar, T. N. 1995. Multiscalar processors. In *Proceedings of the 22nd International Symposium on Computer Architecture*.

Suh, J. and Dubois, M. 2009. Dynamic mips rate stabilization in out-of-order processors. *SIGARCH Comput. Archit. News 37*, 3, 46–56.

Swanson, S., Michelson, K., Schwerin, A., and Oskin, M. 2003. WaveScalar. In *Proceedings of the 36th International Symposium on Microarchitecture*.

Taylor, M. B., Kim, J., Miller, J., Wentzlaff, D., Ghodrat, F., et al. 2002. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro. 22*, 2, 25–35.

Tomasulo, R. M. 1967. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. Res. Devel. 11*, 1, 25–33.

Wang, C., Li, X., Zhang, J., Chen, P., Feng, X., and Zhou, X. 2012. FPM: A flexible programming model for MPSoC on FPGA. In *Proceedings of the International Parallel and Distributed Processing Symposium Workshops and PhD Forum (IPDPSW)*. 477–484.

Wang, C., Zhang, J., Zhou, X., Feng, X., and Nie, X. 2011. SOMP: Service-oriented multi processors. In *Proceedings of the IEEE International Conference on Services Computing*. IEEE Computer Society.

Wawrzynek, J., Oskin, M., Kozyrakis, C., Chiuo, D., Patterson, D., et al. 2007. RAMP: Research accelerator for multiple processors. *IEEE Micro 27*, 2, 46–57.

Xilinx. 2009. Fast simplex link (FSL) specification. http://www.xilinx.com/products/ipcenter/FSL.htm.

Received September 2012; revised October 2012; accepted December 2012