IP Core Design of Task Scheduler to Support Out-of-Order Execution in an MPSoC Environment

Ruchika Bamnote, Priya M. RavaleNerkar

PG Student [VLSI & Embedded Systems], Dept. of ETC, DYPCOE Akurdi, Pune, Maharashtra

ruchikabamnote@gmail.com, 8412824480

Abstract—In this paper the design of an IP core of Multiprocessor System-on-Chip (MPSoC) in the context of microarchitecture of the Scheduling Processor targeted for multicore systems is explored. This scheduling processor schedules the tasks i.e. units of computation in parallel for execution on different processors and IP cores. As this model is dealing with Out-of-Order (OoO) execution, the data dependencies like Read-after-Write (RAW), Write-after-Read (WAR) and Write-after-Write (WAW) imposes the challenging constraints on the direct use of techniques like register renaming and dynamic scheduling at instruction level. The scoreboarding algorithm with parameter renaming technique at task level analyzes the data dependencies in order to solve the stalling problem occurring in OoO execution. Thus the scheduler schedules different tasks in OoO manner. The model for the same has been verified by using resulting timing diagram. The results demonstrate that the model can largely release the burden on programmers as well as uncover the task level parallelism (TLP).

Keywords—Multiprocessor System-on-Chip, Out-of-Order execution, scoreboarding algorithm, register renaming, data dependencies, stalling, task scheduling.

INTRODUCTION

Multiprocessor system on chip has been seen in main stream since last few years [1]. Companies like Xilinx and Altera have prime focus of research on this emerging area. The development of MPSoC begins from the multi-core central processing unit. It is a platform that contains multiple processing elements with specific functionalities which are usually heterogeneous. However, due to its heterogeneous instruction set architectures, software tool chains and programming interfaces, it has presented many challenges to efficient designing and implementation of rapid prototype for diverse applications.

The most promising future processor architectures are considered as the combination of reconfigurable computing and multi-core technologies [1]. However, there are some critical issues like computational capabilities, scalability, programmability, flexibility, power consumption and so on, which are becoming increasingly important. Such raising demands have resulted into the outgrowth of FPGA based MPSoC composed of a variety of heterogeneous computational units. Whereas, OoO execution is a paradigm used in most high-performance microprocessors to make use of instruction cycles that would otherwise be wasted by delay. In this paradigm, a processor executes instructions in out-of-order as soon as the input data is available, instead of their original order in a program. As it allows execution with less waiting time, the performance definitely improves. Also the process technology has improved and per units more transistors can be fitted in the same die area, hence adding new features to the system becomes effectively easy. Thus dynamic scheduling can easily be implemented to build cost effective system.

In basic pipelining the system uses in-order instruction issue technique due to which if an instruction stalls rest all instructions are stalled. In contrast to this, the OoO execution has capability to schedule the ready instructions independently. Therefore to solve the stalling problem OoO execution technique is used for multi-cycle task execution, as this study focuses on multi-cycle task execution. Dynamic scheduling is a useful scheduling technique for multi-cycle instructions systems. Scoreboarding and Tomasulo are two such effective methods for dynamic scheduling, out of which this study implements scoreboarding algorithm for OoO parallel execution of task scheduling. Thus an IP core of task scheduler intending to solve the data dependencies during OoO execution using dynamic scheduling is developed in MPSoC environment.

RELATED STUDY

Rigorous research is being done on MPSoC regarding the critical issues like computational capabilities, programmability, flexibility, scalability and power consumption. By using parallel programming paradigm the computational capabilities can be achieved. Such parallel task execution models have been studied for parallel computing machines during the past decades. Initially, many task based parallel programming models were popular like Cilk [2] to enhance ILP to TLP. Mostly it was focused on symmetric multiprocessor but it was unable to support fully automatic parallelization. So the programmers have to handle the task scheduling and mapping schemes manually. Some of them focused on the utilization of reconfigurable FPGA platform and integration of acceleration engines, such as Chimaera [3], Platune [4] and MOLEN [5].

Also models like Wave Scalar [6] combined both static and dynamic dataflow analysis in order to exploit more parallelism. Later on with the tremendous advancements in chip integration, to solve the programming wall problem, MPSoC programming models such as StarSs [7] [8], Oscar [9] and CellSs [10] are taken into consideration. These models implicitly schedule work and data, thereby saving the efforts of programmer by explicitly managing parallelism. These models share conceptual similarities with the out-of-order superscalar pipelines, such as dataflow scheduling and dynamic data dependency analysis. The traditional algorithms, such as Scoreboarding and Tomasulo, can dynamically schedule the instructions for OoO execution and explore ILP with multiple arithmetic units.

Task Superscalar pipeline is an abstraction of out-of-order superscalar pipelines which can use processors as function units. It dynamically identifies task-level parallelism, detects intertask data dependencies, and executes the tasks out-of-order [11]. An object-based dataflow execution with data dependencies analysis method achieve even more dataflow-like execution and exploit higher degrees of concurrency. Using this model the parallel execution of statically-sequential programs is achieved. In a dataflow fashion, it dynamically parallelizes the execution of suitably-written sequential programs on multiple processing cores [12]. OoOJava is a compiler-assisted approach that uses developer annotations along with the static analysis to provide an easy-to-use deterministic parallel programming model. This method is based on task annotations that instruct the compiler to consider a code block for OoO execution [13].

MP-Tomasulo is a dependency-aware automatic parallel task execution engine for sequential programs. It detects and eliminates WAW and WAR inter-task dependencies in the dataflow execution by applying the instruction-level Tomasulo algorithm to the MPSoC environment. So this system operates tasks in OoO on heterogeneous units but it has the overhead of scheduling which could be reduced [14]. In order to reduce these overheads, Task-scoreboarding was developed. It is a data hazards detection engine for OoO task execution. It considers IP cores and processors as function units and treats tasks as abstract instructions. It can analyze intertask data dependencies at runtime and issues tasks to heterogeneous function units automatically with parameter renaming techniques [15].

PROPOSED DESIGN

The OoO task scheduler proposed here is intended to provide the high speed execution by solving the stalling problem in OoO execution due to data dependency at TLP. The proposed work, execution flow for it and the algorithm implementation is discussed in this section.

A. Proposed work

The distinctive heterogeneous MPSoC hardware consists of multiple processors and a variety of heterogeneous IP cores for dedicated applications to extract the task level parallelism. Accountable components of such systems are computing processors, hardware IP cores, scheduling processor, interconnect modules, memory and peripherals. This paper is going to focus on the scheduling processor part, because handling immense number of tasks is a very critical job. This proposed design is a parallel task execution model for the fast execution of such system at TLP. It supports out of order execution along with register renaming mechanism by dynamically scheduling the tasks.

The proposed block diagram for IP core implementation of task scheduler to support OoO execution in an MPSoC environment is shown in Fig. 1. The block diagram consists of two computing processors and three IP cores interfaced with the scheduling processor which uses scoreboarding algorithm for dynamic scheduling. Scoreboarding can provide a light-weight task hazards detection engine for OoO execution, its architecture is simple, which brings smaller scheduling overheads and for TLP, WAW and WAR data dependency do not encounter as much as at instruction level [15]. Therefore here scoreboarding algorithm is preferred instead of Tomasulo.

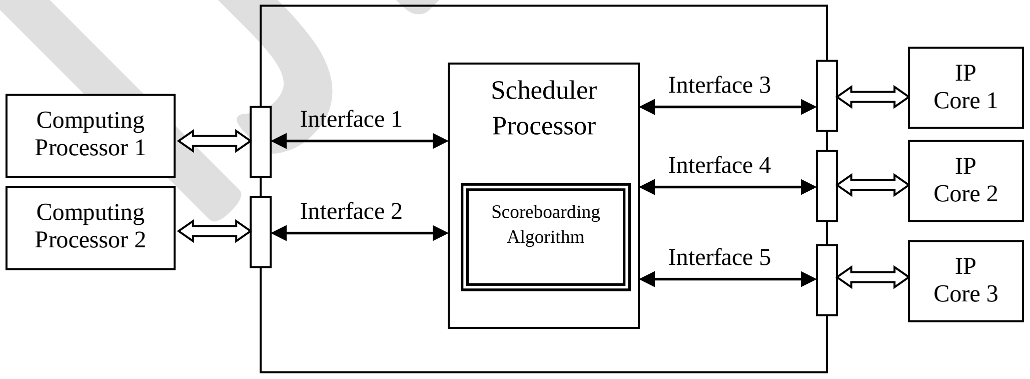

Fig. 1 Block diagram for IP core design of task scheduler to support OoO execution

B. Microarchitecture of scheduler processor

The microarchitecture of scheduler processor is shown in Fig. 2. The scheduler processor fetches the task-instructions from the register file which represents invocation of a task. It decodes each task-instruction and then schedules it to the computing processors and IP cores where the corresponding task is executed. Each task specifies its outputs and inputs to and from the registers in register file. After issuing the task-instruction, register is renamed by using register renaming technique. In this system a merged type rename buffer is used. A merged type feature is only a Register File that contains both the renaming (in-process) registers and architectural (retired) registers [16]. After renaming, the updated values are dispatched to the OoO execution unit. The task scheduler has its own scoreboard memory. The physical register file used here is of load-store type architecture. The tasks are then executed parallelly in the computing processors and IP cores. Then the result is written back in the register file and alternatively scoreboard memory gets updated.

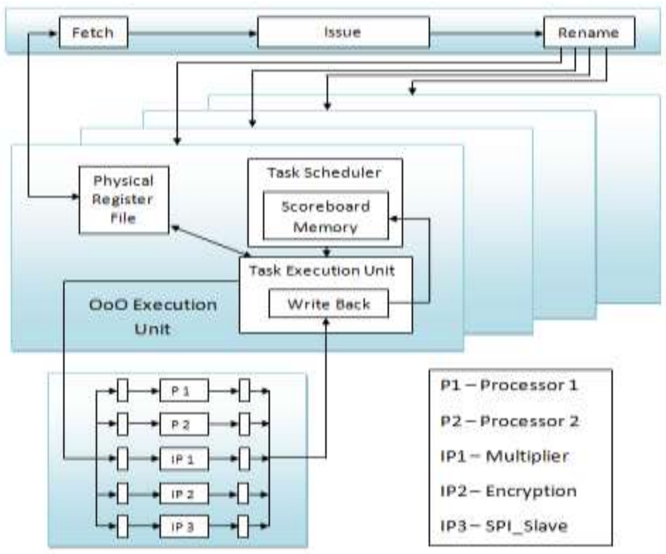

Fig. 2: Scheduler processor microarchitecture

C. *Execution flow for OoO scheduling*

The task sequence is issued and executed through seven stages: fetch, issue, rename, read operand, task partition, execute and write result. From these fetch, issue and rename are in-order stages. Whereas, read operand, task partition and execution stage are out-of-order stages. At last the result is again writing back in in-order. All these stages are similar to the instruction level pipelining, in addition to that register renaming stage is included as stage 3. Each task undergoes through these seven stages as shown in Table 1. The seven stages of execution flow are as follows:

1) *Fetch* – initially the instruction is fetched from the physical register file. Then it is decoded for further execution.
2) *Issue* – after decoding, the scoreboard issues the instruction to the functional unit (FU) and updates its internal data structure if a FU for instruction is free and no other active instruction has same destination register. After confirming that no other active FU wants to write its result into the destination register, WAW- Hazards are avoided. The instruction issue is stalled in the case of a WAW-Hazard or a busy FU.
3) *Rename* – this technique is used for dependency decoding. It assigns a unique storage location with each write-reference to a register.
4) *Read operand* – the scoreboard monitors the availability of the source operands. If no earlier issued active instruction is going to write the register, then that source operand is available and as soon as it gets available the instruction can proceed. In this way it resolves all RAW-Hazards dynamically and allows instructions to execute out of order.
5) *Task partition* – after finishing read operand stage, availability of function unit check is done and then goes for execution.
6) *Execute* – the required FU starts the execution of the instruction. As soon as the result is ready, it informs the scoreboard that it has completed the execution. If another instruction is waiting on the same result, it can be forwarded to the stalled FU.
7) *Write result* – the write back of the instruction is stalled on the existence of a WAR-Hazard until the source operand is read by the dependent instruction which is a preceding instruction in the order of issue.

TABLE 1 PROCESSING FLOW OF SCOREBOARDING

Task status	**Wait until**	**Bookkeeping**
Fetch		Fetch the data from register file

Issue	Not Busy [FU] and not Results [D]	Busy [FU] ←yes; Op [FU]←op; F_i [FU] ←D; F_j [FU] ←S1; F_k [FU] ←S2; Q_j←Result [S1]; Q_k←Result [S2]; R_j←not Q_j; R_k←not Q_k; Result [D] ←FU
Rename	Task issued	Rename with new register from register file
Read Operand	R_j and R_k	R_j ←No; R_k←No;
Task Partition	! ∀ Busy [FU]	Select the FU and replace table entries
Execute	Function unit done	Distribute tasks to function units
Write Result	∀f((F_j [f]≠F_i [FU] or R_j [f] = No) & (F_k [f] ≠F_i [FU] or R_k [f] =No)	∀f(if Q_j [f]=FU then R_j [f] ←Yes); ∀f(if Q_k [f]=FU then R_k [f] ←Yes); Result [F_i [FU]] ←0; Busy [FU] ←No

D. Algorithm implementation

Scheduling is very useful for the multi-cycle instructions, because when instructions are multi-cycled then only the scheduling will work more efficiently; otherwise, single cycle instructions does not need the scheduling. The scoreboard algorithm maintains three status tables to control the execution of the instructions:

- *Instruction Status*: Indicates the existing status of stages for each instruction being executed.
- *Functional Unit Status*: Indicates state of each functional unit. The function unit status is listed in Table 2. Each function unit maintains 9 fields in the table:

1. Busy: Indicates whether the unit is being used or not
2. Op: Operation to be performed in the unit (e.g. MULT, DIV, LOAD, ADD)
3. F_i: Destination register
4. F_j,F_k: Source-register numbers
5. Q_j,Q_k: Functional units that will produce the source registers F_j, F_k
6. R_j,R_k: Flags that indicates when F_j, F_k are ready

- *Register Status*: Indicates which function unit will write results into it for each register.

TABLE 2 Function unit status table

Name	Function Unit Status							
	Busy	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Load								

Multiplier								
AES_ENC								
SPI_slave								

RESULT

In order to solve the problem of stalling, the task scheduler is designed and developed in this research that supports OoO execution in an MPSoC environment. Five processing units are interfaced to the scheduler with two computing processors and three IP cores. The IP cores interfaced here are AES encryption, SPI slave and multiplier and computing processors are of RISC architecture. The task scheduler is designed in ModelSim software using Verilog language. As technology independent modeling is developed, it can be used as an IP core in any system for OoO execution and hazards detection.

Here four tasks are considered for the algorithm implementation as shown in Table 2. The experimental results are taken under four situations: no hazards, RAW, WAW and WAR. Task execution time and task scale, these two parameters are considered for performance evaluation. The task execution time denotes the entire execution time for different types of data hazards. Whereas, task scale is nothing but the total amount of different tasks or number of loop iterations.

The task scale is considered up to 1046 for result analysis. From the analysis it is found that this model is able to solve 25 % and 50 % RAW, WAW and WAR hazards and 100 % WAR hazards as shown in Fig. 3. Also from the RTL level simulation it is found that the model is able to solve the WAW and WAR dependencies at task level which has been avoided in case of ILP.

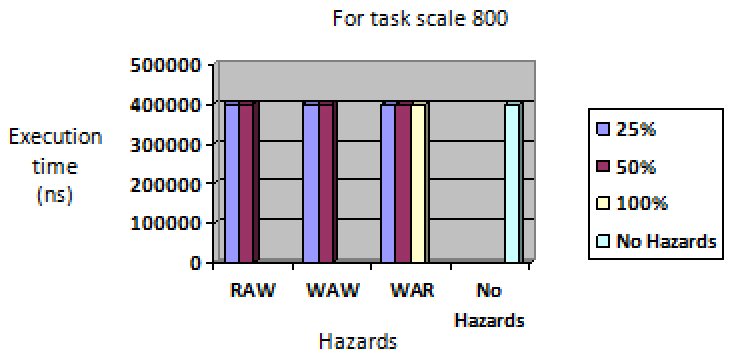

Figure 3: Result analysis for hazard detection

CONCLUSION

In this paper, a synthesizable IP core of task scheduler is designed in an MPSoC environment. The stalling problem in OoO execution is solved using dynamic scheduling scoreboard algorithm for TLP. The algorithm considers the abstract instructions as tasks and; processor and IP cores as function units. The model analyzes the inter-task data dependencies at runtime and after solving dependencies, it issues the tasks to function units. The experimental result shows that the designed IP core can support the OoO execution with data dependency resolution at TLP. It also resolves the WAW and WAR dependencies which are not possible at ILP. This designed IP core can be used in SoCs to support OoO execution for task scheduling. The work can be extended with the implementation of algorithm for superscalar processors.

REFERENCES:

[1] S. Borkar, and A.A. Chien, The future of microprocessors. Communications of ACM, 54(5): 67-77, 2011
[2] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. Journal of parallel and distributed computing, 37(1):55-69, 1996.
[3] Scott Hauck, Thomas W Fry, Matthew M Hosler, and Jeffrey P Kao. The Chimaera reconfigurable functional unit. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 12(2):206-217, 2004
[4] Tony Givargis and Frank Vahid. Platune: a tuning framework for system-on-a-chip platforms. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 21(11):1317-1327, 2002
[5] Georgi Kuzmanov, Georgi Gaydadjiev, and Stamatis Vassiliadis. The Molen processor prototype. 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 296-299, 2004
[6] Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. Wavescalar. Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture, 291-302, 2003
[7] Dallou, Tamer, and Ben Juurlink. Nexus++: A Hardware Task Manager for the StarSs Programming Model, 2011
[8] Josep M. Perez, Rosa M. Badia and Jesus Labarta. A Dependency-Aware Task-Based Programming Environment for Multi-Core Architectures. Cluster Computing, 2008 IEEE International Conference on. IEEE. 142-151, 2008

[9] A. Hayashi, Y. Wada, T. Watanabe, et al. Parallelizing compiler framework and API for power reduction and software productivity of real-time heterogeneous multicores. in Proceedings of the 23rd international conference on Languages and compilers for parallel computing. Houston, TX: Springer-Verlag. 2010

[10] Bellens, Pieter, Josep M. Perez, Rosa M. Badia, and Jesus Labarta. CellSs: a programming model for the Cell BE architecture. In *SC 2006 Conference, Proceedings of the ACM/IEEE*, pp. 5-5. IEEE, 2006

[11] Yoav Etsion, Felipe Cabarcas, Alejandro Rico, Alex Ramirez, Rosa M Badia, Eduard Ayguade. Task superscalar: An out-of-order task pipeline. 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 89-100, 2010

[12] Gagan Gupta and Gurindar S Sohi. Dataow execution of sequential imperative programs on multicore architectures. Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, 59-70, 2011

[13] James Christopher Jenista and Brian Charles Demsky. OoOJava: Software out-of-order execution. ACM SIGPLAN Notices, 46(8):57-68, 2011

[14] Chao Wang, Xi Li, Junneng Zhang, Xuehai Zhou, and Xiaoning Nie. MP-Tomasulo: A dependency-aware automatic parallel execution engine for sequential programs. ACM Transactions on Architecture and Code Optimization (TACO), 10(2):1-9, 2013

[15] C. Wang, X. Li, J. Zhang, P. Chen, Y. Chen, X. Zhou, and R. Cheung. Architecture support for task out-of-order execution in MPSoCs. IEEE Transactions on Computers, 1-14, 2014

[16] D. Capalija, & T. S. Abdelrahman, Microarchitecture of a coarse-grain out-of-order superscalar processor. Parallel and Distributed Systems, IEEE Transactions on, *24*(2), 392-405, 2013