2015 International Conference on Information Processing (ICIP)
Vishwakarma Institute of Technology. Dec 16-19, 2015

Task Dependency Aware IP Core for Dynamic Scheduling in MPSoC Environment

Ruchika Bamnote
Electronics and Telecommunication Engg.
D. Y. Patil College of Engg. Akurdi
Pune, India
ruchikabamnote@gmail.com

Priya M. RavaleNerkar
Electronics and Telecommunication Engg.
D. Y. Patil College of Engg. Akurdi
Pune, India
priyaravale@rediffmail.com

Dr. Mrs. Sulabha S. Apte
Professor and Head C.S.E. Dept., W.I. T. Solapur
University
aptesulabha@gmail.com

***Abstract*—This paper deals with Intellectual Property (IP) core design for dynamic task scheduling to support Out-of-Order (OoO) execution in Multiprocessor System-on-Chip (MPSoC) environment. MPSoC is one of the most promising future processor architecture. But such systems have to face challenges in the context of OoO execution during dynamic scheduling due to data dependencies like Read-after-Write (RAW), Write-after-Write (WAW) and Write-after-Read (WAR). Due to these dependencies stalling problem occur during OoO execution. In order to solve this stalling problem and achieve task level parallelism (TLP), Scoreboarding algorithm with register renaming technique is designed. As these dependencies impose challenging constraints on the direct use of techniques like OoO execution, register renaming and dynamic scheduling, a synthesizable IP core is designed. The simulation results show that the design can analyze all the task dependencies during run-time and resolves them at TLP. This algorithm is able to resolve 100% RAW, WAW and WAR hazards which are not solvable at instruction level parallelism (ILP).**

Keywords—Intellectual Property; Out-of-order execution; MultiprocessorSystem-on-chip; data dependencies; stalling; Scoreboarding algorithm; register renaming

I. INTRODUCTION

Multiprocessor system on chip is an emerging research area and has been into the main stream since last few years [1]. It is a platform that contains multiple heterogeneous processing elements with specific functionalities. The development of MPSoC begins from multi-core central processing units. However, due to heterogeneous instruction set architectures, software tool chains and programming interfaces, it has to face many challenges for efficiently designing and implementing a rapid prototype for various applications.

There are critical issues like computational capabilities, flexibility, scalability, programmability and power consumption which are very much important in the case of MPSoC platform. Such systems need fast processing which can be achieved using parallel execution and OoO execution is one of such techniques. Most high-performance microprocessors make use of instruction cycles that would otherwise be wasted by delay in OoO execution paradigm. While handling such situations, a processor executes instructions according to the availability of input data, instead of their original order in a program during OoO execution. Thus the performance improves more with less waiting time.

To remove one major limitation of in-order pipelining structure of stalling instructions behind them, OoO execution technique was developed. Dynamic scheduling with OoO execution is one of the effective methods in order to increase the speed of complex systems like MPSoC. That is why this study is focusing on multi-cycle OoO task executions. Unlike in-order execution, out-of-order execution has the capability to schedule ready instructions independently of long latency instructions. There are two effective methods, Scoreboarding and Tomasulo, for OoO instruction execution. Out of which this paper focuses on scoreboarding algorithm at TLP, which act as a data hazard detection engine. Reason for selecting scoreboarding algorithm instead Tomasulo is that it has a simple architecture as compare to Tomasulo and at TLP, WAW and WAR dependencies do not encounter as much as at instruction level [2]. Thus to solve the stalling problem in OoO execution, a task dependency aware IP core is designed for dynamic scheduling in MPSoC environment.

The paper is divided into five sections. Literature survey is done in section II. The methodology of proposed design, its execution flow and scheduler microarchitecture is discussed in section III. Section IV illustrates the experimental results. Finally, the paper is concluded in Section V.

II. REALATED STUDY

Rigorous research is being done on MPSoC regarding the critical issues like computational capabilities, programmability, flexibility, scalability and power consumption. Using parallel programming techniques, more efficient computational capabilities can be achieved. Such

978-1-4673-7758-4/15/$31.00 ©2015 IEEE

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 [3] to enhance ILP to TLP. Mostly it was focused on symmetric multiprocessor but this system does not support fully automatic parallelization. So the programmers have to handle 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 [4], Platune [5] and MOLEN [6]. Also models like WaveScalar [7] combined both static and dynamic data flow analysis in order to exploit more parallelism.

Later on with tremendous advancements in chip integration, MPSoC programming models such as StarSs [8], Oscar [9] and CellSs [10] are taken into consideration to solve the programming wall problem. These models implicitly schedule work and data, thereby saving the efforts of programmer by explicitly managing parallelism. These models share conceptual similarities with out-of-order superscalar pipeline techniques, such as data flow scheduling and dynamic data dependency analysis. Task Superscalar pipeline is an abstraction of out-of-order superscalar pipelines. It dynamically identifies task-level parallelism, detects intertask data dependencies and executes tasks out-of-order [11]. An object-based dataflow execution with data dependency analysis method achieve even more dataflow-like execution and exploit higher degrees of concurrency. In [12], a dataflow model dynamically parallelizes the execution of suitably-written sequential programs on multiple processing cores. OoO-Java is a compiler-assisted approach that uses developer defined annotations along with 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].

In [14], a microarchitecture of a coarse-grain superscalar processor was designed which executes in out-of-order manner. This multi-level computing architecture (MLCA) surrounded with a control processor (CP) was scalable and efficient in a coarse-grain context. In [15] both static and dynamic scheduling analysis was done in heterogeneous MPSoC system for checking the trade-offs during task dependency analysis. In this paper, dynamic form analyzes inter-task dependencies at run-time and static form obtains the dataflow graph in prior to the execution. MP-Tomasulo [16] is a dependency-aware automatic parallel task execution engine for sequential programs. It detects and eliminates WAW and WAR inter-task dependencies in dataflow execution by applying instruction-level Tomasulo algorithm to the MPSoC environment. Thus this system operates tasks in OoO on heterogeneous units but it has to handle overheads of scheduling which can be minimized. In order to reduce these overheads, Task-scoreboarding was developed in [2]. It is a data hazards detection engine for OoO task execution. It can analyze inter-task data dependencies at runtime and issues tasks to the heterogeneous function units automatically.

III. METHODOLOGY

The proposed OoO dynamic task scheduler is intended to provide high speed execution by solving data dependency problem and stalling problem. In this section proposed design, its execution flow and algorithm implementation is discussed [17].

A. *Proposed Design*

The classic MPSoC hardware platform consists of multiple processors and a variety of heterogeneous IP cores for dedicated applications to extract the task level parallelism. Typically such systems made up of multiple computing processors, hardware IP cores, scheduling processor, interconnect modules, memory and peripherals. Focus of this paper is on the implementation of dynamic scheduling algorithm for scheduling processor and not on the implementation of whole system-on-chip platform consisting of multiple computing processors and IP cores. Because, handling such immense number of tasks is a very big job. The proposed system is a parallel task execution model for the fast execution of system at TLP. The design supports out of order execution along with register renaming mechanism by dynamically scheduling the tasks.

The proposed block diagram of task dependency aware IP core design which supports OoO execution in 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. There are two effective methods for OoO execution at ILP: Scoreboarding and Tomasulo for solving data dependency problem. Out of which scoreboarding can handle only RAW hazards, whereas Tomasulo can resolve WAW and WAR along with RAW dependency at ILP. But Scoreboarding algorithm is chosen instead of Tomasulo because Scoreboarding can provide a light-weight task hazards detection engine for OoO execution and its architecture is simpler, which brings smaller scheduling overheads. Second, for task level parallelization, WAW and WAR doesn't encounter as much as at instruction level [2].

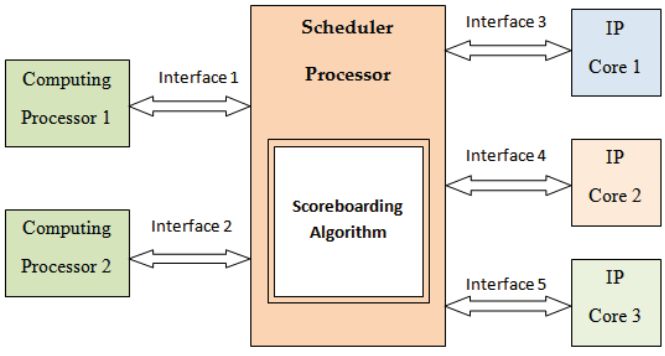

Fig. 1: Implementation block diagram for task dependency aware IP core design

B. Microarchitecture of scheduler processor

The microarchitecture of task dependency aware scheduler processor is shown in Fig. 2. The proposed design considers abstract instructions as tasks and; processor and IP cores as function units. The scheduler processor fetches task-instructions from the physical register file which represents an invocation of a task. It decodes each task-instruction and then schedules it to corresponding computing processors and IP cores where each task executed respectively. Each task-instruction specifies its inputs and outputs from and to 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 features only a Register File which contains both the renaming (in-process) registers and architectural (retired) registers [14]. After renaming, the updated values are dispatched to OoO execution unit. The task scheduler contains 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 physical register file and alternatively scoreboard memory gets updated.

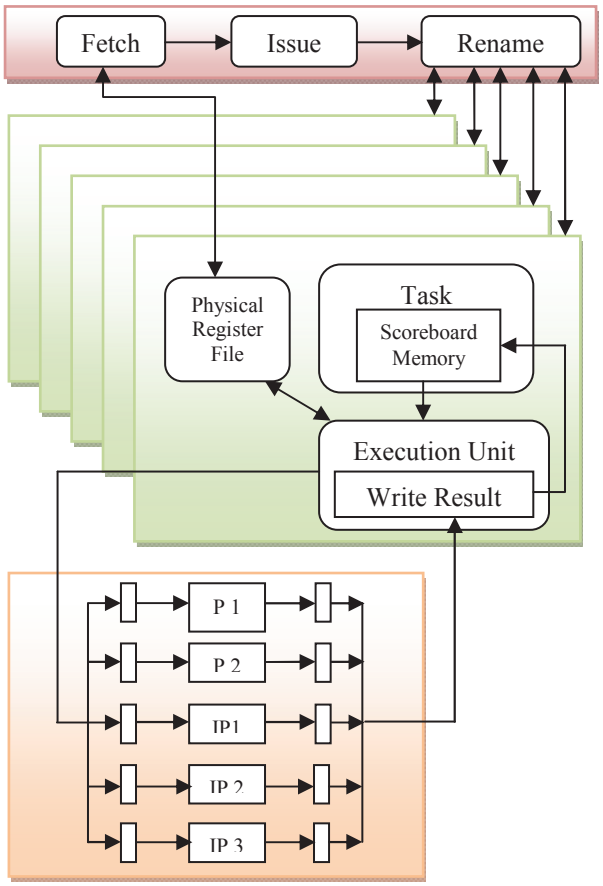

Fig. 2: Proposed scheduler processor microarchitecture

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 stages, fetch, issue and rename are in-order stages. Whereas read operand, task partition and execute are out-of-order stages. At last the result is again writing back in in-order to the register file. All these stages are similar to the instruction level pipelining scoreboarding algorithm, in addition to that register renaming and task partitioning stages are included as stage 3 and 5 respectively. Each task undergoes through these seven stages as shown in Table I. The seven stages are as follows:

1. **Fetch –** initially fetch the task from physical register file. Then it is decoded for further execution.
2. **Issue** – if a functional unit (FU) for execution of task is free and no other active task has the same destination register, the scoreboard issues the task to FU and updates its internal data structure. By ensuring 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 renamed register to source registers.
4. **Read operand** – the scoreboard monitors the availability of the source operands. If no earlier issued active task is going to write the register, then that source operand is available. If the operands are available, the task can proceed. This scheme resolves all RAW-Hazards dynamically. It allows tasks 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 task. As soon as the result is ready, it notifies the scoreboard that it has completed execution. If another task is waiting on this result, it can be forwarded to the stalled FU.
7. **Write result** – on existence of a WAR-Hazard, the write back of the task is stalled, until the source operand is read by the dependent task which is a preceding task in the order of issue.

D. Algorithm implementation

The aim is to design the task dependency aware IP core for task scheduler that supports OoO execution in MPSoC environment by solving stalling problem due to data dependency. In order to achieve this, dynamic scheduling scoreboarding algorithm is implemented in task scheduler. For memory storage, two memories are designed. One is 27 bit scoreboard memory and other is 36 bit wide main memory. The scoreboard algorithm maintains three status tables to control the execution of tasks:

TABLE I. PROCESSING FLOW OF SCOREBOARDING ALGORITHM

Task status	Wait until	Bookkeeping
Fetch		Fetch data from register file
Issue	Not Busy [FU] and not Results [D]	Busy [FU] ←yes; Op [FU]←op; Fi [FU] ←D; Fj [FU] ←S1; Fk [FU] ←S2; Qj←Result[S1]; Qk←Result [S2]; Rj←not Qj; Rk←not Qk; Result [D] ←FU
Rename	Task issued	Rename with new register from register file
Read Operand	Rj and Rk	Rj ←No; Rk←No;
Task Partition	! □ Busy [FU]	Select the FU and replace table entries
Execute	Function unit done	Distribute tasks to function units
Write Result	□f((Fj [f]≠Fi [FU] or Rj [f] = No) & (Fk [f] ≠Fi [FU] or Rk [f] =No)	□f(if Qj [f]=FU then Rj [f] ←Yes); □f(if Qk [f]=FU then Rk [f] ←Yes); Result [Fi [FU]] ←0; Busy [FU] ←No

- Task Status: Indicates the existing status of stages for each task being executed.
- Functional Unit Status: Indicates the state of each functional unit. The function unit status table is listed in Table II. Each FU maintains 9 fields in the table. Value in each field of table changes cycle by cycle. Table II shows the list of all fields:
  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. Fi: Destination register
  4. Fj,Fk: Source-register numbers
  5. Qj,Qk: Functional units that will produce the source registers Fj, Fk
  6. Rj,Rk: Flags that indicates when Fj, Fk are ready
- Register Status: Indicates which function unit will write results into it for each register.

TABLE II. FUNCTION UNIT STATUS

Tasks	Function Unit Status							
	Busy	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Load								
Multiplier								
AES_ENC								
SPI_slave								

IV. RESULTS AND DISCUSSION

The task dependency aware IP core of scheduling processor is designed in order to solve the problem of stalling during OoO execution in an MPSoC environment. Five processing units are interfaced to the scheduler with two computing processors and three IP cores. In this design, AES encryption, SPI slave and multiplier, these IP cores and two computing processors of RISC architecture are interfaced. The task scheduler is designed in ModelSim software using Verilog HDL. A technology independent modeling is developed so that it can be used in other systems also for OoO execution and hazards detection.

For performance evaluation of algorithm, several example test cases using specific functions of IP cores are designed. An example task sequence with 11 tasks is listed in Table III. The comparison of experimental and theoretical results of this task sequence is shown in Fig. 3. In this figure, x-axis refers to the length of task sequence listed in table and y-axis is running time of application in nanoseconds. There is a little gap between the curves which is because of scheduling overheads. Furthermore, Fig. 3 also depicts the OoO task execution. Task no. from 2 to 4 finishes in OoO manner. But tasks no. 4 to 5 are not executing in OoO manner because of dependency. Similarly, task no. 5, 6, 7, 8 and 9, no. 10 and 11 are also executing in OoO manner.

TABLE III. EXAMPLE TASK SEQUENCE

ID	Task Sequence
1	LOAD F6, R2
2	MULT F15, F5, F3
3	ENCRY F12, F6
4	SPI F11
5	LOAD F7, R4
6	MULT F2, F5, F15
7	LOAD F6, R2
8	SPI F3
9	SPI F7
10	LOAD F3, F7
11	ENCRY F12, F6

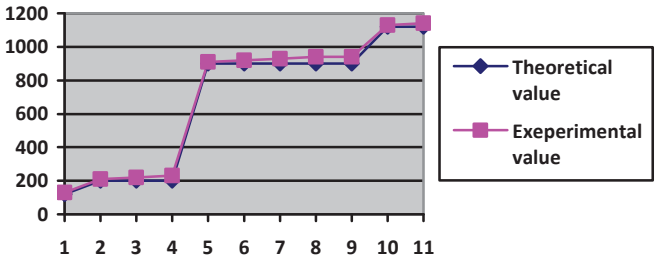

Fig. 3: Experimental results of example task sequence

For the algorithm implementation four tasks are considered as shown in Table II. 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.

For task dependency analysis, the task scale up to 800 is taken. From the analysis it is found that the proposed design is able to solve 25 %, 50 % and 100 % RAW, WAW and WAR hazards as shown in Fig. 4. In this figure x-axis denotes types of hazards and y-axis shows the total execution time needed in nanoseconds for all four situations. The simulation results shows that the design is able to solve the WAW and WAR dependencies at task level which has been avoided in case of ILP.

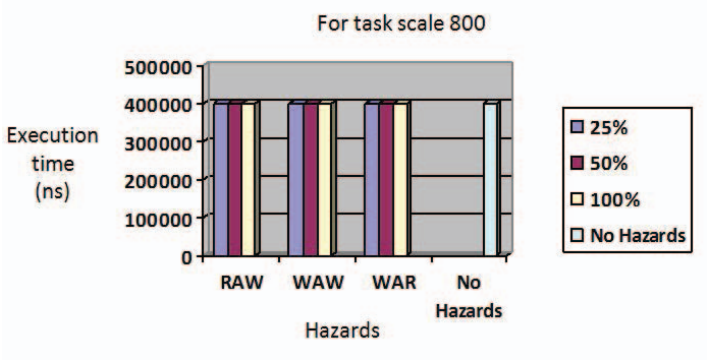

Fig. 4: Result analysis for hazard detection

V. CONCLUSION

A dependency aware IP core is designed for task scheduler in MPSoC environment in this paper. The stalling problem in OoO execution is solved using dynamic scheduling scoreboarding algorithm at TLP. The algorithm considers abstract instructions as tasks, and processor and IP cores as function units. The design successfully analyzes inter-task data dependencies at runtime and then resolves these dependencies. The experimental results show that the designed IP core can support OoO execution with data dependency resolution at TLP. Along with true dependency, it resolves the WAW and WAR artificial dependencies which are not possible to resolve at ILP using scoreboarding algorithm. This designed IP core can be used in SoCs for OoO execution during task scheduling by data dependency analysis.

References

[1] S. Borkar and A.A. Chien. The future of microprocessors. Communications of ACM, 54(5): 67-77, 2011.

[2] J. Zhang, P. Chen, Y. Chen, X. Zhou, C. Wang, X. Li and R. Cheung. Architecture support for task out-of-order execution in MPSoCs. IEEE Transactions on Computers, 64(5):1267-1310, May 2015.

[3] R. D Blumofe, C. F Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall and Y. Zhou. Cilk: An efficient multithreaded runtime system. Journal of parallel and distributed computing, 37(1):55-69, 1996.

[4] S. Hauck, Thomas W Fry, Matthew M Hosler and Jeffrey P Kao. The Chimaera reconfigurable functional unit. IEEE Transactions on Very Large Scale Integration Systems, 12(2):206-217, 2004.

[5] 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.

[6] G. Kuzmanov, G. Gaydadjiev and S. Vassiliadis. The Molen processor prototype. 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 18(7): 296-299, 2004.

[7] Andrew Schwerin Steven Swanson, Ken Michelson and Mark Oskin. Wavescalar. Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture, 8(3):291, 2003.

[8] Dallou, Tamer and Ben Juurlink. Nexus++: A Hardware Task Manager for the StarSs Programming Model, 11(5): 1-4, 2011.

[9] J. M. Perez, R. M. Badia and J. Labarta. A Dependency-Aware Task-Based Programming Environment for Multi-Core Architectures. IEEE International Conference on Cluster Computing,. 6(3): 142-151, 2008.

[10] P. Bellens, J. M. Perez, R. M. Badia and J. Labarta. CellSs: A programming model for the CellBE architecture. IEEE Conference, Proceedings of the IEEE, 12(4): 5-15. 2006.

[11] Y. Etsion, F. Cabarcas, A. Rico, A. Ramirez, R. M Badia and E. Ayguade. Task superscalar: An out-of-order task pipeline. 43rd Annual IEEE/ACM International Symposium on Microarchitecture, 15(7): 89-100, 2010.

[12] Gagan Gupta and Gurindar S Sohi. Dataflow 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] D. Capalija & T. S. Abdelrahman. Microarchitecture of a coarse-grain out-of-order superscalar processor. IEEE Transactions on Parallel and Distributed Systems, 24(2): 392-405, 2013.

[15] Xuehai Zhou, Qi Guo, Chao Wang and Xi Li. Static or dynamic: Trade-offs for task dependency analysis for heterogeneous mpsoc. In 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, 12(5): 903-910. IEEE, 2013.

[16] C. Wang, Xi Li, J. Zhang, X. Zhou, and X. Nie. MP-Tomasulo: A dependency-aware automatic parallel execution engine for sequential programs. ACM Transactions on Architecture and Code Optimization, 10(2):1-9, 2013.

[17] Ruchika Bamnote and Priya RavaleNerkar, IP Core Design of Task Scheduler to Support Out-of-Order Execution in an MPSoC Environment, International Journal of Engineering Research and General Science, 3(3): 435-440, June 2015.