Review on Dynamic Task Scheduling to Support OoO Execution in an MPSoC Environment

Ruchika Bamnote
ME Student
Department of Electronics and Telecommunication Engineering (VLSI & Embedded Systems)
D. Y. College of Engg. Akurdi, Pune, India

Priya M. RavaleNerkar
Assistant professor
Department of Electronics and Telecommunication Engineering (VLSI & Embedded Systems)
D. Y. College of Engg. Akurdi, Pune, India

ABSTRACT

In this work, we present a survey of the different task scheduling parallel programming models in order to support Out-of-Order (OoO) execution for high performance computing in an Multiprocessor System on Chip (MPSoC) environment. Thus, we review different parallel programming approaches, as well as current heterogeneous parallel programming models. In addition, we analyze different OoO execution architectures to solve the data dependency issues. The characteristics, strengths, and weaknesses are presented in all the cases. The study shows that the availability of multi-core CPUs has given new impulse to the OoO programming approach.

Keywords

Out-of-Order execution, MPSoC, heterogeneous parallel programming model.

1. INTRODUCTION

The past decades shows a tremendous invasion of MPSoC, especially in high-performance parallel computing domains. Since 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 [5]. But FPGA can provide a more flexible framework to construct prototypes for different applications as efficiently as compared to the GPU architectures. The integration of hundreds of cores into the current supercomputing machines is possible because of the remarkable evolution of heterogeneous multi-core research paradigms and the intrusion of reconfigurable hardware accelerators.

One of the most promising future processor architectures is considered as the combination of reconfigurable computing and multi-core technologies. However, critical issues like raw computational capabilities, programmability, flexibility, scalability and power consumption are becoming increasingly important. This scenario and raising demands have led to the emergence of FPGA based MPSoC which is composed of a variety of heterogeneous computational units. However, on heterogeneous MPSoC platforms task partitioning and scheduling approaches have encountered serious challenges, especially with intertask dependencies, including structural dependencies, Read-After-Write (RAW), Write-After-Write (WAW), and Write-After-Read (WAR) data dependencies. The task-level parallelism confined by different tasks using same source or destination parameters may cause tasks to run in sequence. Scoreboarding and Tomasulo are the traditional hazards detection and elimination solutions to address the data dependencies problem at instruction level. Both the algorithms provide OoO instruction execution engines when there are sufficient computing resources.

The structure of the paper is broke down as the following. Section 2 outlines different heterogeneous parallel programming models. Section 3 describes the task level parallelism with their different architectures. Section 4 presents the comparison of different task scheduling architectures available for OoO execution. Finally, the paper is concluded in Section 5.

2. PROGRAMMING MODELS

Due to the enhancement in GPU and FPGA based research approaches, reconfigurable heterogeneous hardware accelerators can achieve both high performance and promising flexibility along with increasingly speedups to diverse embedded systems and applications. Also the involvement of reconfigurable hardware platforms decreases the embedded system design time and space costs, as well as shortens the time-to-market (TTM) simultaneously very efficiently. But in particular programmability for MPSoC is still posing serious challenges. Programming models and middleware architecture support should invisibly fill the gaps between different architectures since the hardware is adapted to fit in the applications. Alternatively, there have been creditable MPSoC programming models such as StarSs [6] and CellSs [8], devoted to specific hardware architectures. StarSs and CellSs are the MPSoC programming models which help to attack the programming wall problem with the tremendous invasion of chip integration. These models implicitly schedule work and data, thereby saving the programmer of explicitly managing parallelism. These models share conceptual similarities with out-of order superscalar pipelines, such as dynamic data dependency analysis and dataflow scheduling.

2.1 StarSs

StarSs is a task-based programming model. Regardless of the target architecture it enables exploitation of task-level parallelism. In StarSs, the programmer has to identify pieces of code that can be executed as tasks, as well as their inputs and outputs; for that it provides programmers with pragmas which are annotations added to the serial code. Thereafter, the runtime system (RTS) determines the dependencies between tasks and schedules ready tasks onto worker cores. By enabling programmers to explicitly expose task side-effects, the StarSs programming model supports out-of-order execution of tasks through annotating operands of kernel functions as *input*, *output*, or *inout* (bidirectional). Thus the model can decouple the execution of the thread generating the tasks, from their decoding and execution.

2.2 CellSs

Cell superscalar (CellSs) is an alternative to traditional parallel programming models. Its main objective is to offer a simple and flexible programming model for parallel and heterogeneous architectures. It addresses the automatic exploitation of the functional parallelism of a sequential program through the different processing elements of the Cell BE architecture. A source to source compiler generates the necessary code based on a simple annotation of the source code and a runtime library exploits the existing parallelism by building at runtime a task dependency graph. The runtime deals with the task scheduling and data handling among different processors of heterogeneous architectures. Besides, in order to reduce the overhead of data transfers a locality-aware task scheduling has been implemented. Since the system complexity is growing, the problem of how to design a flexible programming model is becoming increasingly challenging. Both approaches provide superscalar or renaming techniques allowing OoO execution of tasks, but they are limited by their architecture requirements as it is hard to directly apply them to the reconfigurable MPSoC architectures.

3. TASK BASED PARALLELISM

Parallel task execution models have been studied for parallel computing machines during past decades. Taskbased parallel programming model are quite popular to enhance ILP to TLP, such as Cilk [12]. In order to significantly reduce the workload of programmers, this state-of the-art programming paradigm focuses on symmetric multiprocessors. But one of the major drawbacks of this approach is that automatic parallelization is not fully supported, which means programmers are required to handle the task mapping and scheduling schemes manually. Thus the speedup achieved is largely confined by the inadequate experiences of programmers. As a side effect, this could also increase the burden of programmers with synchronization and task scheduling on the symmetric multiprocessor architectures.

Meanwhile, compared to symmetric processors, heterogeneous processors are becoming increasingly dominating in embedded and high performance computing domains. One approach is to utilize reconfigurable FPGA platform and integrate acceleration engine, such as Chimaera [9]. Moreover, there are several creditable general FPGA research platforms, such as Platune [12] and MOLEN [10]. These studies focus on providing reconfigurable FPGA based environments with software tool chains to construct application specific MPSoC. Some other works like Wave Scalar [11] 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. Inter-task data dependency analysis and synchronization problem has posed a significant challenge in order to run tasks OoO for coarse-grained parallelization. Traditional algorithms, such as Scoreboarding and Tomasulo, explore ILP with multiple arithmetic units, which can dynamically schedule the instructions for OoO execution. In this section a brief overview of some of these papers is studied.

3.1 Task Superscalar

Task Superscalar pipeline [7] is an abstraction of out-of-order superscalar pipelines which operates at the task-level. It is a sequential program based framework that achieves function-level parallel execution. It eases runtime analysis of inter-task data dependencies, and out-of-order task execution. Task superscalar requires the user to identify function parameters using pragma (#) directives in which dependences may occur. It builds a dynamic task-flow graph in prior to parallel execution based on memory locations. It 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. Task superscalar generalizes the operational flow of dynamically scheduled out-of-order processors, and provides a native, task-based, dataflow execution engine. Task superscalar therefore combines the effectiveness of out-of-order processors in uncovering parallelism together with the task abstraction. Thus it provides a unified management layer for chip multiprocessors (CMPs) which effectively employ processors as functional units. The task superscalar pipeline dynamically detects intertask data dependencies, identifies task-level parallelism, and executes tasks out -of-order. While simultaneously simplifying the programming model, Task superscalar enables programmers to exploit many core systems effectively.

3.2 OoOJava

The OoOJava [4] is a compiler-assisted approach that leverages developer annotations along with static analysis to provide an easy-to-use deterministic parallel programming model. For out-of-order execution, OoOJava extends Java with a task annotation that instructs the compiler to consider a code block. As soon as the data dependences are resolved OoOJava executes tasks and guarantees that the execution of an annotated program preserves the exact semantics of the original sequential program. Therefore, annotations merely affect its performance and never affect the program’s correctness. OoOJava uses the results of disjoint reachability analysis to generate a handful of lightweight comparisons that allow it to safely dynamically extract parallelism even when the heap accesses cannot be statically determined to be disjoint. It combines this with a new value forwarding approach that is analogous to register renaming and eliminate write-after-write and write-after-read hazards for variables. Together, these techniques allow OoOJava to parallelize a wide range of programs while requiring few changes to sequential code. Thus OoOJava uses static analysis to discover variable and heap dependences. To extract fine-grained and unstructured parallelism, processors execute instructions out-of-order in this model. OoOJava adapts out-of-order execution techniques to parallelize code blocks in its software runtime and guarantees that the execution respects all dependences.

Program dependences can take two forms: control dependences and data dependences. OoOJava handles control dependences implicitly by constraining a task to have a single exit. A task may have a data dependence on another task through a variable or through conflicting heap accesses to the same object. Different dynamic instances of the same task access the same variables. Like register renaming in out-of-order hardware, a critical component of parallelization is to eliminate write-after-write and write-after-read hazards on variables by forwarding values directly to the consuming task. The task dependence relations are restricted to be only parent-child or sibling-sibling in nature by OoOJava. It enforces this structure by attributing a child's dependences to its parent and only retiring a task after all of its children have retired. This structure simplifies the management of dependences by localizing the problem; a parent manages the dependences between itself and its children. Moreover, it allows the implementation to parallelize dependence tracking and therefore supports scaling to systems with a large number of cores. But OoOJava have a strict strategy to serialize all I/O operations in the program and it prohibits parallelism when unrelated tasks access disjoint sets of file descriptors.

3.3 Dataflow execution model

Gupta and Sohi [3] introduced an object-based dataflow execution with data dependencies analysis method to achieve an even more dataflow-like execution and exploit higher degrees of concurrency. The parallel execution of statically-sequential programs is achieved by this model. In a dataflow fashion, it dynamically parallelizes the execution of suitably-written sequential programs on multiple processing cores. Hence the execution is significantly race-free and determinate. Thus the model facilitates the program development and yet exploits available parallelism. Parallel tasks which are nothing but the program functions are dynamically extracted from a sequential program and executed in a dataflow fashion on multiple processing cores using tokens associated with shared data objects. And it employs a token protocol to manage the dependences between tasks. Along with the token protocol, decentralized scheduler is also employed to handle WAW dependences. As the program is sequenced, dependent functions are postponed, and they are introduced into the deques after their dependences have been resolved. However, the WAW and WAR data hazards cannot be solved by renaming techniques in this model.

3.4 MP-Tomasulo

For MP-Tomasulo [2] is a dependency-aware automatic parallel task execution engine for sequential programs. MP-Tomasulo detects and eliminates WAW and WAR inter-task dependencies in the dataflow execution by applying the instruction-level Tomasulo algorithm to MPSoC environments, so that it operates out-of-order task execution on heterogeneous units. 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 returning the task. 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. With the task-adaptive partitioning and scheduling schemes, it can detect RAW, WAW, and WAR data dependencies automatically. Therefore MP Tomasulo can improve the task-level parallelism without burden to programmers. However MP-Tomasulo supports only the architecture in which IP cores are tightly coupled to the processor without shared memory access operations. The IP core is more like a hardware accelerator for specific tasks in such situation. Also updating the reconfigurable IP core is difficult to implement on hardware. MP-Tomasulo is also limited by several factors in eliminating program stalls besides the software implementation.

- *The inter-task parallelism degree* shows whether the independent tasks can be found to execute at large. Then the dynamic scheduling scheme can reduce no further stalls if each task relies on its predecessor.
- *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 bringing overheads when storing tasks and maintaining data concurrency among different entries.
- *The number and types of functional units* determine the impact of structural dependencies in the issue stage. The tasks will stall and no tasks can be issued if there are no more available function units, until these dependencies are cleared.
- *Task partitioning plans* determine the target function unit for each task. The performances evaluation of partitioning method demonstrated that a 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.

3.5 Task-Scoreboarding

Task-Scoreboarding is a data hazards detecting engine for OoO task execution [1]. Task-Scoreboarding treats processors and IP cores as function units and tasks as abstract instructions. At runtime, it can analyze inter-task data dependencies and issue tasks to heterogeneous function units automatically with parameter renaming techniques. Task score boarding allows tasks to execute out of order when the computing resources are sufficient and no data dependences present. Chao Wang, et al. have proposed a novel high level architecture support for automatic OoO task execution on FPGA based heterogeneous MPSoCs. It is composed of a hierarchical middleware with an automatic task level OoO parallel execution engine. The middleware is able to identify the parallel regions and generate the sources codes automatically which is incorporated with OoO layer model. In this framework, both Scoreboarding and Tomasulo algorithms at task level are applied. From these two approaches, Scoreboarding can obtain shorter scheduling overheads, but the WAW tasks can only run in sequences. In the meantime, Tomasulo algorithm can eliminate WAW data hazards by register renaming. Each function call of do_T_* is responsible for the intended behavior of the main program in the scheduler processor at runtime. At each call to these functions, the runtime will do the following actions:

- Analyze data dependency including RAW, WAR and WAW. The data dependency analysis is based on the parameters used by tasks.
- Eliminate the WAW and WAR dependencies by parameter renaming techniques.
- Identify the target function unit to run current task by a task mapping method.

At instruction level, Scoreboarding and Tomasulo are both effective methods for OoO instruction execution. The reasons for which Scoreboarding algorithm is chosen instead of Tomasulo are that the Scoreboarding can provide a light-weight task hazards engine for OoO execution. The architecture is simpler, which brings smaller scheduling overheads. And for TLP, WAW and WAR happens not as much as at instruction level. Most programmers are intended to use different parameters in case of WAR and WAW hazards. Therefore introducing a complex mechanism like Tomasulo is not just fair enough. Due to the scheduling overheads, the performance of both approaches (MP-Tomasulo and Task- Scoreboarding) gets larger than the ideal scenario, whereas MP-Tomasulo always achieves higher scheduling overheads than Task-Scoreboarding approach.

4. COMPARISION

Here the comparison of different OoO execution engines is done. Task Superscalar [7] proposes abstraction of OoO superscalar pipelines regarding processor as function units. A dataflow execution model that achieves parallel execution of statically-sequential programs is presented in [3]. In a data flow fashion, it dynamically parallelizes the execution of appositely written sequential programs, on multiple processing cores. OoOJava [4] is a compiler-assisted approach that leverages developer annotations along with static analysis to provide an easy-to-use deterministic parallel programming model. The method is based on task annotations that instruct the compiler to consider a code block for OoO execution. Although these approaches provide OoO engine by superscalar or renaming engines, they do not clarify on the adaptive mapping for general FPGA platform with reconfigurable IP cores, therefore the flexibility across different architectures is still have scope. MP-Tomasulo and Task-Scoreboarding algorithms provide flexibility. MP-Tomasulo [2] is a dependency-aware automatic parallel execution engine for sequential programs, but this model exhibits the overhead of the scheduling which could be reduced. Task-Scoreboarding [1] is a data hazards detecting engine for OoO task execution. Finally, the summary of state-of-the-art parallel execution engines is listed in Table 1.

5. CONCLUSION

This study presents the survey on dynamic task scheduling to support OoO execution in an MPSoC environment. In this paper different models which solves the data dependency issues in out-of-order execution is studied. Task Superscalar, dataflow execution model and OoOJava do not focus on the adaptive mapping for general FPGA platform with reconfigurable IP cores. Thus they have less flexibility. Whereas MP-Tomasulo and Task-Scoreboarding are flexible enough as it is applicable to FPGA with reconfiguration. But the overhead of the scheduling is more in case of MP Tomasulo as compared to Task-Scoreboarding. In future we can elaborate this study for different applications of heterogeneous MPSoC architectures.

Table 1. Comparison of models

Models	Strength	Weakness
Task Superscalar (7)	Support OoO automatic parallel execution	Not applicable to FPGA with reconfiguration
OoOJava (4)	An OoO compiler for Java runtime	Not applicable to FPGA with reconfiguration
Dataflow (3)	Race free and determinate parallel execution	Not applicable to FPGA with reconfiguration
MP-Tomesulo (2)	Support OoO automatic parallel execution and applicable to FPGA with reconfiguration	More scheduling overheads
Task-Scoreboarding (1)	Applicable to FPGA with reconfiguration with less scheduling overheads	Not adaptive for runtime

6. REFERENCES

[1] 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.

[2] 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):9, 2013.

[3] 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.

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

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

[6] Dallou, Tamer, and Ben Juurlink. Nexus++: A Hardware Task Manager for the StarSs Programming Model, 2011.

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

[8] 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. 5-15. IEEE, 2006.

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

[10] Georgi Kuzmanov, Georgi Gaydadjiev, and Stamatis Vassiliadis. The Molen processor prototype. 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 296-299, 2004.

[11] Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. Wavescalar. Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture, 291-302, 2003.

[12] 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.

[13] 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.