RECONFIGURABLE HETROGENIOUS MPSOC USING PARTICLE SWARM OPTIMIZATION

C.Anupriya
Final Year Student, Department of Electronics and Communication Engineering, Raja College of Engineering and Technology, Madurai, Tamilnadu, India

E.Sushmitha
Final Year Student, Department of Electronics and Communication Engineering, Raja College of Engineering and Technology, Madurai, Tamilnadu, India

E.Gopinath
Assistant Professor, Department of Electronics and Communication Engineering, Raja College of Engineering and Technology, Madurai, Tamilnadu, India

Abstract

Multiprocessor systems on chip have out –of order (OOO) execution schemes shows incredible promise for task-level parallelism. However the main challenge of the OOO execution lies in the analysis of the inter task dependences .In this paper, we address this challenge by applying the instruction-level score boarding algorithm at the task level by using Particle swarm Optimization algorithm (PSO).The basic PSO algorithm to optimize the task runtime by minimizing the make span of a particular task set, and in the same time, maximizing resource utilization. Due to the changes, we should increase the speed and performance of the circuit. We introduce both software – based static and dynamic implementations on top of a heterogeneous MPSOC prototyped on a Field Programmable Gate Array fabric.

Keywords *- Multiprocessors systems-on chip (MPSOC), Out of order (OOO) execution, score boarding algorithm, task level scheduling.*

1. Introduction

Multiprocessor systems on chip (MPSOC) have emerged as a solution in modeling embedded system because they provide acceptable tradeoff between the cost and performance of the system under design. With rising complexity of the modern embedded systems, the goal is to produce a design of high performance with time to market acceptable for consumer electronics. MPSOC is comprised out of various processing element and enable multiple HW/SW partitioning, mapping and scheduling option. Optimization and automation of the MPSOC system synthesis is a complex problem [2].RECONFIGURABLE computing has made major in road with the advent of large scale field programmable gate array (FPGAs). It is predicted that multiprocessor system on chip (MPSOC) will greatly improve computational capabilities of heterogeneous platform in the near future [1].Therefore we expect great promise by combining heterogeneous MPSOC with reconfigurable computing. Learning from current cutting-edge disciplines, heterogeneous MPSOC combine multiple components in a modular manner, (e.g.) microprocessor, digital signal processor

(DSP), or Intellectual property (IP) core/ accelerators. The execution of out of order(OOO) task poses significant challenge to schedule them and map them on the processing element.

At this level, the major drawback of current programming models is that programmer need to handle the task assignments manually, which would seriously increase the burden of programmer .the OOO execution of instruction is dealt with by scoreboarding and Tomasulo algorithm [2] at the instruction level. Both approaches provide techniques for OOO instruction execution when there are sufficient computational resources and no data hazards among instruction. The Tomasulo algorithm is more complex as it can also detect write after write (WAW) and write after read (WAR) hazards through register remaining. Consequently , as programming models require more experience from the programmers, the extension from the instruction-level scheduling algorithm to task level provides an alternate methodology to utilize MPSOC platform effectively .In this paper ,we extend the instruction –level score boarding algorithm to the level of task. The reason why algorithm are as follows

1. Scoreboarding provides a light-weight task hazards engine for OOO execution. The architecture is simpler which brings smaller scheduling overheads.
2. For task-level parallelization, WAW and WAR hazards do not happens as much as at the instruction level. Most programmers intend to use different parameters in the case of WAR and WAW hazards. Therefore, introducing a mechanism as complex as the Tomasulo algorithm is not necessary.

An OOO execution scheme for sequential task execution by using PSO algorithm an MPSOC platform is proposed in this paper. The main contributions are listed as follow

1. We apply a traditional-level score boarding algorithm at the task level. Here, we define specific tasks as assignments for functional units in MPSOC system (e.g.) IP cores and processor.
2. We compose task sequence with different dependences to test the performance of our approach and use Joint Photographic Expert Group (JPEG) encoding to represent a real life application in our case study.
3. A prototype system is implemented on an FPGA board as an experimental platform. Our result demonstrates the performance of the scoreboarding algorithm at the task level.
4. We propose both dynamic and static forms of the score boarding algorithm called runtime MP score boarding and task dependence analyzer (TDA).

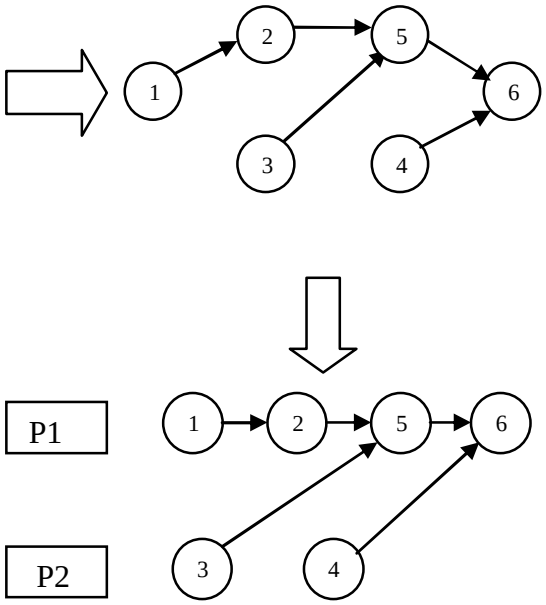
The remainder of paper is organized as follows: Section I present the problem of OOO execution on an MPSOC and the motivation of our approach. section II ,we discuss more details of our design including the hardware platform ,the programming, the software/hardware code sign flow, the score boarding process flow, and the design of static/dynamic scheme.

Section- I
Out-of-Order Execution Model

We follow the description of dataflow execution model in [26], which is extended to a general heterogeneous multi core computing scenario. Dataflow machines handle dependences using tokens to signal production and availability of data. We employ a similar technique, but make two crucial

changes. First, we associate tokens with objects instead of individual memory locations to match the data abstraction .Second, we assign each object multiple read tokens and a single write token to manage both production and consumption of data. In this description, data dependence will occur in the following scenarios.

1. *WAW Hazard:* Some tasks try to acquire write token of an object whose write token is held by another task.
2. *WAR Hazard:* Some tasks try to acquire write token of an object whose read tokens are held by some other tasks.
3. *RAW Hazard:* Some tasks try to acquire read token of an object whose write token is held by another task.



Dependence Graphs

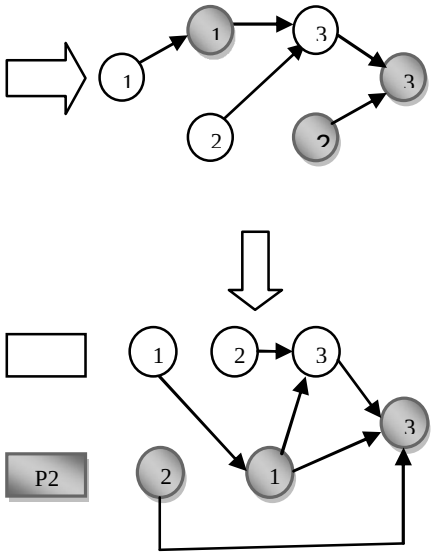
Tasks

1 :{c},{a,b};
2 :{d},{c};
3 :{f},{e};
4 :{h},{g};
5 : {i},{d,f};
6 : {j},{h,i};

Fig. 1 shows the basic OoO execution model on a platform with two general-purpose processors (*P*1 and *P*2). We abstract tasks as OP: {write set}, {read set} pattern, as shown in Fig. 1(a). As described above, OP stands for the operand of different tasks, and the write/read set contains the objects whose write/read token(s) should be acquired by tasks before execution. In Fig. 1, we indicate the objects as letters (e.g., *a*, *b*, and *c*). In general, we can divide the process into two steps. First, to obtain the task dependence graph, the dependence between tasks should be analyzed. Fig. 1(b) shows the data dependence between the tasks. For example, *T* 1 acquires write token of object

c, and thus it has an inter task data dependence [solid arrow lines in Fig. 1(b)] with *T* 2, which needs read token of object *c*. Likewise, *T*2 has a data dependence with *T*5 on object *d*, and *T* 3 has a data dependence with *T* 5 on object *f* as well. These dependences must be preserved if the dynamic execution is to maintain the sequential appearance of the static program. Second, tasks are mapped to a hardware platform according to the task dependence graph. Fig. 1(c) shows the execution of the code for the task execution model on dual processors. The dotted lines indicate the communication between processors. The execution starts with task *T* 1. Attempts to acquire a read token of object *a/b* and a write token of object *c* are successful. Hence, *T* 1 is submitted for execution on an available core (*P*1). Meanwhile, attempts to acquire a write token of object *e* and a read token of object *f* are successful, and thus *T* 3 is scheduled to execute (on core *P*2). The *T* 1 execution advances to *T* 2, which is shelved because a read token of object *c* cannot be acquired since *T* 1 is still being executed and holding the write tokens of object *c*. When the execution of *T*1 is finished, it will release the write token of object *c*, which means *T*2 now can be guaranteed a read token of object *c*. Therefore, *T* 2 can be executed once all its read tokens are ready (in this case, only the token of object *c*).

Considering MPSoC, however, it still poses significant challenges to improve task-level parallelism. To address this problem, OoO execution mentioned above is an effective way. Nevertheless, there are more problems presented, such as how to detect data dependences and how to map tasks to a heterogeneous MPSoC platform. However, most previous works have been done on a homogeneous platform. However, the task scheduling will become more complicated when the platform is heterogeneous two kinds of specific processors (*P*1 and *P*2), in which *P*1 can do tasks *T x* and *P*2 can do task Gx, respectively.



Picture 2 Dependence Graphs

Tasks
T 1 :{c},{a,b};
T 2 :{d},{c};
T 3 :{f},{e};

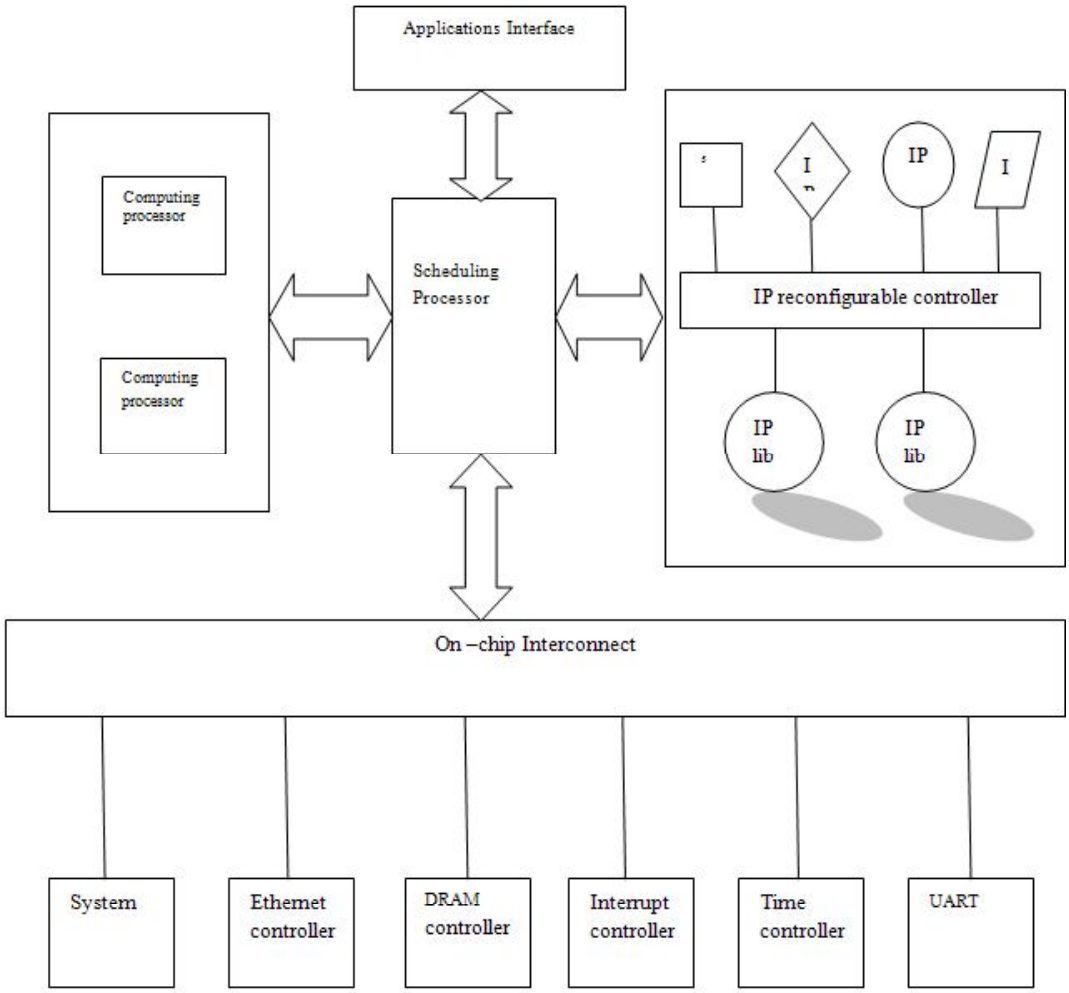
T 4 :{h},{g};
T 5 : {i},{d,f};
T 6 : {j},{h,i};

Compared to the basic OoO execution model in Fig. 1, it is clear that the step of dependence analyzing is same while the task mapping is different and more complicated. In addition, even without the dataflow dependences, structural dependences are also required to be taken into consideration. The hazards between dynamic instances of the same function may not manifest at run time, e.g., dynamic instances of the tasks *T x*, *T* 1, and *T* 2 can be issued at the same time only when there are adequate processors. How to handle these dependences is the key issue of the OoO execution on heterogeneous MPSoC. In our approach, we handle these dependences in two ways: dynamic and static ways. Both are based on the task-level score boarding algorithm, which is extended from instruction level. Subsequently, we do tradeoffs between dynamic and static manifestations.

Section –II
Design Specification
Hardware Architecture

To demonstrate our approach, we built an MPSoC prototype system on an FPGA board. Fig. 3 shows the components of target system.

1. One general-purpose processor serves as a scheduling processor, which is employed to perform task scheduling in dynamic scheme or to do preprocessing in a static way. The original task sequence will be sent to the scheduling processor through application interfaces.
2. Several general-purpose processors connected with the scheduling processor provide a run-time environment for software tasks. They can execute different types of software tasks.
3. Several heterogeneous IP cores serve as functional units (indicated with different shapes) are responsible for specific tasks to achieve acceleration. In particular, the IP cores can be reconfigured from IP library to fit in different applications.
4. The on-chip interconnect is used for data transmission between scheduler and other hardware blocks (computing processors, IP cores). Considering the frequent data exchanges between the scheduling processor and other computing elements, we select star topology with the scheduler-centric interconnect structure.
5. Other peripheral parts are connected through the bus-based interconnect, such as universal asynchronous receiver/transmitter, double data rate dynamic RAM (DDR DRAM), System ACE controller, Ethernet controller, and time and interrupt controller. Based on the hardware prototype, a task-level OoO execution scheme utilizing score boarding algorithm will be illustrated in the following sections.



Block Diagram

Software Design Flow

In the software design flow, programming model and libraries are provided for users to guide the implementation of source code with annotations. There are two kinds of source files, first is Main. c, which is the main program of the application, and the others are Apps.c, which are the application library on each computing processor. Then, the codes with annotations will be processed by the OoO compiler front-end, through which annotated codes and function libraries are merged into translated regular source codes. The regular source codes shall be further processed by the Xilinx software tool chain to generate the executable files. The executable file of main program will be executed on scheduling processor and the executable files of application libraries will be loaded into each computing processor. A certain function will be executed when requested by the main program. All the procedures are transparent to the user. What is required from programmers is to use simple annotations (e.g., #pragma) in the sequential program to indicate which parts of code will be executed concurrently. In the dynamic implementation, the tasks will be mapped to target functional units dynamically, and in the static way, some wait operations will be inserted to the sequential program based on the preanalysis

Scoreboarding Process at Task Level

This section describes how the task sequence is issued and executed using the scoreboarding scheme [24]. In our approach, we treat all computing resources (e.g., processors, IP cores) as functional units. Computing processors are special functional units on which every kind of tasks can be executed while only one kind of task can be executed on IP cores.

We follow the hardware first principle in hardware/software partition. That is to say, when the IP cores are free, task requests will be sent to IP core. Otherwise, they will be sent to processors. In the implementation, we assigned a unique priority number to each functional unit according to its performance, which are listed in Table I (the lower priority value, the higher its actual priority). The most important data structures of the task-level score boarding algorithm are three tables: an instruction status table to indicate the status (e.g., issued, read operands, task mapping, executing, and write result) of tasks, a functional unit status table (FUT) to indicate the status (e.g., busy, free, type, priority) of each functional unit, and a register status table (RST) to indicate the objects (implemented as registers) status. All these tables are derived from the instruction-level score boarding algorithm. For the execution flow, the algorithm is divided into five stages: issue stage, read operands stage, task mapping stage, execution stage, and write result stage. Table II showed the overview of the five stages and they are described in more detail in the following.

1)Issue: In this stage, structural and WAW hazards are detected. First, the RST is checked to make sure there are no other active tasks with the same destination variable to avoid WAW hazards. And then, to detect the structural hazards, the FUT is scanned in priority order (from high to low) to find an available functional unit for a certain task. Since the priority of IP cores are always higher than processors' due to their high performance, the IP cores will be chosen prior to processors. If both WAW and structural hazards are not detected, the task can be issued. Otherwise, the task will stall, and no further tasks can be issued until these hazards are cleared.

2) Read Operands: The score boarding monitors the availability of the source operands. If the operands are not yet available, the score boarding monitor will wait for the results. A source operand is available if no earlier issued active task is going to write it. When both operands are available, the task will be dispatched to a certain functional unit. The score boarding resolves RAW hazards in this step, and tasks may be executed OoO.

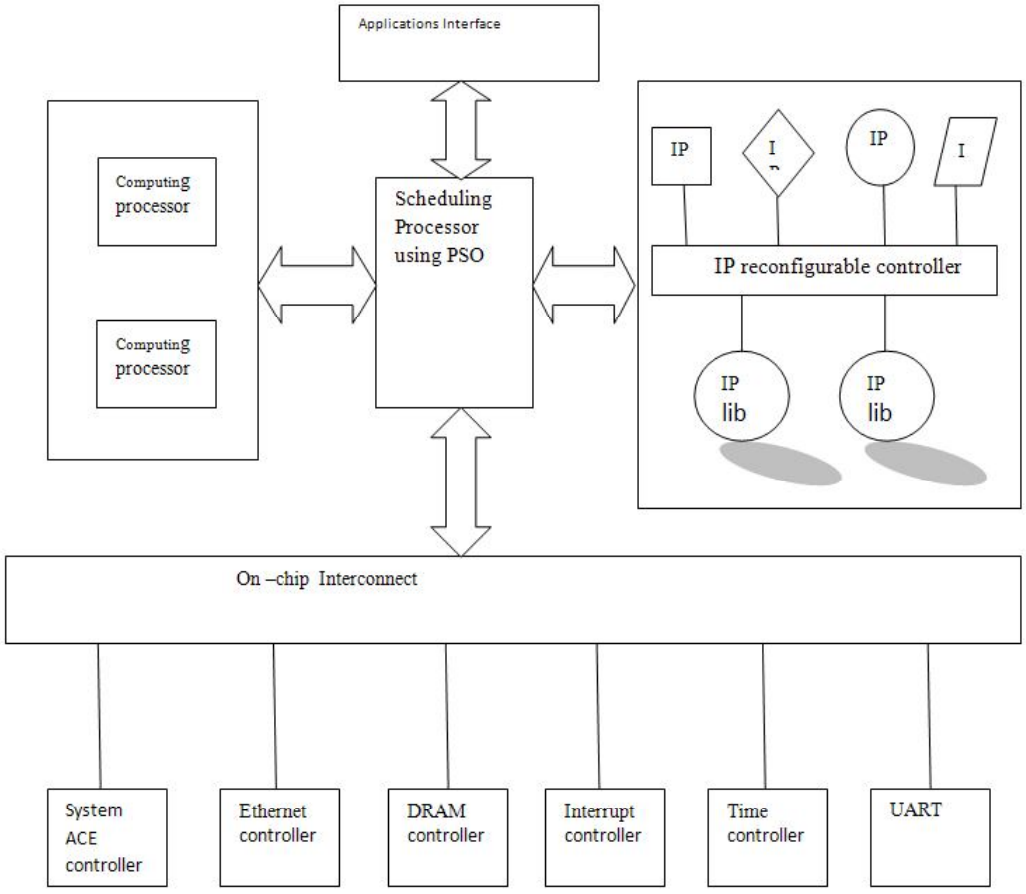
3) Task Mapping: In the issue stage, the allocated functional unit ensures that there are no structural hazards. However, after the read operands stage is finished, there may be other available functional units that have higher priority. In order to find some functional units with higher priority than the current one, the FUT is scanned in the predefined priority order. Then, the tasks are reallocated onto the functional units with higher priority. Therefore, this stage is also called task reallocation stage. The main purpose of this stage is to choose an appropriate functional unit on which the current task can finish as early as possible.

If the reallocated task and the task in issue stage are requesting the same functional unit, we give the tasks in the mapping stage priority and the tasks in the issue stage will stall until there is an available functional unit, because the task in mapping stage can finish execution earlier.

4) Execution*:* The functional unit begins execution once it has received its operands. When the result is ready, it notifies the scheduling processor that it has completed execution. Task distribution and data transfer are both performed through an on-chip interconnect. Using the hardware interconnects the execution results are returned through interrupts. One interrupt controller is integrated to detect interrupt request signals from all the interconnect channels. The interrupt handler assigns the variables with results. In our proposed architecture, since the results from different tasks may be transferred back at the same time, a first-come-first-serve policy is used to deal with interrupts, and no interrupt preempt is supported.

Section III
Proposed System
Scheduling based on Particle Swarm Optimization

In this section, we present a scheduling heuristic for dynamically scheduling workflow applications. The heuristic optimizes the cost of task-resource mapping based on the solution given by particle swarm optimization technique.



Particle Swarm Optimization (PSO)

Particle Swarm Optimization (PSO) is a swarm-based intelligence algorithm [8] influenced by the social behavior of animals such as a flock of birds finding a food source or a school of fish

protecting themselves from a predator. A particle in PSO is analogous to a bird or fish flying through a search (problem) space. The movement of each particle is co-ordinate by a velocity which has both magnitude and direction. Each particle position at any instance of time is influenced by its best position and the position of the best particle in a problem space. The performance of a particle is measured by a fitness value, which is problem specific. The PSO algorithm is similar to other evolutionary algorithms. In PSO, the population is the number of particles in a problem space. Particles are initialized randomly. Each particle will have a fitness value, which will be evaluated by a fitness function to be optimized in each generation. Each particle knows its best position p best and the best position so far among the entire group of particles g best. The p best of a particle is the best result (fitness value) so far reached by the particle, whereas g best is the best particle in terms of fitness in an entire population. In each generation the velocity and the position of particles will be updated as in Eq 6 and 7, respectively.

The PSO algorithm has been introduced by Kennedy and Eberhartin 1995 [6], [24]. According to the PSO algorithm uses two components:

a. The scheduling heuristic as listed in Algorithm 1
b. The PSO steps for task-resource mapping optimization as listed in Algorithm 2.

Algorithm *1:* Scheduling Heuristic

1: Calculate average computation cost of all tasks in all compute resources.
2: Calculate average cost of (communication/size of data between resources.
3: Set task node weight wkj as average computation cost.
4: Set edge weight ek1, K2 as size of file transferred between tasks.
5: Compute PSO ({ti}) /* a set of all tasks i k*/.
6: repeat.
7: for all "ready" tasks {ti}T do.
8: Assign tasks {ti} to resources {pj} according to the solution provided by PSO.
9: end for.
10: Dispatch all the mapped tasks.
11: Wait for polling time.
12: Update the ready task list.
13: Update the average cost of communication between resources according to the current network load.
14: Compute PSO ({ti}).
15: **until** there are unscheduled tasks.

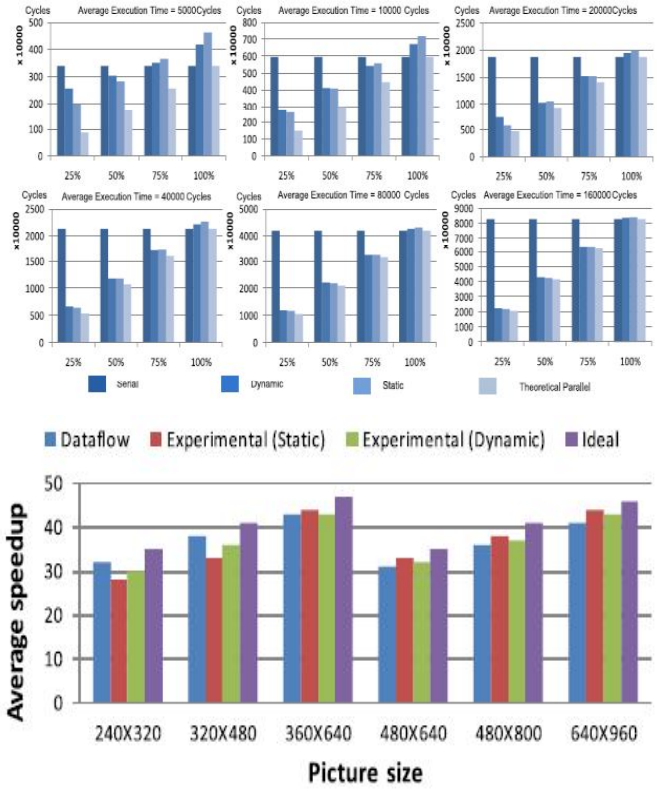
Algorithm 2*: PSO algorithm.*

1: Set particle dimension as equal to the size of ready tasks in {ti} T.
2: Initialize particles position randomly from PC =1... j and velocity vi randomly.
3: For each particle, calculate its fitness value as in Equation4.

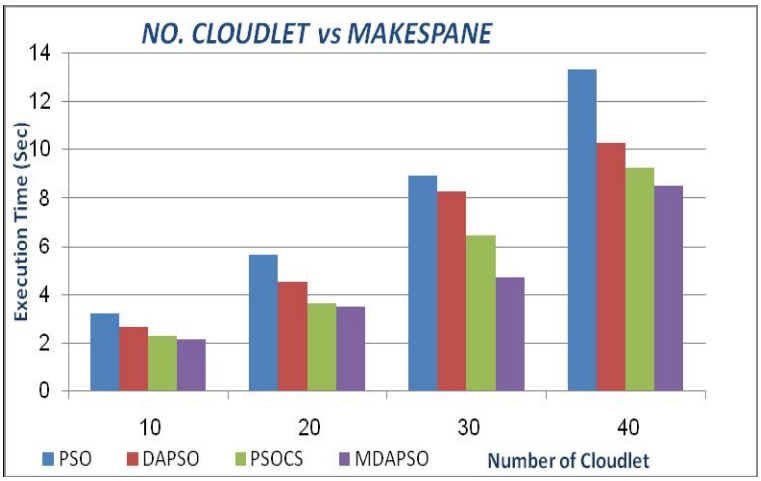
4: If the fitness value is better than the previous best pbest, set the current fitness value as the new pbest.
5: After Steps 3 and 4 for all particles, select the best particle as gbest.
6: For all particles, calculate velocity using Equation 6 and update their positions using Equation 7.
7: If the stopping criteria or maximum iteration is not satisfied, repeat from Step 3.

Graph Analysis

Every test case has been run both dynamically and statically. Presents the execution time of each test case. We take four pillars in one group to illustrate four kinds of time cost. The first pillar stands for time consumption of serial execution, that is to say, the tasks are sent serially. The second pillar indicates the execution time of task sequences using dynamic scheduling method. The tasks are sent to scheduling processor with run-time score boarding running on it. The third pillar represents the execution time of task sequences in static preanalysis way (the tasks are preprocessed by TDA before sent to hardware platform). The last pillar refers to the ideal parallel execution time of tasks ignoring overheads. There are four groups of pillars in a diagram indicating the test cases with different dependence degrees, ranging from 25% to 100%. In addition, the average execution time of one task has been configured from 5k to 160k cycles and one diagram stands for one configuration.



Speedup comparison. In each group, the left bar is the speedup using the dataflow execution, the middle bar is the practical speedup on our experimental platform, and the right bar is the ideal speedup using our platform.



Execution Time

Conclusion

According to this paper the application of the instruction-levels scoreboarding technique to the task level that is commonly executed in current-day heterogeneous MPSoCs. Furthermore, we introduced both static and dynamic implementations of scoreboarding in software. The static implementation is to obtain the dataflow graph prior to the execution. The dynamic implementation analyzes the intertask dependences at run time.

We prototyped our approaches on an FPGA platform with two scenarios. The first scenario entails four manually composed test cases with different dependence degrees, and the second scenario entails the JPEG encoding as a real-life application. Our experimental results show that our approach can achieve 95.75%–98.68% of the theoretical speedup. Finally, we performed an analysis such as when to choose for either of our approaches (static versus dynamic).

As future work, there are numerous directions worth pursuing. We plan to study more hardware extensions, investigate how the scoreboarding approach can be applied to high-performance DSP or GPU cores where multiple functional units or vectors may execute in massive parallel, and investigate how the scoreboarding scheme works under reconfigurable fabric conditions to further improve the performance.

References

1. R. M. Tomasulo, “An efficient algorithm for exploiting multiple arithmetic units,” IBM J. Res. Develop., vol. 11, no. 1, pp. 25–33, 1967.
2. J. Kennedy and R. Eberhart, "Particle swarm optimization”, in Neural Networks, 1995. Proceedings, IEEE International Conference on, vol. 4, (1995), pp. 1942-1948.
3. D. Koch, C. Beckhoff, and J. Teich, communication architecture for complex runtime reconfigurable systems and its implementation on spartan-3 FPGAs,” in Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays, 2009, pp. 253–256.
4. E. Lübbers and M. Platzner, “ReconOS: Multithreaded programming for reconfigurable computers,” ACM Trans. Embedded Comput. Syst., vol. 9, no. 1, 2009, Art. ID 8.