Colored Petri Net model with automatic parallelization on real-time multicore architectures

Chao Wang \textsuperscript{a,b}, Xiaojing Feng \textsuperscript{a,b}, Xi Li \textsuperscript{a,*}, Xuehai Zhou \textsuperscript{a}, Peng Chen \textsuperscript{a,b}

\textsuperscript{a} School of Computer Science, University of Science and Technology of China, Hefei, China
\textsuperscript{b} Suzhou Institution of USTC, Suzhou, China

\textbf{Abstract}

This paper proposes a novel Colored Petri Net (CPN) based dynamic scheduling scheme, which aims at scheduling real-time tasks on multiprocessor system-on-chip (MPSoC) platforms. Our CPN based scheme addresses two key issues on task scheduling problems, dependence detecting and task dispatching. We model inter-task dependences using CPN, including true-dependences, output-dependences, anti-dependences and structural dependences. The dependences can be detected automatically during model execution. Additionally, the proposed model takes the checking of real-time constraints into consideration. We evaluated the scheduling scheme on the state-of-art FPGA based multiprocessor hardware system and modeled the system behavior using CPN tools. Simulations and state space analyses are conducted on the model. Experimental results demonstrate that our scheme can achieve 98.9% of the ideal speedup on a real FPGA based hardware prototype.

\textbf{1. Introduction}

Real-time multiprocessor system-on-chips (MPSoCs) are commonplace recently, and the number of processors on an MPSoC is growing steadily. However, the performance potential of MPSoCs cannot be tapped out unless applications running on them have been highly parallelized. One common approach of parallelizing applications is task scheduling. Task scheduling in real-time multiprocessor systems is to assign tasks to different processors, allowing them execute in parallel so that all time constraints imposed on tasks are satisfied.

Parallel programming models have been widely used to exploit parallelism, such as OpenMP [1], MPI [2], Intel's TBB [3], CUDA [4], OpenCL [5] and Cilk [6]. They are perceived to result in high performance gains. However, the major drawback of these models is that they impose too many burdens to programmers, and therefore lead to low application-development productivity. To lighten the burden of programmers, automatic parallelization technologies have been intensely studied. Recent studies such as task superscalar [7] explore a new research direction into extracting Task Level Parallelism (TLP).

There are numerous approaches for modeling a scheduling system on MPSoCs, such as directed graph, Petri net, UML activity diagram and so on. Of these approaches, model based design provides a promising approach for tackling these problems. Benefiting from verification and simulation on models, design errors can be detected as soon as possible and various paradigms of designs can be evaluated in early design phases. Furthermore, by performing time-based simulations on models, any violation of time constraints can be revealed prior to the implementation of real-time systems.

Model based design methodology has been widely applied in solving task scheduling problems for real-time multiprocessor systems. Nonetheless, little research has been conducted on modeling dynamic scheduling schemes. It is mainly because there are two vital behaviors hard to describe using modeling languages:

1. \textit{Dependence detecting}: In a dynamic scheduling scheme, tasks are usually reordered to perform out-of-order execution for exploiting task-level parallelism. However, the scheduling scheme can lead to correct task execution order only when all inter-task dependences are maintained. It demands that system models are able to describe and detect different types of dependences at run time. Since the out-of-order task execution is characterized by concurrent and asynchronous, it is difficult to model the dynamic behavior of dependence detecting. Most related work takes inter-task dependency as a priori, while rarely addresses the problem of dynamic dependence detecting.

2. \textit{Task dispatching}: Task dispatching here refers to assigning a particular processor to a task when there are multiple processors capable to execute it. Task dispatching strategy also has significant effect on system performance, since the ex-
cution and communication costs of each individual task vary among different dispatching strategies. In a dispatching strategy, the mapping relationship between tasks and processors must be established and modified dynamically. Therefore, modeling the strategy is challenging, especially on the heterogeneity MPSoCs architectures.

In this paper, we construct CPN-based models for Task-Level Score boarding, and consider both partitioning and dependence detection simultaneously. Simulation experiments are performed on CPN Tools [8] finally.

Furthermore, we also implement a prototype system on a single Xilinx Virtex-5 FPGA board and perform a series of experiments to compare the performances between task superscalar and our proposed Task-Level Scoreboarding.

We claim the following contributions:

1. This paper presents a scheduling scheme for MPSoCs, called Task-Level Scoreboarding, which can automatically detect inter-task dependences and extract TLP by allowing tasks to execute out-of-order. Compared with state-of-art approaches, our proposed approach introduces lower time overhead and achieves more significant performance.

2. We construct a CPN-based model for the proposed scheduling scheme. The model can identify different types of dependences and simulate both scheduling and partitioning processes of tasks. Besides, various partitioning strategies can be evaluated based on the model. With the help of our CPN model, different design paradigms can be conveniently evaluated. This will lead to significant reduction on time-to-market.

The rest of this paper is organized as follows. Section 2 summarizes previous work related to task scheduling problems and relevant modeling methods. Section 3 presents the Task-Level Scoreboarding scheduling scheme. The CPN model of Task-Level Scoreboarding is elaborated in Sections 4 and 5 presents the heterogeneous multiprocessor prototype and experimental results. Finally, Section 6 concludes this paper and introduces future work.

2. Related work

In order to tap out the performance potential, there have been many approaches proposed to parallelize the workloads of multiprocessor systems. Recent works employ hardware techniques to reduce runtime overhead, such as carbon [9], ADM [10], task superscalar [7,11]. Especially, task superscalar proposes an abstraction of out-of-order superscalar pipelines. With the support of renaming mechanism, task superscalar is able to eliminate WAR and WAW hazards dynamically and thereby extract more parallelism at task level. However, the runtime time overhead introduced by renaming mechanism is considerable. For application workloads where WAW and WAR hazards do not arise frequently, the objective of applying renaming mechanism is not justified enough.

Directed and undirected graphs provide an intuitive approach to model multiprocessor tasks informally. Various directed and undirected graphs based models for scheduling algorithms are surveyed in [12]. In these models, nodes in a graph are used to represent tasks, while the arcs represent precedence constraints or inter-task interactions. These graphs have the virtues of simple structure and graphical representation. However, although they are competent to modeling the workflows, they do not have the capability to describe data-flow and control-flow for modeled systems. Furthermore, since the modeling languages lack of formal definitions, these models usually lead to inconsistency and ambiguity in system specification. Therefore, there is a need to resort to formal methods, especially in context of complex system designs.

A timed automaton is a finite-state machine equipped with time concepts. It supports modeling of times by annotating state-transition graphs with clock variables and time guards. Transitions in an automaton are conditioned by time guards which compare clock variables with time constants, and firing a transition can affect the values of selected clock variables. This property enables timed automata to model time-dependent systems. When timed automaton was first introduced by Rajeev Alur and David Dill [13], its expressive power was strictly limited. Nevertheless, a lot of efforts have been made towards extensions of timed automata. For example, weighted/priced timed automata were introduced independently in [14,15], which extend cost information on locations and transitions. The timed automata in [16] is extended with deadlines and release times which are two common features in scheduling problems. These extensions increase the expressive power of original timed automata and are employed to model systems in, among others, scheduling problems. To name a few, the extended timed automata model in [16] is adopted in solving the problem of scheduling partially-ordered tasks on parallel machines, while weighted/priced timed automata are applied to optimal scheduling and planning problems in [17]. In these timed automata models, each task and resource must be represented by a single automaton. Since the model structures remain fixed during the execution of models, certain applications which require dynamic creation of new tasks cannot be modeled using timed automata. Furthermore, it is hard to use timed automata to model concurrent systems with shared resources [18]. Due to this limitation with respect to modeling power, the application scope of timed automata is greatly restricted.

On the contrast, Petri nets, especially CPN, have received much attention for modeling scheduling processes on multiprocessor platforms. For instance, Zuberek et al. model the scheduling of multiple tasks on distributed-memory multiprocessors using CPN [19]. The proposed model can be utilized to evaluate the influence of different model parameters on the system performance. In [20], CPN is used to build a model which formally describes the behavior of task distribution and execution within the grid environment. Based on the analysis of the model, the grid service reliability can be evaluated. Reference [21] studies the task scheduling of a robot system with temporal constraints, using timed Petri nets. [22] presents a Petri net based model of task scheduling on dynamically partitioned multiprocessor systems and performs a series of sensitivity analyses on the model. However, none of these models take inter-task data dependences into consideration.

Tavares et al. propose a model based scheduling scheme for multiprocessor systems with timing and energy constraints [23,24]. In the scheme, multiprocessor tasks are modeled using timed Petri nets. The model can describe precedence/exclusion relations among tasks. Hoheisel et al. develop a Petri net based model for workloads in the Fraunhofer Resource Grid (FhRG) environment [25,26]. Their model also considers the precedence constraints on grid tasks. Eskinazi applies timed Petri net within a reconﬁgurable environment and proposes a Petri net model responsible for task dispatching and relocation [27]. Dodd presents a CPN model for real-time task scheduling system of Seahawk helicopter [28]. The model is capable to monitor input/output resources of each task and detect inter-task dependences. However, it does not take task dispatching into account, since the modeled tasks have been statically bound to processors.

To the best of our knowledge, there are no Petri net based scheduling schemes addressing dependence detecting and task dispatching simultaneously. This paper takes both of these problems into
account and proposes a CPN based scheduling scheme. The details of our proposed scheme will be presented later in this paper.

3. Task-Level Scoreboarded scheduling scheme

This section presents the Task-Level Scoreboarding scheme. The description focuses on the architecture and processing flow of the scheme. At the end of this section, the advantages of our Task-Level Scoreboarding scheme are presented.

3.1. Architecture overview of Task-Level Scoreboarding scheduler

The block diagram of Task-Level Scoreboarding scheduler is outlined by dashed lines in Fig. 1. The scheduler takes preprocessed tasks as input, and buffers them in the task queue. Then the scoreboard controller fetches tasks from the queue in order, schedules them, and dispatches them to different processors for out-of-order execution. Processor Status Table is utilized to record the statuses of all processors, while the Variable Result Table indicates which processor will produce the pending result. There are two tables help scoreboard controller to identify inter-task dependences. Monitor module collects the running information of all processors and update the content of those two tables immediately upon perceiving a state change. Under the control of the scheduler, all processors exchange data with shared memory through on-chip interconnection. Note that the scheduler can be implemented either as a software module running on a GPP, or as a standalone hardware module attached to on-chip interconnection.

3.2. Processing flow of scoreboard controller

Under the control of the scoreboard controller, every task will undergo the stages shown in Table 1.

3.2.1. Check destination

A task moves to this stage immediately when entering scoreboard controller. In this stage, the readiness of destination variable is checked. The destination is ready if no other active tasks have the same destination variable. Otherwise, a WAW hazard will be detected. The task will move forward to the next stage if WAW hazards do not exist or they are resolved.

3.2.2. Partition

As mentioned before, there may be more than one suitable processor which a task can be dispatched to. Therefore, a partitioning strategy must be employed to choose one as target processor in this situation. First of all, the controller looks through the statuses of all processors to check whether there are any free processors capable to execute the task. If yes, the controller will substitute the value of the variable PAR with the identifier of the target processor. Otherwise, the task will stall until any suitable processor becomes free.

3.2.3. Issue

The absence of WAW and structural hazards is guaranteed when a task enters issue stage. The controller will update related table entries when a task issues.

3.2.4. Read operands

A source variable is ready when no earlier issued tasks intend to write it. There must be a RAW hazard if either a source variable is not ready. The scoreboard controller keeps on monitoring source variables of issued tasks. Once all the source variables are ready, the processor will access the memory to fetch source operands.

3.2.5. Execution complete

The processor will begin to execute immediately when source operands have been read.

3.2.6. Write results

A WAR hazard arises when any source variable of earlier issued tasks is just the destination variable of the completing task. A processor will write its result back only when there are not any WAR hazards. After that, a task is accomplished.

3.3. Advantages over instruction level scoreboarding

In this section, we present the advantages of our Task-Level Scoreboarding over instruction level scoreboarding, which includes two aspects:

(1) Task partitioning: When extending traditional scoreboarding to operate on MPSoCs, we need consider how to partition tasks to multiple processors. To the best of our knowledge, few literatures consider task partitioning together with out-of-order execution. In our proposed scheme, it is convenient to evaluate different partitioning strategies, since the partition stage is loosely coupled with the scheduling process. For demonstration, we will present how to integrate a greedy partitioning strategy to our Task-Level Scoreboarding in Section 4.5.

(2) Monitoring and profiling: Our Task-Level Scoreboarding monitors and collects the running information of processors, so it is able to trace the whole execution process and locate where the hotspot is through profiling techniques. When applied on reconfigurable platforms, using the profiling result, our Task-Level Scoreboarding can guide the process of hardware reconfiguration at runtime to achieve higher performance.

4. CPN models of Task-Level Scoreboarding

In order to facilitate the verification and evaluation of Task-Level Scoreboarding we proposed, CPN models are built. This section concentrates on the structure and execution process of our model. We discuss how the places and transitions in CPN are built, how the inter-task dependences are identified, and finally how the partitioning strategies are modeled. Besides, taking a greedy algorithm...
as the example, we present how to integrate a partitioning strategy into our model. Note we focus on the detection method of inter-task dependences, therefore the CPN model could be applied with other scheduling policies, such as fixed priorities, EDF, round-robin schemes.

### 4.1. Overview of CPN models

Fig. 2 illustrates the sketch map of the CPN model for a task. The whole model is divided into six parts by dotted lines. Each part represents an individual processing stage, and the name of each stage is denoted by italic letters on the left side of the figure.

In a CPN model, each token can be attached with a value. This value is called the token color. Tokens are distinguished from each other by their colors. Besides, all tokens are accommodated in places. As shown in Fig. 2, the place FU contains three tokens whose colors are represented by strings ("dec", "enc" and "idct", respectively). Likewise, VAR contains 32 tokens and the color of each token is an integer between 0 and 31. In addition, the token color can be a complex value such as a product or a list. For example, the color of the token (task, d, i, j) has a color which is a product of four elements. For convenience, we use the token color to represent a token. For instance, the token d refers to a token which is colored in d.

The behavior of a system can be described by system states and events which cause state changes. It is the number of tokens and the token colors in each individual place who represent system states, while the events are represented by transitions. A transition is said to be enabled when a token of a given color is ready in each input place (a place that has an arc directed towards the transition). Besides, a transition may fire representing the event actually takes place. When a transition fires, it consumes tokens in input places, and produces new tokens in output places (those places that have an arc starting from the transition).

When a transition fires, the color of a token produced or consumed is indicated by arc inscriptions. Some arc inscriptions are drawn besides arcs, while others are omitted since their values have no effect on understanding the model. The arc inscription "dec", "enc" and "idct" are constants, while d, i, j and par are all variables whose values may vary in different situations. In addition, a transition can have an action function which determines the variable values in related arc inscriptions.

It is notable that a CPN model can involve the time concept. When imported the time concept, the execution of a CPN model is controlled by a global clock. Besides, each token has an attached time stamp which indicates the time at which the token is ready to be consumed. A transition will not fire unless all needed tokens have time stamps which are no greater than the current value of the global clock. What is more, each transition has a duration inscription for representing the time taken to execute events. When a transition fires, it produces tokens with a new time stamp which is the sum of the firing time and the duration of the transition. These features make it possible to apply our models for time-based simulation.

### 4.2. Interpretations for places

In Fig. 2, each place is drawn as either a circle or an ellipse. The places drawn as ellipses can be interpreted as system resources which are shared by all tasks, while the implication of each place drawn as a circle is a precondition or a post condition of a related transition.

The place VAR can be interpreted as the abstraction of the Variable Result Table, and each token in VAR represents a free variable. Similarly, a token in the place INH also represents a variable. However, the presence of a token in INH indicates that the corresponding variable is about to be read.

The place FU models the Busy field of Processor Status Table. Each token contained in FU represents a free processor. A token

---

**Table 1**

Processing flow of scoreboard controller.

<table>
<thead>
<tr>
<th>Task status</th>
<th>Wait until</th>
<th>Action or bookkeeping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check destination</td>
<td>Once entering scoreboard controller</td>
<td>While (results[0])</td>
</tr>
<tr>
<td>Partition</td>
<td>Not results [D]</td>
<td>PAR -- target processor determined by the partitioning strategy</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FPAR[PAR] -- S1; FPAR[PAR] -- S2;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Qj -- result[51]; Qk -- result[52];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rj -- not Qj; Rk -- not Qk; result[31] -- PAR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rj -- No; Rk -- No</td>
</tr>
<tr>
<td>Read operands</td>
<td>RJ and RK</td>
<td>Distribute tasks to processors</td>
</tr>
<tr>
<td>Execution complete</td>
<td>Processor done</td>
<td></td>
</tr>
<tr>
<td>Write results</td>
<td>∀(f(Fj[PAR] = FI[PAR] or RJ[f] = No) &amp; (FK[f] = FI[PAR] or RK[f] = No))</td>
<td></td>
</tr>
</tbody>
</table>
will be removed from FU when a task fires, and the same token will be returned back after the task writes its result.

The place ARB acts as a memory arbitrator and contains only one token initially. The transitions connected to this place represent the events of reading/writing the shared memory. A transition will remove the token immediately when it fires, and return the token at the end of the firing. During this time, other transitions cannot fire due to the absence of the token. By this means, the mutual exclusive access to shared memory is guaranteed.

Other places act as preconditions or post conditions of transitions. Their interpretations will be described in the following sub-section, together with those of transitions.

What is more, each place has an initial state before the execution of the model. The initial states of places are also shown in Fig. 2. VAR and FU contain all tokens indicating that all variables and processors in system are ready to be used initially. ARB has a token which indicates that the shared memory is free to access currently. Besides, the place P1 possesses a token that provides the information of a task. All other places have no token when the model begins to execute.

4.3. Interpretations for transitions

Each transition in a CPN model represents an event that causes a state change. The events that transitions represent are summarized in Table 2. There are two nodes in the first stage, the place p1 and the transition t1. Initially, p1 possesses a token which contains the information of the task. The place p1 acts as a precondition for t1, and transition t1 represents the event of checking the readiness of the destination variable. Transition t1 will be enabled if VAR has a token colored in \( d \). When firing, \( t_1 \) will consume the token in \( p_1 \). Besides, it will remove the token \( d \) from VAR first, and then return the token back to VAR at the end of its firing. (Note that the bi-directional arc between \( t_1 \) and VAR is shorthand for two uni-directional arcs with the same arc inscription.)

Transition \( t_2 \) represents the event of task partitioning. The value of the arc inscription par is determined by the partitioning strategy. If FU possesses a token colored in par, it indicates that the target processor is free, and transition \( t_2 \) is enabled. After \( t_2 \) fires, the place \( p_2 \) will obtain a token and enable the transition \( t_3 \).

Transition \( t_3 \) represents the event of task issuing. The state changes caused by the firing of \( t_3 \) are listed as follows:

1. The token \( d \) is removed from VAR, indicating that the task is about to write the corresponding variable, and latter issued tasks are forbidden to read or write it.
2. Two tokens i and j are added to INH, indicating that the task is about to read corresponding variables. So later issued tasks are forbidden to write them.
3. The token par is removed from FU indicating the processor is busy.

<table>
<thead>
<tr>
<th>Transition</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1 )</td>
<td>Check if destination variable is ready</td>
</tr>
<tr>
<td>( t_2 )</td>
<td>Partition</td>
</tr>
<tr>
<td>( t_3 )</td>
<td>Issue</td>
</tr>
<tr>
<td>( t_4 )</td>
<td>Check if source variable i is ready</td>
</tr>
<tr>
<td>( t_5 )</td>
<td>Check if source variable j is ready</td>
</tr>
<tr>
<td>( t_6 )</td>
<td>Read operands</td>
</tr>
<tr>
<td>( t_7 )</td>
<td>Release the lock on source variables</td>
</tr>
<tr>
<td>( t_8 )</td>
<td>Execute a task on different processors</td>
</tr>
<tr>
<td>( t_9 )</td>
<td>Write results</td>
</tr>
</tbody>
</table>

Table 2

Interpretations of transitions.

After \( t_3 \) fires, \( p_4 \) and \( p_5 \) obtain tokens. Place \( p_4 \) acts as a precondition for \( t_4 \), and \( t_4 \) represents the event of checking the readiness of the source variable i. If VAR holds a token colored in i, \( t_4 \) will fire and produce a token in \( p_5 \) which indicates that the variable i is ready to be read. The implications for \( t_5 \), \( p_6 \), and \( p_7 \) can be inferred likewise. If \( p_5 \), \( p_6 \) and ARB hold tokens in them, \( t_5 \) is enabled. The transition \( t_6 \) can be interpreted as the event of reading operands. After firing, \( t_6 \) will produce a token in \( p_6 \). The presence of a token in \( p_6 \) indicates that all the source variables have been read. After that, \( t_7 \) fires and removes from INH the tokens produced by \( t_3 \). Thus, i and j are allowed to be written again.

After the firing of \( t_7 \), the task moves to execution complete stage. Each transition in this stage represents the event of executing the task on a designated processor. The transitions number in this stage equals to that of processors in modeled system. Each transition has a duration inscription, which indicates the execution time of the designated processor. In our demonstration model, there are three transitions in this stage. Of these transitions, only one can be enabled, since \( p_9 \) holds only one token in it. Besides, it is the token color par who determines which transition is able to fire. No matter which transition fires, \( p_{10} \) will get a token, and the execution complete stage is over.

Transition \( t_8 \) represents the event of writing results. It must be noticed that the arc connecting \( t_8 \) and INH ends with an empty circle instead of an arrowhead. This special arc is an inhibitor arc which is used to test for absence of tokens in a place. In other words, \( t_8 \) can fire only when INH does not contain the token d. When \( t_8 \) fires, tokens \( d \) and par will be returned to VAR and FU, respectively, indicating that the corresponding destination variable and processor are available again. Finally, the execution process of a task ends.

4.4. Identification of inter-task dependences

Fig. 3 illustrates a model for a task sequence consisting of two tasks. The left part represents the first task, while the right part represents the latter. The transition \( s_5 \) and surrounding elements are omitted. That is because the latter task has only one source variable. By this means, dead-locks are avoided when multi tasks compete for the occupations on the same destination variable or processor. By observing different cases for this model, we will present how different types of inter-task dependences are identified.

- **Case 1:** the tokens d and d' have the same color. In this case, \( t_3 \) will fire prior to \( s_1 \), and remove the token d from VAR. The needed token will be found missing in VAR when \( s_1 \) is about to fire. Thereby, a WAW hazard is identified. The transition \( s_1 \) will stall until the needed token is returned after the firing of \( t_3 \).
- **Case 2:** the variables par and par' have the same color. Similarly to case 1, \( t_9 \) will remove the token colored in par from FU. Then a structural hazard is indicated by the absence of the needed token in FU when \( s_2 \) is about to fire. As a consequence, the latter task will stall until \( t_3 \) has fired.
- **Case 3:** the variables d and d' have the same color. In this case, the token d is consumed when \( t_3 \) fires. After that, \( s_1 \), \( s_2 \) and \( s_3 \) will fire one after another. However, when \( s_4 \) is about to fire, the needed token is found missing in VAR, so a RAW hazard is detected.
- **Case 4:** the variables d' and i have the same color, and the token j is removed from VAR by an earlier issued task. In this case, \( t_3 \) can fire, and adds the token i in INH. However, \( t_3 \) is unable to fire since the token j is missing in VAR. So the first task stalls in its read operands stage. Meanwhile, the latter task executes ahead of the first one. When the latter task completes execution and \( s_5 \) is about to fire, a WAR hazard will arise if the token j has not been returned to VAR yet. At that time, the model will check...
the absence of the token d’ in INH, and find that INH contains the token i in it. Since the token i have the same color as d’, the inhibitor arc will prevent s9 from firing until the token is removed from INH. Thus, the WAR hazard is resolved.

4.5. Modeling of partitioning strategies

The event of task partitioning is represented by t2 in Fig. 2. Actually, t2 can be substituted by a sub-module of CPN model, if the partitioning strategy is too complicated to be modeled by a single transition. Taking a greedy algorithm as the example, we will present how to model partitioning strategies.

The greedy strategy is intuitive. That is if there is any SP available, a task will be dispatched to a SP. Otherwise, the task will be sent to a free GPP. If there is no processor available, the task must wait. Fig. 4 illustrates the model of this strategy. The components outlined by dashed lines are used to substitute t2 and its surrounding arcs in Fig. 2.

The inscription action( ) near the upper left corner of the transition t2 is an action function written in CPN ML[29]. Fig. 5 shows the pseudo code of the greedy algorithm, which determines the color of output token to p9.

If there is no available processor, t9 will produce a token (FALSE, _), where the symbol _ is a wildcard symbol. Transition t9 will fire and consume the token. After that, P2 will obtain a token and enable t9 once again. Note that t9 has a duration inscription whose value is 1 time unit. As a consequence, t9 will repeat firing once every time unit. When any processor able to execute the task becomes free, t9 will produce a token (TRUE, par) instead of (FALSE, _), and t9 is thereby enabled. After t9 fires, the token par is added in p3, and the task moves from partition stage to the next.

Since the partitioning stage is loosely coupled with other stages, various partitioning strategies can be integrated into our model likewise. Based on the CPN model, our scheduling scheme and different partitioning strategies can be verified and evaluated conveniently.

5. Experiments

In our experiments, we evaluate our scheduling scheme on simulation platform and hardware platform, respectively. Our simulation experiments are performed in CPN Tools version 3.2.1. Time parameters of the simulation model, such as the execution time and data transmission time for IP cores, are all derived from the measurement on our prototype system. The details of the prototype system are presented in Section 5.1.

5.1. Heterogeneous multiprocessor prototype system

In order to evaluate Task-Level Scoreboarding on real hardware platforms, we implement a heterogeneous prototype system on Digilent XUPV5 board with Xilinx Virtex-5 XC5VLX50T FPGA, and the whole system is built in Xilinx ISE Design Suite 12.4.

Fig. 6 illustrates the block diagram of our prototype system. The whole prototype system integrates a MicroBlaze V8.00.a as a GPP and four hardware IP cores as SPs. The MicroBlaze communicates with IP cores by Fast Simplex Link (FSL) v2.11.c buses and connects several peripherals through Peripheral Local Bus (PLB). Besides, The MicroBlaze has a local memory of 64 KB, which accommodates the input/output data for IP cores.

Since we do not have multi FPGA engines (e.g. RAMP [30]), the IP cores which can be simultaneously integrated are quite restricted. We look through AutoBench, ConsumerBench, DenBen and TeleBench benchmark suites in EEMBC [31], and choose four types of test cases (Adder, IDCT, AES_ENC and AES_DEC) to test our Task-Level Scoreboarding. We implement IP cores for all these test cases and integrate them into our prototype system. Besides, the execution time of these IP cores can be configured to customized values. By these means, we can organize tasks with different execution times to form various sample workloads.

Based on the hardware platform, a software edition of Task-Level Scoreboarding algorithm with greedy partitioning strategy is...
implemented on the MicroBlaze. For comparison, we also transplant a software edition of task superscalar [7] to our prototype system.

5.2. Speedups for different task sequences

In this section, we want to verify the performance of our proposed Task-Level Scoreboarding. In order to evaluate Task-Level Scoreboarding, we design several sample task sequences, which are listed in Table 3. The first task sequence consists of four tasks without any data hazards, while other task sequences contain different types of data hazards with varying frequencies. We will repeat executing these sequences for a number of times on both simulation platform and hardware platform, so that we can observe the performance of our approach in different task scales. By comparing the experimental results on our prototype system and CPN model, we can estimate the time overhead of our Task-Level Scoreboarding algorithm.

In order to investigate the influence of task execution time on system performance, we set up several different configurations of the prototype system. The execution times for the same type of IP cores vary per configuration. While in each individual configuration, the execution times of all IP cores are set to the same value.

(1) No hazards.

Fig. 7(1) illustrates the performance comparison of applying our proposed scheme implemented on simulation platform and hardware platform when given task sequence 1. We have conducted both the simulation and hardware experiments in different configurations. The execution times of all IP cores are set to the same value. Theoretically, the four tasks in sequence 1 can be executed totally in parallel. So we conjecture that the upper limit of theoretical speedup in this case should equal to 4.0. This conjecture is demonstrated correct through our experimental verification, given the simulation results as shown in Fig. 7(1).

Observing the performance comparison in each configuration, we can find that: (1) the actual results obtained by hardware experiments are always smaller than the theoretical results obtained by simulation experiments given the task scale and (2) when the task scale is small, the actual results are much smaller than the theoretical results. As the task scales expand, the theoretical speedups gradually increase, while the increases of the actual speedups seem not conspicuous.

The reason of the first phenomenon is that we do not take the time overhead of Task-Level Scoreboarding into account during simulation, but it does exist when performing experiments on prototype platform. The time overhead mainly arises during detecting dependences and partitioning tasks, as well as querying/updating the states of processors and variables. Besides, the overhead will go up in proportion to the increase of task scales. That is why the speedup of actual result does not increase significantly. The reason of significant increment of the theoretical result is that, when task scales are small, the time cost of sequential issuing tasks would has great impact on system speedup, as the task scales expand, the impact would decrease.

When observing the performance comparison with respect to different configurations, we can find that when given the task scale, as task execution time increases, greater speedups would be obtained in both simulation and hardware experiments. When IP core execution times increase to 80,000 cycles, the actual speedup can achieve as high as 3.7, which is competitive to 94.4% of the simulation result. The reason is that for a limited number of tasks, the longer each task execution time becomes, the lower the proportion of scheduling overhead is.

(2) RAW hazards.

Fig. 7(1)–(5) shows the experimental results on both simulation platform and hardware platform when given task sequences 2–5. These sequences have different frequencies of RAW hazards during execution, which are 25%, 50%, 75% and 100%, respectively.

Fig. 6. Block diagram of the prototype system.
Firstly, we want to evaluate our approach when dealing with RAW hazards. As shown in Fig. 7(2), given a fixed task scale, as the execution time increases, the theoretical speedup increases; given a fixed execution time, as the task scale increases, the theoretical speedup increases. When both the task scale and the execution time are large enough, the theoretical speedup is almost as high as the upper limit of theoretical speedup. Moreover, given a fixed task scale and execution time, the actual speedups are always smaller than the theoretical values. And, the gap between the actual speedup and the theoretical speedup decreases, when the execution time for each task increases.

Secondly, we want to show the performance comparison of our proposed approach when dealing with RAW hazards with different frequencies. According to our analysis, the upper limits of theoretical speedups for these sequences are 4.0, 2.67, 2.0 and 1.0, respectively. It is clear that, as the frequency rises, the system speedup decreases due to the reduction of potential parallelism hidden in task sequences.

Observed from Fig. 9(2)–(4) that, as the task scales increase, the resulting speedup would significantly increases, because more potential parallelisms can be tapped out when using our Task-Level Scoreboarding. For sequence 5, there are so many RAW hazards that all tasks must execute in order. Thus, the speedup seems unchanged as task scales increase. When each sequence is repeated executing for 256 times, the actual speedsups of our approach can at most achieve as high as 90.4%, 92.0%, 95.1% and 98.9% of the theoretical values respectively. The performance losses are caused by scheduling overhead.

(3) WAW hazards. Fig. 7(6)–(8) shows the experimental results on both simulation platform and hardware platform when given task sequences 6–8. All of these sequences consist of only four types of task, including add, idct, enc and dec. However, they have different occurring frequencies of WAW hazards during execution, which are 25%, 50% and 100%, respectively. The upper limit of theoretical speedup for these three cases can be calculated based on the following formulas:

\[
\text{Speedup}_{\text{seq6}} = \frac{t_{\text{add}} + t_{\text{idct}} + t_{\text{enc}} + t_{\text{dec}}}{t_{\text{add}} + t_{\text{idct}}}
\]

\[
\text{Speedup}_{\text{seq7}} = \frac{t_{\text{add}} + t_{\text{idct}} + t_{\text{enc}} + t_{\text{dec}}}{t_{\text{add}} + t_{\text{idct}} + t_{\text{enc}}}
\]
In our prototype system, there are the sum of the execution time and data transmission time for each type of task. Actually, the theoretical speedups for sequences 6 and 7 are slightly beyond 2.0 and 1.33, respectively. The theoretical speedup for sequence 8 exactly equals to 1.0, since there are so many hazards that all tasks must execute sequentially.

The best result of the actual speedups for these three cases achieve as high as 93.8%, 95.7% and 96.8% of the theoretical values respectively.

(4) WAR hazards.

In each experiment, we want to evaluate the effectiveness of our approach when dealing with the WAR hazards given task sequence 9. There is a RAW hazard between the first two tasks in this sequence, and the third task can begin to execute prior to the second one. Consequently, a WAR hazard will occur if the third task completes execution before the second one begins to execute. To ensure the presence of WAR hazards, we adjust execution times of IP cores and set up another four configurations of our prototype system. The IP core execution times for different configurations are listed in Table 4. In addition, due to the limitation on the number of IP cores, we cannot design sample sequences with more WAR hazards in our prototype system.

Theoretically, as task scale increases, the value of speedup should approximate 3.0 according to the formula below:

\[
\text{Speedup}_{\text{approx}} = \frac{t_{\text{add}} + t_{\text{dec}} + t_{\text{enc}} + t_{\text{enc}}}{\max(t_{\text{add}}, t_{\text{enc}})} \approx 3.0
\]

The experimental results are shown in Fig. 8. Similar to previous experiments, the speedups for both the CPN model and prototype system increase as the task scale and task execution time rise. In spite of the time overhead introduced by scheduling, the actual speedup can achieve as high as 91.3% of the maximum value. That demonstrates the effectiveness of our approach.

Through the experimental results obtained from the experiments above, we demonstrate that our Task-Level Scoreboarding scheme can effectively accelerate the execution of various applications. Besides, the correctness of simulation models are also demonstrated through the simulation experiments, since the simulation results are well matched with the theoretical analyses.

5.3 Comparative experiments

As mentioned in Section 1, task superscalar [7] proposes a dynamic scheduling scheme based on Tomasulo algorithm, which is most related to our work but taking a different approach. In order to compare our Task-Level Scoreboarding with the state-of-the-art approach, we also implement a software edition of task superscalar on our prototype system. We conduct the comparative experiments on hardware platform given the sample task sequences listed in Table 3. These two approaches will be evaluated and further compared in these cases.

(1) No hazards.

Fig. 9(1) shows the performance comparison of our approach and task superscalar when given task sequence 1, which has no data hazards. Theoretically, the speedups of both approaches are the same, and their upper limit equals to 4.0.

Through the observation on Fig. 9(1) we can see that, our approach performs more effectively than task superscalar when given a fixed task execution time. It demonstrates that our approach introduces less time overhead thanks to its simplicity. Especially, when the execution time for each task is small, the speedups of our approach are much higher than those achieved by Task Superscalar.

For each individual approach, when given a fixed task scale, the speedup rises as the execution time of each task increases. As the execution time for each task increases, the proportion of the scheduling overhead to overall execution time decreases and the speedup increases.

(2) RAW hazards.

Since both task superscalar and our Task-Level Scoreboarding cannot eliminate RAW hazards, so in the cases of sequence 2 and sequence 5, both approaches achieve the same theoretical speedup, which are 4.0 for sequence 2 while 1.0 for sequence 5. Fig. 9(2) and (5) illustrates the comparison of our approach and task superscalar in these two cases. Observing the figures, we can find the same phenomena as the previous experiment: (1) given a fixed task scale, the speedups for both approaches increase as the task execution times go up; (2) given a fixed task sequence and the scale, our approach achieves higher speedups than the other. The reasons for these phenomena are also the same as those of the previous experiment.

Given sequence 3 and sequence 4, the task execution order varies between two approaches due to the influence of structural hazards. With the help of its internal structure Reservation Station (RS), task superscalar can issue a task even if a structural hazard actually exists. Therefore, the following tasks with no hazards can be executed in advance. On the contrary, Task-Level Scoreboarding must block all the following tasks when encountering structural hazards. For the reason above, task superscalar theoretically exhibits more excellent performance than our Task-Level Scoreboarding. When given sequence 3 and sequence 4, the upper limits of theoretical speedups for task superscalar are respectively 4.0 and 2.67, while those for Task-Level Scoreboarding are 2.0 and 1.33, respectively.

The experimental results in these two cases are shown in Fig. 9(3) and (4). From the figures we can see that, task superscalar achieves higher speedups than our approach when the execution time of each task is large enough. The reason is that task superscalar is able to uncover more potential parallelisms than our approach. As task execution time decreases, the influence of scheduling overhead on system performance is enhanced. Therefore, the speedups of both approaches decrease. The speedup of task superscalar decreases more significantly than our approach since the time overhead brought in by task superscalar is much greater than that of our Task-Level Scoreboarding. As a result, when the
execution time for each task is no more than 10,000 cycles, our approach exceeds task superscalar in speedup finally.

(3) WAW hazards.

As mentioned before, task sequences 6–8 have increasing frequencies of WAW hazards, which are 25%, 50% and 100%, respectively. Unlike our Task-Level Scoreboarding, task superscalar can eliminate WAW and WAR hazards using renaming mechanism. Therefore, task superscalar can achieve better performance in these cases theoretically. Given task sequences 6–8, the upper limits of theoretical speedups for task superscalar are all 4.0, while those for our approach are 2.0, 1.33 and 1.0, respectively.

Fig. 9(6)–(8) respectively shows the actual experimental results for comparison of both approaches when given sequence 6–8. Observing Fig. 9, we can find that task superscalar can benefit from tapping out more parallelism and achieve better overall performances when the task execution time is large enough. However, as the task execution time decreases below a threshold, our approach shows better performance than Task Superscalar, benefiting from its lower scheduling overhead. The threshold is between 10,000 and 20,000 cycles for sequence 6, and nearly 10,000 cycles for sequence 7. While for sequence 8, the threshold falls below 10,000 cycles.

Based on the observation on Fig. 9(6)–(8), we can also evaluate the effect of the occurring frequency of WAW hazards on the resulting speedups for both approaches. Given a fixed task execution time, 10,000 cycles for demonstration, task superscalar achieves higher speedup when applied on task sequence 8, where the frequency is 100%. As the frequency decreases, our approach benefits from the increasing parallelism hidden in tasks and thereby achieves higher speedups, while the speedups for task superscalar remain unchanged. When the frequency drops to 50% (sequence 7), the speedups for both approaches are more or less the same. Furthermore, when the frequency drops to 25% (sequence 6), our approach shows better performance than Task Superscalar. It can be concluded that our Task-Level Scoreboarding can achieve greater speedups than task superscalar when the frequency drops below 25%.

(4) WAR hazards.

The comparison of task superscalar and our Task-Level Scoreboarding on sequence 9 is illustrated in Fig. 10. Theoretically speaking, task superscalar should exhibit better performances than our approach, since it can eliminate WAR hazards while our approach cannot. However, the experimental result on hardware platform demonstrates our approach achieves better performances when given a fixed task execution time. Benefiting from lower time overhead, our approach achieves higher speedups, although it can tap out less parallelism than Task Superscalar.
6. Conclusion and future works

In this paper, we have proposed task-level scoreboarding scheduling scheme based on the CPN model for MPSoCs, which can dynamically schedule tasks to perform out-of-order execution and thereby improve the parallelism of applications. Assisted by the modeling power of CPN, various types of inter-task dependencies can be detected automatically. In addition, different dispatching strategies can be easily evaluated in our CPN based scheme.

Our proposed scheme is implemented both in CPN tools and FPGA hardware prototype for comparison. Experimental results on case studies demonstrate our CPN based scheduling scheme could achieve significant speedup against state-of-the-art scheduling schemes.

In the future, we intend to realize our proposed scheduling scheme as a hardware module and embed it into MPSoC platforms. Furthermore, due to the restriction on chip area, the number and category of processors on MPSoC platforms are significantly restricted. Whereas reconfigurable computing provides a promising approach to enhance the utilization of resources on a silicon chip, we are also planning to extend our CPN based scheduling to support dynamic reconfigurable architectures. We hope that it can provide an effective method for evaluating reconfiguration algorithm and software/hardware partitioning strategies for reconfigurable MPSoC platforms.

Acknowledgements

This work was supported by the National Science Foundation of China under Grants Nos. 61379040, 61272131, 61202053, Jiangsu Provincial Natural Science Foundation Grant No. BK201240198, Fundamental Research Funds for the Central Universities No. WK011000034, and China Postdoctoral Science Foundation Grant No. 2012M521252. The authors deeply appreciate many reviewers for their insightful comments and suggestions.

References


Fig. 10. Comparative experiment for task sequence 9.

Chao Wang got his bachelor degree in University of Science and Technology of China in 2006 and Doctor Degree in 2011. His main research interests include Multiprocessor systems on Chip and reconfigurable systems.

Xiaojing Feng is currently a Ph.D. candidate in the School of Computer Science, University of Science and Technology of China. He received his B.E. degree from the Department of Computer Science, University of Science and Technology of China in 2007. His research interests include Multiprocessor systems and reconfigurable systems.

Xi Li is now an Associate Professor in the School of Computer Science, University of Science and Technology of China. His main research interests include embedded operation system and reconfigurable computing.

Chao Wang

Xiaojing Feng

Xi Li

Peng Chen got his bachelor degree in University of Science and Technology of China in 2006. He is a candidate for the master of Computer Architecture in University of Science and Technology of China. His major interest is embedded system.

Xuehai Zhou is a Professor and Ph.D. supervisor in University of Science and Technology of China. His main research interests include embedded system design, reliable system design and reconfigurable computing.

Peng Chen

Xuehai Zhou