第44卷第1期 中国科学技术大学学报 Vol. 44，No. 1
2014年1月 JOURNAL OF UNIVERSITY OF SCIENCE AND TECHNOLOGY OF CHINA Jan. 2014

Article ID：0253-2778(2014)01-0019-15

A colored Petri net based scheduling scheme for multiprocessor system-on-chip

FENG Xiaojing，LI Xi，WANG Chao，CHEN Peng，ZHOU Xuehai

(*School of Computer Science and Technology*，USTC，*Hefei* 230027，*China*；
Suzhou Institution for Advanced Study，USTC，*Suzhou* 215123，*China*)

Abstract：A novel colored Petri net (CPN) based dynamic scheduling scheme was proposed，which aimed at generating a hardware scheduler for multiprocessor system-on-chip (MPSoC) platforms. CPN was employed to model inter-task dependences in the proposed scheduling scheme，including RAW，WAW and WAR data dependences，as well as structural dependences. All the dependences can be automatically detected during model execution. Tasks can be then scheduled and dispatched to different processors for out-of-order execution according to the dependences，achieving the goal of improving task-level parallelism. The scheduling scheme is implemented both with software simulation tools and on an FPGA-based hardware platform. Through state space analyses and comparing experiments，the correctness and effectiveness of the scheduling scheme are demonstrated.

Key words：colored Petri nets；task scheduling；multiprocessor system-on-chip (MPSoC)；model based design

CLC number：TP301.2　　**Document code**：A　　doi：10.3969/j.issn.0253-2778.2014.01.003

Citation：Feng Xiaojing，Li Xi，Wang Chao，et al. A colored Petri net based scheduling scheme for multiprocessor system-on-chip[J]. Journal of University of Science and Technology of China，2014，44(1)：19-33.

基于有色 Petri 网的多处理器片上系统调度方法

冯晓静，李　曦，王　超，陈　鹏，周学海

(中国科学技术大学计算机科学与技术学院，安徽合肥 230027；
中国科学技术大学苏州研究院，苏州 215123)

摘要：为了生成一个适用于多处理器片上系统的硬件调度器，提出一种新型的基于有色 Petri 网(CPN)的动态调度方法. 该调度方法使用 CPN 对包括写后读、写后写、读后写数据相关以及结构相关在内的任务间相关性进行了建模，这些相关会在模型运行的过程中被自动检测出来. 根据相关性，任务会被动态地调度并分配

Received：2013-03-18；**Revised**：2013-04-13

Foundation item：National Natural Science Foundation of China (61202053)，Natural Science Foundation of Jiangsu Province (BK2012194).

Biography：FENG Xiaojing，Male，born in 1984，PhD candidate. His research interests include reconfigurable computing technology，multiprocessor systems and formal verification. E-mail：bangyan@mail.ustc.edu.cn

Corresponding author：ZHOU Xuehai，PhD，Professor. E-mail：xhzhou@ustc.edu.cn

到不同的计算单元上乱序执行,从而达到提高任务级并行度的目的.该调度方法分别在软件仿真平台和基于FPGA 的硬件平台上得以实现.状态空间分析和对比实验的结果证明了调度方法的正确性和有效性.
关键词:有色 Petri 网;任务调度;多处理器片上系统;基于模型的设计方法

0 Introduction

The increase in complexity of modern embedded applications has led to the popularity of Multiprocessor System-on-Chip (MPSoC) architecture, 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 to parallelizing applications is task scheduling. That is, to partition an application into small portions, namely tasks, and then dispatch tasks to different processors to allow them to execute in parallel. A great number of scheduling schemes have been proposed for MPSoCs. Some recent works, such as Carbon[1], ADM[2] and Task Superscalar[3], even employ hardware technologies to accelerate the process of task scheduling.

Although these schemes are effective in exposing hidden parallelism in applications to underlying hardware of MPSoCs, designing these schemes is daunting and time-consuming. The reason lies mainly in two aspects: ① human designers are extremely prone to making mistakes due, on one hand, to the increasing complexity of MPSoCs, and on the other, to the natures of multiprocessor applications such as concurrency and asynchronism. ② Since system performance usually depends on a number of parameters, a large variety of paradigms should be evaluated for a scheduling scheme.

Model based design provides a promising approach for tackling these problems. Benefiting from verification and validation of models, designers can detect errors and flaws as soon as possible. Besides, different paradigms of designs can be conveniently evaluated in early design phases through model based simulations. For these reasons, model based design methodology has been widely employed for solving task scheduling problems. Nonetheless, little research has been conducted in model based dynamic scheduling schemes. This is mainly because there are two vital behaviors hard to describe when modeling dynamic scheduling schemes for MPSoCs:

(Ⅰ) 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 be 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's hard to model the dynamic behavior of dependence detecting. Most related work takes inter-task dependency as a priori, and rarely addresses the problem of dynamic dependence detecting.

(Ⅱ) 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 a significant effect on system performance, since the execution 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. What's worse, the heterogeneity of processors on MPSoCs makes it more difficult.

This paper proposes a colored Petri net (CPN) based dynamic scheduling scheme, which addresses both of the aforementioned problems simultaneously. To achieve the goal of improving task-level parallelism of applications, tasks are divided into several pipeline stages and then

scheduled in our scheme. The scheduling process is modeled using CPN. We use colored tokens to model tasks and system resources. Inter-task dependences, including true-dependences (read-after-write, RAW), output-dependences (write-after-write, WAW), anti-dependences (write-after-read, WAR) and structural dependences are represented as different relations on transitions and tokens, such as producer-consumer relation and competition relation. All dependences can then be identified and resolved dynamically during model execution. Besides, our scheme support the modeling of various task dispatching strategies. For demonstration, we will show how to construct the model for a greedy dispatching strategy in this paper. By performing simulations on our CPN model, task scheduling scheme and dispatching strategies can be easily evaluated.

Our proposed scheme aims at generating a hardware scheduler responsible for dynamically scheduling tasks on MPSoCs. The scheme is generic in the sense that it can be applied in systems with different architectures and specifications. We have applied the scheme to SOMP platform[4] and built the CPN model for the SOMP prototype systems. The model results are confirmed against the measurement derived from prototype systems, demonstrating the correctness of the CPN model.

1 Related work

As the complexity of MPSoCs increases, model based design methodology plays an increasingly important role in hardware and application designs on MPSoC platforms. Now that task scheduling is an important concern in system design, much effort has been devoted to modeling task scheduling processes on MPSoCs, both informally and formally.

Formal methods are system design techniques which build system models using languages with mathematically defined syntax and semantics. They can reveal ambiguities that might go undetected in informal methods, and thereby improve the reliability of system designs. There are numerous modeling languages that can be used in formal methods. The most widely applied ones include Petri nets and timed automata, mainly because they have both mathematical definitions and graphical representations.

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 in Ref. [5], 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 Refs. [6] and [7], which extend cost information on locations and transitions. The timed automata in Ref. [8] 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 Ref. [8] 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 Ref. [9]. 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's hard to use timed automata to model concurrent systems with shared resources[10]. Due to this limitation with respect to modeling power, the application

scope of timed automata is greatly restricted.

In contrast, Petri nets are a specialized class of state-transition graphs with two sets of nodes, places and transitions[11]. Places are employed to describe the status of modeled systems, while transitions represent state changes. Besides, tokens are imported in models to represent input/output data or control information such as logic conditions. These features make Petri nets well suited for modeling concurrent systems, including their shared resources. Furthermore, the event of dynamically creating new tasks can be easily modeled by adding new tokens in the model at runtime. CPN is a high-level extension for Petri nets[12]. With the support to colored tokens and model time etc., CPN possesses enhanced expressive power beyond basic Petri nets.

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[13]. The proposed model can be utilized to evaluate the influence of different model parameters on system performance. In Ref. [14], 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. Ref. [15] studies the task scheduling of a robot system with temporal constraints, using timed Petri nets. Ref. [16] 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[17-18]. 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[19-20]. 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[21]. Although these models have the capability of describing different types of inter-task dependences, the dependences must be given as prerequisites since the models cannot identify them automatically.

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.

2 Preliminaries

2.1 Problem description

The proposed scheduling scheme aims to improve the parallelism of applications at task level. The term "task" here refers to a piece of work that can be completed on a single processor. All input and output of tasks are associated in variables residing in the memory. Thus, each task can be defined as follows:

Definition 2.1 Let T_i be a task. T_i can be defined as a triple, $T_i=(tn_i, \mathrm{TD}_i, \mathrm{TS}_i)$, where:

① $tn_i \in \mathrm{TASK}$ represents the task type;

② TD_i, $\mathrm{TS}_i \subseteq \mathrm{DATA}$ are finite sets, representing output and input data of a task, respectively. $\mathrm{DATA}=\mathrm{U}_i(\mathrm{TD}_i \cup \mathrm{TS}_i)$ is a finite set whose members are input and output variables for all tasks.

The input to the scheduler should be a non-speculative task sequence, which can be derived from applications using a source-to-source compiler.

Definition 2.2 Let T be the input task sequence. T can be defined as a first-in-first-out (FIFO) queue, $T=[T_1, T_2 \cdots T_n]$ where each element of the queue represents a task.

Subsequently, we need to describe the architecture of the MPSoC platforms. In this paper, we consider single-chip heterogeneous architecture for MPSoCs. Heterogeneous MPSoCs combine general purpose processors (GPPs) with a variety of heterogeneous application-specific processors (ASPs) such as digital signal processors (DSPs), intellectual property (IP) cores and custom logics. GPPs usually provide runtime libraries for general computing tasks. On the contrary, ASPs can execute only specific tasks, usually computation-intense tasks. Since ASPs provide greater performance over GPPs for specific tasks, such tasks can be dispatched to ASPs for acceleration. Additionally, we assume a shared-memory multiprocessor architecture for the program execution. A processor must access the shared memory to fetch the input data prior to execution and send the results to the shared memory when accomplishing computations. Data transmission between processors must pass through the shared memory.

Definition 2.3 Let S be the specification of an MPSoC platform. S can be defined as an 8-tuple, S = (GPP, TASK, TIME, SCH, ASP, STASK, STIME, SSCH), where:

① $GPP=\{gp_1, gp_2, \cdots, gp_m\}$ is a finite set of GPPs;

② $TASK=\{t_1, t_2, \cdots, t_n\}$ is a finite set, which represents all types of tasks that can be executed on GPPs;

③ TIME: $GPP \times TASK \rightarrow \{1, 2, 3, \cdots\}$ is a function which determines the execution time for a task running on a GPP;

④ SCH: $GPP \times TASK \rightarrow \{1, 2, 3, \cdots\}$ is a function which determines the time cost for scheduling a task to a GPP.

⑤ $ASP=\{sp_1, sp_2, \cdots, sp_n\}$ is a finite set of ASPs;

⑥ STASK: ASP→TASK is a function which determines the functionality of each ASP;

⑦ STIME: $ASP \rightarrow \{1, 2, 3, \cdots\}$ is a function which determines the execution time for running a task on a particular ASP;

⑧ SSCH: $ASP \rightarrow \{1, 2, 3, \cdots\}$ is a function which determines the time cost for scheduling a task to a ASP.

①+② A MPSoC may integrate several GPPs. With the support of runtime libraries running on it, each GPP can execute all types of tasks which are defined by the set TASK.

③+④ The execution time of executing a task on GPP varies among different types of tasks. Besides, different tasks running on GPPs vary in the scheduling time, which involves the time cost for fetching input data and writing back the results. The execution and scheduling times for different tasks on GPPs are specified by the functions TIME and SCH, respectively.

⑤+⑥ A heterogeneous MPSoC may also integrate several ASPs. Unlike GPPs, an ASP usually can only execute a specific type of tasks. The function STASK defines the task type for each ASP.

⑦+⑧ Since each ASP can execute only one type of tasks, the execution of an individual ASP is fixed. However, the time cost of scheduling a task to ASP varies among task types. The functions STIME and SSCH respectively describe the time costs for executing and scheduling a task on an ASP.

Note that our proposed scheme is well suited for, but not limited to, heterogeneous MPSoC platforms. It can also be applied to homogeneous architectures without any modification.

2.2 Colored Petri nets

Petri nets are a powerful modeling formalism with both graphical representations and formal mathematic definitions. The formal definition of Petri nets is given in Ref. [11]. Petri nets are a particular kind of bipartite directed graphs composed of three types of objects, namely places, transitions and directed arcs. Directed arcs are either from a place to a transition or from a transition to a place. These three objects construct the static structure of a Petri net. Besides, each place can hold a nonnegative number of tokens in it. With the flow of tokens among places, a Petri

net is executable. Therefore, tokens provide Petri net with the ability to model dynamic behaviors of a modeled system.

CPN is a high-level extension for Petri nets. The formal definition of CPN is given in Ref. [22]. The biggest difference between CPN and basic Petri nets lies in the fact that each token in a CPN model can be attached with a value. This value is called token color. Tokens are distinguished from each other by their colors. Token color provides convenience for describing heterogeneity of modeled objects, and thereby enhances the modeling power of Petri nets.

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 that represent system states, while the events are represented by transitions. A transition in a CPN model is either enabled or disabled. 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). An enabled transition can fire, indicating 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). Consequently, the firing of a transition is accompanied with a flow of tokens, and thereby results in a state change of a CPN model.

For the sake of its expressive power, we tentatively use CPN to model the behavior of task scheduling in our scheme.

3 CPN based scheduling scheme

This section presents the details on our proposed scheme. We describe the scheduling algorithm used in our scheme first. Subsequently, the CPN model is presented in a top-down fashion.

3.1 Task scheduling scheme

The scheduling algorithm can be found in Ref. [23]. The execution of a task is divided into several pipeline stages to allow exploiting parallelism among tasks. The pipeline stages and the manipulations at each stage are presented in Tab. 1.

Tab. 1 Processing flow of the scheduling algorithm

Task Status	Wait Until	Action or Bookkeeping
Check Destination	Once entering scoreboard controller	while (Results[D])
Dispatch Task	not Results[D]	PAR ← Target processor determined by the dispatching strategy
Issue	Not Busy [PAR]	Busy[PAR] ←yes; Fi[PAR] ←D; Fj[PAR] ←S1; Fk[PAR] ←S2; Qj←Result[S1]; Qk←Result[S2]; Rj←not Qj; Rk←not Qk; Result[D] ←PAR
Read Operands	Rj and Rk	Rj ←No; Rk←No;
Execution Complete	Processor done	Distribute tasks to processors
Write Results	∀f((Fj[f]≠Fi[PAR] or Rj[f] = No) & (Fk[f] ≠Fi[PAR] or Rk[f] =No))	∀f(if Qj[f]= PAR then Rj[f] ←Yes); ∀f(if Qk[f] = PAR then Rk[f] ←Yes); Result[Fi[PAR]] ←0; Busy[PAR] ←No

3.2 Overview of the CPN model

The CPN model of our scheduling scheme is established in a hierarchical way with a top level module and several low level sub-modules. Fig. 1 illustrates an instance of the top level module. To facilitate discussion on our scheme, we assume that each task in the instance model has only one output variable and no more than two input variables. Also, we'll present how to modify the model to deal with tasks with more input/output variables later in this section.

By convention, a place is drawn as a circle or an ellipse with the place name written inside it. Places are used to store the states of a modeled system. In our model, the places drawn as circles store the states of all tasks, while those drawn as ellipses store the states of system resources.

Each place is assigned a color set, which is denoted by an identifier around it, such as TASK and DATA. The color set specifies the colors for

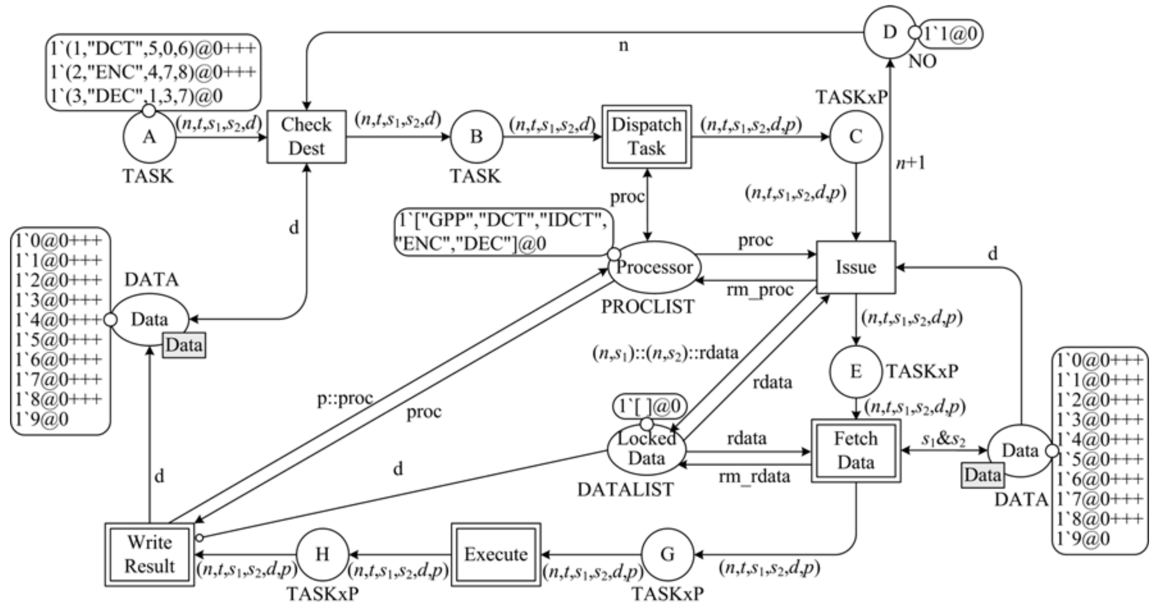

Fig. 1　Top level module of the CPN model in the initial marking

the tokens that can reside in a place. Note that there are two places with the same name Data. These places are called fusion places. Fusion places with the same name function identically as any change that happens to one place always happens to the other ones. They can be drawn in different modules across hierarchical structures. Therefore, fusion places provide a hierarchical way for representing a unique place in multiple locations within a model.

Transitions are drawn as rectangles. Transitions can be interpreted as events of the modeled system. Each transition in the top level module respectively represents the operations on tasks in a particular pipeline stage. Some pipeline stages are too complex to be modeled by a single basic transition. Each of these pipeline stages is modeled in a hierachical way. In the top level module, a stage is represented by a substitution transition which is marked with double line borders. Each substitution transition has a corresponding low level module, and functions as the compound behavior of all model elements in the low level module. A transition may have a code segment, guard function and duration inscription with it. They are not drawn in the figure to preserve the clarity. The details of them will be included in the rest of the paper when necessary.

Arcs are drawn as directed lines. Each arc has an arc expression attached to it. The arcs drawn as bi-directional lines are short for two mono-directional lines with the same arc expression. The arc between the place Locked Data and the transition Write Result is distinct from other arcs, as it's ended with an empty circle instead of an arrowhead. It is an inhibitor arc which is used to test the absence, rather than the presence of tokens, in a place.

All tokens should be accommodated in places. The tokens contained in a place are drawn in the rounded rectangle attached to it. The absence of a rounded rectangle means the place contains no tokens. In our model, tokens are used to model software application (tasks) and system resources (considering processors and variables).

The source and destination variables of tasks are represented by tokens contained in the place Data. Each token has a prefix which indicates the quantity of tokens with the same color, and also a time stamp as the suffix. The token color is an integer number acting as the variable name.

The color of the token in the place processor is of list data types. Each element in the list represents a free processor in the modeled system. We use the token colored in GPP to denote a GPP in the modeled system, while DCT, IDCT, ENC

and DEC to denote ASPs with different functionalities, DCT encoder/decoder and AES encoder/decoder respectively. Likewise, the token contained in the place Locked Data is also colored by a list. The elements of the list represent the variables which are to be read during the execution of the model. At model start-up, the token color is a blank list denoted by the symbol [ ].

Tasks are also modeled by tokens. Token colors are used to distinguish not only different tasks but also the same task in different states. For instance, a newly coming task is modeled by a token in place A whose color is a product of 5 elements, (n, t, s_1, s_2, d). When a task has already been dispatched to a processor, the token color will turn into a product of 6 elements, (n, t, s_1, s_2, d, p). The elements represent the serial number, the task type, two source variables, destination variable and the processor of a task, respectively. For the tasks with only one source variable, s_2 can be assigned a meaningless value.

Fig. 1 also shows the initial marking of our model. An initial marking refers to the tokens residing in each place at the start-up of the model. It is determined by system specification and application, since tasks and system resources are all represented by colored tokens in our model. Unlike many other modeling methods, we just need to modify the initial marking when the modeled system changes, without any modification on the net structures of the model. It brings in higher flexibility and scalability to our model.

3.3 Processing flow of the CPN scheduling scheme

In this sub-section, the processing flow of our CPN based scheduling scheme, with the interpretations on tokens, transitions and places, is presented. Especially, we present how inter-task dependences are modeled and resolved.

3.3.1 Check destination

At the start-up of the model, only the transition Check Dest is enabled. The transition models the first pipeline stage in our scheme. It's connected to three input places A, D, and data. The place A acts as task queue and each token in it represents an individual task to be processed. The place D can be interpreted as a task counter, which forces tasks in the queue enter the scheduler in order. Tokens in place data represent task variables free to write.

Output-dependences can be identified in this stage. They are modeled by transition Check Dest competing for the same token in data. During model execution, the transition Check Dest keeps checking the presence of the token d in the place data. If the token is found missing from the place, it indicates the existence of an output-dependence. So the transition will stay disabled. On the contrary, the presence of the token indicates that no other active tasks have the same destination variable as the incoming task. In this case, the transition Check Dest is able to fire, removing one token from each input place and producing a new token in the place B.

If there are tasks with more than one destination variable, the transition Check Dest should be modified as a substitution transition whose low level module is illustrated in Fig. 2. In this module, the transition t_1 represents the behavior of checking the availability of the first destination variable d_1, while the transition t_2 can

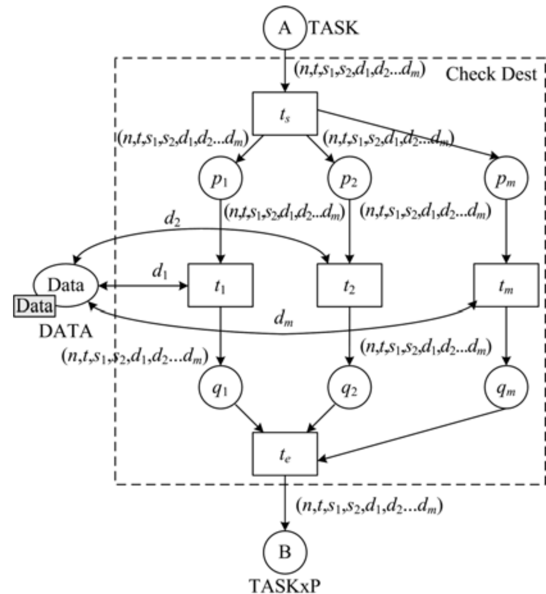

Fig. 2 Model of Check Dest

be interpreted as checking the availability of the second destination variable d_2. If tasks have more destination variables, extra branches must be added into the model. The elements of a branch involves p_m, q_m, t_m and attached arcs, which are marked by thick border lines in Fig. 2. Note that t_e won't be enabled unless all the other transitions have fired, meaning that the destination is ready only when all output variables is free to use.

3.3.2 Dispatch task

The transition dispatch task represents the event of dispatching tasks. As mentioned before, different dispatching strategies can be evaluated within our scheduling scheme. For demonstration, we implement a greedy dispatching strategy in this paper. The greedy strategy is intuitive. That is, if there is any ASP available, a task will be dispatched to an ASP for acceleration. Otherwise, the task will be sent to a free GPP. If neither an ASP nor a GPP is available, then the task must wait. The model of the greedy dispatching strategy is shown in Fig. 3.

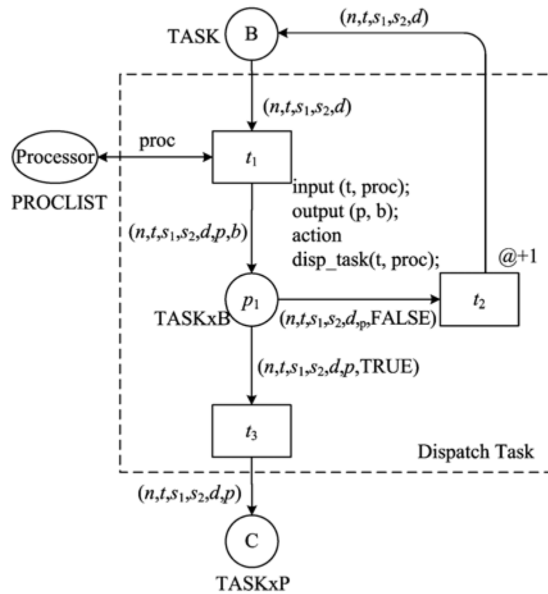

Fig. 3 Model of dispatch tasks

Once the place B receives a token, the transition t_1 will fire and produce a new token whose color is a product of 7 elements (n, t, s_1, s_2, d, p, b). The last element is a Boolean variable indicating whether a structural dependence exists. If there is no free processor capable of executing the task, t_1 will produce a token (n, t, s_1, s_2, d, p, False) to enable the transition t_2, indicating a structural dependence exists. Structural dependences are modeled by competition relations on tokens in the place processor. Subsequently, the transition t_2 will fire and add a token in place B. Then t_1 is enabled once again. Note that the transition t_2 has an inscription @+1 which means that the firing of the transition has the duration of one time unit. As a consequence, t_1 will repeat firing once every time unit. When any processor able to execute the task becomes free, t_1 will produce a token (n, t, s_1, s_2, d, p, True) where True indicates the absence of structural dependences and p represents the processor assigned to the task. The token will enable the transition t_3 rather than t_2. After t_3 fires, the dispatch task stage is over.

The transition t_1 in Fig. 3 has an attached code segment written in CPN ML. The code segment determines the color of the token t_1 produced.

3.3.3 Issue

The third pipeline stage is modeled by the transition Issue. The transition will be enabled immediately when the place C obtains a token. The state changes caused by the firing of the transition are listed below:

① The token in C move to the place E, meaning the task has been issued;

② The token d is removed from data, meaning the task is about to write the corresponding variable;

③ The elements (n, s_1) and (n, s_2) are added to the token in locked data, meaning that the nth task is about to read corresponding variables;

④ The element p is removed from the token in Processor, meaning the processor is busy;

⑤ A token is added in the place D, allowing the next task to enter the scheduler from the task queue.

In our scheme, a task won't be issued unless all needed resources (processor and destination variables) are prepared. Besides, a task is not allowed to enter the scheduler until its previous task has been issued. This guarantees that no later

tasks would preempt the needed resources from an issued task. By this means, dead-locks on critical resources are avoided.

3.3.4 Fetch data

After a task is issued, the assigned processor is to fetch input data. The event is represented by the substitution transition fetch data. The low level module of the transition is illustrated in Fig. 4.

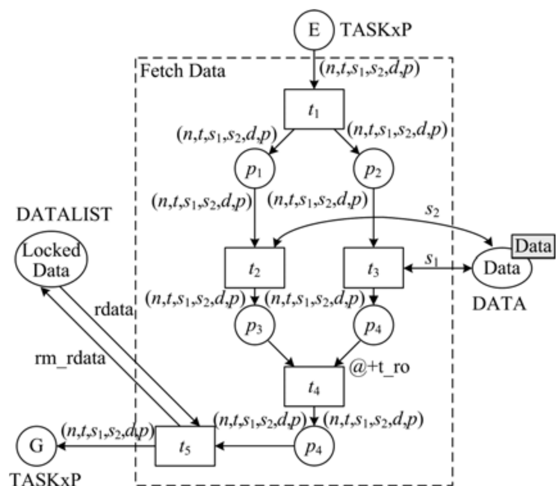

Fig. 4 Model of fetch data

Before the operation of fetching input data, the availability of input data should be checked. Tokens in place data can be interpreted as the variables ready to be read here. The transition t_2 represents the event of checking the availability of the first source variable. A true-dependence is modeled as transition t_2 waiting for needed tokens in place data. The absence of the token s_1 in the place data indicates that a true-dependence exists. The t_2 must wait until the token s_1 is returned to data. Likewise, t_3 represents the event of checking if the other source variable is available. The transition t_4 models the operation of fetching input data. It will be enabled when both t_2 and t_3 have fired. The duration inscription of the transition represents the time spent on fetching data from memory. At the end of this stage, the transition t_5 will remove the elements (n, s_1) and (n, s_2) from the token in locked data to allow later issued tasks writing the variables s_1 and s_2.

The model should be extended with extra elements when a task has more than two source variables.

3.3.5 Execute

The model of the pipeline stage execute may vary with respect to the specification of the modeled system. Fig. 5 shows an instance of the model. In this instance, we assume that the modeled system supports four types of tasks and integrates one ASP for each type of task, as well as a GPP. Thus, there are two ways for executing a task, either on a GPP or on an ASP. The four transitions in the top half of the model represent executing a task on a particular ASP (DCT, IDCT, ENC or DEC). Other transitions represent executing different tasks on the GPP. All these transitions have a same input place G. Tokens in the place represent tasks to be executed, and each token can enable only one transition, meaning a task can only be executed in one way. Each transition has a guard function drawn near its left top corner. It is the guard function that determines which transition can be enabled by a particular token. Besides, each transition has a duration inscription near its top right corner to model the task execution time. No matter which transition fires, it will consume a token in place G and add in place H a token which represents a task to be written back.

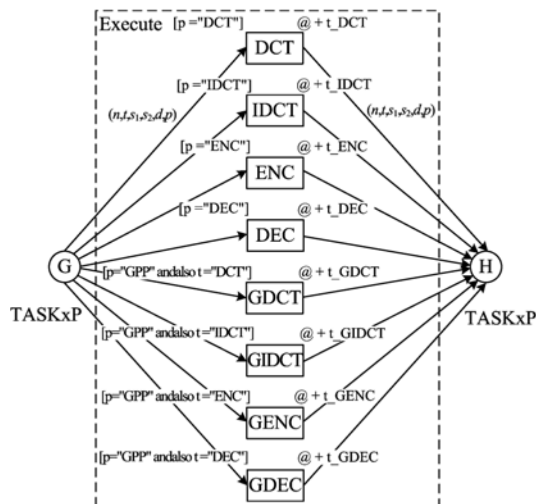

Fig. 5 Model of execute

3.3.6 Write result

Before a task writes back its computing result, the absence of anti-dependences should be

checked prior. In our scheme, an anti-dependence is modeled by the transition waiting for the absence of tokens in place locked data. Theoretically, this relation can be represented as an inhibitor arc. However, there is no direct support for inhibitor arcs in CPN tools. So we model the event using several CPN objects as illustrated in Fig. 6.

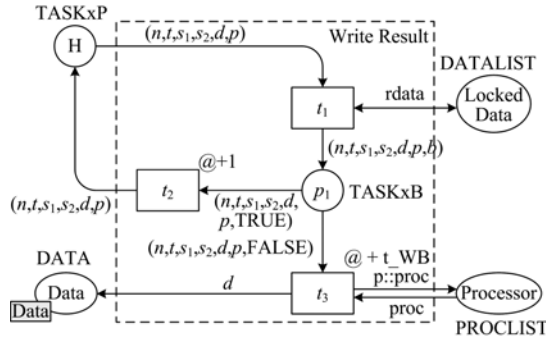

Fig. 6　Model of write result

The token in place locked data is a list which records the variables to be read. Once transition t_1 gets a token, it will check whether any earlier issued tasks are to read the destination variable of the current task. If yes, t_1 will produce a token (n, t, s_1, s_2, d, p, True), meaning there exists an anti-dependence. The produced token will enable the transition t_2. Subsequently, the transition t_2 will fire and enable t_1 again. Consequently, t_1 will keep on firing once every time unit until the earlier task has read the variable. By then, t_1 is able to fire, producing a token (n, t, s_1, s_2, d, p, False) rather than (n, t, s_1, s_2, d, p, True). The token will make the transition t_3 fire. The transition t_3 will return the token d to the place data, allowing the following tasks to write the corresponding variable. Besides, it will add the element p to the token in the place processor, meaning the corresponding processor is free for use again. The duration inscription is used to model the time spent on writing computing results to memory. After the firing of t_3, the processing flow for a task is over.

4　Evaluation

4.1　Prototype system

Our proposed scheduling scheme is generic and can be applied to a wide range of MPSoC platforms. For demonstration, we present the application of our scheme in a real-world platform, SOMP platform, within this section. The modeling results are further evaluated and prove to be correct.

SOMP is an FPGA based multiprocessor system which integrates heterogeneously embedded processors and hardware IP cores on a single chip. Fig. 7 illustrates the architectural view of SOMP platform, and the details on SOMP can be found in Ref. [4]. The scheduler is responsible for scheduling tasks and dispatching them into different processors. It can be implemented as either a software component running on an embedded processor or a standalone hardware module. SOMP offers the ability to fast build a prototype system and its integrated processors can be easily reconfigured. So we select it as our evaluation platform. We have built prototype systems on Xilinx Virtex-5 FPGA boards. Besides, we have applied our scheme to designing prototype systems and implemented a software edition scheduler. The details on the implementation of the scheduler can be found in Ref. [23].

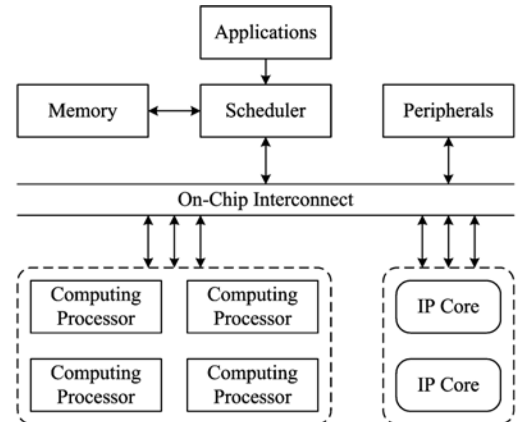

Fig. 7　Architecture of SOMP platform

To apply our scheduling scheme to SOMP, we must map functional components of a prototype system to different model elements. Besides, the initial marking of the model should be configured according to system specification. The CPN model is implemented using CPN tools. To get precise simulation results, all time parameters of CPN

models are configured according to the measurements on prototype systems, including task execution times, memory access times and bus transmission times.

Assisted by the built-in simulator in CPN tools, we conduct time based simulations on CPN models.

During simulation, each step of CPN model execution is recorded and output to a report file. The report file can be used to generate timing diagrams for investigating the process of task scheduling. The diagram shows the firing time and duration time of each transition that occurs during a simulation. By comparing the timing diagram with the actual result derived from prototype systems, we can guarantee the behavior of CPN model is consistent with that of the modeled system and ensure the correctness of the model execution.

4.2 Comparison experiments

In order to evaluate the performance of our proposed scheduling scheme, we also implement software edition of task superscalar[3] on the prototype system and conduct a series of comparison experiments. Task sequences listed in Tab. 2 are selected as test cases in the experiments. The four tasks in sequence 1 have no data dependences during execution, while the other task sequences have different data dependences of various occurring frequencies. Investigating the speedups for these task sequences, we can find out the performance of both task superscalar and our CPN based scheme when dealing with different dependences.

Tab. 2 Sample task sequences

Sequence 1 (no dependences)	Sequence 2 (50%[a] RAW)	Sequence 3 (100% RAW)
add(5,0)	add(5,1)	add(5,1)
idct(6,1)	idct(6,5)	idct(6,5)
enc(7,2,3)	enc(7,3,6)	enc(7,3,6)
dec(8,2,4)	dec(8,4,2)	dec(1,4,7)
Sequence 4 (50% WAW)	**Sequence 5 (100% WAW)**	**Sequence 6 (25% WAR)**
add(5,0)	add(5,0)	add(1,0)
idct(5,1)	idct(5,1)	enc(2,1,3)
enc(5,2,3)	enc(5,2,3)	dec(3,4,0)
dec(4,7,6)	dec(5,2,4)	idct(5,6,0)

【Note】 [a] The percentage presents the occurring frequency of data dependence

The prototype system integrates a GPP as the scheduling processor and four different ASPs (ADD, IDCT, ENC and DEC respectively). Besides, the prototype system is set to be of different configurations and the execution times of ASPs vary among configurations. For task sequences 1～5, all ASPs have the same execution time in each individual configuration, which are 5 000(5 k), 10 000(10 k), 20 000(20 k), 40 000 (40 k) and 80 000(80 k), respectively. For task sequence 6, the execution times for ASPs are shown in Tab. 3 so as to ensure the appearance of WAR dependences during task execution. By these means, we can evaluate the influence of task granularity on the performance of scheduling schemes. Moreover, each sequence is repeated executing for several times to simulate the applications with different task scales.

Tab. 3 Execution times of ASPs

Configuration	Execution times (clock cycles)	
	IDCT / AES_DEC	AES_ENC/ ADD
cfg1	5 000	10 000
cfg2	10 000	20 000
cfg3	20 000	40 000
cfg4	40 000	80 000

The experimental results are illustrated in Fig. 8, where the y coordinate denotes the speedups of each scheduling scheme to totally sequential execution, the x coordinate denotes the number of times that a sequence is repeated, while bars represent the speedups for different system configurations.

4.2.1 No data dependences

Fig. 8(a) shows the experimental results for task sequence 1, which has no data dependences. Theoretically, the speedups of both approaches are the same, and their upper limit equals to 4.0. Actually, however, our approach performs more effectively than task superscalar when given a fixed task granularity and scale. It demonstrates that our scheme introduces less time overhead thanks to its simplicity. Especially, the smaller the task granularity is, the lager the speedup gap between two schemes will be.

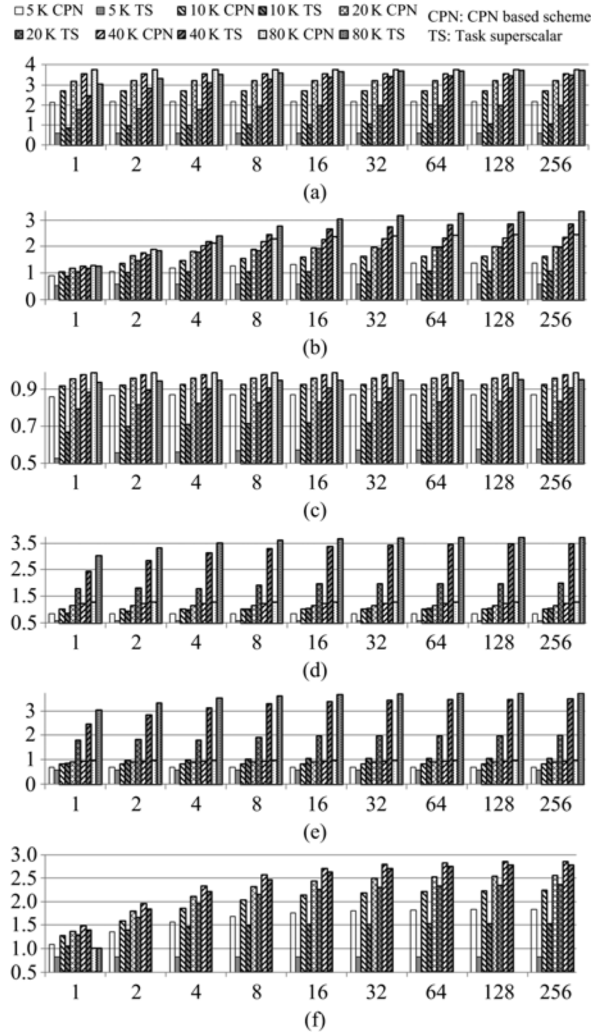

Fig. 8　Comparison of task superscalar and the CPN based scheme

For each individual scheme, when given a fixed task scale, the speedup will rise as the granularity of each task increases. That's because the scheduling overhead is fixed when given a task scale. Therefore, as task granularity increases, the proportion of the scheduling overhead to overall execution time decreases and the speedup goes up.

4.2.2　RAW dependences

Given sequence 2, the task execution order varies between two schemes due to the influence of structural dependences. 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 dependences can be executed in advance. On the contrary, our scheme must block all the following tasks when encountering structural hazards. For the reason above, task superscalar theoretically exhibits more excellent performance than our scheme. When given sequence 2, the upper limits of theoretical speedups for task superscalar are respectively 4.0 and 2.67.

The experimental results in these two cases are shown in Fig. 8(b), where we can see that, task superscalar achieves higher speedups than our approach when the execution time of each task is large enough. That's because task superscalar is able to uncover more potential parallelisms than our scheme. As task execution time decreases, the influence of scheduling overhead on system performance is enhanced. Consequently, the speedups of both approaches decrease. The speedup of task superscalar decreases more significantly than our scheme since the time overhead brought in by task superscalar is much greater than that of our scheme. 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.

For sequence 3, both approaches achieve the same theoretical speedup (equal to 1.0) since neither task superscalar nor our scheme can eliminate RAW dependences. Fig. 8(c) illustrates the comparison of our approach and task superscalar in this case. Observing the figure, we can find the same phenomena as in the pervious experiment: ① given a fixed task scale, the speedups for both approaches increase as the task granularity goes up; ② given a fixed task sequence and the scale, our scheme achieves higher speedups than the other. The reasons for these phenomena are the same as those for sequence 1.

4.2.3　WAW dependences

Task sequences 4 and 5 have increasing frequencies of WAW hazards, which are 50% and 100%, respectively. Unlike our scheme, 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 4 and 5, the upper limits of theoretical speedups for task superscalar are all 4.0, while those for our scheme are 1.33 and 1.0, respectively.

Fig.8(d) and Fig.8(e) respectively show the actual experimental results for comparison of both approaches when given sequences 4 and 5. Observing Fig.8(d), we can find that task superscalar can benefit from tapping out more parallelism and achieve better overall performances when the task granularity is large enough. However, as task granularity decreases below a threshold, our scheme shows better performance than task superscalar, benefiting from its lower scheduling overhead. The threshold is nearly 10 000 cycles for sequence 4, while for sequence 5, the threshold falls below 10 000 cycles.

Based on the observation on Fig.8(d) and Fig.8(e), we can also evaluate the effect of the occurring frequency of data dependences 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 to task sequence 5, where the occurring frequency of WAW dependences is 100%. When the frequency drops to 50% (sequence 4), the speedups for both approaches are more or less the same. That's because, as the frequency decreases, our scheme benefits from the increasing parallelism hidden in tasks and thereby achieves higher speedups, while the speedups for task superscalar remain unchanged. It can be concluded that our scheme can achieve greater speedups than task superscalar when the frequency drops below 50%.

4.2.4 WAR dependences

The comparison of two schemes on sequence 9 is illustrated in Fig.8(f). 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 the hardware platform demonstrates our approach achieves better performances in this case. The reason is that our scheme brings in lower time overhead when scheduling tasks. Benefiting from that, our approach achieves higher speedups, although it can tap out less parallelism than task superscalar.

Through comparison experiments on various sample task sequences, we can make the following conclusions:

① Our scheme introduces less time overhead than task superscalar during scheduling. Benefiting from that, even though both approaches tap out more or less the same quantity of parallelisms, our scheme will achieve higher speedups.

② Although task superscalar can uncover more parallelism by utilizing renaming mechanism when encountering WAW and WAR hazards, our approach still outperforms if the occurring frequency of these hazards is low or the task granularity is small.

5 Conclusion

In this paper, we propose a CPN based scheduling scheme 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 dependences can be detected automatically. Besides, different dispatching strategies can be easily evaluated in our CPN based scheme. Our proposed scheme is implemented in CPN tools. The correctness of our scheme is demonstrated through state space analyses and case studies, and the effectiveness is evaluated by comparison with state-of-art task superscalar.

References

[1] Kumar, Hughes C J, Nguyen A. Carbon: Architectural support for fine-grained parallelism on chip multiprocessors[C]// Proceedings of the 34th Annual International Symposium on Computer Architecture. San Diego, USA: ACM Press, 2007: 162-173.

[2] Sanchez D, Yoo R M, Kozyrakis C. Flexible

architectural support for fine-grain scheduling [C]// Proceedings of the 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems. Pittsburgh, USA: ACM Press, 2010: 311-322.

[3] Etsion Y, Cabarcas F, Rico A, et al. Task superscalar: An out-of-order task pipeline [C]// Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture. Atlanta, USA: IEEE Computer Society, 2010: 89-100.

[4] Wang C, Zhang J N, Zhou X H, et al. SOMP: Service-oriented multi processors[C]// Proceedings of the IEEE International Conference on Services Computing. Washington, USA: IEEE Computer Society, 2011: 709-716.

[5] Alur R, Dill D L. A theory of timed automata[J]. Theoretical Computer Science, 1994, 126 (2): 183-235.

[6] Alur R, La Torre S, Pappas G J. Optimal paths in weighted timed automata[J]. Theoretical Computer Science, 2004, 318(3): 297-322.

[7] Behrmann G, Fehnker A, Hune T, et al. Minimum-cost reachability for priced timed automata [C]// Proceedings of the 4th International Workshop on Hybrid Systems: Computation and Control. Rome, Italy: Springer-Verlag, 2001: 147-161.

[8] Abdeddaïm Y, Kerbaa A, Maler O. Task graph scheduling using timed automata[C]// Proceedings of the 17th International Symposium on Parallel and Distributed Processing. Washington, USA: IEEE Computy Society, 2003: 237. 2(1-8).

[9] Behrmann G, Larsen K G, Rasmussen J I. Optimal scheduling using priced timed automata [J]. ACM SIGMETRICS Performance Evaluation Review, 2005, 32(4): 34-40.

[10] Srba J. Comparing the expressiveness of timed automata and timed extensions of petri nets [C]// Proceedings of the 6th International Conference on Formal Modeling and Analysis of Timed Systemss. Saint Malo, France: Springer-Verlag, 2008: 15-32.

[11] Murata T. Petri nets: Properties, analysis and applications [J]. Proceedings of the IEEE, 1989, 77(4): 541-580.

[12] Jensen K. Coloured Petri nets [A]// Petri Nets: Central Models and Their Properties, Lecture Notes in Computer Science. Springer, 1987, 254: 248-299.

[13] Zuberek W M, Govindarajan R, Suciu F. Timed colored Petri net models of distributed memory multithreaded multiprocessors[C]// Proceedings of the Workshop on Practical Use of Colored Petri Nets and Design. Aarhus, Denmark, 1998: 253-270.

[14] Azgomi M A, Entezari-Maleki R. Task scheduling modelling and reliability evaluation of grid services using coloured Petri nets [J]. Future Generation Computer Systems, 2010, 26(8): 1 141-1 150.

[15] Blej M, Azizi M. Modeling and analysis of a real-time system using the networks of extended Petri [J]. Journal of Computers, 2009, 4(7): 641-645.

[16] Madhukar M, Leuze M, Dowdy L. Petri net model of a dynamically partitioned multiprocessor system[C]// Proceedings of the 6th International Workshop on Petri Nets and Performance Models. Durham, UK: IEEE Computer Society, 1995: 73-82.

[17] Tavares E, Oliveira Jr M, Maciel B, et al. Pre-runtime scheduling considering timing and energy constraints in embedded systems with multiple processors[C]// IFIP working Conference on Model-Driven Design to Resource Management for Distributed Embedded Systems. Braga, Portugal: Springer, 2006: 255-264.

[18] Barreto R, Maciel P, Neves M, et al. A novel approach for off-line multiprocessor scheduling in embedded hard real-time systems[C]// FIP working Conference on Design Methods and Applications for Distributed Embedded Systems. Toulouse, France: Springer, 2004: 157-166.

[19] Neubauer F, Hoheisel A, Geiler J. Workflow-based Grid applications [J]. Future Generation Computer Systems, 2006, 22(1-2): 6-15.

[20] Hoheisel A, Der U. Dynamic workflows for grid applications[C]// Proceedings of the 3rd Cracow Grid Workshop. Krakau, Polen, 2003 (http://www.andreas-hoheisel.de/).

[21] Eskinazi R, de Lima M E, Maciel P R M, et al. A timed Petri net approach for pre-runtime scheduling in partial and dynamic reconfigurable systems [C]// Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium. Denver, USA: IEEE Computer Society, 2005: 330-337.

[22] Jensen K. An introduction to the theoretical aspects of coloured Petri nets [A]// A Decade of Concurrency Reflections and Perspectives, Lecture Notes in Computer Science. Springer, 1994, 803: 230-272.

[23] Wang C, Li X, Chen P, et al. Detecting data hazards in multi-processor system-on-chips on FPGA [C]// Proceedings of the 26th Parallel and Distributed Processing Symposium Workshops & PhD Forum. Shanghai, China: IEEE Press, 2012: 282-287.