Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

Mixed real-time scheduling of multiple DAGs-based applications on heterogeneous multi-core processors

Guoqi Xie[a,b,*], Gang Zeng[c], Liangjiao Liu[a,b], Renfa Li[a,b,*], Keqin Li[a,d]

[a] College of Computer Science and Electronic Engineering, Hunan University, China
[b] Key Laboratory for Embedded and Network Computing of Hunan Province, China
[c] Graduate School of Engineering, Nagoya University, Japan
[d] Department of Computer Science, State University of New York, New Paltz, New York, USA

ARTICLE INFO

Article history:
Received 13 December 2015
Revised 8 February 2016
Accepted 19 April 2016
Available online 20 April 2016

Keywords:
High performance
Heterogeneous multi-core processors
Partial priority scheduling
Mixed real-time scheduling
Timing constraint

ABSTRACT

As multi-core processors continue to scale, more and more multiple distributed applications with precedence-constrained tasks simultaneously and widely exist in multi-functional embedded systems. Scheduling multiple DAGs-based applications on heterogeneous multi-core processors faces conflicting high-performance and real-time requirements. This study presents a multiple DAGs-based applications scheduling optimization with respect to high performance and timing constraint. We first present the fairness and the whole priority scheduling algorithms from high performance and timing constraint perspectives, respectively. Thereafter, we mix these two algorithms to present the partial priority scheduling algorithm to meet the deadlines of more high-priority applications and reduce the overall makespan of the system. The partial priority scheduling is implemented by preferentially scheduling the partial tasks of high-priority applications, and then fairly scheduling their remaining tasks with all the tasks of low-priority applications. Both example and experimental evaluation demonstrate the significant optimization of the partial priority scheduling algorithm.

1. Introduction

1.1. Background

Multi-core processors are increasingly used in the implementation of embedded control systems as well as many other compute-intensive application. To satisfy the demand of multi-functional embedded systems such as real-time image recognition, automotive control systems, and human body interaction plus gesture control, heterogeneous processors are needed [1–4]. A heterogeneous multi-core processor integrates different types of processors in the same chip. As multi-core processors continue to scale, increasing multiple distributed applications (also called functions or dataflows in some studies) with precedence-constrained tasks simultaneously widely exist in multi-functional embedded systems [5]. To make full use of heterogeneous multi-cores processors, scheduling is always an important topic. Traditional real-time scheduling on multi-core processors are usually based on only tasks models [1–3]. That is, these works only consider the scheduling from a "task level" perspective. The real-time community has been recently active in trying to provide new models and update classic scheduling theory to such relatively new platforms and applications [6]. Different parallel applications models have been proposed for multi-core processors, e.g., the fork/join model [7] and the synchronous parallel model [4]. A model called directed acyclic graph (DAG) that reflects the complexity and parallelization of such applications was widely used [8–11]. The nodes represent the tasks and the edges represent the communication messages between the tasks in DAG [8,9]. Multiple DAGs represent multiple applications that run on the same multi-core platform [5].

1.2. Motivation

There is an increasing interest for multi-core processors requiring both high-performance and real-time requirements [6]. However, these two requirements are in conflict with each other in multi-functional real-time scheduling.

Minimizing the overall scheduling length (i.e., makespan) of the system is the major requirement from a high performance perspective [12,13], whereas meeting the deadlines of applications is one of the most important safe requirements from a timing constraint

* Corresponding author. Tel.: +8613203157161 (G. Xie), +8618673110801 (R. Li).
E-mail addresses: xgqman@hnu.edu.cn (G. Xie), sogo@ertl.jp (G. Zeng), llj1984109@qq.com (L. Liu), lirenfa@hnu.edu.cn (R. Li), lik@newpaltz.edu (K. Li).

http://dx.doi.org/10.1016/j.micpro.2016.04.007
0141-9331/© 2016 Elsevier B.V. All rights reserved.

perspective [14]. Generally, fairness polices (e.g., slowdown [12], round-robin [15], etc.) aim to reduce the overall makespan of the system or individual makespans of applications. However, different applications have varied deadlines. The system cannot meet the deadlines of all applications, particularly in resource-constrained embedded environments. Many applications miss their deadlines and thereby their timing requirements cannot be satisfied in this case.

Each application has a priority property, and a high-priority application means it has a highly important and strict timing constraint on the deadline. Thus, missing the deadline of such application may cause severe safe problems. Scheduling applications in decreasing order of priorities can meet the timing constraints of a few high-priority applications, but many other applications would still miss their deadlines. The system would even experience a substantially long makespan because fairness is completely ignored in the aforementioned situation.

In summary, the two requirements are in conflict with each other, and major solutions that consider different requirements are required. New algorithms for multiple DAGs-based applications scheduling should be designed to optimize high performance and timing constraint in resource-constrained embedded systems. The object of this study is devoted to the real-time scheduling of multiple DAGs-based applications on heterogeneous multi-core processors.

1.3. Our contributions

The main contributions of this study are as follows:

(1) We present the static multiple DAGs-based applications scheduling algorithm with round-robin fairness policy to minimize the overall makespan of the system from a high performance perspective.
(2) We present the static multiple DAGs-based applications scheduling algorithm with the whole priority policy to meet the deadlines of a few high-priority applications.
(3) We mix the fairness and whole priority policies and present the static multiple DAGs-based applications scheduling algorithm with the partial priority policy. This policy enable more applications to meet their deadlines from a timing constraint perspective while reducing the overall makespan of the system from a high performance perspective.
(4) We evaluate the different scheduling polices (e.g., fairness, whole priority, and partial priority polices) through several experiments to demonstrate the significant optimization of the proposed partial priority policy.

The rest of this paper is organized as follows. Section 2 reviews the related research. Section 3 presents the multiple applications model. Section 4 proposes the fairness scheduling algorithm. Section 5 proposes the whole priority scheduling algorithm. Section 6 proposes the partial priority scheduling algorithm. Section 7 verifies the performance of all our proposed algorithms. Section 8 concludes this paper.

2. Related works

List scheduling is one of the well-known methods of DAG-based application scheduling [8,9]. List scheduling includes two phases: the first phase orders tasks according to the descending order of priorities (task prioritizing), whereas the second phase allocates each task to a proper core (task allocation) [8]. Scheduling tasks of a DAG-based application for rapid execution is a well-known nondeterministic polynomial (NP)-hard optimization problem, and many heuristic list algorithms have been proposed to generate near-optimal solutions based on global non-preemptive scheduling policy [8,9,16,17]. Meanwhile, the issue of static (i.e., all applications are triggered by the same time) multiple DAGs-based applications scheduling were also conducted [12,18,19]. Merging multiple DAGs-based applications into one application for scheduling [18,19] is a simple but not effective method to minimize the overall makespan, because they do not considers the fairness among multiple DAGs-based application. Zhao et al. [12] first identified the fairness issue in multiple DAGs-based applications scheduling and proposed a fairness scheduling algorithm called Fairness with a slowdown-driven strategy that ensures the fairness of different applications. Online workflow management (OWM) [13] and fairness dynamic workflow scheduling (FDWS) [20] are related dynamic (i.e., all applications arrive at different time instants) multiple DAGs-based applications scheduling approaches. The main objective of the preceding investigations of multiple DAGs-based applications scheduling merely reduces the overall makespan of the system [12,13] or individual makespans of applications [12,20].

Some related scheduling algorithms are concerned about DAG-based application scheduling with deadline constraints [10,21,22]. However, these approaches are merely for single DAG-based application, and not suitable for the issues of multiple DAGs-based applications. Recently, the multiple DAGs-based parallel applications scheduling considering deadline were studied. In [14], Wang et al. presented the algorithm of considering maximize throughput of multi-DAGs under deadline constraint to improve the ratios of applications which can be accomplished within deadlines by timely abandoning the applications exceed their deadlines. However, some high-criticality applications cannot be abandoned in safety-critical systems such that the algorithm cannot be applied to such systems. Hu et al. investigated the scheduling of periodic applications on time-triggered systems [23]. The objective is to schedule all tasks in all instances of all applications in one hyper period to guarantee that all instances of all applications can meet their respective deadlines. However, these works merely focus on meeting the deadlines of all applications and do not include the system performance into consideration. In practical heterogenous systems, the deadlines of all applications cannot be met, particularly in resource-constrained environments. Moreover, different applications shouldn't completely fair and they should have different priorities according to their importance to systems. For example, at least three types of applications exist in automotive embedded systems according to the classification of safety, namely, active safety, passive safety, and non safety applications. Missing the deadline of high-priority (active safety) applications could cause severe safety problems.

In summarizing the above research of multiple DAGs-based applications scheduling, we find that there are two different objectives and no paper considers them together: (1) one objective aims to reduces the overall makespan of the system or individual makespans of applications, and disregard the deadline requirements of applications; (2) the other objective is to meets the deadlines of applications, and ignores the performance requirement of the system. To solve the above problem, the objective of this study is to enable more high-priority applications to meet their deadlines from a timing constraint perspective while reducing the overall makespan of the system from a high performance perspective.

3. Modeling

3.1. System model

We use $P = \{p_1, p_2, ..., p_{|P|}\}$ to represent a set of heterogeneous multi-cores that are completely inter-connected. All communications can be performed concurrently. $|P|$ represents the size of set P. This study uses $|X|$ to denote the size of any set X.

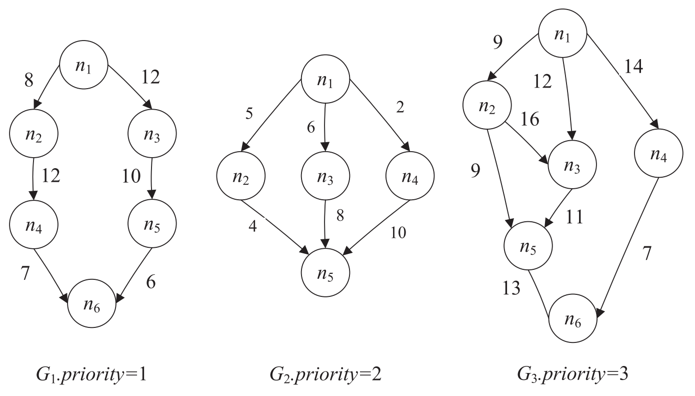

Fig. 1. Example of three applications with precedence-constrained tasks of heterogeneous multi-core processors.

The preceding assumption is equal to many famous classical algorithms (i.e., HEFT [9], critical path on a processor [9], and heterogenous selection value [8]) on heterogenous systems. We employ $PL = \{priority_1, priority_2, ..., priority_{|PL|}\}$ to represent a set of priorities. Each priority is denoted with a positive integer. The value 1 represents the lowest priority; the larger the value, the higher the priority.

A parallel application is represented by DAG G=(N, E, C, W, *priority, lowerbound, deadline, makespan*). N represents a set of $|N|$ nodes and each node $n_i \in N$ represents a task that has different computation times on various cores. E is a set of communication edges and each edge $e_{i,j} \in E$ represents the transmitted message from task n_i to n_j. Accordingly, $c_{i,j} \in C$ represents the communication time of $e_{i,j}$ only when n_i and n_j are assigned to different cores. The communication time is 0 if n_i and n_j are assigned to the same core. *pred*(n_i) represents the set of n_i's immediate predecessor tasks. A task is triggered to execute only if all its predecessor tasks have been executed. *succ*(n_i) represents the set of n_i's immediate successor tasks. The task that has no predecessor task is called n_{entry}, whereas the task that has no successor task is called n_{exit}. W is a $|N| \times |P|$ matrix, in which $w_{i,k}$ denotes the computation time to run task n_i on core p_k. *priority* $\in PL$ is the unique identifier and represents the priority of the application. If two applications G_m and G_n satisfy G_m.*priority* > G_n.*priority*, then G_m has a higher priority than G_n. The *lowerbound* means the minimum makespan of an application when all cores are monopolized by the application using a standard single DAG-based application scheduling algorithm (e.g., HEFT [9]). The *deadline* means the timing constraint of the application; this timing constraint should be larger than or equal to the *lowerbound*. Thus, the deadline should comply with the basic condition *deadline* ≥ *lowerbound*. Additional information on *lowerbound* and *deadline* is provided in Section 4.1. The system consists of multiple DAGs-based applications and is represented by $GS = \{\{G_1, G_2, G_3, ..., G_{|GS|}\}, makespan\}$. The *makespan* represents the overall makespan of GS (i.e., the maximum makespan of all applications) and reflects the performance of the system. Similar to all the works reviewed in Section 2, this study also considers the global non-preemptive scheduling policy.

3.2. Motivating example

Fig. 1 shows an example with three applications of G_1, G_2, and G_3. The priorities of G_1, G_2, and G_3 are 1, 2, and 3, respectively, where G_1 and G_3 have the highest and lowest priorities, respectively. Table 1 shows the computation times for G_1, G_2, and G_3 of Fig. 1. The example shows six, five, and six tasks for G_1, G_2, and G_3, respectively. This example assumes that three cores exist. Although the example is simple, it involves three cores, three application, and three priorities. Hence, this example can reflect the characteristics of heterogenous multi-core processors executing multiple DAGs-based applications with different priorities. The weight of 8 of the edge between tasks $G_1.n_1$ and $G_1.n_2$ in Fig. 1(a) represents the communication time when $G_1.n_1$ and $G_1.n_2$ are not assigned in the same core. The weight of 12 of $G_1.n_1$ and p_1 in Table 1(a) represents the computation time denoted by $G_1.w_{1,1}$=12.

Table 1
Computation time matrixes of the example in Fig. 1.

(a) Computation time matrix of G_1

Tasks	$G_1.n_1$	$G_1.n_2$	$G_1.n_3$	$G_1.n_4$	$G_1.n_5$	$G_1.n_6$
p_1	12	9	7	13	18	15
p_2	8	15	12	15	10	10
p_3	9	11	16	18	20	8
$rank_u$	77	58	55	34	33	11

(b) Computation time matrix of G_2

Tasks	$G_2.n_1$	$G_2.n_2$	$G_2.n_3$	$G_2.n_4$	$G_2.n_5$
p_1	4	9	18	21	7
p_2	5	10	17	15	6
p_3	6	11	16	19	5
$rank_u$	42	20	31	35	6

(c) Computation time matrix of G_3

Tasks	$G_3.n_1$	$G_3.n_2$	$G_3.n_3$	$G_3.n_4$	$G_3.n_5$	$G_3.n_6$
p_1	8	14	9	18	18	5
p_2	11	13	12	15	16	10
p_3	19	8	16	14	20	7
$rank_u$	110	91	63	31	39	8

4. Fairness scheduling

4.1. Lower-bound and deadline

HEFT is the well-known precedence-constrained task scheduling based on the DAG model to reduce makespan to a minimum combined with low complexity and high performance in heterogenous systems [9]. Two important contributions were proposed for the two phases of HEFT.

First, HEFT uses the upward rank value ($rank_u$) of a task given by Eq. (1) as the common task priority standard, where the tasks are arranged according to the decreasing order of $rank_u$. Table 1 shows the upward rank values of all tasks shown in Fig. 1 using Eq. (1)

$$rank_u(n_i) = \overline{w_i} + \max_{n_j \in succ(n_i)} \{c_{i,j} + rank_u(n_j)\}. \tag{1}$$

Second, the attributes $EST(n_j, p_k)$ and $EFT(n_j, p_k)$ represent the earliest start time (EST) and the earliest finish time (EFT), respectively, of task n_j on core p_k. $EFT(n_j, p_k)$ is considered the common task allocation criteria, because it can meet the local optimal of each precedence-constrained task using the greedy policy. Both are calculated by

$$\begin{cases} EST(n_{\text{entry}}, p_k) = 0; \\ EST(n_j, p_k) = max\left(avail[p_k], \max_{n_i \in pred(n_j)} \{AFT(n_i) + c'_{i,j}\}\right); \end{cases} \tag{2}$$

and

$$EFT(n_j, p_k) = EST(n_j, p_k) + w_{j,k}, \tag{3}$$

where $avail[p_k]$ is the earliest available time when core p_k is ready for task execution and $c'_{i,j}$ represents the communication time. If n_i and n_j are allocated to the same core, then $c'_{i,j} = 0$; otherwise, $c'_{i,j} = c_{i,j}$. $AFT(n_i)$ is the actual finish time of task n_i. n_j is allocated to the core with minimum EFT using an insertion scheduling policy. The insertion-based strategy is explained bellow: if n_j can be

Table 2
Properties of tasks and applications in Fig. 1.

Application	G_1	G_2	G_3
Task priority	$G_1.n_1$, $G_1.n_2$, $G_1.n_3$, $G_1.n_4$, $G_1.n_5$, $G_1.n_6$	$G_2.n_1$, $G_2.n_4$, $G_2.n_3$, $G_2.n_2$, $G_2.n_5$	$G_3.n_1$, $G_3.n_2$, $G_3.n_3$, $G_3.n_5$, $G_3.n_4$, $G_3.n_6$
priority	1	2	3
lowerbound	59	36	54
deadline	69	41	64

inserted into one of the slacks of cores, then it is inserted into the slack with the minimum EFT.

Lower-bound means the minimum makespan of an application when all cores are monopolized by the application using a standard single DAG-based application scheduling algorithm as mentioned earlier. HEFT is a well-known algorithm with low complexity and high performance that should be selected as the standard algorithm to assess parallel application. HEFT is used as the standard algorithm to explain the proposed algorithms in this study. Other algorithms can also be easily selected and employed for a simple replacement. The lower-bound of the application G_m is the actual finish time of n_{exit}, namely

$$G_m.lowerbound = AFT(G_m.n_{\text{exit}}), \tag{4}$$

where $G_m.n_{\text{exit}}$ represents the exit task of G_m. The user specifies a deadline for each application based on the value of the lower-bound after it is obtained.

Table 2 lists the properties for each application of the example in Fig. 1. The task priority of each application can be obtained according to the descending order of $rank_u(n_i)$, and then acquire the lower-bounds of the three application: $G_1.lowerbound = 59$, $G_2.lowerbound = 36$, and $G_3.lowerbound = 54$, respectively. The deadline of the application is specificated under the condition $G_m.deadline \geq G_m.lowerbound$. Finally, the deadlines of the three applications are $G_1.deadline = 69$, $G_2.deadline = 41$, and $G_3.deadline = 64$, respectively.

4.2. Scheduling framework

Fairness policies are effective approaches to achieve an improved performance. This subsection presents a round-robin fairness policies for multiple DAGs-based applications scheduling.

First, we present the scheduling framework for multiple DAGs-based applications on heterogeneous multi-core processors (Fig. 2).

The scheduling framework contains three priority queues, namely, task priority, common ready, and task allocation queues.

(1) The task priority queue (*task_priority_queue*) of each application is where tasks are ordered according to the decreasing $rank_u(n_i)$.
(2) The common ready queue (*common_ready_queue*) of the system is for storing ready tasks (one ready task with the maximum $rank_u(n_i)$ is selected from each application) and where the tasks are also ordered according to the decreasing $rank_u(n_i)$.
(3) The task allocation queue (*task_allocation_queue*) of each core is for storing allocated tasks.

Second, each step (see Fig. 2) of the proposed round-robin fairness policy in this paper is described as follows:

Step (1) Task priority: The tasks of each application are putted into corresponding task priority queue *task_priority_queue* according to the decreasing order of $rank_u(n_i)$.
Step (2) Task ready with round-robin fairness policy: ready tasks with the maximum $rank_u(n_i)$ are selected from application, and are putted into the common ready queue *common_ready_queue* according to the decreasing order of $rank_u(n_i)$.
Step (3) Task allocation with round-robin fairness policy: tasks with the maximum $rank_u(n_i)$ is selected from *common_ready_queue*, and are putted into the task allocation queue *task_allocation_queue* of the core with minimum $EFT(n_i)$ using the insertion scheduling policy.
Step (4) Task execution: Tasks are executed according to the corresponding cores after all tasks are assigned to the task allocation queues.

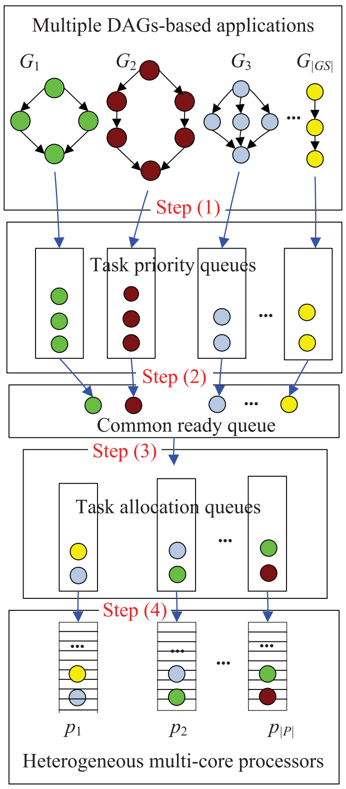

Fig. 2. Scheduling framework of heterogenous multi-core processors.

4.3. The F_MHEFT algorithm

We propose a scheduling algorithm called fairness on multiple HEFT (F_MHEFT) for parallel applications and describe the steps of the algorithm in Algorithm 1.

The time complexity of the F_MHEFT algorithm is analyzed as follows: Scheduling all applications must traverse all applications that can be completed in O(|GS|) time. Scheduling all tasks of an application can be completed in O($N_{\max}$) time, where $N_{\max}$=$max(|G_1.N|, |G_2.N|, \ldots, |G_{|GS|}.N|)$. Computing the EFT values of all tasks can be done in O($N_{\max} \times |P|$) time. Thus, the complexity of the F_MHEFT algorithm is O($|GS| \times N_{\max}^2 \times |P|$).

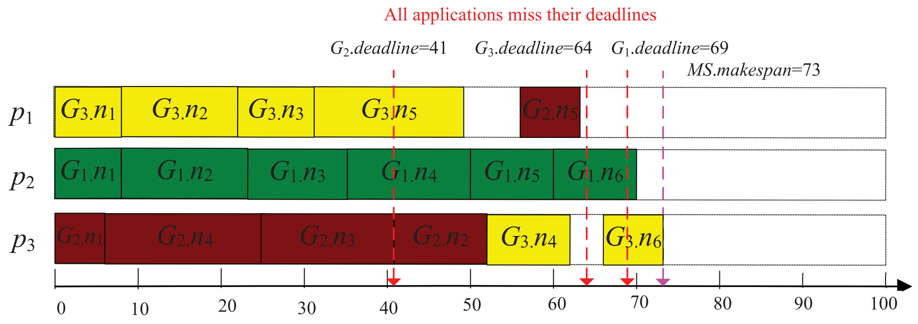

Fig. 3. Scheduling result of the example in Fig. 1 using F_MHEFT.

Algorithm 1 F_MHEFT algorithm

```
Input: $P = \{p_1, p_2, ..., p_{|P|}\}$, $GS = \{G_1, G_2, ..., G_{|GS|}\}$, and all types of queues.
Output: $\{G_1.makespan, G_2.makespan, ..., G_{|GS|}.makespan\}$, $GS.makespan$
1:  Calculate $rank_u(n_i)$ of each task;
2:  Put all tasks into the task priority queues of applications according to the descending order of $rank_u(n_i)$;
3:  $N_{max} = max(|GS.G_1.N|, |GS.G_2.N|, ...., |GS.G_{|GS|}.N|)$;
4:  while ($N_{max} > 0$) do
5:      for ($m = 1; m <= |GS|; m++$) do
6:          $n_i = task\_priority\_queue(G_m).out()$;
7:          $common\_ready\_queue.put(n_i)$;
8:      end for
9:      while ($!common\_ready\_queue.empty()$) do
10:         $n_i = common\_ready\_queue.out()$;
11:         Assign $n_i$ to $task\_allocation\_queue(p_k)$ of the core $p_k$ with the minimum EFT using the insertion-based scheduling policy;
12:     end while
13:     $N_{max}--$;
14: end while
```

Table 3
Task allocation step of the example in Fig. 1 using F_MHEFT.

Steps	Task allocation
1	$G_3.n_1$, $G_1.n_1$, $G_2.n_1$
2	$G_3.n_2$, $G_1.n_2$, $G_2.n_4$
3	$G_3.n_3$, $G_1.n_3$, $G_2.n_3$
4	$G_3.n_5$, $G_1.n_4$, $G_2.n_2$
5	$G_1.n_5$, $G_3.n_4$, $G_2.n_5$
6	$G_1.n_6$, $G_3.n_6$

Fig. 3 shows the scheduling process of the example in Fig. 1 using F_MHEFT. Accordingly, Table 3 shows the steps of the task allocation, where each step is a round-robin fairness for all application. The overall makespan of the system is 73. However, all applications miss their deadlines (Fig. 3).

5. Whole priority scheduling

The example using the F_MHEFT algorithm demonstrated that fairness scheduling is prone to minimize the overall makespan of the system and ignores the timing constraints of applications. Missing the deadlines of low-priority applications may cause no safe problem; however, severe safe problems will occur once the deadlines of high-priority applications are missed.

5.1. The WP_MHEFT algorithm

We propose a scheduling algorithm called whole priority on multiple HEFT (WP_MHEFT) for parallel applications with different priorities. The core idea of WP_MHEFT is preferentially scheduling high-priority applications by sacrificing the fair scheduling opportunity of low-priority application. S queue called *application_priority_queue* is introduced for the priority ordering of application. We describe the steps of this queue in Algorithm 2.

Algorithm 2 WP_MHEFT algorithm

```
Input: $P = \{p_1, p_2, ..., p_{|P|}\}$, $GS = \{G_1, G_2, ..., G_{|GS|}\}$, and all types of queues
Output: $\{G_1.makespan, G_2.makespan, ..., G_{GS}.makespan\}$, $GS.makespan$
1:  Calculate $rank_u(n_i)$ of each task;
2:  Put all tasks into the task priority queues of applications according to the descending order of $rank_u(n_i)$;
3:  Put all applications into the $application\_priority\_queue$ according to the descending order of $priority(G_m)$;
4:  while ($!application\_priority\_queue.empty()$) do
5:      $G_m = application\_priority\_queue(GS).out()$;
6:      while ($!task\_priority\_queue(G_m).empty()$) do
7:          $n_i = task\_priority\_queue(G_m).out()$;
8:          Assign $n_i$ to $task\_allocation\_queue(p_k)$ of the core $p_k$ with the minimum EFT using the insertion-based scheduling policy;
9:      end while
10: end while
```

The time complexity of the WP_MHEFT algorithm is analyzed as follows. Scheduling all applications must traverse all application, which can be completed in O(|*GS*|) time. Scheduling all tasks of an application can be completed in O(N_{max}) time. Computing the EFT values of all tasks can be done in O($N_{max} \times |P|$) time. Thus, the time complexity of the WP_MHEFT algorithm is also O($|GS| \times N^2_{max} \times |P|$).

5.2. Example of the WP_MHEFT algorithm

Fig. 4 shows the scheduling result of the example in Fig. 1 using WP_MHEFT. Table 4 shows the steps of task allocation. Figs. 3 and 4 show that the overall makespan of the system are 73 and 100 using F_MHEFT and WP_MHEFT, respectively. F_MHEFT performs better than WP_MHEFT from the high performance perspective. However, all applications miss their deadlines using F_MHEFT (Fig. 3), whereas G_2 can meet its timing constraint using WP_MHEFT (Fig. 4).

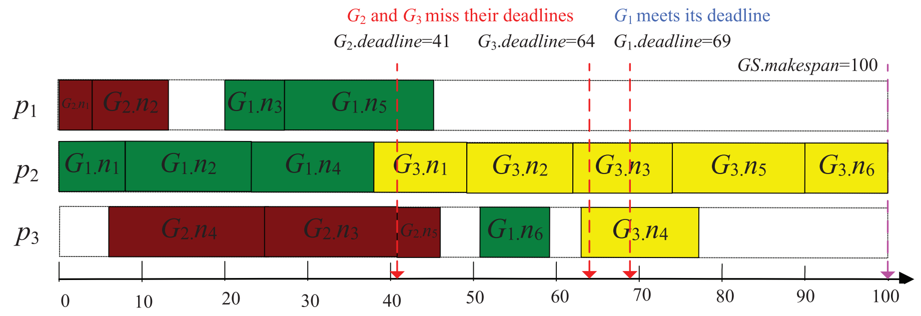

Fig. 4. Scheduling result of the example in Fig. 1 using WP_MHEFT.

Table 4
Task allocation step of the example in Fig. 1 using WP_MHEFT.

step	task allocation
1	$G_1.n_1$, $G_1.n_2$, $G_1.n_3$, $G_1.n_4$, $G_1.n_5$, $G_1.n_6$
2	$G_2.n_1$, $G_2.n_4$, $G_2.n_3$, $G_2.n_2$, $G_2.n_5$
3	$G_3.n_1$, $G_3.n_2$, $G_3.n_3$, $G_3.n_5$, $G_3.n_4$, $G_3.n_6$

6. Partial priority scheduling

On the one hand, the round-robin fairness policy can reduce the overall makespan of the system for an improved performance. We can use the F_MHEFT algorithm (Algorithm 1) to schedule together all applications with different priorities. The overall makespan can be reduced to a minimum, but the deadlines of the vast majority of applications may be missed. Note that a high-priority application means highly important and strict timing constraint on deadline. The F_MHEFT algorithm is significantly optimistic on timing constraint because it is only concerned with the performance of the system and ignores the timing constrains of all application.

On the other hand, the deadlines of all applications cannot be met in resource-constrained embedded systems. The WP_MHEFT algorithm (Algorithm 2) can meet the deadlines of a few high-priority applications by preferentially scheduling whole tasks of these application. However, WP_MHEFT is urgently meets the deadlines of high-priority applications at the start and causes many other applications not to be handled positively. The deadlines of these applications are missed, and the overall makespan of the system is considerably long. The WP_MHEFT algorithm is pessimistic on timing constrain, because it is only concerned with the deadlines of a few high-priority applications, and ignores the deadlines of other high-priority applications and performance of the system.

Timing constraint is not to schedule high-priority applications earlier. In fact, a time span exists between the lower-bound and deadline of an application. Timing constraint is still met as long as the application executes completely before the deadline.

6.1. The PP_MHEFT algorithm

We present a optimized scheduling algorithm called partial priority on multiple HEFT (PP_MHEFT) to make more applications can meet their deadlines and reduce the overall makespan as much as possible. The steps are described in Algorithm 3.

The time complexity of the PP_MHEFT algorithm is analyzed as follows. Scheduling all applications must traverse all application, which can be completed in $O(|GS|)$ time. Scheduling all tasks of an application can be completed in $O(n_{\max})$ time. Scheduling using

Algorithm 3 PP_MHEFT Algorithm

```
Input: $P = \{p_1, p_2, ..., p_{|P|}\}$, $GS = \{G_1, G_2, ..., G_{|GS|}\}$, and all types of queues
Output: $\{G_1.makespan, G_2.makespan, ..., G_{|GS|}.makespan\}$, $GS.makespan$
1: Calculate $rank_u(n_i)$ of each task;
2: Put all tasks into the task priority queues of applications according to the descending order of $rank_u(n_i)$;
3: Put all applications into the $application\_priority\_queue$ according to the descending order of $priority(G_m)$;
4: while ($!application\_pritority\_queue.empty()$) do
5:   $G_m = application\_priority\_queue(GS).out()$;
6:   Obtain the applications the priorities of which are less than or equal to $G_m$ and put them to the low-priority application set $low\_application\_set$;
7:   Probe to fairly schedule $G_m$ and all the applications in $low\_application\_set$ together using the F_MHEFT algorithm;
8:   while ($G_m.makespan > G_m.deadline$) do
9:     Cancel the previous allocation of the F_MHEFT algorithm;
10:    if ($!task\_priority\_queue(G_m).empty()$) then
11:      $n_i = task\_priority\_queue(G_m).out()$;
12:      Allocate $n_i$ to $task\_allocation\_queue(p_k)$ of the core $p_k$ with the minimum EFT using the insertion-based scheduling policy;
13:      Probe to fairly schedule the remaining task of $G_m$ and all the applications in $low\_application\_set$ together using the F_MHEFT algorithm;
14:    end if
15:  end while
16: end while
```

the F_MHEFT algorithm can be completed in $O(|GS| \times N_{\max}^2 \times |P|)$ time. Thus, the complexity of the PP_MHEFT algorithm is $O(|GS|^2 \times N_{\max}^3 \times |P|)$. The key points of the PP_MHEFT algorithm are described as follows.

(1) We probe to fairly schedule G_m and all the low-priority applications together using the F_MHEFT algorithm. If $G_m.makespan \le G_m.deadline$, then the F_MHEFT algorithm can meet the deadline of G_m. Otherwise, the previous allocation of the F_MHEFT algorithm should be canceled. The partial tasks of G_m are preferentially scheduled until G_m can meet its deadline by fairly scheduling the remaining tasks of G_m and its low-priority applications using the F_MHEFT algorithm.

(2) We constantly check the deadline of G_m after a task of G_m is preferentially scheduled such that the number of preferentially scheduled tasks of G_m is minimum.

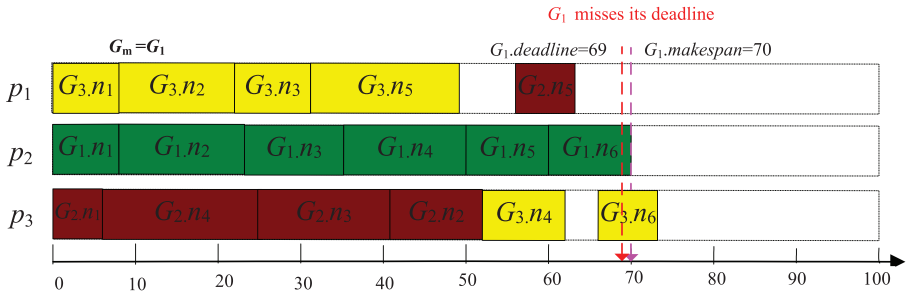

Fig. 5. Scheduling process 1 of the example in Fig. 1 using PP_MHEFT.

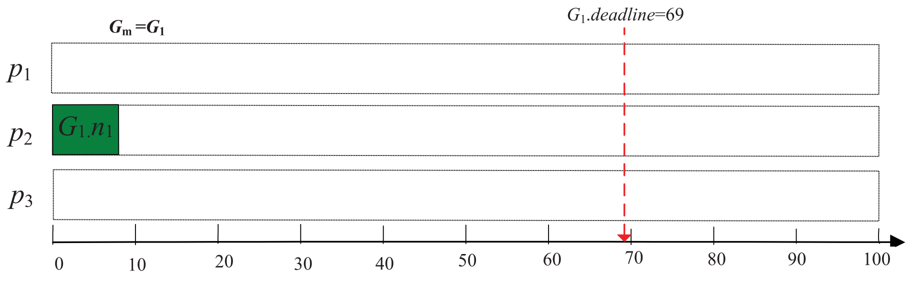

Fig. 6. Scheduling process 2 of the example in Fig. 1 using PP_MHEFT.

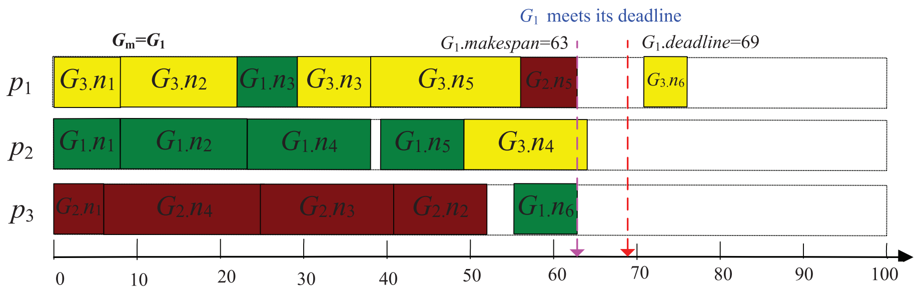

Fig. 7. Scheduling process 3 of the example in Fig. 1 using PP_MHEFT.

6.2. Example of the PP_MHEFT algorithm

Figs. 5–9 show the whole scheduling process of the example in Fig. 1 using PP_MHEFT.

(1) The current application is G_1 because it is the highest priority application of the system. We use the F_MHEFT algorithm to schedule $\{G_1, G_2, G_3\}$ together. The makespan of G_1 is $G_1.makespan = 70$, which is larger than $G_1.deadline = 69$. G_1 misses its deadline (Fig. 5).
(2) To meet the deadline of G_1, we cancel the allocations of $\{G_1, G_2, G_3\}$ and use the partial priority policy to allocate task $G_1.n_1$ to p_2 (Fig. 6).
(3) Schedule the remaining tasks (i.e., $G_1.n_2$, $G_1.n_3$, $G_1.n_4$, $G_1.n_5$, $G_1.n_6$) of G_1 and low-priority applications (i.e., G_2 and G_3) using the F_MHEFT algorithm. The makespan of G_1 is reduced to $G_1.makespan = 63$, which is smaller than $G_1.deadline = 69$ (i.e., G_1 meets its deadline) (Fig. 7).
(4) Given that G_1 has bee scheduled, we change the current application to G_2, which is the highest priority application to be scheduled in the remaining applications. We can see from Fig. 7 that the makespan of G_2 is $G_2.makespan = 63$, which is larger than $G_2.deadline = 41$ (i.e., G_2 misses its deadline). To meet the deadline of G_2, we cancel the previous allocations of $\{G_2, G_3\}$, and use the partial priority policy to allocate tasks $G_2.n_1$, $G_2.n_4$, $G_2.n_3$, and $G_2.n_2$ of G_2 (Fig. 8).
(5) The remaining task $G_2.n_5$ of G_2 and all tasks of G_3 are scheduled using the F_MHEFT algorithm. The makespan of G_2 is reduced to $G_2.makespan = 41$, which is equal to $G_2.deadline = 41$ (i.e, G_2 misses its deadline) (Fig. 9).

Finally, $G_3.makespan = 83$ is larger than $G_3.deadline = 61$ (i.e, G_3 misses its deadline). Thus,the overall makespan of the system is $GS.makespan = 83$.

Table 5 shows the results of the example using the three algorithms. The overall makespan of the system is 83 using PP_MHEFT.

Combined with results of F_MHEFT (Fig. 3) and WP_MHEFT (Fig. 4), we can find the following results.

(1) F_MHEFT has a short overall makespan value of 73, but all applications with different priorities miss their deadlines.
(2) WP_MHEFT preferentially schedules G_1 because it is the highest priority application. Even though the deadline of G_1 is met, the result is an increased overall makespan of the system with value of 100.
(3) PP_MHEFT has a relatively short overall makespan value of 83, which is smaller than WP_MHEFT and longer than F_MHEFT. However, PP_MHEFT meets the deadline of G_1 and G_2 by sacrificing minimum partial fairness to implement minimum partial priority allocation. Obviously, PP_MHEFT makes maximum optimization between fairness and priority on the example. However, PP_MHEFT

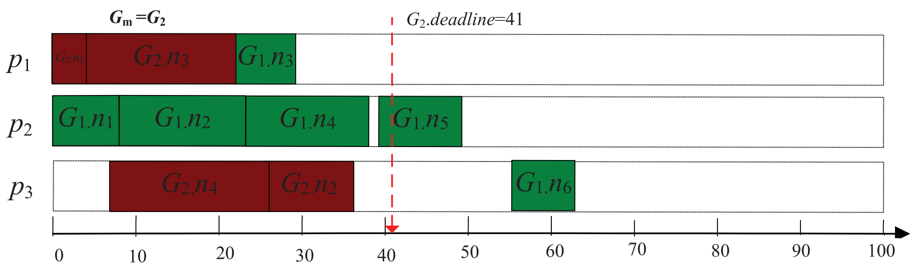

Fig. 8. Scheduling process 4 of the example in Fig. 1 using PP_MHEFT.

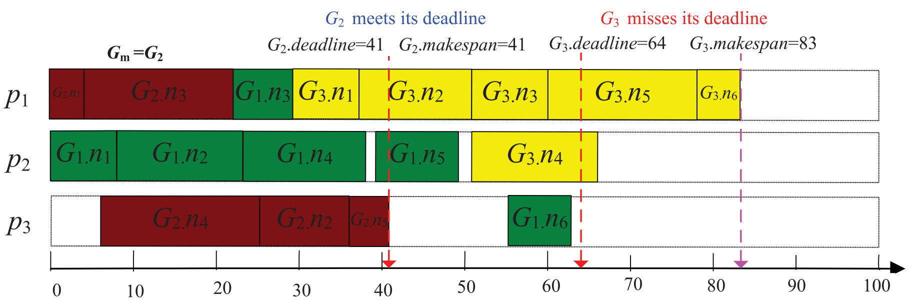

Fig. 9. Scheduling process 5 of the example in Fig. 1 using PP_MHEFT.

Table 5
Results of the example in Fig. 1 using WP_MHEFT.

Algorithm	F_MHEFT	WP_MHEFT	PP_MHEFT
Overall makespan	73	100	83
Applications meeting deadlines		G_1	G_1, G_2
Time complexity	$(\lvert GS\rvert \times N_{\max}^2 \times \lvert P\rvert)$	$(\lvert GS\rvert \times N_{\max}^2 \times \lvert P\rvert)$	$O(\lvert GS\rvert^2 \times N_{\max}^3 \times \lvert P\rvert)$

has a higher time complexity than both F_MHEFT and WP_MHEFT.

6.3. Summarization of the three algorithm

The F_MHEFT, WP_MHEFT, and PP_MHEFT algorithms are compared as follows:

(1) The F_MHEFT algorithm considers that all the applications are scheduled fairly and completely ignore the priorities of different application.
(2) The WP_MHEFT algorithm considers that low-priority applications can be scheduled only if the entire high-priority applications are scheduled.
(3) The PP_MHEFT algorithm first preferentially schedules partial tasks of the highest priority application. Then it fairly schedules the remaining tasks of the highest priority applications and all the tasks of low-priority applications using F_MHEFT. The PP_MHEFT algorithm actually mix the fairness and whole priority policies, and thereby it can meet the deadlines of more high-priority applications and reduce the overall makespan of the system as much as possible.

7. Experiments

7.1. Experimental metrics

The performance metrics selected for comparison are the DMR of applications [24] and overall makespan of the system [20].

Overall makespan is given by

$$MS.makespan = \max_{G_m \in MS}\{G_m.makespan\}. \tag{5}$$

Table 6
Overall makespans (μs) for varying numbers of applications.

Algorithms	FDWS	F_MHEFT	WP_MHEFT	PP_MHEFT
$\lvert MS\rvert = 30$	7978	7484	8135	7715
$\lvert MS\rvert = 40$	7470	7055	7475	7300
$\lvert MS\rvert = 50$	7937	7612	10036	7967
$\lvert MS\rvert = 60$	8738	8352	10694	9208
$\lvert MS\rvert = 70$	9208	8799	12815	9333

DMR is calculated using

$$DMR(X) = \frac{\lvert MS^{miss}(X)\rvert}{\lvert MS(X)\rvert}, \tag{6}$$

where $\lvert MS^{miss}(X)\rvert$ represents the number of the application missing their deadlines and $\lvert MS(X)\rvert$ represents the number of all the applications with parameter X.

We implemented a simulated heterogenous multi-core platform using Java on a standard desktop computer (2.6 GHz Intel CPU and 4 GB memory). This simulated platform contains 32 heterogeneous cores and can generate and run a variety of applications samples with different priorities. Application samples are randomly (an uniform distribution) generated depending on the following realistic parameters of automotive applications. $100\mu s \le w_{i,\,k} \le 400\mu s$, $100\mu s \le c_{i,\,j} \le 400\mu s$, $8 \le \lvert N\rvert \le 23$.

7.2. Experimental results

Experiment 1. This experiment is to compare the overall makespans of the system and DMRs of applications on different scale application sets. The number of applications is changed in the 30 to 70 range with 10 increments. Each application was

Table 7
DMRs for varying numbers of applications.

Algorithms	FDWS			F_MHEFT			WP_MHEFT			PP_MHEFT		
DMR	TOTAL	LOW	HIGH	TOTAL	LOW	HIGH	TOTAL	LOW	HIGH	TOTAL	LOW	HIGH
$\lvert MS\rvert = 30$	0.867	0.867	0.867	0.7	0.6	0.8	0.567	0.733	0.4	0.533	0.667	0.4
$\lvert MS\rvert = 40$	0.925	0.95	0.9	0.9	0.9	0.9	0.75	0.95	0.55	0.625	0.85	0.4
$\lvert MS\rvert = 50$	0.98	0.96	1.0	1.0	1.0	1.0	0.72	0.96	0.48	0.62	0.96	0.28
$\lvert MS\rvert = 60$	1.0	1.0	1.0	1.0	1.0	1.0	0.72	1.0	0.44	0.66	0.9	0.43
$\lvert MS\rvert = 70$	1.0	1.0	1.0	1.0	1.0	1.0	0.814	1.0	0.628	0.8	1.0	0.6

Table 8
DMRs for varying deadlines.

Algorithms	FDWS			F_MHEFT			WP_MHEFT			PP_MHEFT		
DMR	TOTAL	LOW	HIGH	TOTAL	LOW	HIGH	TOTAL	LOW	HIGH	TOTAL	LOW	HIGH
$G_m.deadline = G_m.lowerbound + G_m.lowerbound/40$	0.96	1.0	0.92	1.0	1.0	1.0	0.7	0.92	0.48	0.64	0.92	0.36
$G_m.deadline = G_m.lowerbound + G_m.lowerbound/30$	0.96	1.0	0.92	1.0	1.0	1.0	0.7	0.92	0.48	0.72	0.96	0.48
$G_m.deadline = G_m.lowerbound + G_m.lowerbound/20$	0.96	1.0	0.92	0.98	1.0	0.92	0.64	0.92	0.36	0.7	0.96	0.4
$G_m.deadline = G_m.lowerbound + G_m.lowerbound/10$	0.94	0.96	0.92	0.92	0.92	0.92	0.58	0.88	0.28	0.52	0.8	0.24
$G_m.deadline = G_m.lowerbound + G_m.lowerbound/5$	0.72	0.76	0.68	0.68	0.68	0.68	0.4	0.68	0.12	0.24	0.68	0.04

Table 9
Overall makespans (μs) for varying deadlines.

Algorithm	FDWS	F_DMHEFT	WP_DMHEFT	PP_DMHEFT
$G_m.deadline = G_m.lowerbound + G_m.lowerbound/40$	8481	8172	9251	8510
$G_m.deadline = G_m.lowerbound + G_m.lowerbound/30$	8481	8172	9251	8511
$G_m.deadline = G_m.lowerbound + G_m.lowerbound/20$	8481	8172	9251	8290
$G_m.deadline = G_m.lowerbound + G_m.lowerbound/10$	8481	8172	9251	8875
$G_m.deadline = G_m.lowerbound + G_m.lowerbound/5$	8481	8172	9251	8735

specificated a unique priority. The deadline of each application G_m is calculated as $G_m.deadline = G_m.lowerbound + G_m.lowerbound/40$. Four algorithms (i.e., FDWS [20], F_MHEFT, WP_MHEFT, and PP_MHEFT) are used for the experiment and then compared for verification. Similar to F_MHEFT, FDWS is a state-of-the-art fairness algorithm that minimizes individual makespans of applications. The main difference between FDWS and F_MHEFT in static scheduling is that FDWS and F_MHEFT select the task with highest $rank_{\mathrm{r}}(n_i)$ (Eq. (7)) and $rank_{\mathrm{u}}(n_i)$ from the common ready queue, respectively, in Step (3) of Fig. 2.

$$rank_{\mathrm{r}}(F_m.n_i) = \frac{1}{PRT(F_m)} \times \frac{1}{CPL(F_m)}, \tag{7}$$

where $PRT(F_m)$ and $CPL(F_m)$ represent the percentage of remaining task (PRT) number and the critical path length (CPL) of the function F_m, respectively.

Table 6 shows the overall makespans for varying numbers of applications using the four algorithms. For the algorithms with fairness, we can see that F_MHEFT has lower overall makespans than FDWS in all cases, the difference is about 400 μs. For the proposed algorithms in this study, F_MHEFT and WP_MHEFT generate best and worst performance, respectively; the overall makespans generated by PP_MHEFT are always between the results generated by F_MHEFT and WP_MHEFT.

Table 7 shows the DMRs for varying numbers of applications using the four algorithms. TOTAL represents the DMR of total applications, LOW represents the DMR of the low-priority applications (accounting for half of the total), and HIGH represents the DMR of the high-priority applications (accounting for half of the total). For the algorithms with fairness, the total DMRs are extremely high for FDWS (0.867-1.0) and F_MHEFT (0.7-1.0) in most cases, and all the results reach 1.0 when the applications number reaches 60. For the proposed algorithms in this study, F_MHEFT and PP_MHEFT generate highest and lowest DMRs (including TOTAL, LOW, and HIGH), respectively; the DMRs generated by WP_MHEFT are always between the results generated by F_MHEFT and PP_MHEFT.

Combining the results of Table 6 and 7, we find that PP_MHEFT obtains lowest DMR. Although it sacrifices a certain performance, the results still outperform WP_MHEFT and close to FDWS.

Experiment 2. Given that the system cannot meet the deadlines of all the high-priority applications in Experiment 1, the number of such applications should be reduced. A simple method for meeting the deadlines of all the high-priority applications is to modify the deadline. Hence, this experiment aims to modify the deadline of each application to observe the results. In this experiment, the number of applications is fixed with 50. The deadline of each application G_m is changed from $G_m.lowerbound + G_m.lowerbound/40$ to $G_m.lowerbound + G_m.lowerbound/5$. Note that the deadlines of functions cannot be modified in actual situation; hence, the objective of this experiment is merely to analyze the DMRs and overall makespans generated by different algorithms.

Table 8 shows the DMRs for varying deadlines using the FDWS, F_MHEFT, WP_MHEFT, and PP_MHEFT algorithms. We can see that the DMRs are reduced with the reductions of deadlines. When $G_m.deadline = G_m.lowerbound + G_m.lowerbound/5$, $DMR(HIGH)$ generated by WP_MHEFT and PP_MHEFT are reduced to 0.12 (three high-priority application misses its deadline) and 0.04 (only one high-priority application misses its deadline) miss their deadlines, respectively. $DMR(TOTAL)$ generated by WP_MHEFT and PP_MHEFT are reduced to 0.4 and 0.24, respectively. Note that when $G_m.deadline = G_m.lowerbound + G_m.lowerbound/20$, WP_MHEFT generates lower DMRs than PP_MHEFT in TOTAL, LOW, and HIGH; these results indicate that WP_MHEFT also has the contribution and necessity of existence in our proposed algorithms. Meanwhile, $DMR(HIGH)$ and $DMR(TOTAL)$ generated by FDWS and F_MHEFT are also reduced, but the values are still very high (larger than or equal to 0.68)

Table 9 shows the overall makespans for varying deadlines using the FDWS, F_MHEFT, WP_MHEFT, and PP_MHEFT algorithms.

Table 10
Running time (ms) for varying numbers of applications.

Algorithms	FDWS	F_MHEFT	WP_MHEFT	PP_MHEFT
$\|MS\| = 30$	490	223	186	7898
$\|MS\| = 40$	468	218	261	27635
$\|MS\| = 50$	487	203	184	49828
$\|MS\| = 60$	610	268	264	143344
$\|MS\| = 70$	675	366	353	267411

The overall makespan using FDWS, F_MHEFT, and WP_MHEFT are always fixed as 8481, 8172, and 9251 μs, respectively. We can see that F_MHEFT generates the lowest overall makespan of all the algorithms. Although the DMR generated by PP_DMHEFT is reduced to a minimum when $G_m.deadline = G_m.lowerbound + G_m.lowerbound/5$ (Table 8), the overall makespan is not reduced. In other words, changing the deadlines of applications does not changing the overall makespan of the system greatly for PP_DMHEFT, namely, PP_DMHEFT generates relatively stable overall makespan when the applications number is fixed.

Experiment 3. Given that the time complexity of the PP_MHEFT is higher than that of F_MHEFT and WP_MHEFT in the previous analysis, we are interested in observing the actual running time of different algorithms, although it is not critical in static scheduling.

Table 10 shows the running time of each algorithm for varying numbers of applications. As we expect, FDWS, F_MHEFT, and WP_MHEFT consume merely a few hundred milliseconds, whereas PP_MHEFT needs tens or even hundreds of thousands of milliseconds. With increasing number of applications, the consumed time of PP_MHEFT grows rapidly. For example, although the function number is changed from 60 to 70, the increased time is 267411-143344 = 124067 ms (nearly 2 minutes). Meanwhile, we can see the consumed time of F_MHEFT is about half of that of FDWS. The reason is that FDWS needs recalculate the $rank_r$ (Eq. (7)) of each task in the Step (3) of Fig. 2.

8. Conclusions

A mixed real-time scheduling of multiple DAGs-based applications on heterogeneous multi-core processors has been developed in this study. Considering that the timing constraint is not to schedule high-priority applications earlier and a time span exists between the lower-bound and deadline of an application, we presented the PP_MHEFT algorithm to meet the deadlines of high-priority applications and reduce the overall makespan of the system. PP_MHEFT is implemented by preferentially scheduling partial tasks of these applications; and then fairly scheduling the remaining tasks with other low-priority tasks together. PP_MHEFT actually mixes the fairness and whole priority policies, and thereby it can meet the deadlines of more high-priority applications and reduce the overall makespan of the system as much as possible. Therefore, PP_MHEFT significantly optimizes the static multiple DAGs-based applications scheduling between high performance and timing constraint. Experiments demonstrate that the PP_MHEFT algorithm can achieve useful optimization with high performance and timing constraint for static multiple DAGs-based applications scheduling on heterogeneous multi-core processors.

Acknowledgements

This study was partially supported by the National High-Tech Research and Development Plan of China with Grant No. 2012AA01A301-01, the Key Program of National Natural Science Foundation of China with Grant No. 61432005, and the National Natural Science Foundation of China with Grant Nos. 61173036, 61370095, 61300037, 61300039, 61379115, 61402170, and 61502405.

References

[1] K. Lakshmanan, R.R. Rajkumar, J.P. Lehoczky, Partitioned fixed-priority preemptive scheduling for multi-core processors, in: Real-Time Systems, 2009. ECRTS'09. 21st Euromicro Conference on, IEEE, 2009, pp. 239–248.
[2] N. Guan, M. Stigge, W. Yi, G. Yu, Fixed-priority multiprocessor scheduling with liu and layland's utilization bound, in: 2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium, IEEE, 2010, pp. 165–174.
[3] M. Fan, G. Quan, Harmonic semi-partitioned scheduling for fixed-priority real–time tasks on multi-core platform, in: Proceedings of the Conference on Design, Automation and Test in Europe, EDA Consortium, 2012, pp. 503–508.
[4] A. Saifullah, J. Li, K. Agrawal, C. Lu, C. Gill, Multi-core real-time scheduling for generalized parallel task models, Real-Time Syst. 49 (4) (2013) 404–435.
[5] J.C. Fonseca, V. Nélis, G. Raravi, L.M. Pinho, A multi-dag model for real-time parallel applications with conditional execution, in: Proceedings of the 30th Annual ACM Symposium on Applied Computing, ACM, 2015, pp. 1925–1932.
[6] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, G.C. Buttazzo, Response-time analysis of conditional dag tasks in multiprocessor systems, in: Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on, IEEE, 2015, pp. 211–221.
[7] K. Lakshmanan, S. Kato, R. Rajkumar, Scheduling parallel real-time tasks on multi-core processors, in: Real-Time Systems Symposium (RTSS), 2010 IEEE 31st, IEEE, 2010, pp. 259–268.
[8] G. Xie, R. Li, K. Li, Heterogeneity-driven end-to-end synchronized scheduling for precedence constrained tasks and messages on networked embedded systems, J. Parallel and Distrib. Comput. 83 (2015) 1–12.
[9] H. Topcuoglu, S. Hariri, M.-y. Wu, Performance-effective and low-complexity task scheduling for heterogeneous computing, Parallel and Distrib. Syst. IEEE Trans. 13 (3) (2002) 260–274.
[10] S. Abrishami, M. Naghibzadeh, D.H. Epema, Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds, Future Generation Comput. Syst. 29 (1) (2013) 158–169.
[11] M.A. Vasile, F. Pop, R.I. Tutueanu, V. Cristea, J. ołodziej, Resource-aware hybrid scheduling algorithm in heterogeneous distributed computing, Future Generation Comput. Syst. 51 (2015) 61–71.
[12] H. Zhao, R. Sakellariou, Scheduling multiple dags onto heterogeneous systems, in: Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International, IEEE, 2006, pp. 159–172.
[13] C.-C. Hsu, K.-C. Huang, F.-J. Wang, Online scheduling of workflow applications in grid environments, Future Generation Comput. Syst. 27 (6) (2011) 860–870.
[14] W. Wang, Q. Wu, Y. Tan, F. Wu, Maximize throughput scheduling and cost–fairness optimization for multiple dags with deadline constraint, in: Algorithms and Architectures for Parallel Processing, Springer, 2015, pp. 621–634.
[15] M. Tanaka, O. Tatebe, Workflow scheduling to minimize data movement using multi-constraint graph partitioning, in: Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012), IEEE Computer Society, 2012, pp. 65–72.
[16] C. Augonnet, S. Thibault, R. Namyst, P.-A. Wacrenier, Starpu: a unified platform for task scheduling on heterogeneous multicore architectures, Concurrency Comput.: Prac. Exp. 23 (2) (2011) 187–198.
[17] M.A. Khan, Scheduling for heterogeneous systems using constrained critical paths, Parallel Comput. 38 (4) (2012) 175–193.
[18] U. Hönig, W. Schiffmann, A meta-algorithm for scheduling multiple dags in homogeneous system environments, in: Proceedings of the Eighteenth IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS06), 2006, pp. 147–152.
[19] L.F. Bittencourt, E.R. Madeira, Towards the scheduling of multiple workflows on computational grids, J. Grid Comput. 8 (3) (2010) 419–441.
[20] H. Arabnejad, J. Barbosa, Fairness resource sharing for dynamic workflow scheduling on heterogeneous systems, in: Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on, IEEE, 2012, pp. 633–639.
[21] S. Abrishami, M. Naghibzadeh, D.H. Epema, Cost-driven scheduling of grid workflows using partial critical paths, Parallel and Distrib. Syst. IEEE Trans. 23 (8) (2012) 1400–1414.
[22] M. Mao, M. Humphrey, Auto-scaling to minimize cost and meet application deadlines in cloud workflows, in: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, ACM, 2011, pp. 1–12.
[23] M. Hu, J. Luo, Y. Wang, B. Veeravalli, Scheduling periodic task graphs for safety-critical time-triggered avionic systems, IEEE Trans. Aeros. Electron. Syst. 51 (2015) 2294–2304.
[24] S. Manolache, P. Eles, Z. Peng, Task mapping and priority assignment for soft real-time applications under deadline miss ratio constraints, ACM Trans. Embed. Comput. Syst. (TECS) 7 (2) (2008) 421–434.

Guoqi Xie received his Ph.D. degree in computer science from Hunan University, China, in 2014. He was a postdoctoral researcher at Nagoya University, Japan, from 2014 to 2015. Since 2015 he is working as a Postdoctoral Researcher at Hunan University, China. His major interests include embedded systems, distributed systems, real-time systems, in-vehicle networks, and cyber-physical systems. He is a member of IEEE and ACM.

Gang Zeng is an Associate Professor at the Graduate School of Engineering, Nagoya University. He received his Ph.D. degree in Information Science from Chiba University in 2006. From 2006 to 2010, he was a Researcher, and then Assistant Professor at the Center for Embedded Computing Systems (NCES), the Graduate School of Information Science, Nagoya University. His research interests mainly include power-aware computing and real-time embedded system design. He is a member of IEEE and IPSJ.

Liangjiao Liu received his master degree from Hunan University, China, in 2009. He is currently working towards his Ph.D. degree at Hunan University, China. His research interests include embedded computing, parallel computing, and automotive electronics.

Renfa Li is a full professor of computer science and electronic engineering, and the dean of College of Computer Science and Electronic Engineering, Hunan University, China. He is the director of the Key Laboratory for Embedded and Network Computing of Hunan Province, China. He is also an expert committee member of National Supercomputing Center in Changsha, China. His major research includes embedded computing, parallel computing, and cyber-physical systems. He is a senior member of IEEE, and a senior member of ACM.

Keqin Li is a SUNY Distinguished Professor of computer science. His current research interests include parallel computing and high-performance computing, distributed computing, energy-efficient computing and communication, heterogeneous computing systems, cloud computing, big data computing, CPU-GPU hybrid and cooperative computing, multicore computing, storage and file systems, wireless communication networks, sensor networks, peer-to-peer file sharing systems, mobile computing, service computing, Internet of things and cyber-physical systems. He has published over 370 journal articles, book chapters, and refereed conference papers, and has received several best paper awards. He is currently or has served on the editorial boards of IEEE Transactions on Parallel and Distributed Systems, IEEE Transactions on Computers, IEEE Transactions on Cloud Computing, Journal of Parallel and Distributed Computing. He is an IEEE Fellow.