2013 IEEE 11th International Conference on Dependable, Autonomic and Secure Computing

PK-OMLP: An OMLP based *k*-Exclusion Real-Time Locking Protocol for Multi-GPU Sharing Under Partitioned Scheduling

Maolin Yang[1], Hang Lei[1], Yong Liao[1], Furkan Rabee[2]
[1]School of Information and Software Engineering,
[2]School of computer science and engineering,
University of Electronic Science and Technology of China (UESTC), China
2006 Xiyuan Ave. 611731, Cheng Du, China
maolyang@gmail.com, hlei@uestc.edu.cn, liaoyong@uestc.edu.cn, forkanr@yahoo.com

***Abstract*—With rapid development of Graphics Processing Units (GPU) technologies, GPUs are strongly motivated to be adopted in many real-time applications. However, it is still a challenging work to efficiently integrate multiple GPUs into multicore/multiprocessor real-time systems, due to many real world constraints caused by the non-real-time closed-source GPU drivers. To avoid timing violations, *k*-exclusive locking protocols are developed to arbitrate exclusive access to each of the multiple GPUs. In this paper, a novel *k*-exclusion real-time locking protocol is proposed to handle multi-GPU sharing under partitioned fixed priority (P-FP) scheduling. The proposed protocol improves the prior work, the Clustered *k*-exclusion *O*(*m*) Locking Protocol (CK-OMLP) from two aspects: first, it allows multiple task on each CPU processor to make use of GPUs simultaneously, which improves the flexibility and increases GPU utilization in average cases; second, a suspension-aware analysis is presented to improve the schedulability, where task acquisition delays and GPU executions are modeled as self-suspensions. Schedulability experiments indicate that the proposed protocol outperforms the CK-OMLP in most considered scenarios.**

I. INTRODUCTION

With growing generality and rapid advancement of computing power, Graphics Processing Units (GPU) are strongly motivated to be adopted in real-time systems to speed up the executions of some computationally intensive workloads. Today, multiple GPUs can be interfaced to on-chip multicores as efficient accelerators to increase system level computational performance [1, 2]. For example, GPUs are capable of performing multidimensional Fast Fourier Transformations (FFTs), matrix operations, convolutions, etc. much faster than traditional CPUs [3, 4]. However, current GPU technologies are not real-time oriented. GPUs are considered to be non-preemptive I/O devices and are managed by closed-source non-real-time device drivers. Theses real world constraints challenge the design of real-time scheduling and synchronization algorithms. In the field of real-time systems, GPUs are currently treated as a pool of shared resources and are protected by real-time *k*-exclusion locking protocols [5, 6, 7, 8]. These protocols can be used to remove non-real-time GPU drivers from arbitrating access to GPUs, thus avoiding unbounded GPU effects from violating timing constraints as required by real-time systems. In this paper, we present a new *k*-exclusion locking protocol to support multi-GPU applications in partitioned scheduled multiprocessor real-time systems.

In multiprocessor real-time systems, tasks can be scheduled according to global or partitioned scheduling. Under global scheduling, there is only one ready queue that manage all the ready tasks in the system, and tasks are allowed to execute on any available CPU processor (i.e., task migrations are enabled). While under partitioned scheduling, tasks are bounded to specific processors. Correspondingly, each processor has a ready queue, and each task can only be scheduled on the processor where it is assigned. Experimental studies [9, 10] show that, compared with global scheduling, partitioned scheduling incurs much less runtime overheads, and it is implied to be preferable for hard real-time systems. Moreover, Partitioned Fixed-Priority (P-FP) scheduling has been a widespread choice: it is supported by many Real-Time Operating Systems (RTOSs) such as VxWorks, QNX, RTEMS, etc., it is also supported in industrial standards such as POSIX and AUTOSAR.

When resources (GPUs) are shared among tasks, locking protocols are required to be coupled with scheduling algorithms to bound schedulability penalties resulted from resource contentions. Spinning and suspension are two base mechanisms in designing locking protocols. Intuitively, processor cycles that are wasted by spinning can be saved if tasks suspend when blocked. Experimental results [11, 12, 13] indicate that suspension-based protocols are more efficient compared with spin-based ones, especially when critical sections are long. Since critical sections can be very long (even several seconds in length) in GPU applications, suspension-based locking protocols are well suited in GPU-enabled systems.

Although most *k*-exclusion locking protocols are suspension-based, existing analysis for these protocols are suspension-oblivious, which treats suspensions as normal CPU executions. Consequently, tasks are considered to be scheduled analytically even if they are suspended. This analytical constraint limits the shared resource usage under partitioned scheduling in that, no more than one task on each CPU processor can acquire one of the k ($k \geq 1$) shared resources at any time. Moreover, critical section length, including both GPU processing time and communication

978-1-4799-3381-5/13 $31.00 © 2013 IEEE
DOI 10.1109/DASC.2013.63

overheads between CPU and GPU, tend to be long, resulting in pessimistic results on worst case blockings.

In this paper, we modify the Clustered k-exclusion $O(m)$ Locking Protocol (CK-OMLP) [14], an asymptotically optimal protocol for clustered scheduling under suspension-oblivious analysis, and present a real-time Partitioned k-exclusion OMLP (PK-OMLP) for partitioned scheduled multiprocessors. Then, a suspension-aware analysis for the PK-OMLP is proposed base on our task model. Finally, we conduct schedulability experiments to show the superiority of the PK-OMLP over the CK-OMLP.

The rest of this paper is organized as follows. The related work is discussed in Sect. II. Background and notations is introduced in Sect. III. The PK-OMLP and the associated suspension-aware schedulability analysis are proposed in Sect. IV. Experimental results are shown in Sect. V. Conclusions and future work are presented in Sect. VI.

II. Related Work

To realize mutual exclusion of tasks that require get access to shared resources in multiprocessor real-time systems, many suspension-based locking protocols are proposed. These include the Multiprocessor Priority Ceiling Protocol (MPCP) [15] which favors priority queuing when tasks contend for locks, the Flexible Multiprocessor Locking Protocol (FMLP) [16] which favors FIFO queuing instead, and the OMLP [17] which uses a two-stage queuing mechanism to manage the ordering of contending tasks. Recent research on k-exclusion multiprocessor locking protocols has built upon these results by allowing up to k tasks to hold a pool of k similar or identical shared resources simultaneously.

Brandenburg et al. [14] presented an OMLP extension, the CK-OMLP, which supports k-exclusion locks in clustered scheduled multiprocessor systems. The CK-OMLP employs a priority donation mechanism to ensure that resource-holders are always scheduled, and thus that blocked jobs make progress to acquire the locks. The priority donation mechanism ensures that at most m (m is the number of CPU processors in the system) jobs are allowed to issue resource requests. If the CK-OMLP is used under partitioned scheduling (i.e., each cluster contains only one processor), it is ensures that no job is allowed to issue resource request once another job on that processor has issued a request before. This can be pessimistic for GPU applications under partitioned scheduling in that, only one job on each processor is allowed to get access to a GPU at any time even though there are GPUs available for other jobs on that processor. Further, the priority donation mechanism may cause blockings to all jobs include those do not require use of shared resources.

Elliott and Anderson [7] presented the k-FMLP by extending the FMLP, which is optimal under suspension-aware analysis. Elliott and Anderson [6] also developed the Optimal k-exclusion Global Locking Protocol (O-KGLP) under global scheduling, which employs a global prioritized queue and k FIFO queues of length $\lceil m/k \rceil$. The O-KGLP is implied to be asymptotically optimal under suspension-oblivious analysis. Ward et al. [8] developed a progress mechanism called Replica-Request Priority Donation (RRPD), and constructed the R^{2}DGLP, an k-exclusion locking protocol previously called I-KGLP [18], based on RRPD and priority inheritance. The R^{2}DGLP can avoid release-blocking, that is only the tasks that engage in resource sharing can be blocked. However, these k-exclusion locking protocols are developed under global scheduling, and the schedulability analysis for these protocols can be quite different from that under partitioned scheduling. We focus our efforts on k-exclusion locking protocols under partitioned scheduling in this paper, and the comparison of our work to other protocols under global scheduling remains a future work.

III. Background and Notations

A set of n sporadic tasks $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ are scheduled on a multiprocessor platform that contains k GPUs and m CPU processors p_1, p_2, p_3, …, p_m (we use "processor" to denote "CPU processor" below for simplicity). A subset of n_G tasks, $\Gamma^G \subseteq \Gamma$, are GPU-using tasks that require use of both CPUs and GPUs during execution, and the rest tasks in Γ are CPU-only tasks.

Task model. A sporadic task recurrently generates jobs, the l^{th} job of τ_i is denoted as $J_{i,l}$, and an arbitrary job of τ_i is also denoted as J_i. Each task τ_i is described as (C_i, G_i, $Trans_i$, T_i), wherein C_i is the worst-case CPU execution time (C-WCET) of τ_i, G_i is the worst-case GPU execution time (G-WCET) of τ_i, $Trans_i$ is the data transmission time that τ_i requires to star its GPU programs on a GPU (such time includes that τ_i requires to create buffers, populate data, send commands to a GPU), and T_i is the minimum separation time (or period) between jobs of τ_i. Note that $Trans_i = G_i = 0$ if τ_i is a CPU-only task. The relative deadline of τ_i is assumed to be equal to T_i for simplicity (implicit deadline). The CPU utilization of task τ_i is given by $u_i = C_i/T_i$. Suppose the GPU works f times faster than a GPU and $Trans_i$ is already a part of C_i, we define the effective utilization of task τ_i as following (the same as that in [5]).

$$u_{eff_i} = \frac{C_i - Trans_i + f \cdot (Trans_i + G_i)}{T_i} \qquad (1)$$

Tasks are assumed to be indexed by decreasing order of priorities (e.g., τ_i has a higher priority than that of τ_j if and only if $i<j$). Each job has the same priority as that assigned to the corresponding task. To avoid irrelevant details, we assume that each task in the system has a unique priority. We assume that a job is ready when it releases (i.e., no release jitter). We denote the release time and the finish time of job $J_{i,v}$ to be $r_{i,v}$ and $f_{i,v}$ respectively. The response time of $J_{i,v}$ is $R_{i,v} = f_{i,v} - r_{i,v}$, as shown in Fig. 1. The Worst Case Response Time (WCRT) of task τ_i is defined to be the

maximum response time of its jobs, and is denoted as WR_i, i.e., $WR_i = \max_{\forall v} R_{i,v}$.

Scheduling. We consider P-FP scheduling in this paper. Tasks are statically assigned to processors, and all jobs of a task are required to execute on the processor where this task is assigned. Tasks that are assigned on the same processor as τ_i are called local tasks of τ_i, tasks allocated on any other processors are called remote tasks of τ_i. We denote the set of local tasks of τ_i as Γ_i. Tasks on individual processors are preemptively scheduled according to the Rate Monotonic (RM) scheduling algorithm [19] which is the optimal task level fixed priority scheduling algorithm in uniprocessor systems.

GPU usage patterns. A GPU-using job, $J_{i,v}$, may issue a request, denoted as GR_i, to a GPU. $J_{i,v}$ suspends if GR_i can not be satisfied immediately. When GR_i is satisfied (i.e., $J_{i,v}$ is allocated a GPU), $J_{i,v}$ creates a buffer, transmits GPU operation data and commands to that GPU through API calls, and commences a GPU program, called a kernel, on that GPU (these operations also called data transmissions for short). Then $J_{i,v}$ suspends to waits for results. Computation results will be written to a buffer by the GPU during kernel execution. Finally, $J_{i,v}$ reads back the results. A request that has not been assigned a GPU is called pending request, otherwise it is called GPU-holding request. The execution phases of job $J_{i,v}$ is depicted in Fig. 1. Note that data transmissions may be time-consuming [5], while read-back operations take less time if memory mapping is enabled.

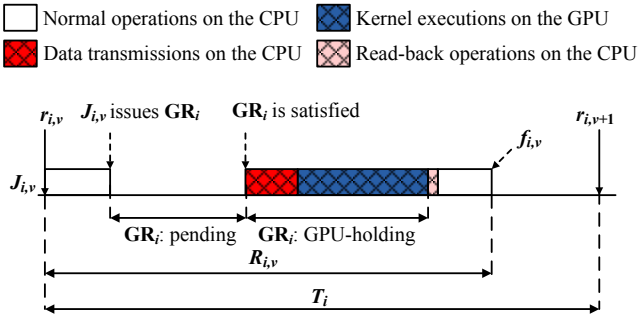

Figure 1. Execution phases of a GPU-using job.

Assumptions. To avoid redundant details, we assume in this paper that

- **A**1 GPU-using jobs do not make nested requests for GPUs;
- **A**2 each GPU-using job acquires at most one of the k GPUs at a time;
- **A**3 Each GPU-using job invokes only one kernel during its execution[1].
- **A**4 All interruption-related delays are ignored.

[1] We make this assumption to prevent wordy description in analysis, though the protocol presented in this paper can also be used when a job invokes more than one kernel. This assumption also coincides with common GPU usage patterns [5, 6].

IV. Locking Protocols for Multi-GPU Sharing

In multiprocessor multi-GPU systems, k-exclusion locking protocols can be used to arbitrate exclusive access to one of the k GPUs. This removes the non-real-time GPU drivers from resource arbitration decisions, and ensures that resource-contention-related delays can be bounded. In this section, we present the structure and the locking rules of the PK-OMLP, and propose a suspension-aware schedulability analysis for the protocol.

A. The PK-OMLP

The PK-OMLP is a two-phase protocol, where jobs compete with their local jobs first, and then the selected jobs contend for GPU allocations. To minimize the wait time of pending requests, GPU-holding jobs should commence operations on the allocated GPUs as soon as possible. Therefore, priority boosting mechanism is used to ensure that jobs can not be preempted during data transmissions. The development procedure we went through is presented thereafter.

Structure. There are two private queues, a priority queue and a FIFO queue, associated with each processor to organize job requests. We denote the queues associated with processor p_x as PPQ_x and PFQ_x respectively. An additional global FIFO queue, denoted as GFQ, of length at most m is used to organize requests issued by jobs assigned to different processors.

Rules. Suppose a GPU-using job J_i that has been assigned to processor p_x issues a GPU request GR_i at time t_0. We present the locking rules governing queuing and locking behaviors as following.

R1 If all GPUs are unavailable at t_0, J_i suspends and GR_i enqueues onto PPQ_x; otherwise, J_i is allocated a GPU.

R2 When GR_i is the head of PPQ_x,

- **R**2.1 GR_i enqueues onto GFQ (and dequeues from PPQ_x) if there is no request issued by local jobs of J_i queuing on GFQ; else
- **R**2.2 If there is a request GR_l issued by a lower priority local job of J_i (i.e., $l>i$, $\tau_l \in \Gamma^G \cap \Gamma_i$) queuing on GFQ, GR_i replaces GR_l; else
- **R**2.3 GR_i has to wait until the ***if*** part of **R**2.1 becomes true.

R3 During the time when GR_i is queuing on GFQ, it can also be replaced by another request GR_h, if which is the head of PPQ_x and with the priority of the associated job higher than that of J_i (i.e., $h<i$, $\tau_l \in \Gamma^G \cap \Gamma_i$).

R4 GR_i enqueues onto PPQ_x (and dequeues from GFQ) if it is replaced by **R**3.

R5 When GR_i is the head of GFQ and there is at least a GPU available, J_i locks a GPU and GR_i enqueues onto PFQ_x (and dequeues from GFQ).

R6 When GR_i is the head of PFQ_x, J_i is scheduled to perform data transmissions with boosted priority.

R7 GR_i dequeues from PFQ_x after data transmissions, and J_i self-suspends.

R8 When the GPU finishes the operation required by J_i (i) it is unlocked; (ii) J_i is signaled and ceases to be suspended; (iii) the head of GFQ, if one exists, is signaled and is granted to lock the newly available GPU.

From **R**6, jobs with GPU-holding requests on each processor are scheduled one after another, and once a job has commenced transmitting data, it can not be preempted by other jobs until it finishes the data transmissions due to priority boosting. From **R**5 and **R**8, at most one job is able to transmit data to get use of a GPU at any time, thus once a job has commenced using a GPU it finishes using that GPU before another job can make use of it. Therefore, pending requests make progress towards acquiring GPUs in PK-OMLP.

It is worth noting that, only data transmissions are treated as critical sections with respect to CPU scheduling. On the other hand, read-back operations do not require mutual exclusion operations thus are considered to be non-critical sections. That is because (i) GPUs can be used by other jobs once it finishes executing a kernel, and (ii) jobs can read back the required GPU execution results until they are scheduled. This observation can result in improved schedulability in that, priority inversion related penalties can be reduced when critical section lengths are limited. In addition, GPU executions are treated to be self-suspensions other than critical sections with respect to CPU scheduling (see **R**7), the corresponding blocking penalties can further be removed from schedulability analysis. We present a suspension-aware analysis below.

B. Schedulability analysis for the PK-OMLP

In this section, we present the schedulability analysis for tasks scheduled under P-FP scheduling with shared resources (i.e., GPUs) arbitrated according to the PK-OMLP. The presented analysis is suspension-aware, and is based on the framework in [11] where tasks with self-suspensions are modeled to have deferred executions. In this paper, the response time of a job can be calculated by summing up (i) the CPU execution time, (ii) the GPU execution time, (iii) the GPU acquisition delay, (iv) the blocking time incurred by data transmissions of lower priority local jobs, and (v) the preemption delays caused by higher priority local jobs. Next, we now bound these terms to derive the upper bounds on WCRT. Since (i) and (ii) can be bounded by C-WCET and G-WCET respectively, and (v) can be bounded by the state-of-the-art iterative techniques in [20], we focus on the upper bounds of (iii) and (iv) herein.

Definition 1 Let $top(v, \mathbf{S})$ be the x tasks with longest data transmissions in the set of **S**, where the data transmission length of task τ_i is given by L_i, and $x=|top(v, \mathbf{S})|=\min(v, |\mathbf{S}|)$.

Lemma 1 *The maximum time a GPU-using job, J_i, can lock a GPU is L_i.*

$$L_i = G_i + Trans_i + \sum_{\tau_j \in top(k, \Gamma_i \cap \Gamma^G / \tau_i)} Trans_j \tag{2}$$

Proof. From **R**5 to **R**8, J_i locks a GPU before data transmission and unlock that GPU when which finishes executing the corresponding kernel. Suppose τ_i is assigned on p_k. From **R**6, all requests that are queued on PFQ_k and ahead of GR_i transmit data (to different GPUs) before J_i does (recall that PFQ_k is a FIFO queue). Since there are at most k+1 requests can queue on PFQ_k, J_i may wait at most $top(x, \Gamma^G \cap \Gamma_i/\tau_i)$ requests to finish the data transmissions before it can reach the head of PFQ_k. Once J_i becomes the head of PFQ_k, it performs data transmissions with boosted priority. Thus it takes $Trans_i$ to complete data transmissions. Finally, it takes G_i for the allocated GPU to perform GPU execution. ■

Definition 2 Let $\Psi(v, \mathbf{A})$ be the x longest GPU-holding requests in the set of requests **A**, where the GPU-holding request length of job J_i is given by L_i, and $x=|\Psi(v, \mathbf{A})|=\min(v, |\mathbf{A}|)$.

Lemma 2 *The time a request, GR_i, spend queuing on GFQ can be bounded by FD_i*

$$FD_i = \begin{cases} 0 & k \geq n_G \\ \dfrac{\sum_{\forall R_j \in \Psi(m+k-1, \Omega(\Gamma^G/\tau_i))} L_j}{k} & k < n_G \end{cases} \tag{3}$$

where $\Omega(\Gamma^G/\tau_i)$ is the set of requests issued by jobs in Γ^G except GR_i.

Proof. Suppose J_i issues GR_i at t_0. If $k \geq n_G$, there are enough GPUs in a system that, each time when a job requires a GPU it can be allocated one. So none GPU-using job needs to wait on GFQ before it is allocated a GPU.

On the other hand, GR_i may queues on GFQ at t_0 if $k < n_G$. Since there are at most m requests on GFQ at any time (due to the constraints of **R**2), there are at most $m-1$ requests on GFQ are in front of GR_i at t_0. In addition, there are at most k GPU-holding requests at t_0. Let these requests except GR_i be denoted by $\mathbf{A_i}$. It can easily be shown that, the cumulative time requirement of the requests in $\mathbf{A_i}$ is at most $\sum_{\forall R_j \in \Psi(m+k-1, \Omega(\Gamma^G/\tau_i))} L_j$.

We assume $FD_i=\Delta t > \frac{\sum_{\forall R_j \in \Psi(m+k-1, \Omega(\Gamma^G/\tau_i))} L_j}{k}$. In that case, all GPUs are locked, while J_i is suspended, for a period of time equal to Δt. That means the cumulative time requirement of

the requests in $\mathbf{A}_i$ is at least $k\times\Delta t > \sum_{\forall R_j \in \Psi(m+k-1, \Omega(\Gamma^G/\tau_i))} L_j$, which contradicts the analysis above.

■

Once a job requires a GPU, it may suspend and queue on the corresponding PPQ and GFQ respectively before it is allocated a GPU. We denote such delay as acquisition delay for that job. Intuitively, acquisition delay of job J_i includes the queuing delays that J_i experience on both PPQ and GFQ.

Lemma 3 *The acquisition delay of a GPU-using job, J_i, can be bounded by AD_i.*

$$AD_i^{n+1} = \sum_{\forall \tau_h \in \Gamma^G \cap \Gamma_i \wedge \tau_h \in h<i} \left\lceil \frac{AD_i^n}{T_h} \right\rceil \cdot \max(FD_h, FD_i) + FD_i \quad (4)$$

where $AD_i^0 = 0$, and $AD_i = AD_i^n$ if $AD_i^{n+1} = AD_i^n$.

Proof. From **R**3, GR_i can be replaced by higher priority local requests when it is queuing on GFQ. Each time when GR_i is replaced by a higher priority local request GR_h (i.e., $h<i$, $\tau_l \in \Gamma^G \cap \Gamma_i$), it has to wait a maximum time of $\max(FD_h, FD_i)$ before it is able to queue on GFQ again. Repeatedly, GR_i may be replaced by all higher priority local requests, and such delay can be bounded by the first item of the right-hand-side of Eq. (4). Additionally, GR_i will experience a maximum of FD_i queuing time on GFQ before it becomes a GPU-holding request.

■

From **R**6, GPU-using jobs can be scheduled with boosted priorities during data transmissions, thus lower priority GPU-using jobs may lead to priority inversions to higher priority jobs. In real-time systems, priority inversions are considered to be blockings [17] and should be counted into schedulability loses.

Definition 3 A job, J_i, is said to be blocked if and only if a lower priority local job of J_i is scheduled while J_i is ready.

Lemma 4 *The blocking time job J_i may experience can be bounded by B_i.*

$$B_i = (n_i^G + 1) \cdot \sum_{\forall \tau_j \in \Gamma_i \cap \Gamma^G \wedge j>i} Trans_j \quad (5)$$

where n_i^G is the number of times that J_i requests GPUs.

Proof. During the time when J_i suspends with GPU-holding request, all lower priority local jobs may execute and issue GPU requests. Right after the time when J_i unlocks a GPU, all these lower priority local jobs may transmit data with boosted priority thus incurring $\sum_{\forall \tau_j \in \Gamma_i \cap \Gamma^G \wedge j>i} Trans_j$ blocking to J_i. In addition, all the lower priority local jobs may execute and issue GPU requests before J_i releases. Therefore, J_i can be blocked for at most $n_i^G + 1$ times.

■

Theorem 1 *The WCRT of task τ_i is bounded as follows*

$$WR_i^{n+1} = C_i + G_i + AD_i + B_i + \sum_{\forall \tau_h \in \Gamma_i \wedge \tau_h \in h<i} \left\lceil \frac{WR_i^n + AD_h + G_h}{T_h} \right\rceil \cdot C_h \quad (6)$$

where $WR_i^0 = 0$, and $WR_i = WR_i^n$ if $WR_i^{n+1} = WR_i^n$.

Proof. The proof follows directly from the above lemmas and the WCRT analysis in [11]

■

τ_i is considered to be schedulable if $WR_i^{n+1} = WR_i^n < T_i$. It is easy to prove that $WR_i^{n+1} \geq WR_i^n$, so that the iteration can also be halted early if $WR_i^{n+1} > T_i$. In that case τ_i is considered to be un-schedulable.

V. Schedulability Experiments

In this section, we conduct schedulability experiments to compare the performance of the CK-OMLP and the PK-OMLP. Task sets with varying characteristics are generated randomly, and are tested for schedulability on an eight CPU system according to Eq. (6). The Worst-Fit-Decreasing (WFD) algorithm is used for task assignment[2], and the priorities of tasks are determined in accordance with the RM algorithm [19].

A. Experimental setup

We use the ratio of schedulable task sets as a metric in our experiments. The task set characteristics are varied by the following factors.

(i) The ratio of G-WCET to C-WCET, which is denoted by α (i.e., $\forall i$, $\alpha = G_i/C_i$);
(ii) The ratio of the length of the C-to-G communications to G-WCET, which is denoted by β (i.e., $\forall i$, $\beta = Trans_i/G_i$);
(iii) The percentage of GPU-using tasks, which is denoted by γ (i.e., $\gamma = |\Gamma^G|/|\Gamma|$);
(iv) The number of GPUs in the system, $k \in [1, 16]$;
(v) The speed up factor for GPUs, $f = 8, 16$.

We set $\alpha \in [0.2, 0.4]$, $\gamma \in [0.1, 0.4]$ for light GPU-usage patterns, while set $\alpha \in [0.5, 0.7]$, $\gamma \in [0.5, 0.9]$ for heavy GPU-usage patterns. A permutation of these five parameters constitutes an experimental scenario. We generate task sets as follows.

Let the effective system utilization be $U_{eff} = \sum_{\forall \tau_i \in \Gamma} u_{eff_i}$. We choose an effective system utilization cap, denoted as UC, in each scenario. From Eq. (2) and the above parameters, we have $UC = \sum_{\forall \tau_i \in \Gamma} u_{eff_i} = \sum_{\forall \tau_i \in \Gamma - \Gamma^G} u_i +$

[2] In order to achieve load balancing, we assign GPU-using tasks first and then CPU-only tasks.

$\sum_{\forall \tau_i \in \Gamma^G} \frac{C_i - \alpha\beta C_i + \alpha f C_i}{T_i}$. Let $\varepsilon = 1 - \alpha\beta + \alpha f$ (which is known for a specific scenario), we have $UC = \sum_{\forall \tau_i \in \Gamma - \Gamma^G} u_i + \varepsilon \sum_{\forall \tau_i \in \Gamma^G} u_i$. Then, we generate task utilizations from [0.02, 0.2] with uniform distribution. So that $u_{aver}=(0.02+0.2)/2=0.101$, and $UC = (|\Gamma| - |\Gamma^G|)u_{aver} + \varepsilon|\Gamma^G|u_{aver}$. Since $\gamma = |\Gamma^G|/|\Gamma|$, we have $|\Gamma^G| = \frac{UC \cdot \gamma}{u_{aver}(\gamma\varepsilon - \gamma + 1)}$. Next, we generate $|\Gamma^G|$ GPU-using tasks and $|\Gamma| - |\Gamma^G|$ CPU-only tasks respectively with uniform distribution. Task periods are generated randomly from [10ms, 100ms]. The C-WCET of each task τ_i is determined by the selected period and utilization, $C_i = T_i \times u_i$.

The generated tasks are added to a task set. Let the cumulative effective system utilization of the task set be denoted by U_c. Once U_c exceeds the selected cap UC, the last generated task is discarded. Then, if $UC - U_c \in [0.02, 0.2]$, we generate a CPU-only task with utilization equal to $U_c - UC$ and add it to the task set; otherwise, we generate a GPU-using task with effective utilization exactly equal to $U_c - UC$ and add it to the task set. A total of 20,000 task sets are generated and tested for each UC.

B. Experimental results

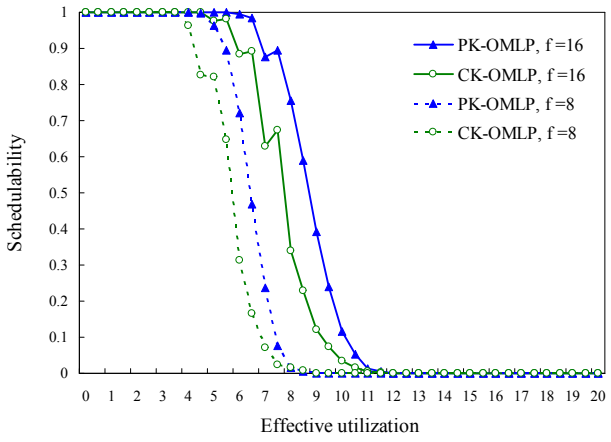

Figure 2. Schedulability in light GPU-usage patterns. α=0.3, β=0.2, γ=0.25, m=8, k=4.

Fig. 2, 3 depict the performances of each protocol under light and heavy GPU-usage patterns respectively. The influence on schedulability caused by α, β, and γ are studied through single factor analysis and are demonstrated in Fig. 4, 5, 6 respectively. Fig. 7, 8 show the performance gains of each protocol with the growing number of GPUs. The presented results show observable trends that the PK-OMLP outperforms the CK-OMLP in most scenarios.

The performances of both protocols, especially the CK-OMLP, fluctuate with increasing system effective utilization, as shown in Fig. 2, 3, 5. The most likely reason is that the task assignment changes with the growing number of tasks (recall that we use WFD for task assignment), and the acquisition delays that some particular tasks may experience change significantly when task assignment changes at some points.

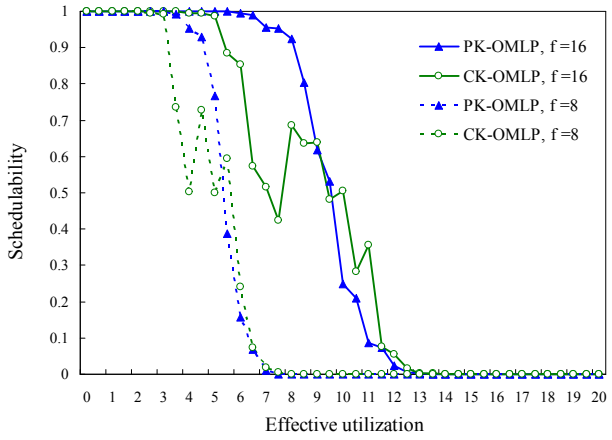

Figure 3. Schedulability in heavy GPU-usage patterns. α=0.6, β=0.2, γ=0.7, m=8, k=4.

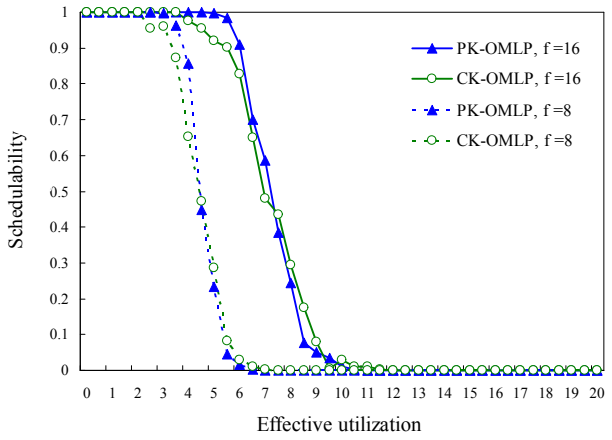

Figure 4. Schedulability with larger value for γ. α=0.3, β=0.2, γ=0.7, m=8, k=4.

Fig. 2 shows that the PK-OMLP strictly dominates the CK-OMLP in light GPU-usage patterns. However, the CK-OMLP performs better than the PK-OMLP in case $U_{eff} > 5$ when f =8 and $U_{eff} > 9$ when f =16 in Fig. 3. Similar trend can also be observed in Fig. 4 where the CK-OMLP performs slightly better than the PK-OMLP when $U_{eff} > 4$ for f =8 and $U_{eff} > 7$ for f =16. Once a job acquires a resource, it is scheduled immediately under the CK-OMLP, while it may be delayed for a bounded amount of time under the PK-OMLP (as discussed in lemma 1 and formulated by the last item in the right-hand-side of Eq. (2)). So that schedulability under the PK-OMLP may perform worse than the CK-OMLP, especially in heavy GPU-usage patterns where such delays can be long.

In the single factor analysis, schedulability under the PK-OMLP matches, even performs worse than the CK-OMLP when the percentage of GPU-using tasks is high, as shown in Fig. 4. The PK-OMLP strictly dominates the CK-OMLP when the ratio of the length of the C-to-G communications

to G-WCET, and especially when the ratio of G-WCET to C-WCET is high, as shown in Fig. 5 and Fig. 6 respectively.

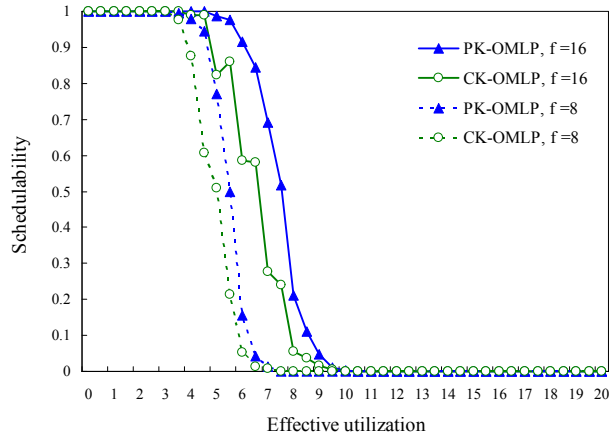

Figure 5. Schedulability with larger value for β. α=0.3, β=0.75, γ=0.25, m=8, k=4.

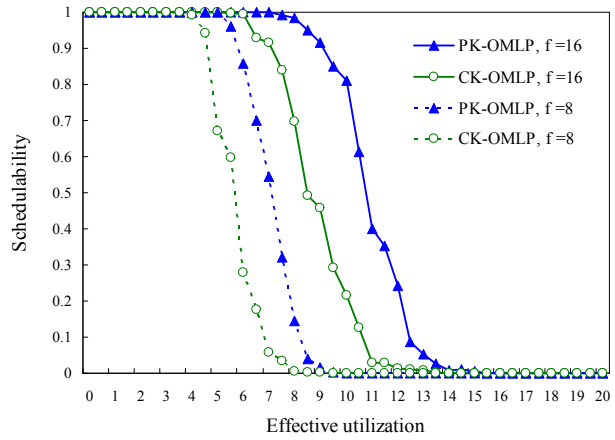

Figure 6. Schedulability with larger value for α. α=0.6, β=0.2, γ=0.25, m=8, k=4.

Fig. 7 and Fig. 8 indicate that, the schedulability of the CK-OMLP remain unchanged while the schedulability of the PK-OMLP may still increase when k>8. Due to the use of the priority donation mechanism, no more than m jobs can get access to GPUs at any time under the CK-OMLP. Therefore is there is little benefit under the CK-OMLP to adopting more than m GPUs in a system. On the other hand, the PK-OMLP supports k>m, and is able to gain schedulability improvement with the growing number of GPUs. Fig. 8 shows that all task sets are schedulable under the PK-OMLP when there are more than 9 GPUs with speed up of 16×.

It can also be observed that the performances of the PK-OMLP will not always increase with increasing number of GPUs. As shown in Fig. 7, the schedulable ratios of the PK-OMLP become unchanged when $k \geq |\Gamma^G|$. Further, jobs that have acquired locks for GPUs may be delayed under the PK-OMLP, and such delays can be long in heavy GPU-usage patterns. The performance of the PK-OMLP may be adversely impacted when there are a large number of GPU-using tasks on each processor and the data transmissions are relatively long, as shown in Fig. 8 when the speed up is 8×.

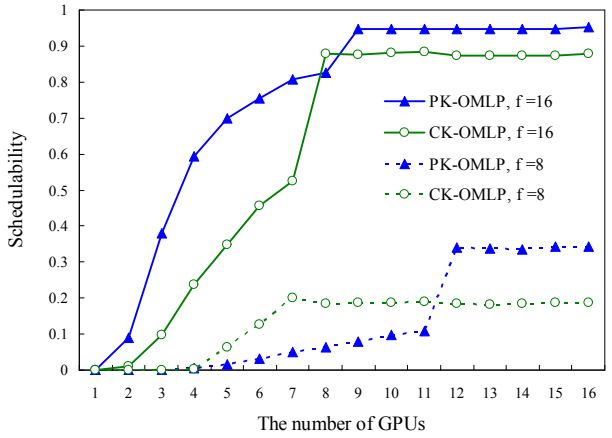

Figure 7. Schedulability in light GPU-usage patterns with increasing number of GPUs. α=0.3, β=0.2, γ=0.25, m=8, U_{eff}=8.5 ($|\Gamma^G|$=9 when f=16, and $|\Gamma^G|$ =12 when f=8).

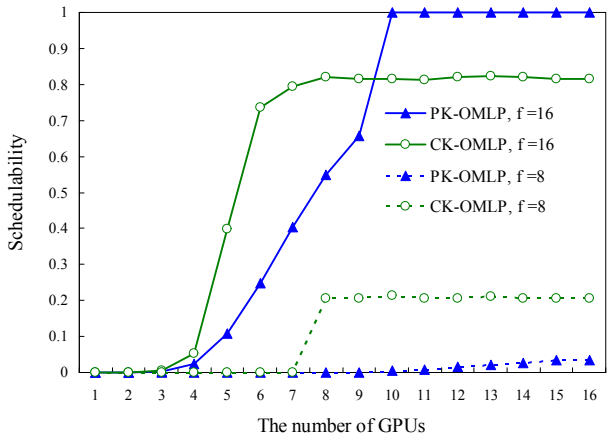

Figure 8. Schedulability in heavy GPU-usage patterns with increasing number of GPUs. α=0.6, β=0.2, γ=0.7, m=8, U_{eff}=12 ($|\Gamma^G|$=10 when f=16, and $|\Gamma^G|$ =20 when f=8).

VI. Conclusions and Future Work

In this paper, we address the problem of k-exclusion locking protocols for multi-GPU applications in partitioned scheduled multiprocessor real-time systems. A novel k-exclusion real-time locking protocol, the PK-OMLP, is presented to arbitrate exclusive access to one of the k GPUs. The PK-OMLP is capable of making full use of k, whether k is more than m or not, GPUs in a system containing m processors, and improves the CK-OMLP from both flexibility and GPU utilizations in hard real-time systems. An associated suspension-aware scheduleability analysis is also presented, in which GPU acquisition delays and GPU executions are treated as self-suspensions and are incorporated in to WCRT analysis. Experimental evaluations indicate that the PK-OMLP achieves better schedulability than the CK-OMLP in most considered scenarios. In future

work, we will address the implementation details of the PK-OMLP in an operating system environment and evaluate the runtime performance. We will also exploit global priority queuing mechanism to construct real-time k-exclusion locks so as to further decrease acquisition delays and priority inversions.

ACKNOWLEDGMENT

This work has been supported in part by the National Natural Science Foundation of China (Grant No. 61103041), the Fundamental Research Funds for the Central Universities of China (Grant No. ZYGX2012J070), the Excellent PhD student Academic Support Program of the UESTC (Grant No. YBXSZC20131028), the Huawei Technology Foundation (Grant No. ZYGX2012J070), and the National High-tech R&D Program of China (Grant No. SQ2011GX02D03708).

REFERENCES

[1] A. Acosta, V. Blanco, F. Almeida, "Towards the dynamic load balancing on heterogeneous multi-GPU systems," in *Proc. 10th IEEE Int. Symp. Parallel and Distributed Processing with Applications (ISPA'12)*, Jul., pp. 646-653, 2012.

[2] Z. Ziming, V. Rychkov, A. Lastovetsky, "Data partitioning on heterogeneous multicore and multi-GPU systems using functional performance models of data-parallel applications," in *Proc. IEEE Int. Conf. Cluster Computing (Cluster'12)*, Sep., pp. 191-199, 2012.

[3] K. Matsumoto, N. Nakasato, S. Sedukhin, "Blocked united algorithm for the all-pairs shortest problem on hybrid CPU-GPU systems," *IEICE Transactions on Information and Systems*, vol. E95D, no. 12, pp. 2795-2768, 2012.

[4] J. Lobeiras, M. Amor, R. Doallo, "Influence of memory access patterns to small-scale FFT performance," *Journal of Supercomputing*, vol. 64, no. 1, pp. 120-131, 2012.

[5] G. Elliott, J. Anderson, "Globally scheduled real-time multiprocessor systems with GPUs," *Journal of Real-Time Systems*, vol. 48, no. 1, pp. 34-74, 2012.

[6] G. Elliott, J. Anderson, "An optimal *k*-exclusion real-time locking protocol motivated by multi-GPU systems," *Journal of Real-Time Systems*, vol. 49, no. 2, pp. 140-170, 2013.

[7] G. Elliott, J. Anderson, "Robust real-time multiprocessor interrupt handling motivated by GPUs," in *Proc. 24th IEEE Euromicro Conference on Real-Time Systems (ECRTS' 12)*, Jul., pp. 267-276, 2012.

[8] B. Ward, G. Elliott, J. Anderson, "Replica-request priority donation: a real-time progress mechanism for global locking protocols," in *Proc. 18th IEEE Int. Conf. Embedded and Real-Time Computing Systems and Applications (RTCSA' 12)*, Aug., pp. 280-289, 2012.

[9] B. Brandenburg. "Scheduling and locking in multiprocessor real-time operating systems," PhD thesis, University of North Carolina at Chapel Hill, 2011.

[10] A. Bastoni, B. Brandenburg, J. Anderson, "An empirical comparison of global, partitioned, and clustered multiprocessor real-time schedulers," in *Proc. 31th IEEE Int. Conf. Real-Time Systems Symposium* (*RTSS' 10*), Nov., pp. 14-24, 2010.

[11] K. Lakshmanan, D. Niz, R. Rajkumar, "Coordinated task scheduling, allocation and synchronization on multiprocessors," in *Proc. 30th IEEE Int. Conf. Real-Time Systems Symposium (RTSS' 09)*, Dec., pp. 469-478, 2009.

[12] B. Brandenburg, J. Calandrino, A. Block, et al., "Real-time synchronization on multiprocessor: to block or not to block, to suspend or spin," in *Proc. 14th IEEE Int. Conf. Real-Time and Embedded Technology and Application Symposium (RTAS'08)*, Apr., pp.342-353, 2008.

[13] J. Ras, A, Cheng, "An evaluation of the dynamic and static multiprocessor priority ceiling protocol and the multiprocessor stack resource policy in an SMP system," in *Proc. 15th IEEE Int. Conf. Real-Time and Embedded Technology and Application Symposium (RTAS'09),* Apr. pp. 13-22, 2009.

[14] B. Brandenburg, J. Anderson, "Real-time resource-sharing under clustered scheduling:mutex, realer-writer, and *k*-exclusion locks," in *Proc. IEEE/ACM Int. Conf. Embedded Software (EMSOFT' 11)*, Oct., pp. 69-78, 2011.

[15] R. Rajkumar, "Real-time synchronization protocols for shared memory multiprocessor," in *Proc. IEEE Int. Conf. Real-Time Systems Symposium (RTSS' 90)*, May, pp. 116-123, 1990.

[16] A. Block, H. Leontyev, B. Brandenburg, "A Flexible real-time locking protocol for multiprocessors," in *Proc. 13th IEEE Int. Conf. Embedded and Real-Time Computing Systems and Applications (RTCSA' 07)*, Aug., pp. 47-56, 2007.

[17] B. Brandenburg, J. Anderson, "Optimality results for multiprocessor real-time locking," in *Proc. 31st IEEE Int. Conf. Real-Time Systems Symposium (RTSS' 10)*, Dec., pp. 49-60, 2010.

[18] B. Ward, J. Anderson, "Supporting nested locking in multiprocessor real-time systems", in *Proc. 24th IEEE Euromicro Conference on Real-Time Systems (ECRTS' 12)*, Jul., pp. 223-232, 2012.

[19] C. Liu, J. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 40-61, 1973.

[20] N. Audsley, A. Burns, M. Richardson, K. Tindell, A. Willings, "Applying new scheduling theory to static priority preemptive scheduling" *Journal of Software Engineering*, vol. 8, no. 5, pp. 284-296, 1993.