

# GPU-Based Parallel EDF-Schedulability Analysis of Multi-Modal Real-Time Systems

Masud Ahmed   Safraz Rampersaud   Nathan Fisher   Daniel Grosu   Loren Schwiebert

Department of Computer Science

Wayne State University, USA.

Email: {masud, safraz, fishern, dgrosu, loren}@wayne.edu

**Abstract**—Real-time multi-modal systems are useful in modeling embedded systems that dynamically change computational requirements over time (e.g., adaptive cruise control systems). For meeting timing constraints of such multi-modal systems, Earliest-Deadline-First (EDF) is an attractive real-time scheduling algorithm due to its optimality on uniprocessor platforms. However, checking EDF-schedulability of a real-time multi-modal system is a difficult problem that requires substantial computational effort. Today’s cost efficient and massively parallel GPU platforms can be effectively leveraged to solve this difficult problem. Existing algorithms for EDF-schedulability of real-time multi-modal systems cannot exploit the entire computational power of a GPU; therefore, in this research, we develop a parallel algorithm leveraging the advantages of a GPU device. Experimental results establish the superior performance of our proposed algorithm upon a low end GPU over the implementation of existing algorithms on a cluster of computers using either MPI or OpenMP. In addition to performance, our proposed algorithm is a cost effective and power efficient alternative against comparable algorithms for multi-core and parallel computing platforms.

## I. INTRODUCTION

Many embedded systems (e.g., smartphones, vehicles, and GPS receivers) exhibit multi-modal operation (e.g., power saving and operational modes), which encourages system designers to consider real-time multi-modal systems for efficiently managing resources and tasks deadlines. These devices utilize recurring tasks that may be scheduled by priority-based algorithms (fixed or dynamic priority). For any of these scheduling algorithms, schedulability analysis may be used to obtain guarantees that the timing constraints of all tasks of a safety-critical (hard) real-time system are *always* met. Several protocols for switching between different software/hardware modes of a multi-modal system along with appropriate schedulability analysis have been proposed [12], [20]. Although Earliest-Deadline-First (EDF), a dynamic priority scheduling algorithm that assigns the highest priority to tasks with the earliest deadline, is an optimal scheduling algorithm for a uniprocessor platform, the existing EDF-schedulability analysis of a multi-modal system is computationally intractable with respect to execution time. A time-efficient schedulability analysis is desirable in a design-space exploration technique that determines the “best” parameters of a multi-modal real time system by repeated application of the schedulability analysis.

Real-time systems’ researchers establish EDF-schedulability using demand-based analysis, which ensures the processor execution supply for a given interval, is always higher than the execution demand for the corresponding interval generated by all tasks of the system. Due to mode changes, EDF-schedulability for multi-modal systems is inherently difficult as both task sets that generate the workload and processor’s modes that execute the workload are not fixed. Recently, Fisher et al. [12] addressed the EDF schedulability for multi-modal systems (considering both hardware and software mode changes) and developed invariants that all schedulable multi-modal systems must satisfy. Ahmed et al. [3] later exploited these five conditions to develop a parallel message-passing algorithm having similar pseudo-polynomial time complexity (polynomially bounded on the value of the system attributes) as the serial algorithm [12]. Although this algorithm utilizes parallel platforms, it may have long execution times even for moderately sized problems with respect to the number of modes. There are systems for which a higher number of modes are typically desirable (e.g., real-time processor control systems [13] where a higher number of modes increases the freedom of choices in unfavorable scenarios) to improve stability. The execution time of the parallel message-passing algorithm is affected by high communication and synchronization overhead. As a result, real-time system designers may achieve a better performance by using cost efficient, scalable, and massively-parallel general-purpose Graphical Processing Unit (GPU) platforms to solve the problem of EDF-schedulability analysis of multi-modal real-time systems. As GPUs are becoming increasingly common for handheld devices, a fast parallel schedulability analysis could be effectively utilized as an online admission controller for these devices if the response time is small for moderately sized problems (e.g., number of modes are smaller than 5).

GPU architectures are being increasingly used as parallel processing platforms for solving large scale problems suitable for the SIMD (Single Instruction Multiple Data) execution model. The scheduler of the GPU uses many threads, organized into warps, that execute the same set of instructions with different data. As a result, branching in the currently executing instruction set of a warp and synchronization among different warps may affect the projected performance of the program. These patterns of execution distinguish the GPU

from many other parallel processing platforms. Therefore, the message passing algorithm of Ahmed et al. [3], which is designed to execute upon a computer cluster, may not perform well on GPU-based platforms. In addition, synchronization requirements among GPU blocks may create possible deadlock scenarios [18].

**Our Contribution.** We utilize the schedulability conditions proposed by Fisher et al. [12] to develop a GPU-based algorithm for checking EDF-schedulability of a multi-modal real-time system. A review of these conditions is included in Section III-C. We exploit the favorable characteristics of a GPU platform to assign workload to GPU blocks and threads. We emphasize the efficient utilization of GPU resources, avoid inter-block synchronization requirements, and ensure a balanced workload distribution among blocks and threads to obtain a better speedup. Using experiments, we investigate the performance of the proposed GPU-based parallel algorithm for EDF-schedulability analysis of multi-modal systems. Our results show that our algorithm achieves significant speedup over existing serial and parallel EDF-schedulability analysis algorithms [3], [12], [20].

**Organization.** The rest of the paper is organized as follows. In Section II, we discuss the relevant work on schedulability analysis, GPU platforms and real-time multi-modal systems. In Section IV-A, we provide basic concepts concerning the GPU architecture. In Section III, we present the models and definitions used in this paper. In Section IV, we present the workload distribution strategies used in the design of the proposed parallel algorithm. In Section V, we present our parallel algorithm for EDF-schedulability analysis. In Section VI, we present and discuss the experimental results. In Section VII, we conclude the paper and discuss future research directions.

## II. RELATED WORK

Several research results on multi-modal real-time systems considered both fixed-priority scheduling and dynamic-priority based (e.g., EDF) scheduling [19], [20], [22], [25], [26]. In this paper, we restrict our attention to EDF-based multi-modal real-time systems that allow the system to change both hardware and software modes [12], [20]. There are server-based multi-mode systems [1], [6], [7], [14], [23], [24] that allow changes of hardware modes (e.g., changing processor frequency), but cannot ensure timing constraints during the transition period between modes. EDF-schedulability analysis that support both hardware and software modes is computationally intractable [20]. Phan et al. [19], [20] performed comprehensive studies on multi-modal systems using a compositional real-time framework [8]. However, their method requires exponential time due to the exploration of a reachability graph. Parallelism is difficult to achieve for their algorithm as the computation at each step depends on results of all previous steps.

Unlike all previous research on multi-modal systems scheduled by EDF, Fisher and Ahmed [12] separated schedulability analysis from determining the minimum resource parameters

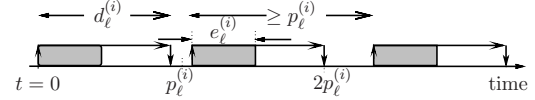


Fig. 1. Job generation pattern of a sporadic task. An upward arrow denotes a job arrival whereas a downward arrow marks the deadline for the job. A shaded region represents the execution requirements for the job.

of a multi-modal system and achieved pseudo-polynomial runtime complexity for checking EDF-schedulability. Later, the authors [2] established that the minimum resource parameters can be determined through a polynomial number of invocations of this schedulability analysis, which results in the pseudo-polynomial complexity of both tasks (checking schedulability and determining minimum resources) combined. To further reduce the execution time, Ahmed et al. [3] developed a parallel algorithm for the schedulability conditions that achieved a significant speedup over the serial algorithm upon a cluster of computers. However, this parallel algorithm is not suitable for today's commercially available GPU platforms which can support approximately 65-million threads [10], [15]. The ease of GPU programming using the CUDA API [11], [17], [18] makes this parallel platform more accessible to system designers.

## III. MODELS & DEFINITIONS

We consider the model (described by Fisher and Ahmed [12]) for real-time multi-modal systems that allows both the real-time application workload (software) and the processing resource (hardware) to have multiple modes. In order to ensure temporal isolation and also hard deadlines, we explicitly couple each software mode to a hardware mode. These couples are used to describe which potential software and hardware modes may be executed together. A single software mode could execute under (and be coupled with) multiple hardware modes and vice versa. We consider a multi-modal system  $\mathcal{M}$  with  $q$  modes. Each mode  $M^{(i)}$ , where  $i \in \{1, \dots, q\}$  is the mode index, is associated with the real-time workload and the minimum processor execution guaranteed by the processing resource. The real-time workload  $\tau^{(i)}$  of mode  $M^{(i)}$  is modeled by the *sporadic tasks model* [16], and the processing resource that guarantees the real-time execution of  $\tau^{(i)}$  is modeled by the *explicit-deadline periodic (EDP) resource model* [9]. Before describing the mode change model of a real-time application, we first describe the sporadic tasks and the EDP resource models. In this paper, we assume that system parameters (i.e., sporadic task and EDP parameters) are integers. This assumption is not restrictive as all timing parameters may be expressed in terms of the number of CPU clock ticks.

**Sporadic Task Model.** A *sporadic task system*  $\tau^{(i)} \stackrel{\text{def}}{=} \{\tau_1^{(i)}, \dots, \tau_{n_i}^{(i)}\}$  is a collection of  $n_i$  sporadic tasks where each task  $\tau_\ell^{(i)} \in \tau^{(i)}$  is characterized by three parameters: *worst-case execution requirement*  $e_\ell^{(i)}$ , *(relative) deadline*  $d_\ell^{(i)}$ , and *minimum inter-arrival separation*  $p_\ell^{(i)}$  (also called the *period*). A sporadic task  $\tau_\ell^{(i)}$  may produce a sequence of

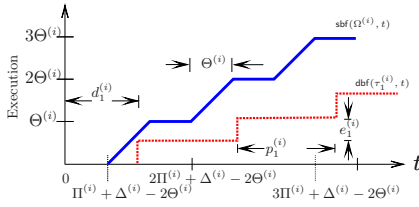


Fig. 2. Demand-bound and supply-bound functions.

jobs, separated by at least  $p_\ell^{(i)}$  time units, where each job has the maximum execution time  $e_\ell^{(i)}$  units, and must complete within  $d_\ell^{(i)}$  time units after its arrival (see Figure 1). We consider *constrained deadline* tasks; that is,  $d_\ell^{(i)} \leq p_\ell^{(i)}$ . The utilization of a task  $\tau_\ell^{(i)}$  is defined as  $u_\ell^{(i)} \stackrel{\text{def}}{=} e_\ell^{(i)} / p_\ell^{(i)}$  and the task system utilization  $u^{(i)}$  equals  $\sum_\ell u_\ell^{(i)}$ . To quantify the maximum workload over an interval of length  $t > 0$ , we consider the *demand-bound function* (dbf) which quantifies the maximum cumulative execution requirements of all jobs of  $\tau_\ell^{(i)}$  that could have both the arrival time and the deadline in any interval of length  $t$ . Baruah et al. [4] have shown that, for a sporadic task  $\tau_\ell^{(i)}$ , dbf can be calculated as follows:

$$\text{dbf}(\tau_\ell^{(i)}, t) = \max \left( 0, \left\lfloor \frac{t - d_\ell^{(i)}}{p_\ell^{(i)}} \right\rfloor + 1 \right) \cdot e_\ell^{(i)}. \quad (1)$$

We denote by  $\text{dbf}(\tau^{(i)}, t) \stackrel{\text{def}}{=} \sum_{\tau_\ell^{(i)} \in \tau^{(i)}} \text{dbf}(\tau_\ell^{(i)}, t)$  the demand-bound function of a sporadic task system  $\tau^{(i)}$ .

**Explicit-Deadline Periodic (EDP) Resource Model.** The *EDP resource model* [9], [23] is a general resource model for characterizing the execution of a system upon a periodically-available, non-continuously-executing resource. The hardware processing resource available to each mode  $M^{(i)}$  is represented by an EDP resource  $\Omega^{(i)} \equiv (\Pi^{(i)}, \Theta^{(i)}, \Delta^{(i)})$  where  $\Pi^{(i)}$  is the *resource period*,  $\Theta^{(i)}$  is the *resource capacity*, and  $\Delta^{(i)}$  is the *resource deadline*. The interpretation of these parameters is that the EDP resource  $\Omega^{(i)}$  guarantees mode  $M^{(i)}$  a total execution of at least  $\Theta^{(i)}$  units over successive  $\Pi^{(i)}$ -length intervals within  $\Delta^{(i)} (\leq \Pi^{(i)})$  units of time (depicted in Figure 3). Furthermore, we assume that EDF is used to schedule the workload at any point when the resource is providing execution. We consider the *supply-bound function*  $\text{sbf}(\Omega^{(i)}, t)$  to quantify the minimum execution supply that a mode  $M^{(i)}$  is guaranteed to receive from  $\Omega^{(i)}$  over any interval of length  $t \geq 0$ . Easwaran et al. [9] have quantified  $\text{sbf}(\Omega^{(i)}, t)$  as follows:

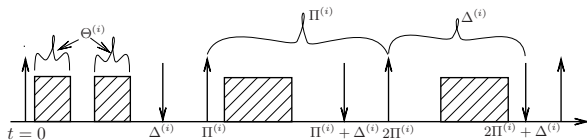


Fig. 3. Execution pattern in EDP resources.

$$\begin{aligned} \text{sbf}(\Omega^{(i)}, t) &= \begin{cases} y\Theta^{(i)} + \max(0, t - x - y\Pi^{(i)}), & \text{if } t \geq \Delta^{(i)} - \Theta^{(i)} \\ 0, & \text{otherwise.} \end{cases} \end{aligned} \quad (2)$$

where  $y = \left\lfloor \frac{t - (\Delta^{(i)} - \Theta^{(i)})}{\Pi^{(i)}} \right\rfloor$  and  $x = (\Pi^{(i)} + \Delta^{(i)} - 2\Theta^{(i)})$ . For schedulability, dbf needs to be smaller than sbf for all  $t$  (See Figure 2).

**Real-Time Mode Change Model.** We now describe the *discrete hardware/software real-time multi-modes* model [12]. Each mode is specified by a three-tuple  $(\tau^{(i)}, \Omega^{(i)}, N^{(i)})$  that respectively characterizes the real-time workload generated by a sporadic task system, the minimum processor execution guaranteed by an EDP resource, and the minimum mode duration in terms of “number of resource periods”  $N^{(i)}$ . The interpretation of  $N^{(i)}$  is that the system remains in mode  $M^{(i)}$  for at least  $N^{(i)} \cdot \Pi^{(i)}$  time units. A *mode-change request*  $\text{mcr}_k \equiv (M^{(i)}, M^{(j)}, t_k)$  consists of *transition time* ( $t_k$ ), the *old mode*  $M^{(i)}$  executing prior to  $t_k$ , and the *new mode*  $M^{(j)}$  executing after  $t_k$  (where  $i, j \in \{1, \dots, q\}$ ). We assume that if  $i < j$  then  $\text{mcr}_i$  occurs prior to  $\text{mcr}_j$ . Mode-change request  $\text{mcr}_0 \stackrel{\text{def}}{=} (M^{(0)}, \cdot, 0)$  represents the transition from the null-mode  $M^{(0)}$  to any mode in  $\{M^{(1)}, \dots, M^{(q)}\}$  at system start time.

Tasks may be divided into groups at the time of mode change request  $\text{mcr}_k \equiv (M^{(i)}, M^{(j)}, t_k)$ . Some tasks can continue to execute without being affected by a mode change request. We call these tasks *unchanged tasks* denoted by  $\tau^{(ij)}$ . Some tasks must be removed from the system immediately at the time of mode change request. We call these tasks *aborted tasks* and denote them by  $\alpha^{(ij)}$ . Finally, for some tasks, immediate termination may leave the system in an inconsistent state, and these tasks must be allowed to complete; we call such tasks *finished tasks* and characterize them as being members of the set  $\tau^{(i)} \setminus (\alpha^{(ij)} \cup \tau^{(ij)})$ . A job from a finished task at the time of mode change request may complete its remaining execution after mode changes. All jobs are scheduled by EDF.

In order to facilitate quick changes of modes, the system designer may allow a transition period, called the *offset*, which we denote by  $\delta_{ij}$  (see Figure 4). During the transition period after  $\text{mcr}_k$ , only the jobs from unchanged tasks and the last generated job from the finished tasks are permitted to execute. At  $t_k + \delta_{ij}$  and after, task system  $\tau^{(j)}$  may generate and execute jobs along with any remaining execution of jobs from  $\tau^{(i)}$ . Finally, there may be some tasks that are common to both modes, but have some properties changed; we treat these tasks as finished tasks in the old mode. During the transition period after  $\text{mcr}_k$ , the system designer may provision a different resource  $\Omega^{(ij)} \stackrel{\text{def}}{=} (\Pi^{(ij)}, \Theta^{(ij)}, \Delta^{(ij)})$  to achieve a quick mode change response. We assume that the offset  $\delta_{ij}$  is some multiple of  $\Pi^{(ij)}$ . Given the above definitions, we may distinguish three phases (see Figure 4) with respect to a mode-change request  $\text{mcr}_k = (M^{(i)}, M^{(j)}, t_k)$  (from the previous request  $\text{mcr}_{k-1} = (M^{(h)}, M^{(i)}, t_{k-1})$ ):

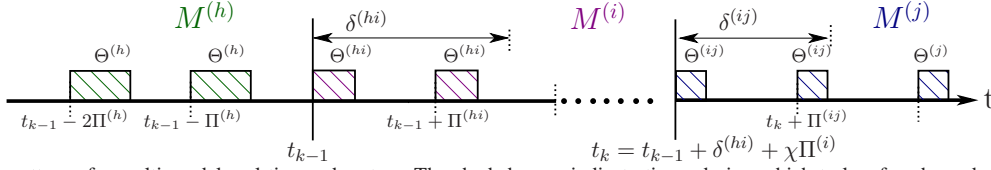


Fig. 4. Execution pattern of a multi-modal real-time subsystem. The shaded areas indicate times during which tasks of each mode execute on the processor.

- 1)  $[t_{k-1} + \delta_{hi}, t_k]$ : jobs of  $\tau^{(i)}$  are executed upon  $\Omega^{(i)}$ ;
- 2)  $[t_k, t_k + \delta_{ij}]$ : non-aborted jobs ( $\tau^{(i)} \setminus \alpha^{(ij)}$ ) with remaining execution at  $t_k$  may execute upon  $\Omega^{(ij)}$ ;
- 3)  $[t_k + \delta_{ij}, t_{k+1}]$ : incomplete jobs of ( $\tau^{(i)} \setminus \alpha^{(ij)}$ ) at  $t_k + \delta_{ij}$  and jobs of  $\tau^{(j)}$  execute upon  $\Omega^{(j)}$ ;

In addition, unchanged tasks ( $\tau^{(ij)}$ ) act independent of mode change request during  $[t_{k-1} + \delta_{hi}, t_{k+1}]$ . The above task classifications follow the taxonomy by Real and Crespo [21]. In this paper, we restrict the mode-change requests and transition intervals to occur only at period boundaries of the EDP model. This assumption is natural for control systems where modes change only at sampling periods. That is, for any mode-change request  $\text{mcr}_k = (M^{(i)}, M^{(j)}, t_k)$ ,  $\delta_{ij}$  must be a multiple of  $\Pi^{(ij)}$ , and for any two successive mode-change requests  $\text{mcr}_{k-1} = (M^{(h)}, M^{(i)}, t_{k-1})$  and  $\text{mcr}_k = (M^{(i)}, M^{(j)}, t_k)$ , we require that

$$t_k = t_{k-1} + \delta_{hi} + \chi\Pi^{(i)} \quad (3)$$

for some  $\chi \in \mathbb{N}^+$  where  $a \geq N^{(i)}$ . We also assume that a non-aborted job may span no more than one mode-change request. In this paper, we consider the following problem.

**EDF-Multi-Mode-Sched Problem:** Given modes  $M^{(1)}, \dots, M^{(q)}$ , resources  $\Omega^{(ij)}$ , offsets  $\delta_{ij}$ , unchanged tasks  $\tau^{(ij)}$ , and aborted tasks  $\alpha^{(ij)}$  for all  $i, j \in \{1, \dots, q\}$  ( $i \neq j$ ), determine whether all jobs, under all legal job arrivals and all possible legal mode-change requests according to Equation 3 are EDF-schedulable.

To solve this problem, Fisher and Ahmed [12] developed schedulability conditions that guarantee that the maximum demand over any interval is always less than the minimum supply received for that interval. For taking mode changes into account, Fisher and Ahmed [12] defined mode-change DBF and SBF functions with respect to any  $\text{mcr}_k = (M^{(i)}, M^{(j)}, t_k)$ . In the rest of the paper, we make use of an indicator function  $\mu_{\geq 0}(x)$  which is zero if  $x < 0$  and is one otherwise; we will also use the notation  $(x)_+ \stackrel{\text{def}}{=} \max(0, x)$ .

#### A. Mode-Change SBF

**Definition 1 (Pre-MCR-SBF):** The function  $\text{sbf}_{\text{prior}}(M^{(i)}, t)$  is the minimum execution guaranteed by  $\Omega^{(i)}$  prior to  $\text{mcr}_k$  over the interval  $[t_k - t, t_k]$ .

**Definition 2 (Transition-MCR-SBF):** The function  $\text{sbf}_{\text{trans}}(M^{(i)}, M^{(j)}, t)$  is the minimum execution during  $\text{mcr}_k$  guaranteed by  $\Omega^{(ij)}$  and  $\Omega^{(j)}$  over the interval  $[t_k, t_k + t]$ .

**Definition 3 (Post-MCR-SBF):** The function  $\text{sbf}_{\text{post}}(M^{(i)}, M^{(j)}, s, t)$  is the minimum execution guaranteed by  $\Omega^{(ij)}$  and  $\Omega^{(j)}$  over the interval  $[t_k + \delta_{ij} - s, t_k + \delta_{ij} + t]$  (for  $0 \leq s \leq \delta_{ij}$ ).

For each of the above defined functions, Fisher and Ahmed [12] derived upper bounds as follows.

$$\text{sbf}_{\text{prior}}(M^{(i)}, t) \geq a\Theta^{(i)} + \min\left(\Theta^{(i)}, \left(t - \left((a+1)\Pi^{(i)} - \Theta^{(i)}\right)\right)_+\right). \quad (4)$$

$$\begin{aligned} \text{sbf}_{\text{trans}}(M^{(i)}, M^{(j)}, t) &\geq b\Theta^{(ij)} + c\Theta^{(j)} \\ &+ \min\left(\Theta^{(ij)}, \left(\min(t, \delta_{ij}) - \left(b\Pi^{(ij)} + \Delta^{(ij)} - \Theta^{(ij)}\right)\right)_+\right) \\ &+ \min\left(\Theta^{(j)}, \left((t - \delta_{ij})_+ - \left(c\Pi^{(j)} + \Delta^{(j)} - \Theta^{(j)}\right)\right)_+\right) \end{aligned} \quad (5)$$

$$\begin{aligned} \text{sbf}_{\text{post}}(M^{(i)}, M^{(j)}, s, t) &\geq d\Theta^{(ij)} \\ &+ \min\left(\Theta^{(ij)}, \left(s - \left((d+1)\Pi^{(ij)} - \Theta^{(ij)}\right)\right)_+\right) \\ &+ f\Theta^{(j)} + \min\left(\Theta^{(j)}, \left(t - \left(f\Pi^{(j)} + \Delta^{(j)} - \Theta^{(j)}\right)\right)_+\right) \end{aligned} \quad (6)$$

where  $a \stackrel{\text{def}}{=} \lfloor \frac{t}{\Pi^{(i)}} \rfloor$ ,  $b \stackrel{\text{def}}{=} \lfloor \frac{\min(t, \delta_{ij})}{\Pi^{(ij)}} \rfloor$ ,  $c \stackrel{\text{def}}{=} \lfloor \frac{(t - \delta_{ij})_+}{\Pi^{(j)}} \rfloor$ ,  $d \stackrel{\text{def}}{=} \lfloor \frac{s}{\Pi^{(ij)}} \rfloor$ , and  $f \stackrel{\text{def}}{=} \lfloor \frac{t}{\Pi^{(j)}} \rfloor$ .

#### B. Mode Change DBF

**Definition 4 (Carry-In Execution):** The carry-in execution  $\text{ci}(M^{(i)}, M^{(j)})$  from mode  $M^{(i)}$  to any other mode  $M^{(j)}$  at time  $t_k$  is an upper bound on the maximum possible remaining execution (over any legal sequence of MCRs) of non-aborted jobs from mode  $M^{(i)}$  for tasks  $\tau^{(i)} \setminus \{\tau^{(ij)} \cup \alpha^{(ij)}\}$  at time  $t_k + \delta_{ij}$  that arrive prior to  $t_k$  and the maximum total execution of unchanged tasks (i.e.,  $\tau^{(ij)}$ ) that have arrival time before  $t_k + \delta_{ij}$ .

The  $\text{ci}(M^{(i)}, M^{(j)})$  function can be obtained from the convergence of the sequence  $\text{ci}_0(M^{(i)}, M^{(j)})$ ,  $\text{ci}_1(M^{(i)}, M^{(j)})$ ,  $\text{ci}_2(M^{(i)}, M^{(j)})$ , ... for all  $i, j (i \neq j) \in \{1, \dots, q\}$ . For any  $\eta \in \mathbb{N}$ ,  $M^{(i)}$ , and  $M^{(j)}$  ( $i \neq j$ ), the function  $\text{ci}_\eta$  is inductively defined as follows.

$$\text{ci}_\eta(M^{(i)}, M^{(j)}) \stackrel{\text{def}}{=} \begin{cases} 0, & \text{if } \eta = 0, \\ \min\left(E_{ij}, F_{ij}\left(\max_{\substack{h=1, \dots, q \\ h \neq i}} \{\text{ci}_{\eta-1}(M^{(h)}, M^{(i)})\}\right)\right), & \text{if } \eta > 0. \end{cases} \quad (7)$$

where

$$E_{ij} \stackrel{\text{def}}{=} \left( \sum_{\tau_\ell^{(i)} \in \tau^{(i)} \setminus \alpha^{(ij)}} e_\ell^{(i)} + \sum_{\tau_\ell^{(ij)} \in \tau^{(ij)}} \left\lfloor \frac{\delta_{ij}}{\Pi^{(ij)}} \right\rfloor e_\ell^{(ij)} \right) + \sum_{\tau_\ell^{(ij)} \in \tau^{(ij)}} e_\ell^{(ij)}, \quad (8)$$

$$F_{ij}(\zeta) \stackrel{\text{def}}{=} \begin{pmatrix} \Psi(M^{(i)}, M^{(j)}, \zeta, \max(\delta_{ij} + d_{\max}^{(ij)}, d_{\max}^{(i)})) \\ - \sum_{\tau_\ell^{(ij)} \in \tau^{(ij)}} \text{dbf}(\tau_\ell^{(ij)}, \max(\delta_{ij} + d_{\max}^{(ij)}, d_{\max}^{(i)}) - \delta_{ij} - p_\ell^{(ij)}) \\ - \text{sbft}_{\text{trans}}(M^{(i)}, M^{(j)}, \delta_{ij}) \end{pmatrix}, \quad (9)$$

$$\Psi(M^{(i)}, M^{(j)}, \zeta, \phi) \stackrel{\text{def}}{=} \sup_{x \geq 0} \{\Psi_x(M^{(i)}, M^{(j)}, \zeta, \phi)\}, \quad (10)$$

$$\Psi_x(M^{(i)}, M^{(j)}, \zeta, \phi) \stackrel{\text{def}}{=} \begin{cases} \sum_{\tau_\ell^{(i)} \in \tau^{(i)} \setminus \{\tau^{(ij)} \cup \alpha^{(ij)}\}} \left( \left\lfloor \frac{(x - \lambda_\ell^{(i)}(\phi))_+}{p_\ell^{(i)}} \right\rfloor + 1 \right) \times e_\ell^{(i)} \\ + \sum_{\tau_\ell^{(i)} \in \alpha^{(ij)}} \left\lfloor \frac{x}{p_\ell^{(i)}} \right\rfloor \times e_\ell^{(i)} \\ + \sum_{\tau_\ell^{(i)} \in \alpha^{(ij)}} \min \left( x - \left\lfloor \frac{x}{p_\ell^{(i)}} \right\rfloor p_\ell^{(i)}, e_\ell^{(i)} \right) \\ + \sum_{\tau_\ell^{(i)} \in \tau^{(ij)}} \text{dbf}(\tau_\ell^{(i)}, x + \phi) \\ + \mu_{\geq 0} \left( x - N_i \Pi^{(i)} \right) \cdot \zeta \\ - \text{sbft}_{\text{prior}}(M^{(i)}, x) \end{cases}, \quad (11)$$

and  $\lambda_\ell^{(i)}(t) \stackrel{\text{def}}{=} \left( d_\ell^{(i)} - t \right)_+^+ .$  The convergence of the above sequence occurs at the smallest  $\eta \in \mathbb{N}$  such that  $\forall_{i,j \in \{1, \dots, q\} \cap i \neq j} \text{ci}_\eta(M^{(i)}, M^{(j)}) = \text{ci}_{\eta-1}(M^{(i)}, M^{(j)})$ . To provide intuition, the function  $\text{ci}_\eta(M^{(i)}, M^{(j)})$  in Equation 7 represents the maximum carry-in from  $M^{(i)}$  to  $M^{(j)}$  if  $\eta - 1$  mode changes have previously occurred. The maximum carry-in may be bounded by the total execution of jobs that are not aborted or are unchanged at the mode change from  $M^{(i)}$  to  $M^{(j)}$  (i.e.,  $E_{ij}$  of Equation 8). The maximum carry-in is also bounded by the total demand generated by jobs of  $M^{(i)}$  accounting for the maximum carry-in from some previous mode change from  $M^{(h)}$  (i.e.,  $\text{ci}_{\eta-1}(M^{(h)}, M^{(i)})$ ). The functions  $F_{ij}$  and  $\Psi$  (Equations 9 and 10) are used to calculate the total demand from “carried-in” from  $M^{(i)}$ , as formalized in the next definition.

**Definition 5:** The *carry-in demand-bound function*  $\text{cidbf}(M^{(i)}, M^{(j)}, \phi)$  for a mode change from  $M^{(i)}$  to  $M^{(j)}$  at time  $t_k$  and  $\phi \in \mathbb{R}_{\geq 0}$  is the maximum remaining execution (over any legal sequence of MCRs prior to  $t_k$ ) of jobs of tasks  $\tau^{(i)} \setminus \alpha^{(ij)}$  that arrive prior to  $t_k$  (or prior to  $t_k + \phi$  for  $\tau^{(ij)}$  tasks) and have deadlines in the interval  $[t_k, t_k + \phi]$ .

It may be shown [12] that  $\Psi(M^{(i)}, M^{(j)}, \text{ci}(M^{(i)}, M^{(j)}), \phi)$  is an upper bound on the carry-in demand-bound function  $\text{cidbf}(M^{(i)}, M^{(j)}, \phi)$  for any mode change from  $M^{(i)}$  to  $M^{(j)}$  over any interval of length  $\phi$ .

### C. Schedulability Conditions

Using definitions of the previous sub-sections, Fisher and Ahmed [12] developed schedulability conditions for multi-modal systems as follows. Over any possible (legal) sequence of mode-change requests, the system is EDF-schedulable, if

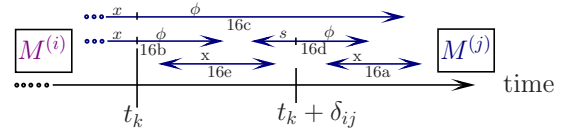


Fig. 5. Busy intervals for each of the schedulability conditions  $\text{SC}_Z$ .

the following five conditions hold for any two distinct modes  $M^{(i)}$  and  $M^{(j)}$ ,

$$\begin{aligned} \text{SC}_1 : & \sum_{\tau_\ell^{(j)} \in \tau^{(j)}} \text{dbf}(\tau_\ell^{(j)}, x) \leq \text{sbft}(\Omega^{(j)}, x), \quad \forall x \in \Upsilon^{\text{SC}_1(j)}; \\ \text{SC}_2 : & \sum_{\tau_\ell^{(j)} \in \tau^{(j)} \setminus \tau^{(ij)}} \text{dbf}(\tau_\ell^{(j)}, x) + \sum_{\tau_\ell^{(ij)} \in \tau^{(ij)}} \text{dbf}(\tau_\ell^{(ij)}, \phi + x) \\ & \leq \text{sbft}_{\text{post}}(M^{(i)}, M^{(j)}, \phi, x), \\ & \quad \forall s, \phi : (0 < \phi \leq \delta_{ij}) \wedge (x \in \Upsilon^{\text{SC}_2(i,j,\phi)}); \\ \text{SC}_3 : & \sum_{\tau_\ell^{(ij)} \in \tau^{(ij)}} \text{dbf}(\tau_\ell^{(ij)}, x) \leq \text{sbft}(\Omega^{(ij)}, x), \quad \forall x \in \Upsilon^{\text{SC}_3(i,j)}; \\ \text{SC}_4 : & \Psi_x(M^{(i)}, M^{(j)}, \zeta_i, x) \leq \text{sbft}_{\text{trans}}(M^{(i)}, M^{(j)}, \phi), \\ & \quad \forall \phi, x : (0 < \phi \leq \delta_{ij}) \wedge (x \in \Upsilon^{\text{SC}_4(i,j,\phi)}); \\ \text{SC}_5 : & \sum_{\tau_\ell^{(ij)} \in \tau^{(j)} \setminus \tau^{(ij)}} \text{dbf}(\tau_\ell^{(j)}, \phi) + \Psi_x(M^{(i)}, M^{(j)}, \zeta_i, \delta_{ij} + \phi) \\ & - \text{sbft}_{\text{trans}}(M^{(i)}, M^{(j)}, \delta_{ij}) \leq \text{sbft}_{\text{post}}(M^{(i)}, M^{(j)}, 0, \phi), \\ & \quad \forall \phi, x : (0 < \phi \leq \mathcal{T}_{ij}) \wedge (x \in \Upsilon^{\text{SC}_5(i,j,\phi)}). \end{aligned} \quad (12)$$

where  $\zeta_i \stackrel{\text{def}}{=} \max_{h=1, \dots, q \wedge h \neq i} \{\text{ci}(M^{(h)}, M^{(i)})\}$ .  $\Upsilon^{\text{SC}_1(i)}$ ,  $\Upsilon^{\text{SC}_2(i,j,s)}$ ,  $\Upsilon^{\text{SC}_4(i,j,\phi)}$ ,  $\Upsilon^{\text{SC}_5(i,j,\phi)}$ , and  $\mathcal{T}_{ij}$  are each a finite set of consecutive positive integers starting from one. Each of these sets are commonly referred as a *testing set*. We use a generic notation of  $\text{SC}_Z(i, j, \phi)$  where  $Z \in \{1, \dots, 5\}$  for the superscript of the testing sets. For example,  $\text{SC}_1(i, \emptyset, \emptyset)$  is the superscript for  $\Upsilon^{\text{SC}_1(i)}$ , which is the testing set of schedulability condition  $\text{SC}_1$  in Equation 12. The last two parameters in this example have the value of  $\emptyset$  as they are not used by  $\text{SC}_1$ . The number of testing sets associated with modes  $M^{(i)}$ ,  $M^{(j)}$ , and schedulability condition  $\text{SC}_Z$  is denoted by  $\text{TS}_Z(i, j)$ . The testing set bounds have been proven for which readers are referred to Fisher and Ahmed [12].  $\mathcal{T}_{ij}$  is specified by the number of  $\text{TS}_5(i, j)$  sets that exist and we will not refer to  $\mathcal{T}_{ij}$  further.

Now, we provide intuitive explanations for each condition of Equation 12. Before missing a deadline by an EDF-schedule, the processor is continuously busy. This interval is known as a busy interval. In the busy interval, the resource demand is larger than the processing supply. Five conditions are used to avoid busy intervals where resource demand is greater than the supply taking mode changes into account. Fisher and Ahmed [12] identified the five different kinds of busy intervals with respect to a mode change request, which are depicted in Figure 5. Therefore,  $\text{SC}_1$  ensures the schedulability of an individual mode.  $\text{SC}_2$  and  $\text{SC}_3$  ensure that an individual mode is schedulable along with the demand from unchanged tasks of the old mode after a mode change.  $\text{SC}_4$  ensures the schedulability during the transition period while accounting for

the carry-in demand from the non-aborted jobs and the mode-change supply function. Similarly,  $SC_5$  ensures schedulability after a transition. The last two conditions account for the carry-in from all past mode change requests through the ci function while analyzing demand of each individual mode.

#### IV. GPU-BASED SCHEDULABILITY

An important step in designing parallel algorithms is to decide the workload distribution among available processors. Workload distribution must account the underlying processing platform in order to achieve higher speedup. In this section, we first describe the GPU platform and then develop policies that take into account advantages and limitations of the GPU platform while distributing testing sets among processing elements. We emphasize a balanced workload distribution to decrease the overhead due to communication/synchronization and thus reduce the execution time. Table I, summarizes the notation used to describe our proposed algorithm.

##### A. GPU Architecture

The CPU directs image processing tasks to the GPU which relies heavily on arithmetic and logical operations, where image data is sent as a stream through a hardware graphics pipeline. This pipeline renders a stream in separate parts to construct an image. A GPU device has streaming multiprocessors (SM) each of which contains a fixed set of processing cores. This streaming architecture executes single instruction multiple data (SIMD) in parallel where each SM is computationally independent from any other SM, making it ideal for problems requiring large data sets processing. The Compute Unified Device Architecture (CUDA) provides the API to submit tasks to and receive results from the graphics processor. The computations are performed by calling a method from the CPU that hosts the GPU device known as a *kernel* function. Processing threads are created and grouped together in *blocks*. The number of threads and blocks are parameters of the kernel function. A block is executed by the GPU scheduler using a set of 32 parallel threads (known as a warp). While executing, each block and thread maintain unique indices `blockIdx` and `threadIdx`.

Any structure placed on the GPU global memory can be accessed by the GPU threads and the CPU. Shared memory structures are local to each streaming multiprocessor with access restricted to a block of threads. Maximizing the number of accesses to shared memory over the global memory is an important optimization since access to the shared memory can be 100 times faster than the global memory. This approach needs to be balanced with the amount of overhead spent on transferring information back and forth between the GPU and CPU memory structures. The cost effective performance of the GPU and ease of use by extending an API based on the C programming language are strong contributors to the success of the CUDA architecture. Ease of use is attributed to the GPU scheduler. The scheduler automatically manages the execution of threads with some explicit synchronization. Thousands of threads can be scheduled efficiently taking advantage of the

TABLE I  
NOTATION USED IN THIS PAPER

| Expression                                           | Description                                                              |
|------------------------------------------------------|--------------------------------------------------------------------------|
| $\tau^{(i)}$                                         | Real-time workload of $M^{(i)}$                                          |
| $\Omega^{(i)}$                                       | Minimum hardware resource of $M^{(i)}$                                   |
| $\tau_\ell^{(i)}$                                    | The $\ell$ -th sporadic task of $\tau^{(i)}$                             |
| $e_\ell^{(i)}$                                       | Execution requirement of $\tau_\ell^{(i)}$                               |
| $d_\ell^{(i)}$                                       | Relative deadline of $\tau_\ell^{(i)}$                                   |
| $p_\ell^{(i)}$                                       | Period of $\tau_\ell^{(i)}$                                              |
| $u_\ell^{(i)}$                                       | Utilization of $\tau_\ell^{(i)}$                                         |
| $\Pi^{(i)}$                                          | Resource period for $\Omega^{(i)}$                                       |
| $\Theta^{(i)}$                                       | Resource capacity of $\Omega^{(i)}$                                      |
| $\Delta^{(i)}$                                       | Resource deadline of $\Omega^{(i)}$                                      |
| $G$                                                  | Total number of groups                                                   |
| $B$                                                  | Total number of blocks in a group                                        |
| $T$                                                  | Total number of threads per block                                        |
| $\mathcal{P}^g$                                      | The $g$ -th group                                                        |
| $\mathcal{P}_\beta^g$                                | The $\beta$ -th block of group $\mathcal{P}^g$                           |
| $\mathcal{P}_{\beta,\gamma}^g$                       | The $\gamma$ -th thread of block $\mathcal{P}_\beta^g$                   |
| $r$                                                  | Data distribution root variable for $\mathcal{P}_{\beta,\gamma}^g$       |
| $\Upsilon^{SC_Z(i,j,\phi)}$                          | Testing sets for the pair $(M^{(i)}, M^{(j)})$ at $\phi$                 |
| $\Upsilon^{SC_Z}$                                    | Set of testing sets for condition $SC_Z$                                 |
| $\Upsilon_g^{SC_Z}$                                  | Testing sets of $SC_Z$ for group $g$                                     |
| $\Upsilon_{\psi(g,\beta,\gamma),r}^{SC_Z(i,j,\phi)}$ | Elements for which thread $\mathcal{P}_{\beta,\gamma}^g$ is responsible. |

available parallelism. Careful tuning of design parameters such as shared memory, number of threads/blocks, and thread vs block synchronization can result in significant performance gains.

In order to efficiently utilize the full computational power of a GPU, we evaluate each of the five schedulability conditions using *groups* of blocks  $\{\mathcal{P}^0, \dots, \mathcal{P}^{G-1}\}$ , where  $G$  is the number of groups on the GPU. The number of blocks in each  $\mathcal{P}^g$  is denoted by  $B$ . We invoke a GPU kernel for each condition  $SC_Z$  once. Our policy is to evaluate each testing set by a single group  $\mathcal{P}^g$ , which implies that no two groups evaluate the same testing set. Since the size of testing sets varies with each condition  $SC_Z$ , the system designer may change  $B$  when evaluating different  $SC_Z$  to obtain similar execution times for all testing sets. The number of testing sets also varies significantly with  $SC_Z$ . Therefore, the system designer may utilize different  $G$  with different  $SC_Z$ , but we restrict  $B$  and  $G$  to be fixed during the evaluation of a single  $SC_Z$  (i.e., throughout the execution of a single kernel invocation). The blocks per group  $B$  must be an integer and each group should contain at least one block (i.e.,  $B \geq 1$ ). Without loss of generality, we fix the number of threads per block to  $T$ . So we denote each group by  $\mathcal{P}^g \stackrel{\text{def}}{=} \{\mathcal{P}_0^g, \mathcal{P}_1^g, \dots, \mathcal{P}_{B-1}^g\}$ , where  $\mathcal{P}_\beta^g$  is a block composed of threads  $\{\mathcal{P}_{\beta,0}^g, \dots, \mathcal{P}_{\beta,T-1}^g\}$ .

##### B. Testing Set Distribution

In Section III, we denoted by  $\Upsilon^{SC_Z(i,j,\phi)}$  the individual testing set of the condition  $SC_Z$  for the pair  $(M^{(i)}, M^{(j)})$  at an interval of length  $\phi$ . We use the notation  $\Upsilon^{SC_Z}$  and  $\Upsilon^{SC_Z(i,j)}$  respectively to denote the set of all testing sets associated with the condition  $SC_Z$  and all testing sets of the pair  $(M^{(i)}, M^{(j)})$  for condition  $SC_Z$ . These notations are defined as follows

$$\Upsilon^{\text{SC}_Z}(i,j) = \left\{ \Upsilon^{\text{SC}_Z}(i,j,\phi) \mid 0 < \phi \leq \text{TS}_Z(i,j) \right\} \quad (13)$$

and

$$\Upsilon^{\text{SC}_Z} = \left\{ \Upsilon^{\text{SC}_Z}(i,j) \mid (0 \leq i, j < q) \wedge (i \neq j) \right\}. \quad (14)$$

Each testing set of  $\Upsilon^{\text{SC}_Z}$  can be uniquely identified by  $i$ ,  $j$ , and  $\phi$ . To obtain a unique index for each testing set  $\Upsilon^{\text{SC}_Z}(i,j,\phi)$ , we define the following function:

$$\mathcal{X}_Z(i,j,\phi) = \sum_{x=1}^{i-1} \sum_{\substack{1 \leq y \leq q \\ x \neq y}}^q \text{TS}_Z(x,y) + \sum_{\substack{y=1 \\ i \neq y}}^{j-1} \text{TS}_Z(i,y) + \phi. \quad (15)$$

We distribute all testing sets among the available groups (recall that each testing set is executed by a single group). A testing set  $\Upsilon^{\text{SC}_Z}(i,j,\phi) \in \Upsilon^{\text{SC}_Z}$  will be assigned to a particular group based on the index function  $\mathcal{X}_Z(i,j,\phi)$ . The testing sets corresponding to a group  $\mathcal{P}^g$  can be determined as follows:

$$\Upsilon_g^{\text{SC}_Z} = \left\{ \Upsilon^{\text{SC}_Z}(i,j,\phi) \mid (g \equiv \mathcal{X}_Z(i,j,\phi) \pmod{G}) \right\}. \quad (16)$$

The elements of  $\Upsilon^{\text{SC}_Z}(i,j,\phi) \in \Upsilon_g^{\text{SC}_Z}$  are distributed among the blocks of a group  $\mathcal{P}^g$ . Each testing set is divided into partitions of size  $\left\lceil \frac{\Upsilon^{\text{SC}_Z}(i,j,\phi)}{B} \right\rceil$  (the workload distribution is depicted in Figure 6). A block  $\mathcal{P}_\beta^g$  is responsible for the  $\beta$ -th partition of that testing set and distributes the partition among its threads  $\mathcal{P}_{\beta,\gamma}^g \in \mathcal{P}_\beta^g$ . We emphasize a balanced distribution of partition elements among these threads to increase speedup. We consider a round-robin distribution of a partition among threads of  $\mathcal{P}_\beta^g$ . We take into account the continuation of the last partition: that is, we start assigning the first element of a partition to a thread  $\mathcal{P}_{\beta,\gamma}^g$  where the previous thread  $\mathcal{P}_{\beta,\gamma-1}^g$  evaluates the last element of the previous partition. In order to achieve this, each thread  $\mathcal{P}_{\beta,\gamma}^g$  maintains a root variable  $r$  where  $r \in \{0, \dots, (T-1)\}$  (motivated by Ahmed et al. [3]). The variable  $r$  indicates the index of the first element assigned to  $\mathcal{P}_{\beta,\gamma}^g$  from the next partition. The set below represents the partition of  $\Upsilon^{\text{SC}_Z}(i,j,\phi)$  assigned to a thread  $\mathcal{P}_{\beta,\gamma}^g$  where  $\mathcal{P}_{\beta,\gamma}^g$ 's root variable is  $r$ .

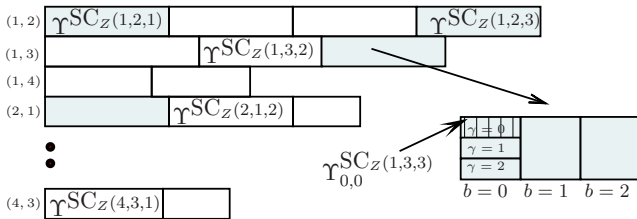


Fig. 6. Workload distribution for four modes where  $G$ ,  $B$  and  $T$  are assumed to be 3. Each row represents testing sets of  $\Upsilon^{\text{SC}_Z}(i,j)$  (associated with mode pair  $M^{(i)}$  and  $M^{(j)}$ ) where each rectangle denotes an individual testing set  $\Upsilon^{\text{SC}_Z}(i,j,\phi)$ . Shaded rectangles belong to  $\Upsilon_0^{\text{SC}_Z}$  which is assigned to the group 0. The bottom rectangle at the right depicts the distribution of the testing set  $\Upsilon^{\text{SC}_Z}(1,3,3)$  among all threads of the group 0.

$$\begin{aligned} \Upsilon_{\kappa,r}^{\text{SC}_Z}(i,j,\phi) &= \left\{ x \mid (x \in \Upsilon^{\text{SC}_Z}(i,j,\phi)) \wedge (r \equiv x \pmod{T}) \wedge \right. \\ &\quad \left. \left( \beta \times \left\lceil \frac{\Upsilon^{\text{SC}_Z}(i,j,\phi)}{B} \right\rceil \leq x < (\beta+1) \left\lceil \frac{\Upsilon^{\text{SC}_Z}(i,j,\phi)}{B} \right\rceil \right) \right\} \end{aligned} \quad (17)$$

where  $\kappa = \psi(g, \beta, \gamma)$  such that

$$\psi(g, \beta, \gamma) = g \times B \times T + \beta \times T + \gamma. \quad (18)$$

Note that the function  $\psi(g, \beta, \gamma)$  gives a system-wide unique identifier to each thread  $\mathcal{P}_{\beta,\gamma}^g$ . For  $\mathcal{P}_{\beta,\gamma}^g$  with root variable  $r$ , we determine  $\mathcal{P}_{\beta,\rho}^g$  that has  $r$  equal to zero for some  $\rho \in \{0, \dots, (T-1)\}$ . The expression  $X \stackrel{\text{def}}{=} ((\gamma - r) \pmod{T})$  identifies this thread. By distributing each of the elements of  $\Upsilon^{\text{SC}_Z}(i,j,\phi)$  in a round-robin fashion, the first thread to receive an element has the index equal to  $\rho' \stackrel{\text{def}}{=} ((X + \left\lceil \frac{\Upsilon^{\text{SC}_Z}(i,j,\phi)}{B} \right\rceil) \pmod{T})$ . To ensure balanced distribution,  $\rho'$  identifies the thread that receives the first element in the next partition. Thus, for any other thread  $\mathcal{P}_{\beta,\gamma}^g$  to determine the new root variable, we calculate  $((\gamma - \rho') \pmod{T})$ . After completing a partition, each thread updates  $r$  as

$$r = \left( \gamma - \left( (X + \left\lceil \frac{\Upsilon^{\text{SC}_Z}(i,j,\phi)}{B} \right\rceil) \pmod{T} \right) \right) \pmod{T}. \quad (19)$$

## V. ALGORITHM GpuSA

In this section, we develop a parallel algorithm for checking schedulability of a multi-modal system using policies defined in the previous section. The main algorithm is GpuSA, which utilizes three subroutines to perform the schedulability analysis. Conditions  $\text{SC}_4$  and  $\text{SC}_5$  require the maximum carry-in for all pairs of modes, which are calculated by MaxCarry.

### A. Description of the Algorithm

The pseudo-code for GpuSA is given in Algorithm 1, which is designed to execute in the CPU. GpuSA invokes CheckConditions for each of the five conditions with varying numbers of groups denoted by  $G$  where  $G = q$  if  $Z = 1$ , otherwise,  $G = q(q-1)$ . As discussed in the section for workload distribution, the number of testing sets and their sizes varies significantly over different conditions. For example,  $\text{SC}_1$  requires only the evaluation of  $q$  testing sets whereas

---

#### Algorithm 1 GpuSA( $\mathcal{M}, B$ )

---

```

1: {CPU executes:}
2:  $G \leftarrow q$ 
3: proceed  $\leftarrow$  true
4: for  $Z = 1$  to 5 do
5:   if  $Z = 4$  then
6:      $\zeta \leftarrow \text{MaxCarry}(\mathcal{M}, B)$ 
7:   Invoke CheckConditions( $Z, \zeta, B$ ) with  $G$  groups.
8:   Perform AND on returned values from line 7 to set proceed.
9:   if proceed = false then
10:    break;
11:    $G \leftarrow q(q-1)$ 
12: return proceed

```

---

---

**Algorithm 2** CheckConditions( $Z, \zeta, B$ ).

---

```
1: {Thread  $\mathcal{P}_{\beta, \gamma}^g$  executes:}
2:  $(g, \beta, \gamma) \leftarrow \text{Init}(\text{blockIdx}, \text{threadIdx}, B)$ 
3:  $r \leftarrow \text{threadIdx}$ 
4: for all  $(i, j)$  of  $q(q-1)$  pairs do
5:   for all  $\Upsilon^{\text{SC}_Z}(i, j, \phi)$  in  $\Upsilon_{\kappa, r}^g$  do
6:     for all  $x$  in  $\Upsilon_{\kappa, r}^{\text{SC}_Z}(i, j, \phi)$  do
7:       if  $Z = 1$  then
8:         if  $\text{sbf}(\Omega^{(i)}, x) < \text{dbf}(\tau^{(i)}, x)$  then
9:           return false
10:        else if  $Z = 2$  then
11:          if  $\text{dbf}(\tau^{(j)} \setminus \tau^{(ij)}, x) + \text{dbf}(\tau^{(ij)}, x + \phi) > \text{sbf}_{\text{post}}(M^{(i)}, M^{(j)}, \phi, x)$ 
then
12:            return false
13:          else if  $Z = 3$  then
14:            if  $\text{dbf}(\tau^{(ij)}, x) > \text{sbf}(\Omega^{(ij)}, x)$  then
15:              return false
16:            else if  $Z = 4$  then
17:              if  $\Psi_x(M^{(i)}, M^{(j)}, \zeta_{ij}, \phi) > \text{sbf}_{\text{trans}}(M^{(i)}, M^{(j)}, x)$  then
18:                return false
19:              else if  $Z = 5$  then
20:                 $\text{carry} \leftarrow \Psi_x(M^{(i)}, M^{(j)}, \zeta_{ij}, \phi) - \text{sbf}_{\text{trans}}(M^{(i)}, M^{(j)}, \delta_{ij})$ 
21:                if  $\text{carry} + \text{dbf}(\tau^{(i)} \setminus \tau^{(ij)}, x) > \text{sbf}_{\text{post}}(M^{(i)}, M^{(j)}, 0, x)$  then
22:                  return false
23:                Update  $r$  using Equation 19.
24: return true
```

---

all other conditions require at least  $q(q-1)$  testing sets (the condition  $\text{SC}_5$  requires the largest number of testing sets, which is  $\sum_{i,j \leq q} \text{TS}_5(i, j)$ ). Therefore, the system designer may want to use a varying number of groups for different  $\text{SC}_Z$  as a higher number of groups may increase the parallel efficiency of a GPU platform; however, a total number of groups greater than the number of testing sets for any  $\text{SC}_Z$  may not be a good policy as all groups with a group identifier  $g$  greater than the number of testing sets will not find any work to complete; thus, they will increase only the overhead.

### B. The CheckConditions Subroutine

The pseudo-code for CheckConditions is given in Algorithm 2. CheckConditions is designed to execute in the GPU to check each condition  $\text{SC}_Z$ . The number of groups  $G$  is passed to CheckConditions as an argument, but the actual GPU block should be a multiple of  $B$  if  $B > 1$ . The built-in variables blockIdx and threadIdx are initialized by the runtime system and accessed within the kernel. A block's group index  $g$  equals  $\lfloor \frac{\text{blockIdx}}{B} \rfloor$  and its block index  $\beta$  within a group equals  $(\text{blockIdx} \bmod B)$ . The thread index  $\gamma$  is initialized by the construct threadIdx. We create a function  $\text{Init}(\text{blockIdx}, \text{threadIdx}, B)$  on line 2 that initializes  $(g, \beta, \gamma)$  using the aforementioned rules. The root variable  $r$  is initialized with the GPU thread identifier threadIdx. The *for*-loop in lines 4 to 23 iterates over each pair for all pairs of modes (optionally, this *for*-loop could be replaced by two cascaded *for*-loops iterating over  $i, j < q$  and  $i \neq j$ ). After identifying  $(i, j)$ , the loop in lines 5 to 23 iterates over all testing sets assigned to group  $\mathcal{P}^g$ . For each  $\Upsilon^{\text{SC}_Z}(i, j, \phi)$ , the thread  $\mathcal{P}_{\beta, \gamma}^g$  considers its partition  $\Upsilon_{\kappa, r}^{\text{SC}_Z}(i, j, \phi)$  only where  $k \stackrel{\text{def}}{=} \psi(g, \beta, \gamma)$  and iterates each element  $x$  of this partition using the *for*-loop in lines 6 to 22. Based on  $Z$ , CheckConditions

---

**Algorithm 3** GetCarry( $\mathcal{M}, \zeta, B$ ).

---

```
1: {Each thread  $\mathcal{P}_{\beta, \gamma}^g$  executes:}
2:  $(g, \beta, \gamma) \leftarrow \text{Init}(\text{blockIdx}, \text{threadIdx}, B)$ 
3: change  $\leftarrow$  false
4:  $i \leftarrow \lfloor \frac{g}{q-1} \rfloor$ 
5: if  $i < g \bmod (q-1)$  then
6:    $j \leftarrow g \bmod (q-1)$ 
7: else
8:    $j \leftarrow g \bmod (q-1) + 1$ 
9: Compute  $E_{ij}$  using Equation 8.
10:  $\text{ci}_{\text{max}}^{(i)} \leftarrow \max_{\substack{h=1, \dots, q \\ h \neq i}} \{\zeta_{hi}\}$ 
11:  $c \leftarrow 0$ 
12:  $d \leftarrow \max(\delta_{ij} + d_{\text{max}}^{(ij)}, d_{\text{max}}^{(i)})$ 
13: for all  $x$  in  $\Upsilon_{\kappa, r}^{\text{SC}_4}(i, j, \delta_{ij})$  do
14:    $c \leftarrow \max(c, \Psi_x(M^{(i)}, M^{(j)}, \text{ci}_{\text{max}}^{(i)}, d))$ 
15:  $c \leftarrow c - \sum_{\tau_{\ell}^{(ij)} \in \tau^{(ij)}} \text{dbf}(\tau_{\ell}^{(ij)}, d - \delta_{ij} - p_{\ell}^{(ij)}) - \text{sbf}_{\text{trans}}(M^{(i)}, M^{(j)}, \delta_{ij})$ 
16: if  $\min(c, E_{ij}) > \zeta_{ij}$  then
17:    $\zeta_{ij} \leftarrow \min(c, E_{ij})$ 
18:   change  $\leftarrow$  true
19: return  $(\zeta_{ij}, \text{change})$ 
```

---

chooses the appropriate condition using a nested *if-then-else* block. Although a subroutine with branching does not perform well inside a GPU thread, we use an *if-then-else* form for ease of presentation. An implementation of CheckConditions may use five subroutines instead of nested *if-then-else*. The algorithm returns with false if  $\text{SC}_Z$  does not hold. These returned values are collected by GpuSA from the global memory. After completing a partition, each thread updates  $r$  at line 23.

We now develop algorithms that calculate the maximum carry-in for all pairs of modes. The maximum carry-in that a mode  $M^{(i)}$  can forward to  $M^{(j)}$ , taking into account all possible previous mode-change requests, is given by  $\text{ci}(M^{(i)}, M^{(j)})$ . The  $\text{ci}(M^{(i)}, M^{(j)})$  is calculated using the sequence  $\text{ci}_0(M^{(i)}, M^{(j)})$ ,  $\text{ci}_1(M^{(i)}, M^{(j)})$ ,  $\text{ci}_2(M^{(i)}, M^{(j)})$ ,  $\dots$ ,  $\text{ci}_{\eta}(M^{(i)}, M^{(j)})$ ,  $\dots$ . Fisher and Ahmed [12] showed that the maximum value of  $\eta$  is limited by the total execution requirements of any mode which is  $\max_{1 \leq i \leq q} \sum_{1 \leq \ell \leq n(i)} e_{\ell}^{(i)}$ . The entire sequence may not be evaluated using a single kernel invocation of the GPU as the calculation of  $\text{ci}_{\eta}(M^{(i)}, M^{(j)})$  requires the synchronization of results from all participating blocks. This calculation also depends on values calculated at the immediate previous step ( $\text{ci}_{\eta-1}(M^{(h)}, M^{(i)})$  for all  $h, i \leq q$ ); therefore, an implementation of  $\text{ci}$  using a single

---

**Algorithm 4** MaxCarry( $\mathcal{M}, B$ ).

---

```
1: {CPU executes:}
2: {Returns a  $[q \times q]$  matrix  $\zeta$ .}
3:  $\zeta \leftarrow 0$ 
4: repeat
5:   proceed  $\leftarrow$  false.
6:   Invoke GetCarry( $\mathcal{M}, \zeta, B$ ) using  $q(q-1)$  groups.
7:   Perform OR on change from  $q(q-1)$  groups to set proceed.
8: until proceed = false
9: return  $\zeta$ 
```

---

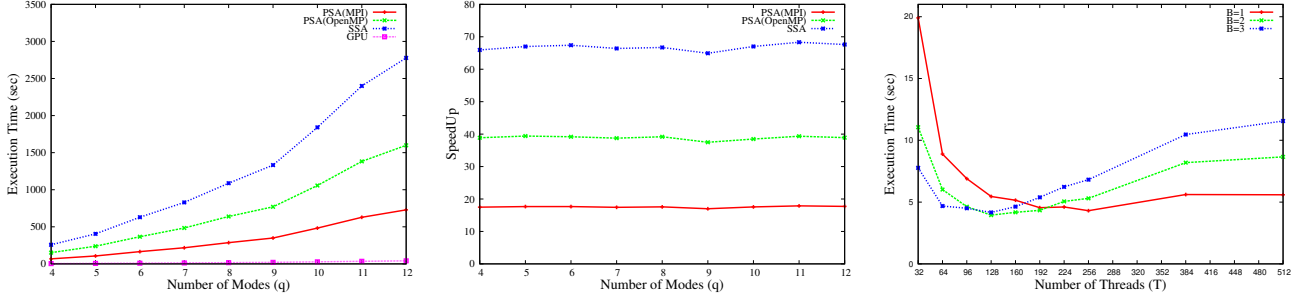


Fig. 7. Performance of GpuSA with respect to execution time (a), speedup (b), and scalability (c).

kernel invocation may require inter-block synchronization. Xiao and Feng [27] addressed the inter-block synchronization which may occur at the end of a thread-execution; however, the same technique may not be suitable for the scenario where synchronization among threads of different blocks may occur inside a loop. These inter-block synchronizations may result in deadlock where all the GPU blocks are not executed concurrently by a GPU scheduler. We instead divide the calculation of  $ci$  into two procedures: **GetCarry** and **MaxCarry**. **MaxCarry** is a CPU function that iterates over  $\eta$  and invokes the GPU function **GetCarry** to calculate  $ci_\eta(M^{(i)}, M^{(j)})$  where values calculated at the  $(\eta - 1)$ -th step is given as an argument.

### C. The GetCarry Subroutine

The device subroutine **GetCarry** is presented in Algorithm 3. Variables  $(g, \beta, \gamma)$  are initialized at the beginning by **Init(blockIdx, threadIdx, B)**. Each group calculates carry-in for a single pair of modes. Lines 5 to 8 determine mode indices  $(i, j)$  from the group index  $g$ . For each pair of modes  $(i, j)$ , there is a single testing set  $(\Upsilon^{sc_4(i,j,\phi)})$  to be evaluated. The block  $\mathcal{P}_\beta^g$ , in the group  $\mathcal{P}^g$ , is assigned a partition of consecutive testing set elements. Next, **GetCarry** calculates  $ci_\eta(M^{(i)}, M^{(j)})$  according to Equations 8, 9, and 10 and stores the new value in  $\zeta_{ij}$  if it is greater than the old value. The function marks the change by setting the **change** flag to true at line 18.

### D. The MaxCarry Subroutine

**MaxCarry** subroutine presented in Algorithm 4 calculates  $ci$  for all pairs of modes  $M^{(i)}$  and  $M^{(j)}$  using a *repeat-until* loop that invokes **GetCarry** with  $q(q - 1)$  groups at each iteration. **GetCarry** at each group calculates  $ci_\eta(M^{(i)}, M^{(j)})$  for different pairs of modes from the value calculated at the previous step and stores the new value in  $\zeta_{ij}$ . To determine proceed, **MaxCarry** performs an associative *or* operation (line 7) on all values of **change** returned from all groups. This step determines whether **change** is set to true by at least one thread. The function proceeds to the next step only if **proceed** is true. Otherwise, the function exits with the current values stored in  $\zeta$ . The last step is analogous to All-to-All-Reduce with an OR operator.

## VI. EXPERIMENTAL RESULTS

We compared **GpuSA** with two existing algorithms for Equation 12: a serial schedulability analysis (denoted as SSA) by Fisher and Ahmed [12] and a parallel schedulability analysis (PSA) by Ahmed et al [3]. PSA was executed upon a cluster of AMD Opteron computers on the Wayne State University grid. Each node has two 2.4GHz dual-core processors with 16GB of RAM. We have used two well-known parallel programming interfaces MPI and OpenMP to implement PSA. Both MPI and OpenMP implementations were executed on 2 processors (4 cores of a single node) in the cluster whereas SSA was executed on a single core from the same cluster. For **GpuSA**, we used a GeForce GT 440 GPU upon a computer with a 2.33GHz Intel® Core™ 2 Duo processor and 2.0GB RAM. The GT 440 has two streaming multiprocessors (SM) each of which contains 48 cores. For generating the multi-modal systems, we have used the following parameters and value ranges:

- 1) The number of modes  $q$  of a multi-modal system is taken from the range  $\{4, \dots, 12\}$ .
- 2) For the real-time workload of a multi-modal system, the UUniFast algorithm [5] is used to randomly generate a pool of 16 sporadic tasks by uniform distribution with total utilization 1. Each task period  $p_\ell^{(i)}$  is uniformly drawn from  $\{200, \dots, 2500\}$  and  $d_\ell^{(i)}$  is set to  $p_\ell^{(i)}$ .
- 3) For each mode  $M^{(i)}$ , 6 tasks have been chosen randomly from the pool of tasks using a uniform distribution.
- 4) Resource parameter  $\Omega^{(i)}$  is set to the highest  $(\Theta^{(i)} = \Pi^{(i)})$  which implies each mode  $M^{(i)}$  fully occupies the processing resource.

We performed two set of experiments. For the first set of experiments, we varied  $q$  from 4 to 12. We then measured the execution times for the **GpuSA** algorithm upon the GPU platform, SSA upon a single node on the cluster, and both OpenMP and MPI implementations of PSA upon the cluster. For each  $q$ , we have generated 25 multi-modal systems based on the policy described in the previous section. In Figure 7(a), the horizontal axis represents the total number of modes  $q$  while the vertical axis represents the mean execution time associated with each  $q$ . Among the implementations, the MPI version of PSA performs better than the OpenMP (it may be due to large number of variables OpenMP needs to propagate

among threads); however, the execution time of GpuSA is significantly less than that of any PSA implementation. This result is observed in Figure 7(b) where speedup is calculated from execution times of existing algorithms divided by the execution time of the GpuSA. The GpuSA achieves a speedup of approximately 17 while measured against a node with two 2.4GHz dual-core processors. For comparable performance, a parallel computing system requires 34 or more 2.4GHz dual-core processors even if PSA achieves the maximum speedup; thus, GpuSA saves both power and cost.

The second set of experiments were performed completely upon the GPU in order to measure the performance of GpuSA. We varied the total blocks ( $B$ ) per group and the total threads ( $T$ ) per block associated with each kernel invocation (for each  $SC_Z$ ) while checking the schedulability of a multi-modal system consisting of only 4 modes. Our experiment varied  $B$  from 1 to 3,  $T$  was varied in multiples of 32 from 32 to 512. In Figure 7(c), we present the execution times on the vertical axis and  $T$  on the horizontal axis. GpuSA obtains a better performance with  $T = 128$  and  $B = 2$ . One possible explanation could be the GPU scheduler might utilize all 96 cores efficiently if  $T = 128$  and  $B = 2$ . A SM schedules threads in groups of 32 parallel threads called warps [17]. Each SM has two warp schedulers which select two warps at a time and issues one instruction from each warp to a group of sixteen cores; therefore, having 128 threads per block provides enough work to keep the cores of each SM busy.

## VII. CONCLUSION AND FUTURE WORK

In this paper, we proposed a GPU-based schedulability analysis of real-time multi-modal systems that obtained significant speedup over existing schedulability analysis algorithms. The proposed algorithm takes advantage of favorable features (e.g., shared memory, SIMD architecture) and avoids limitations (e.g., difficulty in achieving inter-block synchronization) of a GPU platform. The increased speedup will be beneficial for determining minimum resource parameters of a multi-modal system by repetitive application of a schedulability analysis with varying resources. This process is referred to as design-space exploration. A fast schedulability analysis can further enhance the development of interactive tools (e.g., an intelligent personal assistant software for handheld devices) on top of real-time multi-modal systems.

## ACKNOWLEDGMENTS

This research is supported in part by an NSF-CAREER Grant (CNS-0953585), an NSF-CSR Grant (CNS-1116787), an NSF-IGERT Grant (DGE-0654014), and funding from Wayne State University's Office of Vice President of Research.

## REFERENCES

- [1] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proc. of the Real-Time Systems Symp.*, Spain, December 1998. IEEE Computer Society Press.
- [2] M. Ahmed and N. Fisher. Tractable real-time schedulability analysis for resource and application mode changes under temporal isolation. Technical report, 2012. Available at <http://paris.cs.wayne.edu/~dz3716/mme2012.pdf>.
- [3] M. Ahmed, N. Fisher, and D. Grosu. A parallel algorithm for edf-schedulability analysis of multi-modal real time systems. In *Proc. of the 18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Seoul, 2012.
- [4] S. Baruah, A. Mok, and L. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proc. of the 11th Real-Time Systems Symp.*, Florida, 1990.
- [5] E. Bini and G. Buttazzo. Biasing effects in schedulability measures. In *Proc. of the 16th Euromicro Conference on Real-Time Systems*. IEEE Computer Society, 2004.
- [6] S. Brandt, S. Banachowski, C. Lin, and T. Bisson. Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes. In *Proc. of the 24th IEEE Real-Time Systems Symp.*, 2003.
- [7] G. Buttazzo and L. Abeni. Adaptive rate control through elastic scheduling. In *Proc. of the 24th IEEE Conference on Decision and Control*, 2000.
- [8] A. Easwaran. *Compositional Schedulability Analysis Supporting Associativity, Optimality, Dependency and Concurrency*. PhD thesis, Computer and Information Science, U of Penn., 2007.
- [9] A. Easwaran, M. Anand, and I. Lee. Compositional analysis framework using EDP resource models. In *Proc. of the IEEE Real-time Systems Symp.*, Tuscon, Arizona, December 2007.
- [10] G. Elliott and J. Anderson. Globally scheduled real-time multiprocessor systems with GPUs. *Real-Time Systems*, 48(1):34–74, 2012.
- [11] J. Feng, S. Chakraborty, B. Schmidt, U. W. Liu, and U. Bordoloi. Fast schedulability analysis using commodity graphics hardware. In *Proc. of 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Daegu, Korea, August 2007.
- [12] N. Fisher and M. Ahmed. Tractable real-time schedulability analysis for mode changes under temporal isolation. In *Proc. of the 9th IEEE Symp. on Embedded Systems for Real-Time Multimedia*. IEEE Computer Society, October 2011.
- [13] P. M. Hettiarachchi, N. Fisher, M. Ahmed, L. Y. Wang, S. Wang, and W. Shi. The design and analysis of thermal-resilient hard-real-time systems. In *Proc. of the IEEE Real-Time and Embedded Technology and Applications Symp.*, April 2012.
- [14] J. Kim, I. Jung, and B. Lee. A scheduling technique based on the variable bandwidth server for multimedia systems. In *Proc. of the IEEE Region 10 Conference on Computers, Communications, Control and Power Engineering*, 2002.
- [15] D. B. Kirk and W.-m. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.
- [16] A. K. Mok. *Fundamental Design Problems of Distributed Systems for The Hard-Real-Time Environment*. PhD thesis, Laboratory for Computer Science, MIT, 1983. Technical Report.
- [17] NVIDIA. NVIDIA CUDA C programming guide, Feb 2011.
- [18] NVIDIA. CUDA in action - research and apps, Feb 2012.
- [19] L. Phan, S. Chakraborty, and I. Lee. Timing analysis of mixed time/event-triggered multi-mode systems. In *Proc. of the IEEE Real-Time Systems Symp.*, December 2009.
- [20] L. Phan, I. Lee, and O. Sokolsky. Compositional analysis of multi-mode systems. In *Proc. of the 22nd Euromicro Conference on Real-Time Systems (ECRTS)*, July 2010.
- [21] J. Real and A. Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Systems*, 26(2):161–197, 2004.
- [22] L. Santinelli, G. Buttazzo, and E. Bini. Multi-moded resource reservations. In *Proc. of the 17th IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS)*, 2011.
- [23] I. Shin and I. Lee. Compositional real-time scheduling framework with periodic model. *ACM Trans. Embed. Comput. Syst.*, 7(3):30–69, 2008.
- [24] B. Sprunt, L. Sha, and J. P. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems*, 1(1):27–69, 1989.
- [25] N. Stoimenov, S. Perathoner, and L. Thiele. Reliable mode changes in real-time systems with fixed priority or edf scheduling. In *Proc. of the Conference on Design, Automation and Test in Europe, DATE*, 2009.
- [26] N. Stoimenov, L. Thiele, L. Santinelli, and G. Buttazzo. Resource adaptations with servers for hard real-time systems. In *Proc. of the Tenth ACM International Conference on Embedded Software*, EMSOFT, New York, NY, USA, 2010. ACM.
- [27] S. Xiao and W. Feng. Inter-block gpu communication via fast barrier synchronization. In *Proc. of the IEEE Parallel and Distributed Processing Symposium*, April 2010.