Response Time Analysis of Hierarchical Scheduling: the Synchronized Deferrable Servers Approach

Haitao Zhu, Steve Goddard, Matthew B. Dwyer
Department of Computer Science and Engineering
University of Nebraska - Lincoln, Lincoln, NE 68588
{hzhu,goddard,dwyer}@cse.unl.edu

***Abstract*—Hierarchical scheduling allows reservation of processor bandwidth and the use of different schedulers for different applications on a single platform. We propose a hierarchical scheduling interface called synchronized deferrable servers that can reserve different processor bandwidth on each core, and can combine global and partitioned scheduling on a multi-core platform. Significant challenges will arise in the response time analysis of a task set if the tasks are globally scheduled on a multiprocessor platform *and* the processor bandwidth reserved for the tasks on each processor is different; as a result, existing works on response time analysis for dedicated scheduling on identical multiprocessor platforms are no longer applicable. A new response time analysis that overcomes these challenges is presented and evaluated by simulations. Based on this new analysis, we show that evenly allocating bandwidth across cores is "better" than other allocation schemes in terms of schedulability, and that the threshold between lightweight and heavyweight tasks under hierarchical scheduling may be different from the threshold under dedicated scheduling.**

***Keywords*-real-time systems, hierarchical scheduling, response time analysis, multi-core processors, multiprocessors**

I. Introduction

Hierarchical scheduling [1]–[6] allows reservation of processor bandwidth and the use of different schedulers for different applications on a single platform. In hierarchical scheduling, an *interface* specifies how processor resources are provided over time, e.g., "a server provides capacity of 5 time units at a period of 20 time units." While designing hierarchical scheduling, the designers need to choose an interface that can meet the application requirements. In practice, it is desirable to choose interfaces that are easy to implement while permitting schedulability to be guaranteed.

With the increasing use of multi-core architectures in real-time systems, multiprocessor scheduling has received growing attention. Conventional multiprocessor scheduling can be divided into two categories: *partitioned* and *global scheduling*. Under partitioned scheduling, tasks are statically bound to a processor, and cannot migrate across different processors. In contrast, under global scheduling, tasks can migrate across different processors. It is well-known that both partitioned and global scheduling have their own advantages and disadvantages, and neither dominates the other [7]. Hierarchical scheduling allows the use of partitioned and global scheduling in one system, for example, works towards hierarchical scheduling in earliest-deadline-first (EDF) systems have been reported in [4], [5].

We propose a multi-core hierarchical scheduling interface for *fixed-priority preemptive* systems called Synchronized Deferrable Servers (SDS). We consider the deferrable server in this paper as it is a bandwidth-preserving technique with good performance and low implementation complexity [8]. Our work can be generalized to non-bandwidth-preserving servers such as periodic servers, which will not be discussed in this paper but available in a longer version of this paper [9]. In an SDS interface, each core[1] hosts one Deferrable Server (DS) [10] with the same period and possibly different capacity. In this paper we focus on the case where only one highest-priority DS exists on a core, while presenting generalization to arbitrary-priority multiple DS on a core in a longer version [9]. The tasks in a system are divided into two categories: the *migrating* tasks and the *non-migrating* tasks. A migrating task can migrate across different cores, and thus can be processed by different DS. A non-migrating task is not processed by SDS, and is statically bound to a core with migration disallowed.

While clock synchronization is in general difficult for multiprocessors *without a global clock*, various multi-core architectures with a global clock for all cores have been used in real projects.[2] A periodic timer can then be used to achieve tight synchronization of the SDS. For synchronized servers, the primary source of synchronization error is delay in capacity replenishment. The periodic timer eliminates accumulated drift from a common period (except for clock drift). Under this scenario, the degree of synchronization achieved is the same as the synchronization of a single DS to its replenishment period or a set of tasks to their release periods.

Recent advances in multi-core architectures allow applica-

This work is supported in part by the National Science Foundation through awards CNS-0720757 and CCF-0912566, the National Aeronautics and Space Administration under grant number NNX08AV20A, and by the Air Force Office of Scientific Research through awards FA9550-09-1-0129 and FA9550-09-1-0687.

[1]We shall interchangeably use the terms *processor* and *core* in the rest of this paper.

[2]In a communication with Brandenburg, the maintainer of $LITMUS^{RT}$ project [11], he wrote: "...The supported x86 and ARM platforms (as well as the UltraSPARC platform supported in prior versions) all have a global clock signal that is accessible to all processors (in the form of cycle counter registers). Linux bases its notion of time on this global clock source, so there is no drift among processors."

1052-8725/11 $26.00 © 2011 IEEE
DOI 10.1109/RTSS.2011.29

IEEE computer society

tions built from a diverse collection of tasks to be realized on a single hardware platform. For some applications, migrating tasks across different cores may incur significant overhead, e.g., from cache misses, and these tasks should be statically bound to fixed cores. For other tasks, e.g., those that process volatile data, allowing migration offers the usual advantages of global scheduling.

An example demonstrating this diversity of tasks arises in run-time monitoring [12]. In run-time monitoring, a set of *monitor tasks* are used to collect and process *events* generated by *target tasks*. To overcome the monitoring overhead, researchers have exploited multiple execution cores to *hide* monitoring costs. The work in [13] proposes to use a dedicated core for monitoring. While dedicating an entire core is clearly a simple solution, it is not the most resource-efficient strategy.

In predictable monitoring [14], one is interested in guaranteeing that errors detected through monitoring will be reported within a bounded latency—latency is directly related to the maximum response time of the monitoring tasks. Effective processor utilization will enable more errors to be detected within their prescribed bounds. To achieve this, rather than dedicate a single core to monitor tasks one might instead spread the total monitoring bandwidth across each core. This resource efficient solution can be achieved using the relatively simple interface of SDS by first partitioning the target tasks among all the cores, and then use the unused but available bandwidth on each core for monitoring tasks.

Our Contributions: First, we propose a new multi-core hierarchical scheduling interface that allows the use of partitioned and global scheduling in one system. In conventional partitioned scheduling, some cores may have available but unused processor bandwidth after tasks are partitioned. Our interface can collect this bandwidth for migrating tasks and thus improve system utilization.

Our second contribution is the Response Time Analysis (RTA) of multi-core hierarchical scheduling for fixed-priority preemptive systems. Unlike existing RTA [15], [16] for *dedicated scheduling*[3] on identical multiprocessor platforms, our RTA can be applied to multi-core systems where each core provides different bandwidth. We present a sufficient condition to bound from above a task's response time, which is also applicable to dedicated scheduling considered in [15], [16]. In addition we show that, under hierarchical scheduling, a task's response time is affected by lower-priority tasks as well as higher-priority tasks.

Our third contribution is a demonstration that, given a fixed amount of total bandwidth on all cores, evenly allocating bandwidth across cores is superior to approaches that dedicate full bandwidth on individual cores. Thus, for improved schedulability, dedicating entire cores for a whole migrating task sets, which seems a "natural choice" in engineering practice, should be avoided.

Finally, our experimental results reveal *heavyweight tasks*' effect on a task set's schedulability. Heavyweight tasks have utilization or density exceeding a certain threshold. We show that, under hierarchical scheduling, the threshold for judging a task as heavyweight depends not only on its utilization or density, but also on the average bandwidth per core, which holds even when some cores are dedicated to tasks.

This paper is organized as follows. We introduce related work in Section II, and in Section III, the background and system model. In Section IV, we present our RTA for SDS. In Section V, we present our evaluation results, and based on these results, we discuss bandwidth allocation schemes and the effect of heavyweight tasks on tasks' schedulability. Section VI concludes this paper and identifies future work.

II. Related Work

Significant research has been done on hierarchical scheduling [1]–[6]. On uniprocessors, RTA of preemptive tasks executed by fixed-priority bandwidth-preserving servers, e.g., DS, has been studied [2], [3]. These approaches are based on the fact that the job with the maximum response time is released at a *critical instant*. In general, for multiprocessor scheduling, the critical instant of a task is unknown. Therefore, these approaches for uniprocessors cannot be applied to multiprocessor scheduling.

Baruah *et al.* studied Constant-Bandwidth Servers and Total Bandwidth Server on dynamic-priority multiprocessors [17], [18]. Recently, hierarchical scheduling framework combining partitioned and global scheduling on EDF systems has been reported in [4], [5]. A more general framework is proposed by Lipari and Bini [6], which allows designers to trade off resource usage and flexibility in determining virtual platform parameters.

While there are other significant works on hierarchical scheduling, we are not able to list them all in this paper. However, to the best of our knowledge, none of them present RTA for hierarchical scheduling on multiprocessor platforms.

RTA for dedicated scheduling on identical multiprocessor platforms has been studied in [15], [16]. The essential idea of these approaches is to bound from above higher-priority tasks' *interference* on the task of interest. However, these approaches cannot be applied to hierarchical scheduling on multiprocessor platforms.

A recent research topic is *semi-partitioned* scheduling [19]–[24]. Both semi-partitioned scheduling and our hierarchical scheduling interface combine the use of partitioned and global scheduling in one system, and can improve system utilization. However, our work overcomes some limitations of semi-partitioned scheduling. First, the existing semi-partitioned scheduling algorithms cannot be applied to dynamic task systems where tasks may join or leave the system during execution. If the task set is changed, the whole

[3]In this paper, dedicated scheduling means conventional scheduling where 100% bandwidth of each core is dedicated to all the tasks on that core, i.e., hierarchical scheduling is not used.

task set must be re-partitioned. Second, semi-partitioned scheduling algorithms assume that all tasks can migrate at the partitioning phase, even though only a subset of the tasks will migrate during execution. Third, semi-partitioned scheduling assumes each core dedicates full processor bandwidth to tasks, and thus no temporal protection among different applications is provided.

In global dedicated scheduling for identical multiprocessors, the "Dhall Effect" says that, when heavyweight tasks exist, the task set is less likely to be schedulable, even if the average task utilization is low. To circumvent this effect, researchers have invented algorithms that handle heavyweight tasks differently from lightweight tasks [25]–[28]. While the threshold for judging heavyweight tasks varies in previous work, determination of the threshold has not involved the bandwidth of the processors. For example, in [25], [27], [28], a utilization of 0.5 is usually regarded as the threshold of being heavyweight or not. Our results show that determination of the threshold should consider both task utilization (or density) and processor bandwidth.

III. Background and System Model

A DS [10] is a described by a 2-tuple $(T_{\mathrm{S}}, C_{\mathrm{S}})$, where T_{S} is the replenishment period, and C_{S} is the maximum capacity provided by the DS in a replenishment period. A DS works as follows. When a DS with available capacity obtains the processor, it processes pending workload; if no workload is pending, it simply holds its capacity. A DS waits for the next replenishment after it exhausts all its capacity in a replenishment period. Take as time 0 a DS's first replenishment, it will replenish its capacity with C_{S} time units at time $i \cdot T_{\mathrm{S}}$, $i \in \mathbb{N}_0$, and any unused capacity before a replenishment will be discarded.

We extend the DS concept to multi-core systems, and propose the concept of SDS. A set of m SDS, denoted by m-SDS, consists of m DS $(T_{\mathrm{S}}, C_{\mathrm{S}}^i)$, $1 \leq i \leq m$, where T_{S} is the common replenishment period, and C_{S}^i is the maximum capacity of the i-th DS, denoted by s_i, in a replenishment period. Since each DS has the same replenishment period, an m-SDS can be described by an $(m+1)$-tuple $(T_{\mathrm{S}}, C_{\mathrm{S}}^1, C_{\mathrm{S}}^2, ...C_{\mathrm{S}}^m)$. Without loss of generality, we order the DS such that, $\forall i$, $C_{\mathrm{S}}^i \geq C_{\mathrm{S}}^{i+1}$. Each DS consumes and replenishes capacity like a conventional uniprocessor DS.

While it is possible to have multiple sets of SDS each set of which has a different replenishment period in one system, in this paper we focus the case where only one set of SDS exists, and each DS has the highest priority on its host core. In practice, a system designer has the freedom of choosing a DS's replenishment period. To use optimal fixed-priority scheduling algorithms, such as Rate Monotonic (RM), one can choose a sufficiently short replenishment period. In a longer version of this paper [9], we present generalization to arbitrary priority DS and multiple DS on one core.

A. Non-migrating Tasks and Migrating Tasks

All tasks in the system are preemptable, and are divided into two categories: *non-migrating tasks* and *migrating tasks*. A non-migrating task is *not* processed by the SDS, and is statically bound to a core with migration disallowed.

A migrating task can migrate across different cores, and thus can be processed by different DS. Let n be the number of the migrating tasks, and the i-th migrating task be denoted by τ_i. Each migrating task is modeled as a sporadic task (T_i, C_i, D_i) where T_i is the minimum inter-arrival time, C_i is the Worst Case Execution Time (WCET), and D_i is the relative deadline. In practice, migration of a task has a certain overhead; and as in [29], the "cost of pre-emption, migration, and the runtime operation of the scheduler is assumed to be either negligible, or subsumed into the worst-case execution time of each task." In this paper, we consider *constrained-deadline* systems where $D_i \leq T_i$, while deferring the discussion of $D_i > T_i$ to future work. The priorities of the migrating tasks are ordered such that, $\forall i, \tau_i$ has a higher priority than τ_{i+1}.

In our system model, τ_i cannot be processed by two or more DS *at the same time*.

In this paper, we focus on the RTA of *migrating* tasks, and when no confusion arises, we shall refer to a migrating task simply as a task. The schedulability analysis of *non-migrating* tasks is essentially a uniprocessor scheduling problem, which has been well-studied [30]–[32].

B. Scheduling Policy

The scheduling policy of SDS consists of two levels: *intra*-core scheduling, and *inter*-core scheduling.

The intra-core scheduling policy utilizes a fixed-priority uniprocessor scheduling algorithm to locally schedule non-migrating tasks and DS on each core.

The inter-core scheduling policy determines on which DS a job (of a migrating task) executes. The system maintains a global job queue $\mathcal{Q}$ and a dispatcher $\mathcal{P}$. The job at $\mathcal{Q}$'s head has a higher priority than any other jobs in $\mathcal{Q}$. The policy is described as follows:

1) After a new job is released, it is added to $\mathcal{Q}$.
2) After a job is dispatched, it is removed from $\mathcal{Q}$.
3) After a running job is suspended, it is put back in $\mathcal{Q}$.
4) $\mathcal{P}$ makes a dispatching decision when one of the following events occurs: 1) a job is added to $\mathcal{Q}$; 2) a job is finished; 3) a suspended DS obtains the processor (e.g., a DS's capacity is replenished.)
5) $\mathcal{P}$ dispatches a job from $\mathcal{Q}$ to a DS as follows.
   - If there is an idle DS with available capacity, $\mathcal{P}$ dispatches the job at $\mathcal{Q}$'s head to that DS. If multiple such DS exist, $\mathcal{P}$ selects the DS with the smallest index.
   - If each DS with available capacity is processing a job, and the lowest-priority *running* job has a priority lower than the job at $\mathcal{Q}$'s head, it will be preempted.
   - If no DS has capacity, no dispatching will be made.

IV. Response Time Analysis

Denote by J_k^i the i-th job of the k-th task τ_k, and J_k^i's release time and response time are respectively denoted by r_k^i and R_k^i. When there is no need to distinguish which job of τ_k it is, we omit the superscripts i and simply use the notations of J_k, r_k and R_k. The job of τ_k that has the maximum response time is denoted by $J_k^{\max}$, and its response time is denoted by $R_k^{\max}$.

A job's *scheduling window* is the interval between when this job is released and when it is finished. By definition, the length of a job's scheduling window is its response time. A job's scheduling window can be divided into two parts: the *head* and the *body*. The head of a job J_k is the interval between r_k, the release time of the job, and the first replenishment after r_k. The body of J_k's scheduling window is the whole sub-interval following the head. Fig. 1 (a) illustrates the head and body of the scheduling window of a job J_2 released at time 14 and finished at 64.

Following the classic time demand analysis used in [2], [31], our RTA is performed by solving a recurrence equation:

$$R_k^{\max} = \Re(R_k^{\max}). \tag{1}$$

The works in [2], [31] make use of a *critical instant* concept, which is the release time of $J_k^{\max}$, the job with the maximum response time. Following this concept, we define the *critical head*, denoted by H_k^{C} (H stands for *Head*, C for *Critical*, and k indicates the k-th task), the *critical body*, denoted by B_k^{C} (B for *Body*), to be the head and the body of $J_k^{\max}$. With a little abuse of notation, we shall also use H_k^{C} and B_k^{C} to denote the *lengths* of their corresponding intervals, when no confusion arises. By definition,

$$R_k^{\max} = H_k^{\mathrm{C}} + B_k^{\mathrm{C}}. \tag{2}$$

If we know the *exact* critical instant of a task, we can determine the *exact* H_k^{C} and B_k^{C}, and thus calculate the exact $R_k^{\max}$. Unfortunately, in multiprocessor scheduling, the critical instant of a task is generally unknown, and thus, calculating the exact H_k^{C} and B_k^{C} is not possible. If, however, we can respectively calculate H_k^{C}'s upper bound, denoted by $\widehat{H_k^{\mathrm{C}}}$, and B_k^{C}'s upper bound, denoted by $\widehat{B_k^{\mathrm{C}}}$, we can then obtain an upper bound for $R_k^{\max}$. In the following discussion, we present how to calculate $\widehat{H_k^{\mathrm{C}}}$ and $\widehat{B_k^{\mathrm{C}}}$.

A. $\widehat{H_k^{\mathrm{C}}}$: Upper Bound of H_k^{C}

Denote by t_k^{C} the critical instant of τ_k, and by t_k^0 the last replenishment time before t_k^{C}. By definition,

$$H_k^{\mathrm{C}} = t_k^0 + T_{\mathrm{S}} - t_k^{\mathrm{C}}. \tag{3}$$

On a uniprocessor, if τ_k is processed by a DS, and there are other tasks consuming the same DS's capacity, t_k^{C} is the earliest instant when the capacity of the current replenishment period can be exhausted [33], which is C_{S} time units after a replenishment period begins. Therefore, on a uniprocessor, $t_k^{\mathrm{C}} = t_k^0 + C_{\mathrm{S}}$.

However, t_k^{C} on a set of SDS is not always the earliest instant when all DS exhaust their capacity, which is $t_k^0 + C_{\mathrm{S}}^1$ (recall that C_{S}^1 is the largest capacity of all the DS). This is as illustrated by Example IV.1.

Example IV.1. *Consider two tasks:* $\tau_1 = (100, 18, 100)$ *and* $\tau_2 = (150, 34, 150)$ *processed by a 2-SDS* $= (20, 14, 10)$. *Take as time* 0 *the beginning of the replenishment period within which a job* J_2 *of* τ_2 *is released. The earliest instant when all DS's capacity is consumed is* 14*, and this scenario is illustrated in Fig. 1 (a). If* $r_2 = 14$ *and all the DS's available capacity* before 14 *is consumed, then* J_2*'s response time is as most* 50*, as illustrated in Fig. 1 (a). However, consider another scenario in Fig. 1 (b), if* $r_2 = 10$*, and a job* J_1 *of* τ_1 *is also released at time* 10*, then* J_2*'s response time is* 54*, as illustrated in Fig. 1 (b).*

While the exact value of t_k^{C} is generally unknown, the following Lemma 1 states that $t_k^{\mathrm{C}} \geq t_k^0 + C_{\mathrm{S}}^m$, where C_{S}^m is the smallest capacity of all the DS.

Lemma 1. $t_k^{\mathrm{C}} \geq t_k^0 + C_{\mathrm{S}}^m$

Proof: See Appendix A. ■

Corollary 1. $H_k^{\mathrm{C}} \leq \widehat{H_k^{\mathrm{C}}}$ *where*

$$\widehat{H_k^{\mathrm{C}}} = T_{\mathrm{S}} - C_{\mathrm{S}}^m \tag{4}$$

Proof: By (3), $t_k^{\mathrm{C}} = t_k^0 + T_{\mathrm{S}} - H_k^{\mathrm{C}}$, and by Lemma 1, $t_k^0 + T_{\mathrm{S}} - H_k^{\mathrm{C}} \geq t_k^0 + C_{\mathrm{S}}^m \implies H_k^{\mathrm{C}} \leq T_{\mathrm{S}} - C_{\mathrm{S}}^m$. ■

A longer head does not necessarily lead to a longer response time, as a longer head also means possibly more available capacity in the head. Thus more workload can be processed within the head, and this in turn may decrease the length of the body. Therefore, Corollary 1 does *not* state that we can simply use $\widehat{H_k^{\mathrm{C}}} = T_{\mathrm{S}} - C_{\mathrm{S}}^m$ as the *exact* critical head. However, if we use $\widehat{H_k^{\mathrm{C}}}$ as the upper bound of the critical head, *and* "discard" all the capacity within the critical head, that is, all the workload within $J_k^{\max}$'s scheduling window is processed in the body B_k^{C}, then we can bound H_k^{C} and B_k^{C} from above at the same time. How to bound B_k^{C} from above is discussed next.

B. $\widehat{B_k^{\mathrm{C}}}$: Upper Bound of B_k^{C}

By (2), the sufficient condition to bound from above a job J_k's response time R_k is also the sufficient condition to bound from above the body B_k^{C}. Next we present a sufficient condition to bound R_k from above.

1) Sufficient Condition for Bounding R_k: For further discussion, we define the concepts of *work-conserving* and *capacity-conserving* as follows.

Work-conserving: A scheduling algorithm is work-conserving if it will never idle a processor whenever there is pending workload on that processor.

Capacity-conserving: Denote by $Cpty([t_s, t_e])$ the capacity available for processing workload within an interval

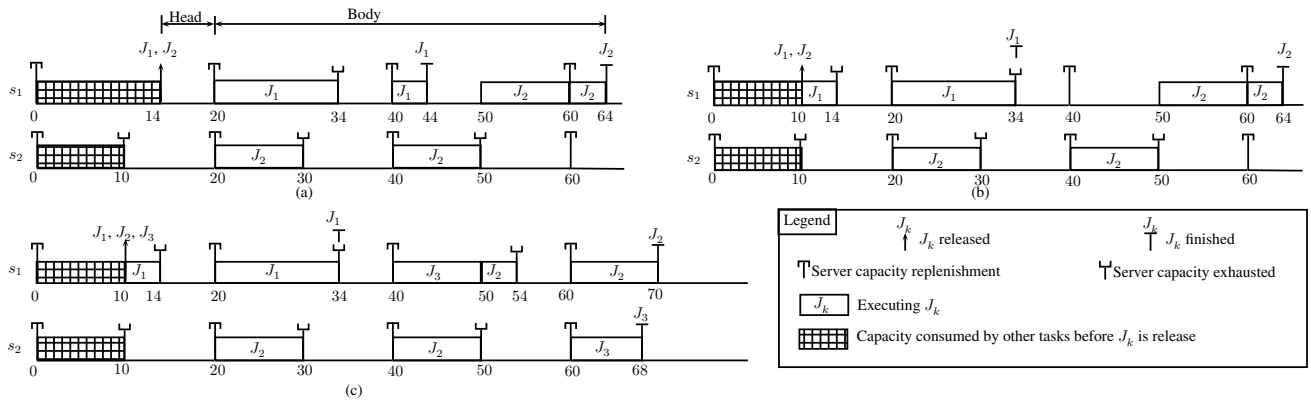

Fig. 1. RTA of Tasks on a 2-SDS

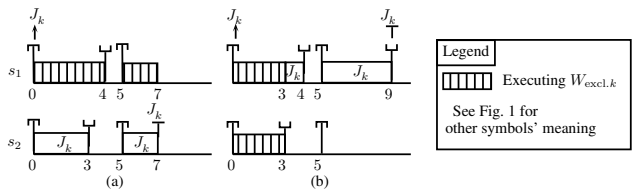

Fig. 2. Example for Intuition of Lemma 2

$[t_s, t_e]$ (when all processors are running under full load), a scheduling algorithm is capacity-conserving if: $t''_e > t'_e \implies Cpty([t_s, t''_e]) \geq Cpty([t_s, t'_e])$. Intuitively, capacity-conserving implies that, beginning at an instant, the capacity in a longer interval is no less than the capacity in a shorter interval.

Most common scheduling algorithms, e.g., RM and Deadline Monotonic (DM), as well as DS, are both work- and capacity-conserving. The concepts of work- and capacity-conserving will be used in the following Lemma 2 that bounds R_k (and thus B_k^{C}) from above.

Lemma 2. *Let $W_{\text{excl.}k}$ be the total workload of jobs excluding J_k within J_k's scheduling window. If J_k and a fixed amount of $W_{\text{excl.}k}$ is processed within J_k's scheduling window by a work- and capacity-conserving algorithm, R_k has its upper bound if J_k does not start execution before* all *the workload $W_{\text{excl.}k}$ is* completely *finished.*

Proof: See Appendix B. ■

The intuition of Lemma 2 is illustrated by an example in Fig. 2. Consider a workload of $W_{\text{excl.}k} = 6$ units and a task $\tau_k = (10, 5, 10)$ processed by a 2-SDS $(5, 4, 3)$. Suppose $W_{\text{excl.}k}$ and J_k are both ready at time 0 when a replenishment occurs. In Fig. 2 (a), if J_k and $W_{\text{excl.}k}$ are processed at the same time on different DS, then J_k is finished at time 7. However, consider another scenario in Fig. 2 (b) where J_k does not start execution until $W_{\text{excl.}k}$ is completely finished (at time 3), J_k's finish time is delayed to 9, even $W_{\text{excl.}k}$ is finished earlier under this scenario.

It is worth noting that Lemma 2 is also applicable to dedicated scheduling on identical multiprocessor platforms considered in [15], [16], and provides for these platforms a sufficient condition to bound τ_k's maximum response time from above. For dedicated scheduling on identical multiprocessor platforms, if J_k does not start execution until $W_{\text{excl.}k}$ is completely finished, then all the processors must be busy executing $W_{\text{excl.}k}$, which takes $\frac{W_{\text{excl.}k}}{m}$ time units (here $W_{\text{excl.}k}$ is interpreted as the higher-priority tasks' workload processed under the worst-case scenario within an interval[4], while m is interpreted as the number of processors). R_k is then bounded from above by $\frac{W_{\text{excl.}k}}{m} + C_k$, as reported in [15], [16].[5]

Note that the sufficient condition stated in Lemma 2 may *not* necessarily occur for a particular task set and SDS. Lemma 2 states that, under the scenario where the sufficient condition holds, R_k will not be less than what it is under any other scenario where the same amount of $W_{\text{excl.}k}$ and J_k are processed.

2) Bounding B_k^{C} from Above: After the sufficient condition to bound R_k from above is presented, we show how to bound B_k^{C} from above.

Denote respectively by $W_{\text{HP}}^k(R_k^{\max})$ and $W_{\text{LP}}^k(R_k^{\max})$ the upper bounds of the higher- and lower- priority[6] tasks' workload processed within $J_k^{\max}$'s scheduling window under the worst-case scenario. Let

$$W_{\text{HL}}^k(R_k^{\max}) = W_{\text{HP}}^k(R_k^{\max}) + W_{\text{LP}}^k(R_k^{\max}) \tag{5}$$

How to calculate $W_{\text{HP}}^k(R_k^{\max})$, $W_{\text{LP}}^k(R_k^{\max})$ and thus $W_{\text{HL}}^k(R_k^{\max})$ will be presented in Section IV-B3 and IV-B4. For simplicity, readers can assume for now that they are known, and this will not affect understanding the following Corollary 2.

[4]In [15], this interval is $J_k^{\max}$'s scheduling window, while in [16], it is an extended busy window.

[5] $\frac{W_{\text{excl.}k}}{m}$ bounds from above $J_k^{\max}$'s *interference* in [15], [16]. One major difference between these works lies in how to bound $W_{\text{excl.}k}$ from above.

[6]Lower-priority tasks need to be considered as they also consume capacity, as will be explained in Section IV-B4.

Corollary 2. *Under the scenario where $J_k^{\max}$ does not start execution before* all *the workload $W_{\mathrm{HL}}^k(R_k^{\max})$ is* completely *finished, let $R_{\mathrm{HL}/k}$ be the time to process the workload $W_{\mathrm{HL}}^k(R_k^{\max})$, and $R_{k/\mathrm{HL}}$ be the time to process J_k* after $W_{\mathrm{HL}}^k(R_k^{\max})$ is processed, *$B_k^{\mathrm{C}} \leq \widehat{B_k^{\mathrm{C}}}$ where*

$$\widehat{B_k^{\mathrm{C}}} = R_{\mathrm{HL}/k} + R_{k/\mathrm{HL}} \tag{6}$$

Proof: By (2), the sufficient condition to bound from above a job J_k's response time R_k is also the sufficient condition to bound from above the body B_k^{C}. Based Lemma 2, we have this corollary. ■

$R_{\mathrm{HL}/k}$ and $R_{k/\mathrm{HL}}$ can be respectively calculated by Lemma 3 and Lemma 4.

Lemma 3.

$$R_{\mathrm{HL}/k} = (\lceil \frac{W_{\mathrm{HL}}^k(R_k^{\max})}{\sum_{i=1}^m C_{\mathrm{S}}^i} \rceil - 1) \cdot T_{\mathrm{S}} + t_{\mathrm{res}}^{\mathrm{HL}} \tag{7}$$

where

$$t_{\mathrm{res}}^{\mathrm{HL}} = \begin{cases} \frac{W_{\mathrm{res}}^{\mathrm{HL}}}{m}, & \text{if } W_{\mathrm{res}}^{\mathrm{HL}} \leq \delta(m) \\ C_{\mathrm{S}}^{i+1} + \frac{W_{\mathrm{res}}^{\mathrm{HL}} - \delta(i+1)}{i}, & \text{if } \delta(i+1) < W_{\mathrm{res}}^{\mathrm{HL}} \leq \delta(i), \\ & \forall i : 1 \leq i \leq m-1 \end{cases} \tag{8}$$

$$W_{\mathrm{res}}^{\mathrm{HL}} = W_{\mathrm{HL}}^k(R_k^{\max}) - (\lceil \frac{W_{\mathrm{HL}}^k(R_k^{\max})}{\sum_{i=1}^m C_{\mathrm{S}}^i} \rceil - 1) \cdot \sum_{i=1}^m C_{\mathrm{S}}^i \tag{9}$$

$$\forall i, 1 \leq i \leq m : \delta(i) = \sum_{j=i}^m C_{\mathrm{S}}^j + C_{\mathrm{S}}^i \cdot (i-1) \tag{10}$$

Proof sketch*:* Under work-conserving scheduling, if $J_k^{\max}$ does not start execution before $W_{\mathrm{HL}}^k(R_k^{\max})$ is *completely* finished, then before $J_k^{\max}$ starts execution, *all* DS with available capacity are executing $W_{\mathrm{HL}}^k(R_k^{\max})$. In other words, before $W_{\mathrm{HL}}^k(R_k^{\max})$ is finished, each DS's capacity is used to process $W_{\mathrm{HL}}^k(R_k^{\max})$. This is the key observation for calculating $R_{\mathrm{HL}/k}$. For details, see Appendix C. ■

Lemma 4.

$$R_{k/\mathrm{HL}} = \begin{cases} C_k, & \text{if } C_{\mathrm{S}}^{\mathrm{rmn},k} \geq C_k \\ T_{\mathrm{S}} - t_{\mathrm{res}}^{\mathrm{HL}} + CRP_k \cdot T_{\mathrm{S}} + C_k - & \\ CRP_k \cdot \min(\sum_{i=1}^m C_{\mathrm{S}}^i, T_{\mathrm{S}}), & \text{otherwise} \end{cases} \tag{11}$$

where

$$CRP_k = \lceil \frac{C_k - C_{\mathrm{S}}^{\mathrm{rmn},k}}{\min(\sum_{i=1}^m C_{\mathrm{S}}^i, T_{\mathrm{S}})} \rceil - 1 \tag{12}$$

$$C_{\mathrm{S}}^{\mathrm{rmn},k} = \min(\sum_{i=1}^m C_{\mathrm{S}}^i - W_{\mathrm{res}}^{\mathrm{HL}}, T_{\mathrm{S}} - t_{\mathrm{res}}^{\mathrm{HL}}) \tag{13}$$

and $t_{\mathrm{res}}^{\mathrm{HL}}$ and $W_{\mathrm{res}}^{\mathrm{HL}}$ are respectively given by (8) and (9).

Proof: See Appendix D. ■

Calculation of $R_{\mathrm{HL}/k}$ and $R_{k/\mathrm{HL}}$ relies on $W_{\mathrm{HL}}^k(R_k^{\max})$ that (, based on (5),) is determined by $W_{\mathrm{HP}}^k(R_k^{\max})$ and $W_{\mathrm{LP}}^k(R_k^{\max})$, as discussed in the next two sub-sections.

3) To Calculate $W_{\mathrm{HP}}^k(R_k^{\max})$: Let $RW_{\mathrm{HP}}^k(L)$ denote the upper bound of the *requested* workload of the tasks with a priority higher than τ_k in an interval of length L.

Under the worst-case scenario, for the higher-priority tasks, *all* the requested workload in $J_k^{\max}$'s scheduling window will be processed no later than when $J_k^{\max}$ is finished, that is,

$$W_{\mathrm{HP}}^k(R_k^{\max}) = RW_{\mathrm{HP}}^k(R_k^{\max}). \tag{14}$$

It is important to note that the *requested* workload and the *processed* workload under the worst-case scenario may be different for lower-priority tasks, as will be shortly discussed in Section IV-B4. This is the reason for distinguishing the concepts of requested and processed workloads.

Let $RW_i^k(L)$ denote the upper bound of a task τ_i's workload requested in an interval of L, then

$$RW_{\mathrm{HP}}^k(L) = \sum_{i<k} RW_i^k(L). \tag{15}$$

The calculation of $RW_i^k(L)$ has been studied in [15], [16], [34], [35]. Among these works, [16], [35] have tighter results than the other works. In [16], [35], the authors extend the beginning of $J_k^{\max}$'s scheduling window to an earlier instant so as to obtain a tighter upper bound of the *carry-in* (and thus the workload) of the *higher*-priority tasks. However, it is unknown whether an instant before $J_k^{\max}$'s release time can be found such that both the higher- and lower-priority tasks' carry-in can be bounded from above *tightly at the same time*. As we shall see shortly, both the higher- and lower-priority tasks must be considered in the RTA of SDS. Therefore, the techniques in [16], [35] cannot be applied here.

Without more efficient and accurate techniques at hand, we resort to the technique proposed in [15]:

$$\begin{aligned} &\forall i \neq k : RW_i^k(L) = \\ &N_i(L) \cdot C_i + \min(C_i, L + D_i - C_i - N_i(L) \cdot T_i) \end{aligned} \tag{16}$$

where $N_i(L) = \lfloor \frac{L + D_i - C_i}{T_i} \rfloor$.

While (16) can be used to bound from above the requested workloads of both the higher- and lower-priority tasks at the same time, for the lower-priority tasks, we can obtain a tighter upper bound of $W_{\mathrm{LP}}^k(R_k^{\max})$ by utilizing the fact that not all requested workload of the lower-priority tasks needs to be processed within $J_k^{\max}$'s scheduling window even under the worst-case scenario, as discussed next.

4) To Calculate $W_{\mathrm{LP}}^k(R_k^{\max})$: Under dedicated scheduling, a job J_k's response time is not affected by lower-priority tasks. Under hierarchical scheduling, however, its response time may be affected by lower-priority tasks, since while it is executing on one DS, there may be lower-priority tasks running on other DS, which will decrease the total capacity available to J_k, and thus increase J_k's response time, as illustrated in Example IV.2.

Example IV.2. *Consider again the two tasks and SDS given in Example IV.1, and now there is a third task $\tau_3 =$*

$(100, 18, 100)$. *In Fig. 1 (c), if a job J_3 is also released at time 10 together with J_1 and J_2, $R_2 = 60$ is longer than $R_2 = 54$ in Fig. 1 (b).*

Example IV.2 indicates a significant difference between the RTA for hierarchical scheduling and the RTA for dedicated scheduling [2], [3], [15], [16] where only higher-priority tasks need to be considered.

Let $RW^k_{\mathrm{LP}}(L)$ denote the upper bound of the *requested* workload of the tasks with priority lower than τ_k in an interval of length L:

$$RW^k_{\mathrm{LP}}(L) = \sum_{i>k} RW^k_i(L) \tag{17}$$

where $RW^k_i(L)$ is given by (16). In general, within a job $J^{\max}_k$'s scheduling window, $RW^k_{\mathrm{LP}}(R^{\max}_k)$ is different from $W^k_{\mathrm{LP}}(R^{\max}_k)$, the *processed* workload of the lower-priority tasks. Lower-priority tasks can run only when $J^{\max}_k$ is running, and it is possible that only a portion of the requested workload in $J^{\max}_k$'s scheduling window is processed, depending on how much *cumulative capacity* is available for the lower-priority tasks in $J^{\max}_k$'s scheduling window. The cumulative capacity for a task within an interval is the amount of this task's workload that can be processed within this interval (when no DS is idle in this interval). Let $CCL_k(L)$ denote the cumulative capacity for processing lower-priority tasks within an interval of length L, then

$$W^k_{\mathrm{LP}}(L) = \min(RW^k_{\mathrm{LP}}(L), CCL_k(L)) \tag{18}$$

Within $J^{\max}_k$'s scheduling window, $CCL_k(R^{\max}_k)$ can be bounded from above based on the following observations:

1) Lower-priority tasks run only when $J^{\max}_k$ is running;
2) While $J^{\max}_k$ is running, at most $m-1$ DS are executing lower-priority tasks;
3) $J^{\max}_k$ runs for C_k time units.

Based on these observations, $CCL_k(R^{\max}_k)$ is bounded by

$$CCL_k(R^{\max}_k) = (m-1) \cdot C_k \tag{19}$$

C. Putting the Pieces Together

We now give a theorem that bounds $R^{\max}_k$ from above.

Theorem 1.

$$R^{\max}_k \leq \widehat{H^{\mathrm{C}}_k} + \widehat{B^{\mathrm{C}}_k}$$

where $\widehat{H^{\mathrm{C}}_k}$ and $\widehat{B^{\mathrm{C}}_k}$ are given by (4) and (6).

Proof: It follows from (2), Corollaries 1 and 2. ■

$R^{\max}_k$ is then bounded by the smallest solution to

$$x = \widehat{H^{\mathrm{C}}_k} + \widehat{B^{\mathrm{C}}_k} \tag{20}$$

(20) can be solved via iteration starting with $x = \widehat{H^{\mathrm{C}}_k} + C_k$, and it terminates if a solution is found, or $x > D_k$.

V. Evaluation and Discussion

In this section we evaluate our work under different settings. While designing SDS, given a fixed amount of total bandwidth and the freedom of choosing how to allocate the bandwidth to each DS, a question of interest is: How does a bandwidth allocation scheme affect the task set's schedulability? Detailed discussion of this question is presented in Section V-B.

Another question of interest is: How does a task's utilization/density affect a task set's schedulability? In dedicated scheduling on identical multiprocessor platforms, *heavyweight* tasks, i.e., tasks with high utilization, will decrease the chance of a task set being schedulable, even if the total utilization of the task set is low. This is recognized as the "Dhall Effect" [36]. For task sets scheduled by SDS, we are interested not only in whether a similar effect exists, but also in what "heavyweight" means in this context. Detailed discussion of this question is presented Section V-C.

A. Experiment Settings

We examined 4-SDS and 8-SDS with average bandwidth per DS equal to 0.15, 0.3 and 0.5. For a specific set of SDS, the replenishment period is varied among 1000, 2000, ..., and 10000. Given a set of SDS with a certain average bandwidth, three types of bandwidth allocation schemes are considered. In the first allocation scheme, denoted by *EQUAL*, each DS has the same bandwidth. In the second scheme, denoted by *FIRST-FIT*, the total bandwidth is allocated to each DS in a *First-Fit* style: The first DS has as much bandwidth as possible (up to 1), and the second DS has as much of the rest of the bandwidth as possible, and so on. In the third scheme, denoted by *RANDOM*, the total bandwidth is randomly distributed across all DS.

Task sets of two different sizes, $n = 10$ and 20, are randomly generated. Since we are considering constrained-deadline task sets in this paper, *density* instead of utilization is used as an evaluation parameter. A task's density is the ratio of this task's WCET to the smaller value of its deadline and period.

We say that task sets with the same size n and the same average density per task belong to the same *task set class*. In our experiment, each task set class has 1000 task sets.

To generate tasks' density, we use the *UUniFast-Discard* algorithm [29]. After a task's density is determined, its deadline is randomly generated between 10000 and 100000 with uniform distribution. Each task's period is randomly chosen between its deadline and 1.5 times its deadline.

Due to space restriction, the discussion in the next two sub-sections is based on a representative subset of the results.

B. Bandwidth Allocation

The aforementioned three allocation schemes for an 8-SDS with an average bandwidth of 0.3 are examined. Fig. 3 shows the percentages of the task sets respecting their deadlines,

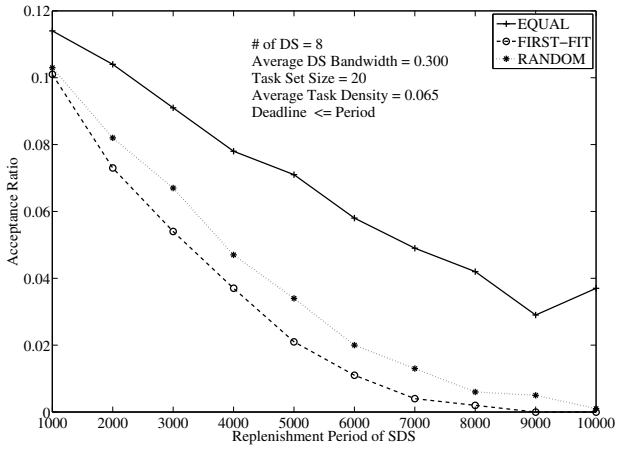

Fig. 3. Acceptance ratio v.s. replenishment period under different allocation schemes (for heavyweight tasks)

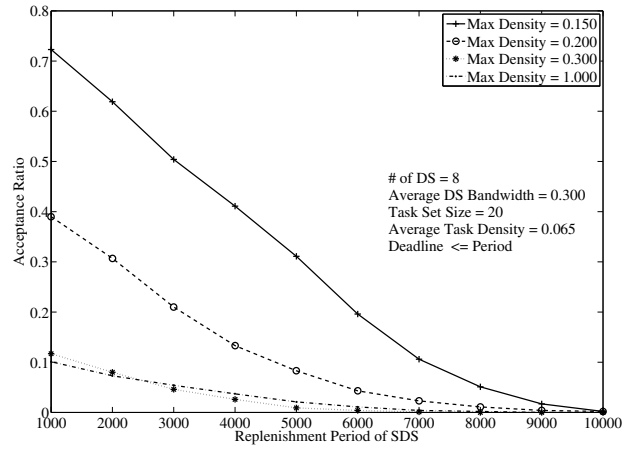

Fig. 4. Acceptance ratio v.s. replenishment period under FIRST-FIT allocation scheme

termed the *acceptance ratio*, when applying our RTA to 1000 task sets, each of which contains 20 tasks with an average density per task of 0.065. Fig. 3 indicates that, given a fixed amount of total bandwidth, *EQUAL* is the most likely to schedule a task set. This is because *EQUAL* has the highest degree of parallelism (among different tasks), and the interference suffered by a task is shorter under this allocation scheme than the other two schemes. In contrast, *FIRST-FIT* has the lowest degree of parallelism, and thus the interference on a task is longer.

The above result and conclusion seem to be contrary to the result in a recent work [6], wherein the authors argue that *FIRST-FIT* would be preferable in terms of tasks' schedulability. However, there is no contradiction here. The result shown in Fig. 3 is a statistical result, therefore, the above conclusion may not hold for a specific task. For example, consider the highest-priority task τ_1. Under *FIRST-FIT*, the bandwidth that can be consumed by lower-priority tasks during τ_1's execution is lower than the bandwidth under the other two schemes. In this regard *FIRST-FIT* favors the higher-priority task's schedulability. Further study is needed to understand the relationship between the bandwidth allocation scheme and a task at a particular priority.

C. Lightweight versus Heavyweight Tasks

The acceptance ratios in Fig. 3 are extremely low, and it turns out that this is related to the *UUniFast-Discard* algorithm used to generate the task density.

While the task sets generated by the *UUniFast-Discard* algorithm are regarded as *unbiased*, we noticed that this algorithm tends to generate task sets with at least one *heavyweight* task. Given a set of SDS with an average bandwidth U_S^{avg}, we define a heavyweight task running on the SDS to be a task with a density greater than $U_S^{avg}/2$. Our results show that if a heavyweight task is present, then this task is unlikely to meet its deadline; but if no heavyweight task exists in a task set, then the task set is more likely to be schedulable. This is illustrated by Fig. 4. In Fig. 4, an 8-SDS with an average bandwidth of 0.3 is studied, and the bandwidth allocation scheme is *FIRST-FIT*. Four task set classes with the same size and the same average density are studied. These four task set classes differ from each other in the maximum possible densities of the tasks in each class, which are 0.15, 0.2, 0.3 and 1 respectively.

As illustrated in Fig. 4, when the maximum density is no less than the average bandwidth, 0.3 in this case, the acceptance ratio is low. The acceptance ratio increases as the maximum density decreases. And when the maximum density is $1/2$ of the average bandwidth, 0.15 in this case, the acceptance ratio is much higher than the case where the maximum density is equal to the average bandwidth for smaller replenishment periods.

In another two experiments (whose results are not presented due to space limitations), the acceptance ratios of the four task set classes studied in Fig. 4 are also calculated under EQUAL and RANDOM allocation schemes, and the trend of the plotted curves are similar to Fig. 4. All of these results suggest that the threshold for a task to be heavyweight turns out to be irrelevant to the allocation scheme. In Fig. 4, under *FIRST-FIT*, there are two dedicated cores (bandwidth of 1), but the threshold of a heavyweight task is 0.15 instead of 0.5. For global scheduling on identical multiprocessors, it is well-known that, if a task set contains tasks with large utilizations or densities, it may be unschedulable even if the average task utilization or density is low [36]. An identical multiprocessor can be regarded as a special instance of SDS where each DS has bandwidth of 1. In this regard, our result extends the previous result. A quantitative analysis of how a task's utilization or density affects the system schedulability will be considered in future work.

VI. Conclusion and Future Work

We propose a fixed-priority preemptive hierarchical multiprocessor scheduling interface called SDS, and present the

RTA for migrating tasks. We identify the effect of lower-priority tasks in hierarchical scheduling RTA. Guidelines for designing SDS are discussed, and the effect of heavyweight on schedulability in hierarchical scheduling is studied.

The RTA presented in this paper suffers pessimism in three respects: First, during the critical head, no capacity is taken into account. Although this inaccuracy can be alleviated by selecting a shorter replenishment period, future work will be considered to bound the head more tightly.

Second, the higher-priority tasks' workload is bounded with the approach in [15], which has been shown to not be tight [16]. However, the tighter result in [16] cannot be used here as mentioned in Section IV-B3. Future work will investigate tighter bound on the higher-priority tasks' workload.

Third, to simplify the computational complexity, we assume that during τ_k's execution, all other DS are executing lower-priority tasks. Future work will improve the tightness by utilizing the fact that some DS may not execute due to exhaustion of capacity while τ_k is executing.

Appendix A

Proof of Lemma 1*:* We prove the lemma by showing that if a job J_k is released at time $r_k^{'}, t_k^0 \leq r_k^{'} < t_k^0 + C_{\mathrm{S}}^m$, its response time $R_k^{'}$ is less than its response time $R_k^{''}$ when it is released at time $r_k^{''} = t_k^0 + C_{\mathrm{S}}^m$.

$r_k^{''}$ is the earliest instant when the DS with the smallest capacity can exhaust its capacity in a replenishment period, so within $[r_k^{'}, r_k^{''})$, all DS have available capacity. Now consider an interval of length L. The available cumulative capacity within $[r_k^{'}, r_k^{'} + L)$ is greater than the available cumulative capacity within $[r_k^{''}, r_k^{''} + L)$ for *any* L.[7] As a result, to process the same amount of workload, the time it takes when $r_k^{'} < r_k^{''}$ is no greater than the time when $r_k^{'} = r_k^{''}$. Therefore, $J_k^{\max}$ cannot be released before $r_k^{''} = t_k^0 + C_{\mathrm{S}}^m$, that is, $t_k^{\mathrm{C}} \geq t_k^0 + C_{\mathrm{S}}^m$. ■

Appendix B

Proof of Lemma 2*:* Consider two scenarios: **I**. J_k does not execute before $W_{\mathrm{excl}.k}$ is *completely* finished, and **II**. J_k executes for a certain amount of time before $W_{\mathrm{excl}.k}$ is finished. For both scenarios, J_k's scheduling window has 3 types of intervals:

1) The intervals wherein at least one processor is executing but no processor is executing J_k. Let $I_{I,busy}$ and $I_{II,busy}$ respectively be the *total* lengths of such intervals in J_k's scheduling window under Scenario I and II;

2) The intervals wherein no processor is executing $W_{\mathrm{excl}.k}$ or J_k.[8] Let $I_{I,wait}$ and $I_{II,wait}$ be the *total* lengths of such intervals in J_k's scheduling window under Scenario I and II;

[7]Note that this property may not hold if $r_k^{'} \geq t_k^0 + C_{\mathrm{S}}^m$.

[8]due to, e.g., that no capacity is available to process $W_{\mathrm{excl}.k}$ or J_k. Such an interval does not exist in dedicated identical multiprocessor scheduling, but can exist in hierarchical scheduling.

3) The intervals wherein one processor is executing J_k. The *total* length of such intervals in J_k's scheduling window is J_k's WCET, C_k, for both scenarios.

Le $R_{I,k}$ and $R_{II,k}$ respectively be J_k's response times under the above two scenarios. We have

$$R_{I,k} = I_{I,busy} + I_{I,wait} + C_k \tag{21}$$

$$R_{II,k} = I_{II,busy} + I_{II,wait} + C_k \tag{22}$$

We prove $R_{II,k} \leq R_{I,k}$ by contradiction. Under Scenario II, when J_k is executing, there may or may not be other processors executing $W_{\mathrm{excl}.k}$, but under work-conserving scheduling, for either case, $I_{I,busy} \geq I_{II,busy}$, as the same amount of $W_{\mathrm{excl}.k}$ is processed under Scenario I and II. Based on (21) and (22):

$$I_{I,busy} \geq I_{II,busy}, R_{II,k} > R_{I,k} \implies I_{II,wait} > I_{I,wait} \tag{23}$$

Under dedicated scheduling, $I_{I,wait} = I_{II,wait} = 0$, so (23) leads to a contradiction.

Under hierarchical scheduling, (23) is possible only when $Cpty([r_k, r_k + R_k^I]) > Cpty([r_k, r_k + R_k^{II}])$ where r_k is J_k's release time. However, since $R_{II,k} > R_{I,k}$, this violates the capacity-conserving property. This finishes the proof. ■

Appendix C

Proof of Lemma 3*:* Under work-conserving scheduling, if $J_k^{\max}$ does not start execution before $W_{\mathrm{HL}}^k(R_k^{\max})$ is *completely* finished, then before $J_k^{\max}$ starts execution, *all* DS with available capacity are *busy* executing $W_{\mathrm{HL}}^k(R_k^{\max})$. In other words, before $W_{\mathrm{HL}}^k(R_k^{\max})$ is finished, each DS's capacity is used to process $W_{\mathrm{HL}}^k(R_k^{\max})$.

Starting from the beginning of the interval B_k^{C}, in each complete replenishment period, a total capacity of $\sum_{i=1}^m C_{\mathrm{S}}^i$ units is used to processed $W_{\mathrm{HL}}^k(R_k^{\max})$ and it requires at most $(\lceil \frac{W_{\mathrm{HL}}^k(R_k^{\max})}{\sum_{i=1}^m C_{\mathrm{S}}^i} \rceil - 1)$ *complete* replenishment periods.

In the last replenishment period during which all the workload $W_{\mathrm{HL}}^k(R_k^{\max})$ is finished, the residual workload $W_{\mathrm{res}}^{\mathrm{HL}}$ to be processed in this period is given by (9).

We now calculate the time, denoted by $t_{\mathrm{res}}^{\mathrm{HL}}$, to process the residual workload $W_{\mathrm{res}}^{\mathrm{HL}}$. Since some DS may exhaust their capacity before the residual workload is finished, $t_{\mathrm{res}}^{\mathrm{HL}}$ cannot be calculated simply by $W_{\mathrm{res}}^{\mathrm{HL}} / \sum_{i=1}^m C_{\mathrm{S}}^i$. To calculate $t_{\mathrm{res}}^{\mathrm{HL}}$, we first define the function $\delta(i)$, $1 \leq i \leq m$ in (10), which calculates the total cumulative capacity between the beginning of a replenishment period and the earliest instant when the i-th DS exhausts its capacity in the same replenishment period, which is C_{S}^i time units after the beginning of this period. More details are discussed in [9].

With some mathematical manipulations, $t_{\mathrm{res}}^{\mathrm{HL}}$ is given by (8) whose details are discussed in [9].

In (8), given a value of i, $1 \leq i \leq m$, $\delta(i)$ has a unique value, so given a value of $W_{\mathrm{res}}^{\mathrm{HL}}$, the value of (8) can be uniquely determined. To sum up, $R_{\mathrm{HL}/k}$ is given by (7). ■

APPENDIX D

Proof of Lemma 4: In the last replenishment period during which the workload $W_{\mathrm{HL}}^{k}(R_k^{\max})$ is finished, the amount of the residual workload, denoted by $W_{\mathrm{res}}^{\mathrm{HL}}$, in this period is given by (9). After the workload $W_{\mathrm{res}}^{\mathrm{HL}}$ is finished, the remaining capacity in the same replenishment period is $\sum_{i=1}^{m} C_{\mathrm{S}}^{i} - W_{\mathrm{res}}^{\mathrm{HL}}$. Since $J_k^{\max}$ cannot execute on more than one processor *at the same time*, the remaining capacity that can be used by $J_k^{\max}$ is given by $C_{\mathrm{S}}^{\mathrm{rmn},k}$ in (13), where $T_{\mathrm{S}} - t_{\mathrm{res}}^{\mathrm{HL}}$ is the length of the interval between when $J_k^{\max}$ starts execution and the first replenishment after $J_k^{\max}$ starts execution.

If $C_{\mathrm{S}}^{\mathrm{rmn},k} \geq C_k$, $J_k^{\max}$ will be finished before the next replenishment period, then the time to process $J_k^{\max}$ *after the workload of* $W_{\mathrm{HL}}^{k}(R_k^{\max})$ *units is finished* is C_k.

If $C_{\mathrm{S}}^{\mathrm{rmn},k} < C_k$, $J_k^{\max}$ will not finish before the next replenishment. In each complete replenishment period after $J_k^{\max}$ starts execution, the capacity that can be used to $J_k^{\max}$ is $\min(\sum_{i=1}^{m} C_{\mathrm{S}}^{i}, T_{\mathrm{S}})$. The time to process $J_k^{\max}$ consists of three components: I) the length of the interval between when $J_k^{\max}$ starts execution and the first replenishment after $J_k^{\max}$ starts execution, which is $T_{\mathrm{S}} - t_{\mathrm{res}}^{\mathrm{HL}}$, II) the number of the *complete* replenishment periods, CRP_k in (12), and III) the time to process $J_k^{\max}$'s residual workload in the last replenishment period wherein $J_k^{\max}$ is finished, which is $C_k - CRP_k \cdot \min(\sum_{i=1}^{m} C_{\mathrm{S}}^{i}, T_{\mathrm{S}})$. The time to process $J_k^{\max}$ *after the workload of* $W_{\mathrm{HL}}^{k}(R_k^{\max})$ *units is finished* is then the sum of all these three components. This finishes the proof. ■

REFERENCES

[1] Z. Deng, J. Liu, and J. Sun, "A scheme for scheduling hard real-time applications in open system environment," in *Proceedings Ninth Euromicro Workshop on Real Time Systems*, 1997, pp. 191–199.

[2] R. I. Davis and A. Burns, "Hierarchical fixed priority pre-emptive scheduling," in *RTSS*, 2005, pp. 389–398.

[3] P. Cuijpers and R. Bril, "Towards Budgeting in Real-Time Calculus: Deferrable Servers," in *LNCS*. Springer, 2007, vol. 4763, p. 98.

[4] J. M. Calandrino, J. H. Anderson, and D. P. Baumberger, "A hybrid Real-Time scheduling approach for Large-Scale multicore platforms," in *ECRTS*, 2007, pp. 247–258.

[5] I. Shin, A. Easwaran, and I. Lee, "Hierarchical scheduling framework for virtual clustering of multiprocessors," in *ECRTS*, 2008, pp. 181–190.

[6] G. Lipari and E. Bini, "A framework for hierarchical scheduling on multiprocessors: from application requirements to run-time allocation," in *RTSS*, 2010, pp. 249–258.

[7] J. Y. T. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic, real-time tasks," *Performance Evaluation*, vol. 2, no. 4, pp. 237–250, Dec. 1982.

[8] G. Bernat and A. Burns, "New results on fixed priority aperiodic servers," in *RTSS*, 1999, p. 68.

[9] H. Zhu, S. Goddard, and M. B. Dwyer, "Response time analysis of hierarchical scheduling: the synchronized deferrable servers approach," University of Nebraska-Lincoln, Tech. Rep., 2011. [Online]. Available: http://ponca.unl.edu/facdb/csefacdb/TechReportArchive/TR-UNL-CSE-2011-0006.pdf

[10] J. K. Strosnider, J. P. Lehoczky, and L. Sha, "The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments," *IEEE Trans. Comput.*, vol. 44, no. 1, pp. 73–91, 1995.

[11] http://www.cs.unc.edu/~anderson/litmus-rt/.

[12] http://www.runtime-verification.org.

[13] J. Ha, M. Arnold, S. M. Blackburn, and K. S. McKinley, "A concurrent dynamic analysis framework for multicore hardware," in *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2009, pp. 155–174.

[14] H. Zhu, M. B. Dwyer, and S. Goddard, "Predictable runtime monitoring," in *ECRTS*, 2009, pp. 173–183.

[15] M. Bertogna and M. Cirinei, "Response-time analysis for globally scheduled symmetric multiprocessor platforms," in *RTSS*, 2007, pp. 149–160.

[16] N. Guan, M. Stigge, W. Yi, and G. Yu, "New response time bounds for fixed priority multiprocessor scheduling," in *RTSS*, 2009, pp. 387 –397.

[17] S. Baruah, J. Goossens, and G. Lipari, "Implementing constant-bandwidth servers upon multiprocessor platforms," in *RTAS*, 2002, pp. 154–163.

[18] S. Baruah and G. Lipari, "A multiprocessor implementation of the total bandwidth server," in *IPDPS*, 2004, pp. 40–49.

[19] B. Andersson and E. Tovar, "Multiprocessor scheduling with few preemptions," in *RTCSA*, 2006, pp. 322–334.

[20] B. Andersson and K. Bletsas, "Sporadic multiprocessor scheduling with few preemptions," in *ECRTS*, 2008, pp. 243–252.

[21] S. Kato and N. Yamasaki, "Portioned Static-Priority Scheduling on Multiprocessors," in *IPDPS*, 2008.

[22] ——, "Semi-Partitioned Fixed-Priority Scheduling on Multiprocessors," in *RTAS*, 2009, pp. 23–32.

[23] K. Lakshmanan, R. R. Rajkumar, and J. P. Lehoczky, "Partitioned fixed-priority preemptive scheduling for multi-core processors," in *ECRTS*, 2009, pp. 239 – 248.

[24] N. Guan, M. Stigge, W. Yi, and G. Yu, "Fixed-Priority multiprocessor scheduling with Liu and Layland's utilization bound," in *RTAS*, 2010, pp. 165–174.

[25] A. Srinivasan and S. Baruah, "Deadline-based scheduling of periodic task systems on multiprocessors," *Information Processing Letters*, vol. 84, no. 2, pp. 93–98, 2002.

[26] J. Goossens, S. Funk, and S. Baruah, "Priority-driven scheduling of periodic task systems on multiprocessors," *Real-Time Systems*, vol. 25, no. 2, pp. 187–205, 2003.

[27] T. Baker, "An analysis of EDF schedulability on a multiprocessor," *IEEE Transactions on Parallel and Distributed Systems*, pp. 760–768, 2005.

[28] T. Baker and S. Baruah, "Schedulability analysis of multiprocessor sporadic task systems," in *Handbook of Realtime and Embedded Systems*, 2007.

[29] R. Davis and A. Burns, "Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems," in *RTSS*, 2009, pp. 398–409.

[30] M. Joseph and P. Pandya, "Finding response times in a real-time system," *The Computer Journal*, vol. 29, no. 5, p. 390, 1986.

[31] J. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: exact characterization and average case behavior," in *RTSS*, 1989, pp. 166–171.

[32] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Engineering Journal*, vol. 8, no. 5, pp. 284–292, 1993.

[33] S. Saewong, R. Rajkumar, J. Lehoczky, and M. Klein, "Analysis of hierarchical fixed-priority scheduling," in *ECRTS*, 2002, pp. 152–160.

[34] T. Baker, "Multiprocessor EDF and deadline monotonic schedulability analysis," in *RTSS*, 2003, pp. 120–129.

[35] S. Baruah, "Techniques for multiprocessor global schedulability analysis," in *RTSS*, 2007, pp. 119–128.

[36] S. Dhall and C. Liu, "On a real-time scheduling problem," *Operations Research*, pp. 127–140, 1978.