CS-TR-2012-003, UTSA

Available online on Feb. 25, 2012; revised on March 14, 2012

Synchronization-Aware Energy Management for VFI-based Multicore Real-Time Systems (Extended Version)

Jian-Jun Han, Xiaodong Wu, Dakai Zhu, Hai Jin, Laurence T. Yang and Jean-Luc Gaudiot

Abstract—Multicore processors have emerged to be the popular and powerful computing engines to address the increasing performance demands and energy efficiency requirements of modern real-time applications. *Voltage and frequency island (VFI)* was recently adopted as an effective energy management technique for multicore processors. For a set of periodic real-time tasks that access shared resources running on a VFI-based multicore system with *dynamic voltage and frequency scaling (DVFS)* capability, we study both static and dynamic *synchronization-aware* energy management schemes. First, based on the enhanced MSRP resource access protocol with a suspension mechanism, we devise a synchronization-aware task mapping heuristic for partitioned-EDF scheduling. The heuristic assigns tasks that access similar set of resources to the same core to reduce the synchronization overhead and thus improve schedulability. Based on the result task-to-core mapping, static energy management schemes with both a uniform and different scaled frequencies for VFIs are studied. To further exploit dynamic slack for more energy savings, we propose an integrated synchronization-aware slack management framework to appropriately reclaim, preserve, release and steal slack at runtime to slow down the execution of tasks subject to the common voltage/frequency limitation of VFIs and timing/synchronization constraints of tasks. Taking the additional delay due to task synchronization into consideration, the new scheme allocates slack in a fair manner and scales down the execution of both non-critical and critical sections of tasks for more energy savings. Simulation results show that, the synchronization-aware task mapping scheme can significantly improve the schedulability of tasks. The energy savings obtained by the static scheme with different frequencies for VFIs is close to that of an optimal INLP (integer non-linear programming) solution. Moreover, compared to the simple extension of existing solutions for uniprocessor systems, our dynamic scheme can obtain much better energy savings (up to 40%) with comparable DVFS overhead.

Index Terms—Real-time systems; Multicore; Shared resources; Voltage frequency island (VFI); Energy management; DVFS;

1 Introduction

THe performance of modern computing systems has been significantly improved in last few decades with increasing processing frequency and level of integration. However, such performance improvements came at the cost of drastically increased power consumption, which has promoted energy to be a first-class system resource and energy management to be an important research area. In the past decade, many hardware and software power management techniques have been proposed for various computing systems from battery-powered embedded computing devices that have limited energy budget to high performance servers that are connected directly to the power grid [6], [15], [41], [43], [49], [57].

The basic principle for saving energy in computing systems is to operate system components at low-performance (and thus low-power) states, whenever possible [6], [42]. For instance, as a widely-deployed power management technique, *dynamic voltage and frequency scaling (DVFS)* reduces simultaneously the supply voltage and processing frequency of processors to save energy when performance demand is low [50]. However, considering its importance and the variety of computing systems, effective energy management remains to be one of the grand challenges for both research and engineering communities [30].

The emergence of multicore architecture, which integrates multiple processing cores on a single chip [40], has quickly led us into a new multicore computing era. The central idea is to exploit the parallelism in applications for higher performance and better energy efficiency. For instance, an application can be executed in parallel on multiple cores where each core runs at a lower frequency to achieve a given performance and save energy. Considering its great features of high-performance and energy efficiency, multicore has been adopted by major chip manufacturers and several lines of multicore processors have been developed (e.g., Intel Core2 Quad [31] and AMD Phenom [23]). Note that, most state-of-the-art commercial multicore processors have a common power supply voltage

- *J.-J. Han, X. Wu, H. Jin and L. T. Yang are with School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China. E-mail:jasonhan@mail.hust.edu.cn, xiaodongwu@smail.hust.edu.cn, hjin@hust.edu.cn.*
- *D. Zhu is with the Computer Science Department, University of Texas at San Antonio, San Antonio, TX 78249. E-mail: dzhu@cs.utsa.edu.*
- *J.-L. Gaudiot is with the Department of Electrical Engineering and Computer Science, University of California, Irvine, CA 92697-2625. Email:gaudiot@uci.edu.*

for all cores on the chip [23], [31], which requires the cores to run at the same frequency with limited power management flexibility (and thus results in sub-optimal energy savings).

To support more flexible power management for multicore systems, *voltage and frequency island (VFI)* technique has been studied recently, where cores are partitioned into groups and each group of cores on an island share one supply voltage and have the same processing frequency [20], [32], [39]. Several recent studies have investigated efficient VFI configurations and showed that having an independent VFI for each core can provide the most flexible support for managing energy [22], [42]. However, as the number of cores on a single chip continues to increase (where extensive research activity is underway to build chips with tens and even hundreds of cores [11], [29]), the potential energy gains of such a per-core VFI option are likely to be too modest to justify the increased design complexity and the associated area overhead of the required supply voltages and on-chip voltage regulators [28], [33]. With the development of fast on-chip voltage regulators, VFI has been adopted in some modern multicore processors, such as Intel Itanium i7 [2] and IBM Power 7 series [1].

Real-time systems have been deployed in a wide range of applications (such as vehicle control systems and multimedia processing), which generally have various timing constraints. Considering the increasing performance demands and parallel nature of computation tasks (e.g., processing multi-channel video/audio signals), it is expected that multicore processors will become the computing engines in real-time embedded systems as well. For instance, several multicore processors have been developed for automotive electrical control units (ECUs) [4], [48]. Although many energy management schemes have been studied for real-time systems based on the popular DVFS technique [6], [7], [15], [41], [43], [49], [57], the common supply voltage and frequency limitation for the cores on a multicore chip (or VFI) brings additional complications and much less work has focused on energy management for multicore-based real-time embedded systems [21], [51].

In real-time systems, tasks may need to exclusively access re-usable shared resources (such as global variables or I/O channels). Such resource contention can lead to *priority inversion*, where a lower-priority task accesses a shared resource non-preemptively and blocks the execution of a higher-priority task [44]. The additional blocking time due to priority inversion can significantly affect the schedulability of tasks and result in very low system utilization. To tackle such task synchronization problems, several lock-based resource access protocols have been studied for both uniprocessor (e.g., PCP [44], [47] and SRP [8]) and multiprocessor (e.g., MSRP [26] and FMLP [10]) real-time systems to guarantee all tasks meet their deadlines when accessing shared resources.

Based on these resource access protocols, a few studies have investigated energy management schemes for real-time tasks under synchronization constraints [17], [35], [54]. However, these studies have focused exclusively on uniprocessor systems and, to the best of our knowledge, there is no existing work on the problem for multiprocessor systems. Note that, for the execution of real-time tasks governed by the resource access protocols in multiprocessor systems, the schedulability analysis generally relies on the worst-case waiting time due to synchronization requirements [8], [26]. As such analysis can be very pessimistic, there exist great opportunities to slow down the execution of tasks and thus save energy. Moreover, considering the increasing popularity of multicore-based realtime embedded systems, it has become a necessity to develop effective energy management schemes for tasks that access shared resources in such systems.

As the first work to address this problem, we consider VFI-based multicore real-time systems and study both static and dynamic synchronization-aware energy management schemes for a set of periodic tasks that access shared resources. Specifically, by extending the MSRP [26] and OMLP [12] resource access protocols and focusing on partitioned-EDF scheduling, we study a *synchronization-aware* task mapping heuristic. It assigns tasks that access similar set of resources to the same core to reduce the blocking and waiting time among tasks, which can significantly improve tasks' schedulability. Then, we study static schemes that exploit such opportunities and assign scaled frequencies for VFIs to save energy.

Moreover, to exploit the vast amount of dynamic slack at runtime due to workload variations of real-time tasks as well as less than the worst-case synchronization overhead, we propose an integrated synchronization-aware slack management framework to appropriately reclaim, preserve, release and steal slack to further slow down the execution of tasks while taking the common voltage/frequency limitation of VFI-based multicore systems into consideration. In addition to non-critical sections of tasks, different from most existing techniques, we consider slowing down the execution of tasks' critical sections as well. The scheme ensures tasks' time constraints by incorporating the additional delay due to scaled execution into the analysis. Finally, the proposed schemes are evaluated through extensive simulations.

The remainder of this paper is organized as follows. Section 2 reviews closely-related research on task synchronization and energy management in real-time systems. In Section 3, we present the system models and basic notations before discussing the resource access protocol adopted in this work. Synchronization-aware task mapping and static energy management schemes are presented in Section 4. In Section 5, the integrated slack management framework and the dynamic synchronization-aware energy management scheme are addressed. Simulation results are presented and discussed in Section 6 and Section 7 concludes the paper.

2 Related Work

Since the seminal work of Weiser *et al.* on reducing energy consumption in processors [50], based on the DVFS technique, many energy management schemes have been studied considering different systems, scheduling policies and tasks with various timing constraints. Although the vast majority of energy management studies in real-time systems have focused on uniprocessor systems (e.g., [6], [15], [34], [36], [41], [43], [45]), significant research effort has also been made for energy management in multiprocessor real-time systems [7], [18], [51], [56]. However, considering the complications of

common voltage/frequency limitation in multicore processors, the research on energy management for multicore real-time systems is rather limited [9], [21], [46].

Based on the *partitioned scheduling* policy, Aydin and Yang studied the problem of how to partition real-time tasks to processors for minimizing energy consumption for multiprocessor systems [7]. They showed that, for *earliest deadline first (EDF)* scheduling, balancing the workload among all processors evenly gives the optimal energy consumption. However, the general partition problem for minimizing energy consumption in multiprocessor real-time system is NP-hard [3], [7]. For frame-based and periodic real-time tasks, Chen *et al.* proposed a series of approximation algorithms for maximizing energy-efficiency of multiprocessor systems, with and without leakage power being considered [14], [16].

In [27], Goh *et al.* introduced a gradient-based real-time scheduling of dependent tasks for heterogeneous embedded multiprocessor systems, where ILP was adopted to improve energy efficiency. For systems with large number of processors, Chu *et al.* developed a nonlinear ILP-based mapping policy with pruning heuristics to map tasks to processors for energy savings in heterogeneous multiprocessor systems [19]. In [52], for real-time streaming applications on multicore systems, Xu *et al.* studied a hill climbing based scheme for energy savings while exploiting pipelining to satisfy the throughput requirements. More recently, Choi and Melhem studied the interplay between parallelism of an application, program performance and energy consumption [18]. For an application with given ratio of serial and parallel portions and the number of processors, the authors derived optimal frequencies allocated to the serial and parallel regions in an application to either minimize the total energy consumption or minimize the energy-delay product [18].

Considering workload variations in real-time tasks and the dynamic slack generated at runtime, Zhu *et al.* studied *global scheduling* based energy management scheme for real-time multiprocessor systems, where the *slack sharing* technique is used for load balancing and thus better energy savings [56]. For soft real-time applications running on multicore systems, Bautista *et al.* recently studied a novel fairness-based power aware scheduler that utilizes the global frequency for all cores at the same time and evaluated the power efficiency of multicore processors [9]. The scheduler pursues to minimize the number of DVFS transitions by increasing or decreasing the voltage and frequency of all the cores simultaneously. Along the same line of assuming all cores on a chip share the same frequency, Seo *et al.* studied one dynamic re-partitioning algorithm for real-time systems which balances the task loads on the cores at runtime to optimize overall power consumption [46].

More recently, Devadas and Aydin proposed an on-line scheduling scheme which tries to put the idle processing cores into sleep state for the reduction in energy consumption [21]. Considering the under-utilized systems and assuming that a task can be executed in parallel, Lee *et al.* introduced an energy-saving scheduling scheme that turns off rarely used cores [38]. In [42], Qi and Zhu studied energy efficiency for different partitioning of cores to VFIs and proposed an asymmetric buddy VFI configuration that can effectively support various workload with great energy savings.

For real-time tasks that access non-preemptable shared resources, several resource access protocols have been studied for uniprocessor real-time systems, such as Priority Ceiling Protocol (PCP) [47] and Stack Resource Policy (SRP) [8]. The protocols have been extended for multiprocessor real-time systems, such as MPCP (Multiprocessor PCP) [37] and MSRP (Multiprocessor SRP) [26], to tackle the blocking caused by tasks on different processors accessing the same global shared resources. In [24], Easwaran *et al.* studied a parallel PCP (P-PCP) resource-sharing protocol for fixed-priority tasks in multiprocessor systems and developed the corresponding schedulability conditions. More recently, Block *et al.* introduced a Flexible Multiprocessor Locking Protocol (FMLP) [10] and Brandenburg *et al.* studied suspension-based optimal locking protocols (namely, OMLP) [12]. Both FMLP and OMLP are applicable to global and partitioned scheduling.

Based on the resource access protocols, a few studies have focused on energy management with task synchronization for uniprocessor systems. Jejurikar and Gupta studied a static *Uniform Slowdown with Frequency Inheritance (USFI)* scheme, where a job inherits the maximum frequency of its blocked jobs when it executes a critical section [35]. With intuition that executing low priority tasks at a higher speed can be more effective to reduce the blocking time between tasks, Chen *et al.* studied two static frequency assignment algorithms for uniprocessor systems with task synchronization to reduce voltage/frequency switching overhead caused by DVFS and to obtain better energy savings [17]. However, both studies did not consider dynamic energy management.

In [54], Zhang and Chanson proposed a *Dual-Speed (DS)* algorithm for uniprocessor systems, where two (i.e., high and low) speeds are calculated off-line for each task to satisfy the feasibility condition under the worst-case scenarios. At runtime, when no job is blocked, the low speed with slack reclamation is applied for energy savings. However, when blocking occurs, the high speed is needed to guarantee deadlines of blocked tasks.

Different from all existing work that has focused on uniprocessor systems, in this paper, we focus on the energy management problem for VFI-based multicore real-time systems with shared resources. We study both static and dynamic synchronization-aware schemes for periodic tasks, which exploit spare system utilization and dynamic slack, respectively, for energy savings.

3 Preliminaries

In this section, we first layout the scope of this work by presenting the system, task and resource models and stating our assumptions. Moreover, the resource access protocol, which extends MSRP [26] by incorporating the idea of suspension (from OMLP [12]), and the corresponding schedulability conditions for partitioned-EDF scheduling are discussed.

3.1 System and Power Models

We consider real-time systems with a multicore processor, which consists of nc processing cores $\{\mathcal{O}_1, \ldots, \mathcal{O}_{nc}\}$ that

have identical functions and capabilities. The multicore processor is assumed to adopt the voltage and frequency island (VFI) technique and the cores are organized into ng VFIs, where each VFI consists of a sub-group of cores, denoted as $\{G_1, \ldots, G_{ng}\}$. For simplicity, we assume that the VFIs are *homogeneous* [1] and each of them contains the same number of cores. That is, nc is assumed to be a multiple of ng and, for the ith VFI, there is $|G_i| = nc/ng$ $(i = 1, \cdots, ng)$.

Each VFI is assumed to have a DVFS-enabled voltage supply that can provide nd discrete supply voltages v_i and corresponding highest processing frequencies f_i $(1 \leq i \leq nd)$. Cores on the same VFI share the common DVFS-enabled voltage supply and have the same processing frequency. However, considering that cores can be easily put into power saving states [1], [2], each core is assumed to have two states: *active* and *sleep*. When a core does not have workload to process, it becomes inactive and can be put to sleep state for energy savings. When all cores on one VFI are in sleep state, we can power off the VFI to save more energy.

For the widely used CMOS technology, dynamic power dissipation is generally a dominant component, which is quadratically related to supply voltage and linearly related to processing frequency [13]. However, with scaled feature size and increased level of integration, leakage power has been ever-increasing and it has become a necessity for effective energy management to consider both dynamic and leakage power components [5], [15], [36]. In this work, we adopt the power model [2] utilized in [46], where the power consumption of the system with a multicore processor is given by:

$$P_{sys} = \sum_{i=1}^{nc} \hbar_i \cdot P_d(f(\mathcal{O}_i)) + P_l(f(\mathcal{O}_i)) \tag{1}$$

Here $\hbar_i$ indicates the state of core $\mathcal{O}_i$. When $\mathcal{O}_i$ is active, $\hbar_i = 1$; otherwise $\hbar_i = 0$. When core $\mathcal{O}_i$ is active and operates at frequency $f(\mathcal{O}_i)$ (and corresponding voltage $v(\mathcal{O}_i)$), its dynamic power consumption can be given as [13]:

$$P_d(f(\mathcal{O}_i)) = C_{ef} \cdot v(\mathcal{O}_i)^2 \cdot f(\mathcal{O}_i) \tag{2}$$

The leakage power $P_l(f(\mathcal{O}_i))$, which exists even when a core is put to sleep, can be given as [46]:

$$P_l(f(\mathcal{O}_i)) = L_g \cdot (v(\mathcal{O}_i) \cdot I_{sub} + |V_{bs}| \cdot I_k) \tag{3}$$

$$I_{sub} = K_3 \cdot e^{K_4 \cdot v(\mathcal{O}_i)} \cdot e^{K_5 \cdot V_{bs}} \tag{4}$$

where I_{sub} and I_k are leakage currents and V_{bs}, L_g, K_3, K_4 and K_5 are system-dependent constants [46].

Note that energy is the integral of power over time. From the above equations, we can see that, although reducing processing frequency can lower the energy consumption due to dynamic power, the energy consumption due to leakage power will increase. Therefore, there exists a critical frequency f_{crit}, below which more energy may be consumed [5], [15], [36], [42]. To simplify our discussion, we assume that the minimum available frequency $f_1 \geq f_{crit}$. The maximum frequency is assumed to be $f_{max} = f_{nd}$, which is normalized to 1 as we consider normalized frequencies. Moreover, we assume that the overhead caused by voltage and frequency changes is negligible (or such overhead can be incorporated into slack reclamation or the worst-case execution time of tasks [56]).

Although Equation 1 represents a much simplified power model, we point out that the synchronization-aware power management schemes studied in this work do not depend on the specific power model. As long as the voltage/frequency levels and the corresponding power consumption form a convex relation, the schemes studied in this paper can be applied.

1. Although it is possible to have heterogeneous VFIs that contain different number of cores [42], such options can increase processor design complexity.

2. Note that, similar power models have been adopted in other recent work on power management [5], [15], [36].

3.2 Task and Resource Models

We consider a set of nt periodic real-time tasks $\Gamma = \{T_1, \ldots, T_{nt}\}$, where each task T_i has a period p_i that is also its relative deadline. The jth job (or task instance) of task T_i is denoted by $J_{i,j}$, which arrives at time $(j-1) \cdot p_i$ and has to complete by its absolute deadline $j \cdot p_i$. Unless specified otherwise, the terms task and job are used exchangeably in the remainder of this paper.

The system has a set of nr global resources $SR = \{\mathcal{R}_1, \ldots, \mathcal{R}_{nr}\}$, which are shared by all tasks. The resources are non-preemptable but serially re-usable. To ensure exclusive access of shared resources, a task can only access a shared resource in its critical sections. For task T_i, there are $N_{cs}(T_i)$ number of critical sections and its jth critical section is denoted by $z_{i,j}$, during which the task will access resource $s(z_{i,j}) \in SR$. The worst-case execution time (WCET) for the jth critical section of task T_i at the maximum processing frequency is $c_{cs}(z_{i,j})$.

In this work, we assume that there are no nested critical sections since they occur rather infrequently in practice and can be dealt with by group locks [10]. That is, a task can only access one resource at any time. Suppose that the WCET of all non-critical sections of task T_i is $c_{ns}(T_i)$, the WCET of task T_i at the maximum frequency f_{max} can be given as $c_i = c_{ns}(T_i) + \sum_{j=1}^{N_{cs}(T_i)} c_{cs}(z_{i,j})$.

When a task is executed at a lower frequency, its execution time is assumed to scale linearly, which has been a widely adopted assumption in other recent work [15], [46]. Hence, if task T_i runs at scaled frequency $f(T_i)$ $(f_1 \leq f(T_i) < f_{max})$, its execution time will be $\frac{c_i}{f(T_i)}$.

The subset of global resources that are accessed by task T_i is denoted as SR_i. Here, a task may access a global resource multiple times. Therefore, the number of distinct global resources accessed by task T_i is no more than the number of its critical sections (i.e., $|SR_i| \leq N_{cs}(T_i)$).

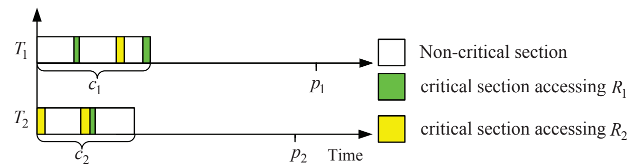


Fig. 1: An example of tasks and resource access patterns.

For example, Figure 1 shows a task system with two tasks (i.e., T_1 and T_2) and two shared global resources (i.e., $\mathcal{R}_1$ and

$\mathcal{R}_2$). Here, the first task T_1 has three critical sections, during which it accesses the resource $\mathcal{R}_1$ twice and the resource $\mathcal{R}_2$ once. Similarly, for task T_2, it accesses the resources $\mathcal{R}_1$ and $\mathcal{R}_2$ once and twice, respectively. We can have $SR_1 = SR_2 = \{\mathcal{R}_1, \mathcal{R}_2\}$.

3.3 Resource Access Protocols and Schedulability

Due to synchronization requirements, global resources have to be accessed exclusively in task's critical sections that are not preemptable. Therefore, the execution of a task, regardless of its priority, can be blocked when it attempts to access a global resource that is held (and accessed) by another task. Hence, the exact sequence of tasks accessing the resources depends on not only the scheduling algorithm but also the resource access protocol.

In this work, based on partitioned-EDF scheduling, we adopt a suspension-based resource access protocol that extends MSRP [26] and OMLP [12]. Specifically, when a task attempts to access a global resource that is held by another task running on a different core, instead of allowing the task to spin-lock with busy-waiting as in the original MSRP [26], the extended protocol suspends the task's execution and lets the core run other tasks to improve system performance [12]. However, to limit resource access requests issued by tasks from a core, there exists a *resource access contention token* guarded by a binary semaphore on each core. Only after a task obtained the token of its core, can it issue a resource access request.

With the suspension mechanism and resource access contention tokens on the processing cores, the basic steps and rules for the extended resource access protocol can be summarized as follows [12], [26]:

- **Rule 1:** A task T_i needs to obtain the contention token on its processing core before issuing a resource access request. If it fails (because of another task on the same core holding the token), the task is put into the prioritized waiting queue of the core's contention token;
- **Rule 2:** Once a task T_i obtained the contention token of its core, task T_i can issue a request to access its resource $\mathcal{R}_a$. If $\mathcal{R}_a$ is free, task T_i will lock and access it by executing the critical section non-preemptively; Otherwise, if $\mathcal{R}_a$ is currently held by another task (on a different processing core), task T_i will be added to the FIFO queue of resource $\mathcal{R}_a$ and then is suspended;
- **Rule 3:** Once a task T_i is suspended on core $\mathcal{O}_k$ due to waiting for a resource $\mathcal{R}_a$ held by a task on another core, $\mathcal{O}_k$ will *repeatedly* execute the next highest priority task T_j non-preemptively until either a) T_i is ready to access resource $\mathcal{R}_a$ (i.e., $\mathcal{R}_a$ is released and T_i is the header task in $\mathcal{R}_a$'s FIFO queue); or b) T_j completes its execution; or c) T_j attempts to execute one of its critical sections (but failed since T_i holds $\mathcal{O}_k$'s contention token);
- **Rule 4:** Once a task T_i finishes accessing a resource $\mathcal{R}_a$ by completing a critical section, the task will release both the contention token (such that other tasks on the same core can have the opportunity to access resources) and resource $\mathcal{R}_a$. If the FIFO queue of $\mathcal{R}_a$ is not empty (i.e., there are tasks from other cores waiting for accessing $\mathcal{R}_a$), the header task is de-queued and starts accessing resource $\mathcal{R}_a$; otherwise, resource $\mathcal{R}_a$ is unlocked; After that, task T_i becomes preemptable again.

From the protocol, we can see that the execution of a task T_i on core $\mathcal{O}_k$ can be blocked at two different occasions: First, when T_i tries to access a resource $\mathcal{R}_a$ that is currently held and accessed by a task on another core, it has to wait in $\mathcal{R}_a$'s FIFO queue and the duration is denoted as *global waiting time*. Second, when a low priority task (which has a later deadline than T_i) on core $\mathcal{O}_k$ is in its critical section and accessing a resource or is waiting for a resource (that is currently held and accessed by a task on another core), T_i can be blocked and the duration is denoted as *local blocking time*.

Hence, for the above resource access protocol, we can have the following properties that are inherited directly from the MSRP and OMLP protocols [12], [26]:

Property 1. *For local blocking time, a task can be blocked at most once by a low priority task on the same core [26].*

Property 2. *The local blocking time for a task is upper-bounded by the longest duration for executing a low priority task's critical section (including the low priority task's global waiting time, if any) on the same core [26].*

Property 3. *For any core, at any given time, there exists at most one task that is either a) accessing a resource; or b) suspended and waiting for a resource (which is currently accessed by a task on another core) [12].*

Schedulability Condition: Note that, due to task synchronization, the execution of a task takes longer because of local blocking time and global waiting time. For a given partitioning of tasks to cores, we discuss next the schedulability condition for EDF on each core under the above resource access protocol. First, for the ease of presentation and discussion, we define some necessary notations as follows:

- Ψ_k: represents the subset of tasks that are allocated to processing core $\mathcal{O}_k$;
- $\mathcal{O}(T_i)$: denotes the core to which task T_i is allocated;
- $tt_i^{max}(\mathcal{R}_a)$: is the maximum amount of time for task T_i to access resource $\mathcal{R}_a$ once. It equals to zero if task T_i does not access $\mathcal{R}_a$ (i.e., $\mathcal{R}_a \notin SR_i$); otherwise, we have:

$$tt_i^{max}(\mathcal{R}_a) = \max\{c_{cs}(z_{i,x}) | \forall z_{i,x} : s(z_{i,x}) = \mathcal{R}_a\} \quad (5)$$

- $tp_k^{max}(\mathcal{R}_a)$: refers to the maximum amount of time for any task on core $\mathcal{O}_k$ to access resource $\mathcal{R}_a$ once;
- $BW(z_{i,j})$: indicates the maximum amount of time that task T_i waits for executing its critical section $z_{i,j}$;
- BW_i: denotes the worst-case global waiting time that can be experienced by task T_i to access all its resources,

$$BW_i = \sum_{j=1}^{N_{cs}(T_i)} BW(z_{i,j}) \quad (6)$$

- B_i: is the maximum local blocking time for task T_i.

Next, we calculate BW_i and B_i for task T_i on core $\mathcal{O}_k$ under the resource access protocol discussed above. Suppose that task T_i attempts to access resource $\mathcal{R}_a$ in its critical

section $z_{i,j}$ and is put into the FIFO queue of $\mathcal{R}_a$ due to task synchronization. Recall that nc is the number of cores in the system. From Property 3, at most $(nc-1)$ requests to access resource $\mathcal{R}_a$, which have been issued by tasks on cores other than $\mathcal{O}_k$, can precede task T_i's request.

For the request issued by a task on core $\mathcal{O}_m$ ($m \neq k$), the worst-case amount of time to access resource $\mathcal{R}_a$ once is:

$$tp_m^{max}(\mathcal{R}_a) = \max\{tt_x^{max}(\mathcal{R}_a) | \forall T_x \in \Psi_m\} \quad (7)$$

Therefore, the maximum global waiting time for task T_i on core $\mathcal{O}_k$ when executing its critical section $z_{i,j}$ will be:

$$BW(z_{i,j}) = \sum_{\forall \mathcal{O}_m, m \neq k} tp_m^{max}(s(z_{i,j})) \quad (8)$$

Substituting $BW(z_{i,j})$ in (6) with (8), we can obtain BW_i, the worst-case global waiting time for task T_i to access all its resources. From Property 1, task T_i on core $\mathcal{O}_k$ can only be blocked at most once by a task T_x where $T_x \in \Psi_k$ and $p_x > p_i$. Therefore, according to Property 2, we have

$$B_i = \max\{BW(z_{x,y}) + c_{cs}(z_{x,y}) | \forall z_{x,y} : T_x \in \Psi_k, \\ p_x > p_i, 1 \leq y \leq N_{cs}(T_x)\} \quad (9)$$

Note that, with the suspension-based resource access protocol, tasks can utilize the processing cores more efficiently (especially, Rule 3 of the protocol). However, such an approach cannot reduce the worst-case blocking and waiting times as calculated above. Therefore, based on the feasibility results for MSRP [8], [26], we can get the following proposition.

Proposition 1. *For a given task-to-core mapping, the tasks are schedulable under EDF on their cores if, for every core $\mathcal{O}_k$ ($k = 1, \ldots, nc$), there is:*

$$\forall T_i \in \Psi_k, \frac{B_i}{p_i} + \sum_{\forall T_j \in \Psi_k, p_j \leq p_i} \frac{c_j + BW_j}{p_j} \leq 1 \quad (10)$$

4 Synchronization-Aware Task Mapping and Static Power Management Schemes

In [7], without considering task synchronization, Aydin and Yang showed that the maximum energy savings can be obtained with balanced workload among all processors. However, finding the optimal partitioning of tasks to processors for minimizing energy consumption in multiprocessor real-time systems is NP-hard. Note that, the problem of energy management with task synchronization in multicore real-time systems being considered in this paper is a generalization of what being considered in [7]. Thus, it is NP-hard as well.

In what follows, we first give the formalized problem description of task-to-core mapping for minimizing energy consumption, under the constraints of task synchronization and common voltage/frequency limitation of VFIs. Then, we focus on efficient task-to-core mapping heuristics for minimizing energy consumption while taking task synchronization into consideration. Specifically, we propose a *synchronization-aware* task mapping heuristic that tries to partition tasks that access the similar group of resources to the same processing core. From Equation 10, we can see that such mapping can improve tasks' schedulability with reduced blocking and waiting time due to task synchronization. With improved schedulability, the mapping also provides better opportunity to slow down the execution of tasks for more energy savings. Finally, based on the result mapping of tasks to cores, the static scheme considers two frequency assignment approaches: a *uniform* scaled frequency for all VFIs and *different* frequency for each VFI.

4.1 Formal Problem Description

Note that, it can be more energy efficient to assign more than one frequencies for VFIs to execute tasks in different intervals. However, exploring the full extent of this direction is well beyond the scope of this paper and will be addressed in our future work. In this paper, we focus on schemes that assign a single frequency for each VFI. Before presenting the problem description, some auxiliary notations are defined as follow:

- $\xi_{i,j}$: indicates whether task T_i is mapped onto core $\mathcal{O}_j$ or not;
- $\phi_{i,j}$: denotes whether core $\mathcal{O}_i$ is assigned the jth discrete processing frequency f_j or not;
- $G(\mathcal{O}_i)$: corresponds to the VFI to which core $\mathcal{O}_i$ belongs;
- $f(\mathcal{O}_i)$: refers to the processing frequency of core $\mathcal{O}_i$;
- $f(T_i)$: is the processing frequency of task T_i.

Note that, both task synchronization constraints and VFI common frequency limitation need to be considered. Thus, with the objective of finding the proper task-to-core mapping and the scaled frequencies of VFIs to minimize energy consumption for a set of periodic tasks running on a VFI-based multicore systems, the problem can be formalized as:

$$\text{Minimize} \quad P_{sys} \quad (11)$$

Subject to:

$$\xi_{i,j} \in \{0,1\} \ (i = 1, \cdots, nt, \ j = 1, \cdots, nc) \quad (12)$$

$$\sum_{j=1}^{nc} \xi_{i,j} = 1 \ (i = 1, \cdots, nt) \quad (13)$$

$$\phi_{i,j} \in \ \{0,1\} \ (i = 1, \cdots, nc, \ j = 1, \cdots, nd) \quad (14)$$

$$\sum_{j=1}^{nd} \phi_{i,j} = 1 \ (i = 1, \cdots, nc) \quad (15)$$

$$f(\mathcal{O}_i) = \sum_{j=1}^{nd} (\phi_{i,j} \cdot f_j) \ \ (i = 1, \cdots, nc) \quad (16)$$

$$f(\mathcal{O}_i) = f(\mathcal{O}_j), \text{ if } G(\mathcal{O}_i) = G(\mathcal{O}_j) \\ (i = 1, \cdots, nc, \ j = 1, \cdots, nc) \quad (17)$$

$$f(T_i) = \sum_{j=1}^{nc} (\xi_{i,j} \cdot f(\mathcal{O}_j)) \ \ (i = 1, \cdots, nt) \quad (18)$$

$$U_i = \max\left\{\frac{B_j}{p_j} + \sum_{\forall T_m:\xi_{m,i}=1}^{p_m \le p_j} \frac{\frac{c_m \cdot f_{max}}{f(T_m)} + BW_m}{p_m} | \forall T_j : \xi_{j,i} = 1\right\} \quad (i = 1, \cdots, nc) \tag{19}$$

$$U_i \le 1 \ (i = 1, \cdots, nc) \tag{20}$$

$$BW_i = \sum_{j=1}^{N_{cs}(T_i)} BW(z_{i,j}) \ (i = 1, \cdots, nt) \tag{21}$$

$$B_i = \max\{BW(z_{x,y}) + \frac{c_{cs}(z_{x,y}) \cdot f_{max}}{f(T_x)} | \forall z_{x,y} : \xi_{x,k} = 1 \wedge p_x > p_i \wedge 1 \le y \le N_{cs}(T_x)\}, \text{ if } \xi_{i,k} = 1 \quad (i = 1, \cdots, nt) \tag{22}$$

$$BW(z_{i,j}) = \sum_{\forall \mathcal{O}_m : m \ne k} tp_m^{max}(s(z_{i,j})), \text{ if } \xi_{i,k} = 1 \quad (i = 1, \cdots, nt, \ j = 1, \cdots, N_{cs}(T_i)) \tag{23}$$

$$tp_m^{max}(\mathcal{R}_a) = \max\{tt_x^{max}(\mathcal{R}_a) | \forall T_x : \xi_{x,m} = 1\} \quad (m = 1, \cdots, nc, \ \forall \mathcal{R}_a \in SR) \tag{24}$$

$$tt_i^{max}(\mathcal{R}_a) = \frac{\max\{c_{cs}(z_{i,x}) | \forall z_{i,x} : s(z_{i,x}) = \mathcal{R}_a\} \cdot f_{max}}{f(T_i)} \quad (i = 1, \cdots, nt, \ \forall \mathcal{R}_a \in SR) \tag{25}$$

For simplicity, we assume that no core is put into sleep mode within the LCM (least common multiple of the tasks' periods). Thus, we only need to consider minimizing the total power consumption, which is given by Equation (11).

Equations (12) and (13) indicates that a task can only be mapped onto one core. With discrete voltage/frequency setting being considered, we assume that a core can only be assigned one discrete processing frequency within $[f_1, f_{nd}]$, as shown in Equations (14), (15) and (16). The common voltage/frequency limitation of a VFI is further indicated by Equation (17). Moreover, the scaled frequency of a task is also subject to the discrete voltage/frequency levels (see Equation (18)). For a feasible schedule, the utilization of a core (see Equation (19)) can be calculated similar to Equation (31) in Section 4.2, which has to be no more than one as shown in Equation (20).

Since VFIs can run at different frequencies independently, the synchronization overhead of a task due to shared resource access must conform to such speed settings. The worst-case global waiting time and local blocking time of a task can be obtained by Equations (21) and (22), which are derived from Equations (6) and (9), respectively. Also, the maximum global waiting time for a task to execute a critical section (see Equation (23)), the maximum amount of time for any task on a specific core to access a resource once (see Equation 24) and that for a specific task to access a resource once (see Equation 25) can be respectively calculated by modifying Equations (8), (7) and (5).

From such a problem formulation, the optimal task-to-core mapping ($\xi_{i,j}$) and the scaled frequency assignment of all cores ($\phi_{i,j}$) can be obtained with an INLP (Integer Non-Linear Programming) approach. More specifically, we have implemented such an INLP solution using Lingo tool [53].

4.2 Synchronization-Aware Mapping Algorithm

For partitioned scheduling in multiprocessor real-time systems, several efficient mapping heuristics have been studied based on task's utilization, such as *First-Fit (FF), Best-Fit (BF)* and *Worst-Fit (WF)*. In particular, the *Worst-Fit Decreasing (WFD)* heuristic aims at obtaining the partitioning with balanced workload on each processor and has been exploited previously for energy management [7]. When task synchronization is considered, from Equation 10, the schedulability of tasks on each core depends on not only the accumulated task utilization (i.e., workload) but also the blocking and waiting time of tasks due to accessing shared resources.

Hence, as a variation of WFD, our novel *synchronization-aware* mapping scheme (denoted as *SA-WFD*) partitions tasks to cores based on the resources that a task accesses in addition to its utilization. First, to quantify the relationship of the resources accessed by tasks, the *resource similarity* between two tasks T_i and T_j is defined as $\omega_{i,j} = |SR_i \cap SR_j|$. Recall that SR_i denotes the subset of resources that are accessed by task T_i. That is, $\omega_{i,j}$ represents the number of resources that are accessed by both tasks T_i and T_j.

Second, instead of using task utilization as in the conventional WFD heuristic, SA-WFD relies on *estimated utilization* of tasks that takes the global waiting time into consideration. Note that, from Equation 6, the accurate estimation of a task T_i's global waiting time BW_i rather depends on the task-to-core mapping. To ensure schedulability of tasks in the worst-case scenario, more specifically, SA-WFD utilizes the *pessimistic estimated utilization* (peu_i) of each task T_i that incorporates its *maximum* global waiting time BW_i^{max}, regardless of a specific task-to-core mapping. More formally, we have

$$peu_i = \frac{c_i + BW_i^{max}}{p_i} \tag{26}$$

$$BW_i^{max} = \sum_{j=1}^{N_{cs}(T_i)} BW^{max}(z_{i,j}) \tag{27}$$

$$BW^{max}(z_{i,j}) = \sum_{\forall T_x \in \Theta(z_{i,j})} tt_x^{max}(s(z_{i,j})) \tag{28}$$

Here, the task set $\Theta(z_{i,j})$ contains up to $(nc - 1)$ other tasks that access the resource $s(z_{i,j})$ and have the longest access time. That is, whenever task T_i accesses a resource $s(z_{i,j})$, it is assumed to wait for other tasks, up to $(nc - 1)$, on different cores to access $s(z_{i,j})$ for the longest time. If no other task accesses the resource $s(z_{i,j})$, there are $\Theta(z_{i,j}) = \emptyset$ and $BW^{max}(z_{i,j}) = 0$.

Based on the resource similarity and pessimistic estimated utilizations of tasks, the basic steps of SA-WFD task mapping scheme are summarized in Algorithm 1. First, the algorithm calculates peu_i of tasks based on Equations 26, 27 and 28

Algorithm 1 The SA-WFD task mapping algorithm

Input: A set Γ of nt tasks; and nc processing cores;
Output: $\Psi_j (j = 1, \cdots, nc)$; system utilization U;
1: **for** (each task $T_i \in \Gamma$) **do**
2: calculate BW_i^{max} from Equation 27;
3: calculate $peu_i = \frac{c_i + BW_i^{max}}{p_i}$;
4: **end for**
5: Initialize $\Psi_j = \emptyset$ $(j = 1, \cdots, nc)$;
6: Sort tasks in non-increasing order of peu_i (if $peu_i = peu_j$ and $i < j$, T_i has higher priority and is in front of T_j);
7: **for** (each task T_i in above sorted order) **do**
8: Find core $\mathcal{O}_x$ with the maximum $\Omega_x(i)$ (if more cores have the same maximum $\Omega(i)$, the core with the minimum $EU_x(\Psi_x)$ is chosen; if there is still a tie, the core with smaller index is chosen);
9: **if** $(EU_x(\Psi_x \cup \{T_i\}) \leq \max_{j=1}^{nc} EU_j(\Psi_j))$ **then**
10: Allocate T_i to core $\mathcal{O}_x$: $\Psi_x = \Psi_x \cup \{T_i\}$;
11: **else**
12: Find core $\mathcal{O}_y$ with the minimum $EU_y(\Psi_y)$ (if there is a tie, the core with smaller index is chosen);
13: Allocate T_i to core $\mathcal{O}_y$: $\Psi_y = \Psi_y \cup \{T_i\}$;
14: **end if**
15: **end for**
16: For each task T_i: based on Ψ_k $(k = 1, \cdots, nc)$, calculate BW_i and B_i from Equations 6 and 9, respectively;
17: For each core $\mathcal{O}_k$: calculate U_k from Equation 31;
18: System utilization: $U = \max\{U_k | k = 1, \cdots, nc\}$;

(lines 1 to 4). The task set for each core is then initialized (line 5). Next, tasks are allocated to cores one at a time in the descending order of their pessimistic estimated utilizations (tie is broken to favor the task with smaller index). Here, different from the conventional WFD heuristic that makes decisions solely based on the accumulated utilization on each core, SA-WFD considers both the utilization as well as the resource similarity between tasks. For the next task T_i to be allocated, we first find a core based on the *overall resource similarity* between task T_i and core $\mathcal{O}_k$, which is defined as:

$$\Omega_k(i) = \sum_{\forall T_j \in \Psi_k} \omega_{i,j} \tag{29}$$

We choose the core $\mathcal{O}_x$ that has the maximum $\Omega_x(i)$. In case that there are more cores having the same maximum $\Omega(i)$, SA-WFD chooses the core further based on the *overall estimated utilization* of these cores, which is defined as:

$$EU_x(\Psi_x) = \sum_{\forall T_j \in \Psi_x} peu_j \tag{30}$$

If there is still a tie, the core with smaller index is chosen (line 8). Task T_i will be allocated to core $\mathcal{O}_x$ as long as such allocation does not increase the maximum overall estimated utilization of cores (lines 9 and 10). That is, provided that the estimated workload on cores is in balance, SA-WFD tries to allocate the tasks that access similar group of resources to the same core as such allocation can reduce blocking and waiting time of tasks and thus improve their schedulability.

However, if the allocation of task T_i to core $\mathcal{O}_x$ would increase the maximum overall estimated utilization of all cores (which may result in unbalanced workload on cores), SA-WFD will select the cores following the original principle of WFD. That is, the core $\mathcal{O}_y$ that has the minimum overall estimated utilization will have the task T_i (lines 12 and 13).

Once the mapping of tasks to cores (i.e., Ψ_k, $k = 1, \cdots, nc$) is determined following the above steps, the global waiting time BW_i and local blocking time B_i of each task T_i can be calculated from Equations 6 and 9, respectively (line 16). Then, based on the schedulability condition represented by Equation 10, we calculate the *utilization* of each core $\mathcal{O}_k$ (line 17), which is defined as follows:

$$U_k = \max_{\forall T_i \in \Psi_k} \left\{ \frac{B_i}{p_i} + \sum_{\forall T_j \in \Psi_k}^{p_j \leq p_i} \frac{c_j + BW_j}{p_j} \right\} \tag{31}$$

Finally, the *system utilization* (U) is found as the maximum of U_k of all cores (line 18). Note that, if $U > 1$, it means that our SA-WFD scheme fails to obtain a feasible task-to-core mapping. Otherwise (i.e., $U \leq 1$), the task sets Ψ_k $(k = 1, \cdots, nc)$ represent the feasible mapping of tasks to cores and each set Ψ_k is schedulable on core $\mathcal{O}_k$ under the EDF scheduling (see Equation 10).

Complexity of SA-WFD: Suppose that the maximum number of critical sections of a task is a constant. From Equations 27 and 28, finding out the maximum waiting time for each task can be done in $O(nt \cdot log(nt))$ time, where nt is the number of tasks in the system. Therefore, the pessimistic estimated utilizations of all tasks can be calculated in $O(nt^2 \cdot log(nt))$ time (i.e., the first for-loop in Algorithm 1). Then, sorting the tasks according to the estimated utilization can be done in $O(nt \cdot log(nt))$ time. When mapping tasks to cores (the second for-loop in Algorithm 1), finding the appropriate core for each task requires the calculation of overall resource similarity and can be done in $O(nc \cdot nt)$ time, where nc is the number of available cores. Thus, the complexity for the second for-loop can be found as $O(nc \cdot nt^2)$. Similarly, the complexity for lines 16, 17 and 18 can be found as $O(nt^2)$, $O(nc \cdot nt^2)$ and $O(nc)$, respectively. Therefore, the complexity of SA-WFD can be found as $O(\max\{log(nt), nc\} \cdot nt^2)$.

An Example: Next, we further illustrate how our SA-WFD task-to-core mapping heuristic works through a concrete example. Here, we consider a multicore system with three cores ($\mathcal{O}_1, \mathcal{O}_2$ and $\mathcal{O}_3$). For simplicity, the maximum processing frequency of cores is normalized to $f_{max} = 1$ and there are 10 frequency levels from 0.1 to 1 with the step of 0.1. The task set contains six tasks that will access two global resources. The timing parameters of the tasks are shown in Table 1 and their resource access patterns are shown in Figure 2.

First, the maximum global waiting time of the tasks are calculated according to Equation 27, which are shown in Table 2. For clarity, the intermediate parameters and the pessimistic estimated utilizations of tasks are also shown in Table 2. From the tasks' estimated utilizations, we can see

TABLE 2: The maximum waiting time and estimated utilizations of tasks; and the final task-to-core mapping for the example.

T_i	$tt_i^{max}(\mathcal{R}_1)$	$tt_i^{max}(\mathcal{R}_2)$	$BW^{max}(z_{i,1})$	$BW^{max}(z_{i,2})$	BW_i^{max}	peu_i	$\mathcal{O}(T_i)$	BW_i	B_i	BW_i/S_s	B_i/S_s
T_1	0	1	3	-	3	$\frac{2+3}{10} = 0.5$	$\mathcal{O}_2$	2	0	2.5	0
T_2	2	0	2	2	4	$\frac{7+4}{30} = 0.367$	$\mathcal{O}_1$	2	0	2.5	0
T_3	0	1	3	-	3	$\frac{2+3}{10} = 0.5$	$\mathcal{O}_3$	1	3	1.25	3.75
T_4	0	2	2	-	2	$\frac{4+2}{30} = 0.2$	$\mathcal{O}_3$	1	0	1.25	0
T_5	1	0	3	-	3	$\frac{3.1+3}{10} = 0.61$	$\mathcal{O}_1$	1	3	1.25	3.75
T_6	1	0	3	-	3	$\frac{2+3}{10} = 0.5$	$\mathcal{O}_2$	2	0	2.5	0

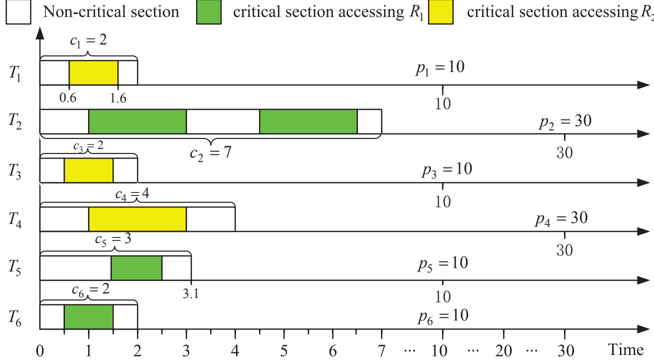


Fig. 2: An example: task and resource access patterns.

TABLE 1: The timing parameters of tasks

T_i	c_i	p_i	$u_i(\frac{c_i}{p_i})$	$N_{cs}(T_i)$	$\{s(z_{i,j})\}$	$\{c_{cs}(z_{i,j})\}$
T_1	2	10	0.2	1	$\{\mathcal{R}_2\}$	{1}
T_2	7	30	0.233	2	$\{\mathcal{R}_1, \mathcal{R}_1\}$	{2,2}
T_3	2	10	0.2	1	$\{\mathcal{R}_2\}$	{1}
T_4	4	30	0.133	1	$\{\mathcal{R}_2\}$	{2}
T_5	3.1	10	0.31	1	$\{\mathcal{R}_1\}$	{1}
T_6	2	10	0.2	1	$\{\mathcal{R}_1\}$	{1}

that the order of tasks to be allocated under SA-WFD will be T_5, T_1, T_3, T_6, T_2 and T_4.

Initially, there is no task on any core and $\Psi_j = \emptyset$ $(j = 1, 2, 3)$. The allocation of tasks T_5 and T_1 results in the following partition: $\Psi_1 = \{T_5\}$, $\Psi_2 = \{T_1\}$ and $\Psi_3 = \emptyset$, which is the same as WFD since these two tasks access different resources. For task T_3, it accesses the same resource ($\mathcal{R}_2$) as task T_1 and has the preference of being allocated to core $\mathcal{O}_2$. However, if task T_3 were allocated to core $\mathcal{O}_2$, its overall estimated utilization (1.0) would be more than the current maximum overall estimated utilization of all cores (0.61). In this case, the same as in WFD, core $\mathcal{O}_3$ is chosen as it has the minimum overall estimated utilization and we have $\Psi_3 = \{T_3\}$. Similarly, task T_6 will be allocated to core $\mathcal{O}_2$ that results in $\Psi_2 = \{T_1, T_6\}$. Moreover, there is $\max_j\{EU_j(\Psi_j)\} = EU_2(\Psi_2) = 1.0$.

For task T_2, which accesses the same resource $\mathcal{R}_1$ as that of tasks T_5 and T_6, both cores $\mathcal{O}_1$ and $\mathcal{O}_2$ have the same overall resource similarity $\Omega_1(2) = \Omega_2(2) = 1$. Here, core $\mathcal{O}_1$ has smaller overall estimated utilization ($EU_1(\Psi_1) = 0.61$) and will be task T_2's preference. Note that, allocating task T_2 to core $\mathcal{O}_1$ leads to $EU_1(\Psi_1 \cup \{T_2\}) = 0.977$, which is smaller than that of core $\mathcal{O}_2$ (which has the current maximum overall estimated utilization 1.0). That is, SA-WFD finally allocates task T_2 to core $\mathcal{O}_1$ and there is $\Psi_1 = \{T_5, T_2\}$.

Following the same steps, after task T_4 is allocated, the final task-to-core mapping can be obtained as: $\Psi_1 = \{T_5, T_2\}$, $\Psi_2 = \{T_1, T_6\}$ and $\Psi_3 = \{T_3, T_4\}$. From this mapping, the final BW_i and B_i for each task can be calculated by Equations 6 and 9, respectively, which are also shown in Table 2. Then, from Equation 31, we can get the utilizations on the cores as $U_1 = 0.71$, $U_2 = 0.8$ and $U_3 = 0.6$, respectively. Finally, the system utilization can be determined as $U = \max_{j=1}^{3}\{U_j\} = 0.8$.

For comparison, based solely on tasks' utilizations (as shown in Table 1), the order of tasks being allocated under the conventional WFD is T_5, T_2, T_1, T_3, T_6 and T_4. The result WFD mapping can be found as $\Psi_1^{WFD} = \{T_4, T_5\}$, $\Psi_2^{WFD} = \{T_2, T_6\}$ and $\Psi_3^{WFD} = \{T_1, T_3\}$ as shown in Table 3. Based on the mapping, we can further get the utilizations for the cores as $U_1^{WFD} = 0.81$, $U_2^{WFD} = 0.6$ and $U_3^{WFD} = 0.8$ from Equation 31, and thus we have $U^{WFD} = \max_{j=1}^{3}\{U_j^{WFD}\} = 0.81$.

Although the task set is schedulable under both WFD and SA-WFD, the average utilization for cores under WFD is higher than that of SA-WFD. Moreover, from Section 6.2, we can see that SA-WFD can significantly improve the schedulability of tasks.

TABLE 3: The result task-to-core mapping of WFD

T_i	$\mathcal{O}(T_i)$	$BW(z_{i,1})$	$BW(z_{i,2})$	BW_i	B_i
T_1	$\mathcal{O}_3$	2	-	2	0
T_2	$\mathcal{O}_2$	1	1	2	0
T_3	$\mathcal{O}_3$	2	-	2	0
T_4	$\mathcal{O}_1$	1	-	1	0
T_5	$\mathcal{O}_1$	2	-	2	3
T_6	$\mathcal{O}_2$	1	-	1	3

4.3 Static Uniform Slowdown Scheme

Taking the common voltage/frequency limitation for cores on a VFI into consideration, the simplest static approach is to assign a *uniform* scaled frequency for all (cores and) VFIs.

Recall that the execution time of a task is assumed to scale linearly with its running speed. When all tasks are executed at the scaled speed U, from Equations 6, 9 and 31, we know that the blocking and waiting times of tasks as well as system utilization are all scaled linearly. To ensure the schedulability of tasks on all cores, the uniform frequency can be safely found as $S_s = f_{l+1}$, where $f_l < U \leq f_{l+1}$ [34]. Hence, for the previous example, we have $S_s = 0.8$ for the SA-WFD

mapping. In contrast, $S_s = 0.9$ for the WFD mapping.

Schedulability of Uniform Slowdown Scheme: For the uniform slowdown scheme, we can directly derive the new sufficient feasibility condition for any processing core $\mathcal{O}_k$ from Equation (10) as:

$$\forall T_i \in \Psi_k, \left(\frac{B_i}{p_i} + \sum_{\forall T_j \in \Psi_k}^{p_j \le p_i} \frac{c_j + BW_j}{p_j} \right) \cdot \frac{1}{S_s} \le 1 \qquad (32)$$

Hence, the system utilization with a given scaled frequency S_s can be found as $U(S_s) = \max\{U_k(S_s) | k = 1, \cdots, nc\}$, where $U_k(S_s)$ can be obtained from Equation 31.

Lemma 1. *The uniform scaled frequency scheme guarantees the schedulability of all tasks if $U(S_s) \le 1$.*

Proof: We prove the claim by contradiction, which is similar to the proofs in [8], [54]. Suppose that this claim is false and a job $J_{i,j}$ on core $\mathcal{O}_a (T_i \in \Psi_a)$ misses its deadline at time t. This implies that $J_{i,j}$'s deadline is t. We find a time point t' before t, which is the latest time such that no active job which arrived before t' has a deadline equal to or sooner than t on $\mathcal{O}_a$. By appropriately choosing t', $\mathcal{O}_a$ is never idle during $[t', t]$. If such t' does not exist, we let $t' = 0$.

We define $\mathbb{A}$ as the set of jobs that arrived in the time interval $[t', t]$ on $\mathcal{O}_a$ and have deadlines equal to or sooner than t. Let $J_{m,n}$ (if any exists) be a job with deadline later than t on $\mathcal{O}_a$, which is waiting for or accessing a resource at time t'. Observe that $J_{m,n}$ must issue a request for this resource prior to time t': if not, $J_{m,n}$ is preempted at time t' and cannot continue its execution until time t subject to our adopted resource access protocol.

According to above assumptions and analysis, the maximum execution time of $J_{m,n}$ in $[t', t]$ is upper-bounded by $\frac{B_i}{S_s}$ according to a given feasible task-to-core mapping (e.g., SA-WFD). Let $X = t - t'$. During $[t', t]$, the maximum processor demand by jobs pertaining to set $\mathbb{A} \cup \{J_{m,n}\}$ is:

$$\frac{f_{max}}{S_s} \cdot \left(B_i + \sum_{\forall T_k \in \Psi_a}^{p_k \le p_i} (c_k + BW_k) \cdot (\lfloor \frac{X - p_k}{p_k} \rfloor + 1) \right)$$

Since $J_{i,j}$ misses its deadline at time t, the total processor demand in $[t', t]$ exceeds X. Then, we have:

$$\frac{f_{max}}{S_s} \cdot \left(B_i + \sum_{\forall T_k \in \Psi_a}^{p_k \le p_i} (c_k + BW_k) \cdot (\lfloor \frac{X - p_k}{p_k} \rfloor + 1) \right) > X$$

For $\forall J_{k,x} (J_{k,x} \in \mathbb{A})$, we have $\frac{X}{p_k} \ge \lfloor \frac{X}{p_k} \rfloor$ and $p_k \le p_i \le X$. Then, there is:

$$\frac{f_{max}}{S_s} \cdot \left(\frac{B_i}{X} + \sum_{\forall T_k \in \Psi_a}^{p_k \le p_i} (c_k + BW_k) \cdot (\frac{X - p_k}{p_k} + 1) \cdot \frac{1}{X} \right) > 1$$

$$\Rightarrow \frac{f_{max}}{S_s} \cdot \left(\frac{B_i}{p_i} + \sum_{\forall T_k \in \Psi_a}^{p_k \le p_i} \frac{c_k + BW_k}{p_k} \right) > 1$$

which contradicts with Equation 32. □

Algorithm 2 SA-WFD-Diff: find frequencies for VFIs

Input: A feasible task-to-core mapping $\Psi_i (i = 1, \cdots, nc)$, where $U_i \ge U_j$ for $1 \le i < j \le nc$;
Output: Scaled frequencies $S_{G_i} (i = 1, \cdots, ng)$;
1: $S_{G_1} = S_s$; //the first VFI has the highest frequency
2: **for** (each VFI represented by G_i, $i = 2, \cdots, ng$) **do**
3: Find the lowest frequency $f_k (f_1 \le f_k \le S_{G_{i-1}})$ for the remaining VFIs: G_i to G_{ng}, such that:
4: //Recalculate BW_j,B_j, U_k with $S_{G_1}, \ldots, S_{G_{i-1}}$, f_x;
5: $\max\{U_k | k = 1, \ldots, nc\} \le 1$;
6: $S_{G_i} = f_k$;
7: $i = i + 1$;
8: **end for**

4.4 One Frequency per VFI

Although SA-WFD aims at obtaining a workload-balanced task-to-core mapping, the utilization on each core can vary. Thus, instead of assigning a single uniform frequency, we can assign individual frequencies for different VFIs on the chip to get more energy savings.

Without loss of generality, for the result mapping of tasks to cores, we assume that there is $U_i \ge U_j (1 \le i < j \le nc)$. Otherwise, we can simply switch the task sets for the cores. Moreover, we use G_1 to represent the first VFI that consists of the first group of $\frac{nc}{ng}$ cores $\{\mathcal{O}_1, \cdots, \mathcal{O}_{\frac{nc}{ng}}\}$, and so on. That is, G_1 has the most loaded cores while G_{ng} has the least loaded ones. Following the intuition that a VFI with less loaded cores should have a lower frequency, the scaled frequencies for the VFIs can be found iteratively as shown in Algorithm 2. Here, the static processing frequency of VFI G_i is denoted by S_{G_i}.

Initially, we assign S_s to the first VFI (line 1). From above discussions, we know that it would be safe to assign S_s for all other VFIs. However, for less loaded cores in other VFIs, it is more energy efficient to have lower frequencies. Note that, scaling down the frequency for one VFI will affect the schedulability of tasks on other cores due to increased global waiting time. Thus, for the remaining VFIs, we find the lowest common frequency f_k (which is no more than S_s) that can guarantee the schedulability of tasks on all cores (lines 3 to 5). That is, we need to re-calculate all cores' utilizations (and thus all tasks' BW_j and B_j) with the new frequency. Please refer to the formalized description of task mapping problem as stated in Section 4.1. Then, f_k is assigned to the second VFI (line 6). Repeat the steps, the frequencies for all VFIs can be found iteratively.

Time complexity analysis: Suppose that the maximum number of critical sections of a task and that of discrete voltage/frequency levels are constants. The utilization adjustment of VFIs can be done in $O(nc \cdot log(nc))$ time. From Section 4.1, finding out the maximum global waiting time of a task requires $O(nt)$ time; while calculating the maximum blocking time for a task can also be done in $O(nt)$ time. The calculation of a core's utilization (line 4 in Algorithm 2) requires $O(nt^2)$ time. As the number of VFIs is at most nc, the time complexity of SA-WFD-Diff can be

found as $O(nc^2 \cdot nt^2)$.

Schedulability Analysis of One Frequency per VFI: When each VFI has a different scaled frequency, the sufficient feasibility condition for any processing core $\mathcal{O}_k \in G_g$ can be easily transformed from Equation 32:

$$\forall T_i \in \Psi_k, \left(\frac{B_i}{p_i} + \sum_{\forall T_j \in \Psi_k}^{p_j \leq p_i} \frac{\frac{c_j}{S_{G_g}} + BW_j}{p_j} \right) \leq 1 \qquad (33)$$

It is noteworthy that the synchronization overhead of task T_i (e.g., BW_i and B_i) has been scaled down from Equations 21 and 22 in Section 4.1. Following with similar reasonings of Lemma 1, we can easily derive the following lemma.

Lemma 2. *The static scheme with a different scaled frequency for each VFI guarantees the schedulability of all tasks if the original mapping is scheduleable.*

5 Online Synchronization-Aware DVFS

It is well-known that real-time tasks typically take only a small fraction of their WCETs at runtime [25]. The actual synchronization overhead of a task is usually less than the worst-case one that is determined at off-line stage as well. Moreover, with the suspension-based resource access protocol, a core can execute other tasks while a task is waiting for its resource that is held by another task on different core (see Rule 3 in Section 3.3), which can effectively reduce the actual timing overhead due to task synchronization. Therefore, significant amount of dynamic slack can be expected at runtime that could be exploited for better energy savings.

However, different from most existing dynamic energy management schemes [6], [41], [56], exploiting dynamic slack for tasks with shared resources in VFI-based multicore real-time systems introduces several **new challenges**.

- First, slowing down the execution of critical sections when tasks access shared resources can affect the actual waiting time of other tasks on different cores, which may lead to deadline misses.
- Second, the amount of dynamic slack reclaimed by a task and its scaled frequency need to conform to the common frequency constraint for the cores on a VFI.
- Third, the pre-execution of tasks on a core when one of its tasks is waiting for a resource requires non-trivial slack management techniques.

To address these challenges, we propose a *synchronization-aware DVFS (SA-DVFS)* framework that consists of a set of slack management policies, which aim at fully and appropriately utilizing dynamic slack to improve energy savings while guaranteeing that there is no deadline miss at runtime even with task synchronization being considered. In what follows, to precisely describe our approaches, we distinguish the term *task* from *job*.

As the foundation of our SA-DVFS framework, we first review the essential ideas and basic operations of wrapper-jobs, which have been studied as an effective slack management mechanism for a single processor system under EDF scheduling [55].

5.1 Slack Management with Wrapper-Jobs

A wrapper-job *WJ* represents a piece of dynamic slack with two parameters (c, d): the size c that denotes the amount of slack and the deadline d that equals to that of the job giving rise to this slack. Wrapper-jobs are kept in a *wrapper-job queue (WJQ)* with the increasing order of their deadlines (i.e., wrapper-jobs with smaller deadlines are in the front of *WJQ*).

The basic operations of wrapper-jobs and *WJQ* can be summarized as follows [55]:

- *GenerateSlack(c, d, WJQ):* Create a wrapper-job *WJ* with parameters (c, d) and add it to *WJQ* with increasing deadline order. Here, all wrapper-jobs in *WJQ* represent dynamic slack with different deadlines. Therefore, the newly created *WJ* may merge with an existing wrapper-job in *WJQ* if they have the same deadline;
- *CheckSlack(d, WJQ):* Find out the total size of all wrapper-jobs that have their deadlines no later than d;
- *ReclaimSlack(c, WJQ):* Remove wrapper-jobs from the front of *WJQ* with accumulated size of c. The last one may be partially removed by adjusting its remaining size.

At runtime, unclaimed wrapper-jobs compete the processor with ready jobs in a *ready-job queue (RJQ)* that are kept in the increasing order of their deadlines as well (tie is broken to favor the job of task with smaller index). When both queues are not empty and the header wrapper-job WJ_h of *WJQ* has earlier deadline (i.e., higher priority) than that of the header ready job J_h of *RJQ*, WJ_h will wrap the execution of J_h by lending its time to job J_h. When the wrapped execution completes, job J_h returns its borrowed slack by creating a new piece slack with the size of wrapped execution length and J_h's deadline (essentially, the slack is pushed forward with a later deadline). When *RJQ* is empty, the header wrapper-job WJ_h executes no-ops and the corresponding slack is wasted. Interested readers can refer to [55] for more details.

After the static scheme obtains the feasible task-to-core mapping and uniform scaled speed S_s (see Section 4), the *effective* utilization for core $\mathcal{O}_k$ can be found as $U_k(S_s) = \frac{U_k \cdot f_{max}}{S_s}$. If $U_k(S_s) < 1$, there exists spare capacity on core $\mathcal{O}_k$. To incorporate such spare capacity at runtime, a dummy task $T_{k,0}$ with utilization $u_{k,0} = 1 - U_k(S_s)$ can be created for core $\mathcal{O}_k$. Here, the period of $T_{k,0}$ can be set as $p_{k,0} = \min\{p_i | \forall T_i \in \Psi_k\}$ (i.e., the minimum period of tasks assigned to core $\mathcal{O}_k$) [55]. The dummy task does not access any resource and has its actual execution time always being zero. Therefore, it will not affect the schedulability of other tasks under EDF. Similarly, the dummy task scheme can be applied to the SA-WFD-Diff, where each VFI can be assigned an individual initial frequency, with only minor modifications.

Moreover, the dummy task can periodically transform the spare capacity as dynamic slack (with the amount of $p_{k,0} \cdot u_{k,0}$ for every period of $p_{k,0}$), which can be reclaimed together with other dynamic slack for more energy savings. In what follows, we assume that the task set for each core $\mathcal{O}_k$ is augmented with a dummy task (i.e., $\Psi_k = \Psi_k + \{T_{k,0}\}$). Furthermore, with partitioned-EDF scheduling, each core $\mathcal{O}_k$ has its own ready job queue RJQ_k and wrapper-job queue WJQ_k.

Algorithm 3 : High-level steps for SA-DVFS

1: //Core $\mathcal{O}_k$ on a VFI G_g to execute job $J_{i,j}$ at time t;
2: **if** ($J_{i,j}$'s current section is **non-critical**) **then**
3: **if** (non-critical section is ready to run) **then**
4: Slack *reclamation* policy; //Section 5.3
5: **else if** (non-critical section stops due to completion, a resource request, being preempted or interrupted) **then**
6: Slack *release and preservation* policies; //Section 5.4
7: **end if**
8: **else if** ($J_{i,j}$'s current section is **critical**) **then**
9: **if** (critical section is suspended) **then**
10: Slack *stealing* policy; //Section 5.5
11: **else if** (suspension is over) **then**
12: Slack *preservation for suspension*; //Section 5.5
13: **else if** ($J_{i,j}$ is ready to execute its critical section) **then**
14: *Constrained slack allocation* policy; //Section 5.6
15: **else if** (critical section completes) **then**
16: Slack *release* policy;
17: **end if**
18: **end if**

5.2 Overview of Synchronization-Aware DVFS

Based on the wrapper-job mechanism to manage slack, the high-level steps for SA-DVFS are summarized in Algorithm 3. Our integrated slack management framework deals with the non-critical and critical sections, respectively. The policies applied to non-critical sections include:

- There is a *slack reclamation* policy for the non-critical sections of jobs that can reclaim dynamic slack *independently* on each core, which obtains the desired scaled frequency for the job based on available slack (line 4); However, the job's actual execution speed is determined by the common frequency of the VFI that its core are on;
- When the execution of a job's non-critical section stops due to completion, a resource request, being preempted or interrupted, the *slack release and preservation* policies will calculate the actual amount of slack utilized, push forward borrowed slack, release unused slack and preserve slack needed in the future (line 6).

For critical sections, the corresponding approaches are:

- When a job is suspended while waiting for its resource that is held by another job on a different core, there is a *slack stealing* policy to pre-execute the non-critical sections of other jobs on the same core (line 10);
- Once the suspension is over, the slack preservation policy is applied to release slack generated due to less task synchronization overhead and to push forward the borrowed slack if there is any (line 12);
- Moreover, when the job is ready to execute its current critical section, we have a *constrained slack allocation* policy; It limits the amount of slack reclaimed by a job's critical section to guarantee there is no deadline miss for those jobs who may be affected by this slack reclamation (line 14);
- Finally, when a job's critical section completes, the slack release policy is needed (line 16).

Next, we discuss each of these slack management policies in detail. To simplify the discussion, we assume that *all VFIs initially have the same static uniform scaled frequency S_s*. Later, we will prove that the SA-DVFS framework can work with different initial frequencies for VFIs (i.e., SA-WFD-Diff) as well.

From [55], we have the following supposition to help understand the schedulability investigation of our SA-DVFS slack management policies.

Hypothesis 1. *Before applying our slack management policies, at each invocation time, the feasibility of jobs subject to the wrapper-job scheme is guaranteed.*

5.3 Slack Reclamation for Non-Critical Sections

For the non-critical section of a real-time job, its execution on one core has no synchronization effect on other jobs. Therefore, slack reclamation for non-critical sections can be performed independently on each core provided that it does not affect the schedulability of jobs on the same core.

Suppose that core $\mathcal{O}_k$ on VFI G_g is about to process the non-critical section of job $J_{i,j}$ at time t. Next, we define a few auxiliary notations for easy presentation of the slack management policies:

- $c_{ns}^{rem}(J_{i,j}, t)$: the remaining WCET of $J_{i,j}$'s non-critical section at time t; Initially, there is $c_{ns}^{rem}(J_{i,j}, t_{i,j}^r) = c_{ns}(T_i)$ at $J_{i,j}$'s release time $t_{i,j}^r$;
- $FS_{i,j}^{ns}(t)$: the *feasible speed* for $J_{i,j}$'s non-critical section at time t; $FS_{i,j}^{ns}(t)$ guarantees that $J_{i,j}$'s non-critical section can complete in time; Initially, $FS_{i,j}^{ns}(t_{i,j}^r) = S_s$;
- $trs(J_{i,j}, t)$: the total amount of *reclaimable* slack for job $J_{i,j}$ at time t; Note that a job can only reclaim the slack that has its deadline no later than that of the job [6], [55];
- CF_k^{exp}: the expected running frequency of core $\mathcal{O}_k$; it equals to the feasible speed of $\mathcal{O}_k$'s current job $J_{i,j}$;
- F_g: the actual frequency for active cores on a VFI G_g;
- $W(t', t, F_g)$: the amount of work being performed in the interval $[t', t]$ at speed F_g, that is, $(t - t') \cdot F_g$; If the $J_{i,j}$'s non-critical section runs at speed F_g in the interval $[t', t]$, there is

$$
\begin{aligned}
c_{ns}^{rem}(J_{i,j}, t) &= c_{ns}^{rem}(J_{i,j}, t') - \frac{W(t', t, F_g)}{f_{max}} \\
&= c_{ns}^{rem}(J_{i,j}, t') - \frac{(t - t') \cdot F_g}{f_{max}}
\end{aligned}
\tag{34}
$$

With these notations, the slack reclamation policy for non-critical section of job $J_{i,j}$ at time t is straightforward and its steps can be easily summarized as follows.

- **Step 1:** find out the amount of reclaimable slack for $J_{i,j}$ as:

  $$trs(J_{i,j}, t) = CheckSlack(d_{i,j}, WJQ_k)$$

  where $d_{i,j}$ is $J_{i,j}$'s deadline;
- **Step 2:** determine the feasible speed for $J_{i,j}$'s non-critical section $FS_{i,j}^{ns}(t)$, which can be given as:

$$FS_{i,j}^{ns}(t) = \frac{c_{ns}^{rem}(J_{i,j}, t) \cdot f_{max}}{trs(J_{i,j}, t) + \frac{c_{ns}^{rem}(J_{i,j}, t) \cdot f_{max}}{FS_{i,j}^{ns}(t^{last})}} \tag{35}$$

where t^{last} is the end time for $J_{i,j}$'s non-critical section's last execution and $FS^{ns}_{i,j}(t^{last})$ is its old feasible speed. Again, due to the limitation of discrete frequencies, we have $FS^{ns}_{i,j}(t) = f_{l+1}$ if $f_l < FS^{ns}_{i,j}(t) \leq f_{l+1}$. Thus, the amount of slack reclaimed by $J_{i,j}$'s non-critical section is:

$$Sk_{rec} = \left(\frac{c^{rem}_{ns}(J_{i,j}, t)}{FS^{ns}_{i,j}(t)} - \frac{c^{rem}_{ns}(J_{i,j}, t)}{FS^{ns}_{i,j}(t^{last})} \right) \cdot f_{max} \quad (36)$$

which is removed from WJQ_k with operation $ReclaimSlack(Sk_{rec}, WJQ_k)$;

- **Step 3:** set core $\mathcal{O}_k$'s expected frequency as $CF^{exp}_k = FS^{ns}_{i,j}(t)$. Finally, to guarantee the feasibility of jobs on all cores in the VFI G_g, the actual processing frequency is determined as:

$$F_g = \max\{CF^{exp}_x | \forall \mathcal{O}_x \in G_g\}$$

Note that, if the value of F_g does not change during this process, $J_{i,j}$'s non-critical section will be processed at speed F_g. However, when the newly determined F_g is different from its old value, a DVFS synchronization is first performed where the execution of jobs on all active cores in G_g is interrupted to update the execution timing information of jobs (such as remaining WCET; see Section 5.4). Then, the new frequency F_g will be set through a DVFS operation.

Incorporating the wrapper-job scheme and Hypothesis 1, it is easy to have:

Lemma 3. *The slack reclamation policy for non-critical sections meets time constraints of all jobs.*

5.4 Slack Release and Preservation Policies

From the slack reclamation policy, we can see that the actual processing speed of a job can be higher than its feasible speed due to the common frequency limitation for cores on a VFI. That is, a job may not be able to use up its reclaimed slack. Therefore, when the execution of a job's non-critical section stops [3], it can release some of its reclaimed slack *early*, which can be exploited by the next running jobs. However, instead of releasing all its un-used slack, the job needs to preserve some of the reclaimed slack to guarantee that its remaining non-critical sections can complete in time with its previous feasible speed. Here, the motivation is to efficiently and fairly utilize the available slack such that the jobs can run at similar frequencies for more energy savings.

Moreover, due to the limit of discrete frequencies, a job may not have reclaimed all its reclaimable slack. Hence, the job may borrow some of the remaining reclaimable slack, which needs to be pushed forward due to the wrapped-execution (see Section 5.1). Furthermore, when the non-critical sections of a job did not use up its WCET and complete early, new slack will be generated.

Suppose that the non-critical section of job $J_{i,j}$ was invoked at time t' on core $\mathcal{O}_k$ ($\in G_g$) and stops its execution at time t. The steps for the slack release and preservation policies can be summarized as follows:

- **Step 1:** Update the remaining work of $J_{i,j}$'s non-critical sections from Equation 34;
- **Step 2:** If the actual processing speed is higher than the feasible speed of $J_{i,j}$'s non-critical sections (i.e., $F_g > FS^{ns}_{i,j}(t')$), the amount of *early released slack* is:

$$Sk_{rel} = \left(\frac{F_g}{FS^{ns}_{i,j}(t')} - 1 \right) \cdot (t - t') \quad (37)$$

otherwise, $Sk_{rel} = 0$;

- **Step 3:** Suppose that, after slack reclamation, the amount of remaining reclaimable slack on $\mathcal{O}_k$ for $J_{i,j}$ is Sk_{rem} = *CheckSlack*($d_{i,j}$, *WJQ*$_k$). During the interval $[t', t]$, the amount of *borrowed slack* by $J_{i,j}$ is:

$$Sk_{bor} = \min\{Sk_{rem}, (t - t')\} \quad (38)$$

which should be first removed from *WJQ* with operation *ReclaimSlack*(Sk_{bor}, *WJQ*$_k$);

- **Step 4:** If $J_{i,j}$'s non-critical sections did not finish, no *new slack* is generated. We have $Sk_{gen} = 0$ and $FS^{ns}_{i,j}(t) = FS^{ns}_{i,j}(t')$; otherwise,

$$Sk_{gen} = \frac{c^{rem}_{ns}(J_{i,j}, t) \cdot f_{max}}{FS^{ns}_{i,j}(t')} \quad (39)$$

- **Step 5:** Finally, the borrowed slack is pushed forward, and the early released slack and the newly generated slack are added to *WJQ*$_k$ with a combined operation:

$$GenerateSlack(Sk_{rel} + Sk_{bor} + Sk_{gen}, d_{i,j}, WJQ_k)$$

Lemma 4. *The approach to calculate the amount of early released slack (Equation 37) ensures no deadline miss.*

Proof: At time t', job $J_{i,j}$'s feasible speed is set as $FS^{ns}_{i,j}(t')$. Then, it is expected that the work of $J_{i,j}$ being performed during $[t', t]$ will be $FS^{ns}_{i,j}(t') \cdot (t - t')$. As its actual work during the interval $[t', t]$ is $F_g \cdot (t - t')$, the extra work that has been done is $(F_g - FS^{ns}_{i,j}(t')) \cdot (t - t')$.

Note that, our slack management framework ensures the deadlines of jobs subject to their newest feasible speeds. Job $J_{i,j}$'s feasible speed remains invariable at time t from Step 4 even when it finishes at time t. Therefore, adhering to the current feasible speed of $J_{i,j}$'s non-critical section (i.e., $FS^{ns}_{i,j}(t')$), the amount of its early released slack Sk_{rel} can be determined as:

$$\frac{(F_g - FS^{ns}_{i,j}(t')) \cdot (t - t')}{FS^{ns}_{i,j}(t')} = \left(\frac{F_g}{FS^{ns}_{i,j}(t')} - 1 \right) \cdot (t - t')$$

□

From the wrapper-job scheme, Hypothesis 1 and Lemma 4, we can have:

Lemma 5. *Our slack release and preservation policies guarantee that all jobs can finish in time.*

3. From previous discussions, the execution of a job's non-critical section may stop due to a) being preempted by a newly arrived high priority job with earlier deadline; b) being interrupted due to a DVFS operation; c) the start of the job's next critical section; or d) (early) completion.

5.5 Slack Stealing Policy for Pre-Execution

Recall that the schedulability condition represented in Equation 10 incorporates the global waiting time for every critical section of a job. However, the suspension-based resource access protocol (Rule 3; see Section 3.3) adopted in this work allows a core to execute the non-critical sections of other ready jobs when one of its ready jobs is suspended and waiting for a resource that is currently held by jobs on other cores. Such pre-execution by *stealing* the waiting time ensures that those pre-executed jobs will complete early subject to their current feasible speeds and more dynamic slack can be generated for better energy savings.

Suppose that core $\mathcal{O}_k$ starts suspending the execution of a job $J_{i,j}$'s critical section $z_{i,x}$ because its resource $s(z_{i,x})$ is not available at time t_{start}. According to Rule 3 of the resource access protocol, $\mathcal{O}_k$ will *repeatedly* pre-execute the non-critical section of its earliest deadline job *non-preemptively* until $s(z_{i,x})$ becomes accessible for $J_{i,j}$ at time t_{end}.

To avoid being an obstacle for a lower frequency of the VFI and enable such pre-execution be processed at the lowest possible speed, the expected running frequency of core $\mathcal{O}_k$ is temporarily set as $CF_k^{exp} = f_1$ at time t_{start}. After that, a DVFS operation arises whenever possible. Please be noted that the current feasible speed of the pre-executed job's non-critical section remains invariable to guarantee no deadline miss. When there is no job that has its non-critical section ready for execution, $\mathcal{O}_k$ becomes idle and real waiting occurs.

For each job $J_{a,b}$ that is pre-executed by core $\mathcal{O}_k$ during the time interval $[t', t]$ at speed F_g, the remaining WCET of $J_{a,b}$'s non-critical section is updated from Equation 34. The amount of newly generated slack Sk_{gen} due to such pre-execution can be found as:

$$Sk_{gen} = \frac{(t - t') \cdot F_g}{FS_{a,b}^{ns}(t')} \tag{40}$$

Specifically, if $J_{a,b}$'s non-critical section completes at time t, additional amount of new slack due to early completion will be generated and we have

$$Sk_{gen} = Sk_{gen} + \frac{c_{ns}^{rem}(J_{a,b}, t) \cdot f_{max}}{FS_{a,b}^{ns}(t')}$$

In the end, the new slack will be added to WJQ_k through operation *GenerateSlack(*$Sk_{gen}, d_{a,b}, WJQ_k$*)*.

Slack preservation for suspension: Once the resource $s(z_{i,x})$ becomes accessible for job $J_{i,j}$ at time t_{end}, the amount of *actual* waiting time for resource $s(z_{i,x})$ is

$$BW^{act}(z_{i,x}) = t_{end} - t_{start}$$

Thus, the amount of slack borrowed by job $J_{i,j}$ during its waiting time is

$$Sk_{bor} = \min\{BW^{act}(z_{i,x}), trs(J_{i,j}, t_{end})\} \tag{41}$$

Moreover, due to possible shorter waiting time for resource $s(z_{i,x})$, the amount of new slack generated is

$$Sk_{gen} = \frac{BW(z_{i,x}) \cdot f_{max}}{S_s} - BW^{act}(z_{i,x}) \tag{42}$$

Then, the slack in WJQ_k will be updated with two operations:

$$ReclaimSlack(Sk_{bor}, WJQ_k)$$

$$GenerateSlack(Sk_{bor} + Sk_{gen}, d_{i,j}, WJQ_k)$$

to push forward the borrowed slack and add the newly generated slack.

By incorporating the wrapper-job scheme and Hypothesis 1, there is:

Lemma 6. *The slack stealing policy for pre-execution ensures the deadlines of all jobs.*

5.6 Constrained Slack Allocation: Critical Sections

Once the critical section $z_{i,x}$ of job $J_{i,j}$ obtains the permission to access its resource $\mathcal{R}_r = s(z_{i,x})$ at time t on core $\mathcal{O}_k(\in G_g)$, we may be able to reclaim some slack and process $z_{i,x}$ at a speed lower than its initial uniform speed S_s. However, as discussed earlier, the execution of a job's critical section affects not only the local blocking time of high priority jobs on the same core but also the global waiting time of other jobs' critical sections on different cores.

Therefore, in addition to $J_{i,j}$'s reclaimable slack $trs(J_{i,j}, t)$ at time t, the following two constraints have to be satisfied to ensure that all jobs can finish in time:

- **C1:** For any job $J_{a,b}$ on $\mathcal{O}_k$ that can be blocked by $J_{i,j}$, its actual blocking time needs to be bounded by $\frac{B_a \cdot f_{max}}{S_s}$;
- **C2:** For any job $J_{e,l}$ that has a critical section $z_{e,y}$ to access the resource $\mathcal{R}_r$ on other cores, its actual waiting time for the execution of $J_{i,j}$'s critical section $z_{i,x}$ needs to be bounded by $\frac{BW(z_{e,y}) \cdot f_{max}}{S_s}$.

Each of the above constraints essentially imposes a limit on the amount of slack that can be reclaimed by $J_{i,j}$'s critical section $z_{i,x}$. In what follows, we discuss how to obtain such slack limits s_{limit}^{c1} and s_{limit}^{c2}, which correspond to the first and second constraints, respectively.

Slack limit due to local blocking time: From Property 1 in Section 3.3, we know that $J_{i,j}$'s critical section $z_{i,x}$ can only block jobs of tasks in the set $\Psi_k^i = \{T_m | T_m \in \Psi_k \wedge p_m < p_i\}$. If task T_i has the smallest period (i.e., relative deadline) in core $\mathcal{O}_k$'s task set Ψ_k, we have $\Psi_k^i = \emptyset$ and there is no slack limit because of local blocking time. That is, $s_{limit}^{c1} = +\infty$.

Otherwise, for every task $T_m \in \Psi_k^i$, we need to have:

$$BW^{act}(z_{i,x}) + \frac{c_{cs}(z_{i,x}) \cdot f_{max}}{S_s} + s_{limit}^{c1} \leq \frac{B_m \cdot f_{max}}{S_s} \tag{43}$$

where $BW^{act}(z_{i,x})$ represents the actual time waiting for $J_{i,j}$'s critical section $z_{i,x}$. Here, the left side of the inequality stands for the longest blocking time that can be experienced by T_m's jobs due to the invocation of $z_{i,x}$. Therefore, s_{limit}^{c1} can be obtained as:

$$s_{limit}^{c1} = \min\{\frac{f_{max} \cdot (B_m - c_{cs}(z_{i,x}))}{S_s} - BW^{act}(z_{i,x}) | \forall T_m \in \Psi_k^i\} \tag{44}$$

Slack limit due to global waiting time: For jobs on other cores that have critical sections to access resource $\mathcal{R}_r$, not all

their global waiting times will be affected when the execution of $J_{i,j}$'s critical section $z_{i,x}$ is scaled down. Note that, for the jobs that are currently in $\mathcal{R}_r$'s FIFO waiting queue at time t, each of them belongs to a different core by Property 3. Suppose that these jobs form a job set Φ_r and their cores form a set $\mathbb{X}$.

It can be seen from Property 3 that, for the cores in $\mathbb{X}$, no other job's waiting time will be affected by $z_{i,x}$ except the ones in Φ_r. If $\Phi_r = \emptyset$, there is no limit due to global waiting time and $s^{c2}_{limit} = +\infty$.

Otherwise, for any job $J_{q,a} \in \Phi_r$, suppose that it enters $\mathcal{R}_r$'s FIFO queue at time t_q and its current critical section that needs to access $\mathcal{R}_r$ is $z_{q,c}$. Without taking slack reclamation into consideration, the expected waiting time for $z_{q,c}$ can be found as:

$$BW^{exp}(z_{q,c}) = \frac{c_{cs}(z_{i,x}) + \sum_{\forall J_{b,l} \in \Phi_r}^{t_b < t_q} c_{cs}(z_{b,e})}{S_s} \cdot f_{max} + t - t_q \quad (45)$$

where $z_{b,e}$ is the current critical section of job $J_{b,l}$ that is in front of $J_{q,a}$ in $\mathcal{R}_r$'s FIFO queue.

Recall that $\frac{BW(z_{q,c}) \cdot f_{max}}{S_s}$ represents $z_{q,c}$'s worst case waiting time. Therefore, we have

$$s^{c2}_{limit} = \min\left\{\frac{BW(z_{q,c}) \cdot f_{max}}{S_s} - BW^{exp}(z_{q,c}) | \forall J_{q,a} \in \Phi_r\right\} \quad (46)$$

While for other cores having at least one task that needs to access $\mathcal{R}_r$ but not in $\mathbb{Y} = (\mathbb{X} \cup \{\mathcal{O}_k\})$, we define another set as:

$$\mathbb{Z} = \{\mathcal{O}_m | \exists T_q \in \Psi_m \wedge \mathcal{R}_r \in SR_q \wedge \mathcal{O}_m \notin \mathbb{Y}\} \quad (47)$$

If $\mathbb{Z} = \emptyset$, there is no other constraint for s^{c2}_{limit}. Otherwise, we further define the set $\mathbb{O}$ that consists of any core $\mathcal{O}_m \in \mathbb{Z}$ where all its tasks that need to access $\mathcal{R}_r$ complete before/at time t and release their next jobs on/after $J_{i,j}$'s deadline $d_{i,j}$. We can see that, on each core in set $\mathbb{O}$, no job will access resource $\mathcal{R}_r$ during the execution of $z_{i,x}$, and thus the slowdown of $z_{i,x}$ cannot affect the schedulability of any job on any core in set $\mathbb{O}$ from Hypothesis 1.

In case $\mathbb{O} = \mathbb{Z}$ (i.e., no task on cores in $\mathbb{Z}$ will be affected by $z_{i,x}$), there is no other constraint for s^{c2}_{limit} as well. For the case of $\mathbb{Z} \neq \mathbb{O}$, it is possible for a job on a core in set $(\mathbb{Z} - \mathbb{O})$ being affected by $z_{i,x}$. Here, if such a job waits for $J_{i,j}$'s critical section $z_{i,x}$, it will need to wait for all jobs in Φ_r. Thus, based on the waiting time defined in Equation 8, the other upper-bound of s^{c2}_{limit} is

$$\frac{f_{max}}{S_s} \cdot (\sum_{\mathcal{O}_m \in \mathbb{O}} tp^{max}_m(\mathcal{R}_r) + \sum_{\mathcal{O}_y \in \mathbb{Y}} (tp^{max}_y(\mathcal{R}_r) - c_{cs}(z_{q,c}))) \quad (48)$$

where $z_{q,c}$ is the current critical section of the job on core $\mathcal{O}_y$ to access resource $\mathcal{R}_r$.

The first part in Equation 48 represents the slack generated due to the fact that on any core in set $\mathbb{O}$, no job will access resource $\mathcal{R}_r$ during the execution of critical section $z_{i,x}$; while the second part corresponds to the slack generated due to shorter time for accessing resource $\mathcal{R}_r$ on each core in set $\mathbb{Y}$ than the worst case estimation.

Finally, the amount of slack that can be safely reclaimed by $J_{i,j}$'s critical section $z_{i,x}$ can be obtained as

$$s_{safe} = \min\{trs(J_{i,j}, t), s^{c1}_{limit}, s^{c2}_{limit}\} \quad (49)$$

Then, similar to Section 5.3, the feasible speed for $z_{i,x}$, the expected frequency for core $\mathcal{O}_k$ and the real processing frequency of the VFI G_g can be set accordingly. Once the execution of $z_{i,x}$ completes, the slack release and preservation policies discussed in Section 5.4 can be applied as well except that no slack is needed to be preserved for $z_{i,x}$.

5.7 An Example Execution of SA-DVFS

In what follows, we illustrate how SA-DVFS works by considering one actual execution of the tasks in Figure 2 discussed in Section 4.3. We assume that each job takes half of its WCET except that the non-critical section of job $J_{5,1}$ takes 1 unit at the maximum speed. Note that, the cores $\mathcal{O}_1$, $\mathcal{O}_2$ and $\mathcal{O}_3$ constitute one VFI G_1 and $S_s = 0.8$.

As described in Section 5.1, the wrapper-jobs corresponding with dummy tasks on cores $\mathcal{O}_1$, $\mathcal{O}_2$ and $\mathcal{O}_3$ are respectively $(1.125, 10)$, $\emptyset$ and $(2.5, 10)$ at time 0 .

At time 0, $FS^{ns}_{3,1}(0)$ is $\frac{1}{1/0.8+2.5}$ by Equation 35 and is set as 0.3 subject to discrete frequencies. Then, $Sk_{rec} = \frac{1}{0.3} - \frac{1}{0.8} = 2.0833$ by Equation 36. Similarly, we can obtain $FS^{ns}_{1,1}(0) = 0.8$, $FS^{ns}_{5,1}(0) = 0.6$, and the corresponding Sk_{rec} of jobs $J_{1,1}$ and $J_{5,1}$ are respectively 0 and 0.875. Thus, the shared frequency F_1 is 0.8 at time 0 and the wrapper-jobs are adjusted as shown in Figure 3.

At time 0.3125, $J_{3,1}$ issues a request for resource $\mathcal{R}_2$. $c^{rem}_{ns}(J_{3,1}, 0.3125)$ is 0.75 $(1-0.8 \cdot 0.3125)$ and its Sk_{rel} due to early released slack is calculated as 0.5208 $((\frac{0.8}{0.3} - 1) \cdot 0.3125)$ from Equation 37. Moreover, the amount of slack borrowed during $[0, 0.3125]$ on $\mathcal{O}_3$ is 0.3125 $(\min\{0.3125, 0.3125-0\})$ from Equation 38 and returned without changing the corresponding wrapper-job. Thus, the size of wrapper-job on core $\mathcal{O}_3$ is 0.9375 $(0.4167 + 0.5208)$. Similarly, Sk_{rel} of jobs $J_{1,1}$ and $J_{5,1}$ are 0 $((\frac{0.8}{0.8} - 1) \cdot 0.3125)$ and 0.1042 $((\frac{0.8}{0.6} - 1) \cdot 0.3125)$. Thus, the size of wrapper-job on $\mathcal{O}_1$ is 0.3542 units $(0.1042 + 0.25)$.

When $J_{3,1}$ enters the queue of $\mathcal{R}_2$ at time 0.3125, it immediately holds $\mathcal{R}_2$. Since $\frac{BW(z_{3,1})}{S_s}$ is 1.25 as shown in Table 2, the available slack time on $\mathcal{O}_3$ is 2.1875 units $(1.25+0.9375)$ now from Equation 42. We can have $s^{c1}_{limit} = +\infty$ $(\Psi^3_3 = \emptyset)$ and $s^{c2}_{limit} = \frac{2-1}{0.8} = 1.25$ from Equation 48 ($\Phi_r = \emptyset$, $\mathbb{Z} = \{\mathcal{O}_2\}$ and $\mathbb{O} = \emptyset$). Thus, s_{safe} is 1.25 by Equation 49 and the feasible speed for $J_{3,1}$ to execute $z_{3,1}$is 0.4 $(\frac{1}{\frac{1}{0.8}+1.25})$ from Equation 35. In the end, the available slack time on $\mathcal{O}_3$ is 0.9375 units $(2.1875 - 1.25)$ at time 0.3125.

At time 0.375, $J_{1,1}$ issues a request to access resource $\mathcal{R}_2$ that is currently held by $J_{3,1}$. We can have that Sk_{rel} of jobs $J_{3,1}$, $J_{1,1}$ and $J_{5,1}$ are respectively 0.0625 $((\frac{0.8}{0.4} - 1) \cdot (0.375 - 0.3125))$, 0 and 0.0208 $((\frac{0.8}{0.6} - 1) \cdot (0.375 - 0.3125))$ from Equation 37. Thus, the amount of available slack on $\mathcal{O}_3$ is 1 unit and that on $\mathcal{O}_1$ is 0.375 units now.

When $J_{6,1}$ is pre-executed by our slack stealing policy at time 0.375, the frequency of the VFI is set as 0.6

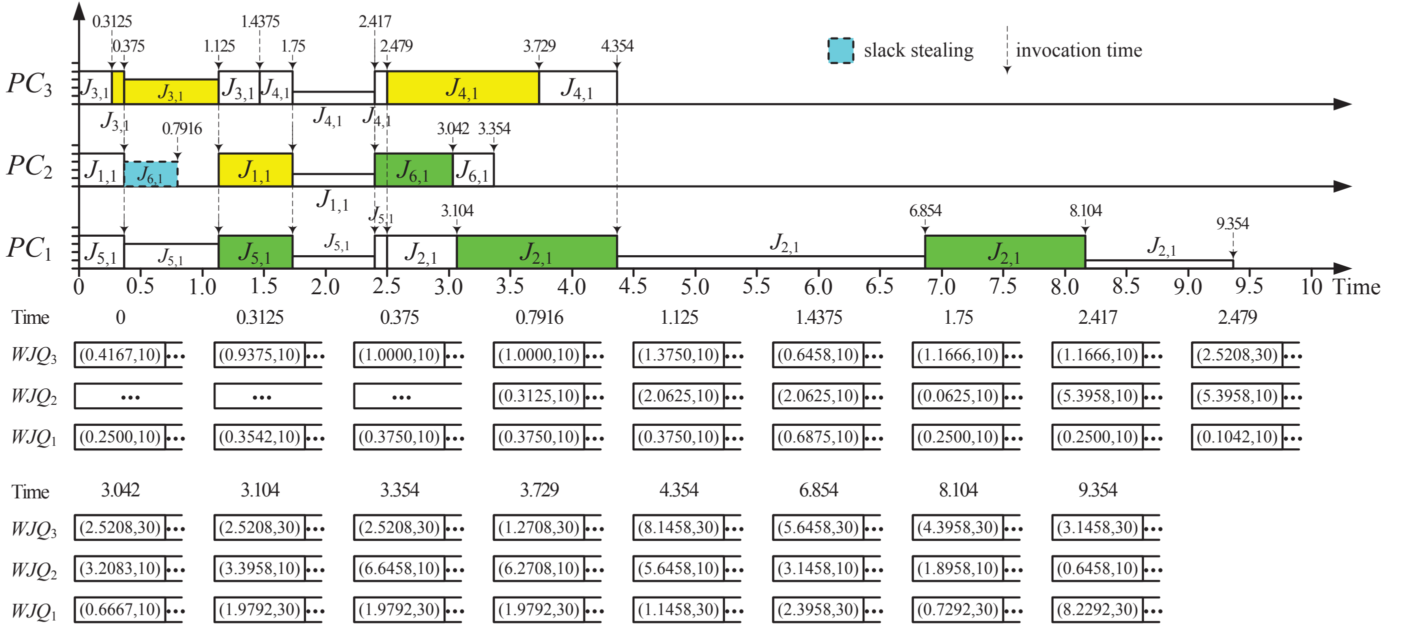


Fig. 3: The execution of the first job for each task.

($\max\{0.6, 0.1, 0.4\}$) as the expected running frequency of $\mathcal{O}_2$ is set as 0.1 (i.e., f_1) by our approach.

At time 0.7916, when $J_{6,1}$ intends to access $\mathcal{R}_1$, it is suspended subject to Rule 3 (see Section 3.3). By our slack stealing policy for pre-execution, the size of slack stolen by $J_{6,1}$ is $0.6 \cdot \frac{0.7916-0.375}{0.8} = 0.3125$ from Equation 40. Then, a wrapper job with size of 0.3125 units and deadline of 10 units (i.e., the deadline of $J_{6,1}$) is created and added to WJQ_2.

At time 1.125, job $J_{3,1}$ completes executing $z_{3,1}$ and releases resource $\mathcal{R}_2$ on core $\mathcal{O}_3$. The amount of early released slack is 0.375 units ($(\frac{0.6}{0.4} - 1) \cdot (1.125 - 0.375)$) from Equation 37, while the amount of slack due to early completion of critical section $z_{3,1}$ is 1.25 units ($\frac{0.5}{0.4}$) from Equation 39. In addition, 0.75 units of slack ($\min\{1.125 - 0.375, 1\}$) is borrowed and returned. Now, the amount of available slack on $\mathcal{O}_3$ at time 1.125 is 2.625 units ($1 + 0.375 + 1.25$). Then, the feasible speed of non-critical section of $J_{3,1}$ is re-set as 0.146 ($\frac{0.75}{\frac{0.75}{0.3}+2.625}$) and thus 0.2 due to discrete frequency levels. In the end, the amount of slack reclaimed by $J_{3,1}$ is 1.25 units ($0.75 \cdot (\frac{1}{0.2} - \frac{1}{0.3})$) from Equation 36 and thus the amount of available slack on $\mathcal{O}_3$ is 1.375 units ($2.625 - 1.25$).

Job $J_{1,1}$ on core $\mathcal{O}_2$ holds resource $\mathcal{R}_2$ at time 1.125. We can see that 1.75 units ($2.5 - (1.125 - 0.375)$) of slack is released from Equation 42. The slack with 0.3125 units stolen by $J_{6.1}$ is borrowed during [0.7916, 1.125] and is returned at time 1.125 (see Equation 41). Thus, the amount of available slack on $\mathcal{O}_2$ is 2.0625 ($0.3125 + 1.75$) units now. When $J_{1,1}$ attempts to access $\mathcal{R}_2$, we have $s^{c1}_{limit} = +\infty$ ($\Psi^1_2 = \emptyset$) and $s^{c2}_{limit} = \frac{1-1}{0.8} = 0$ from Equation 48 ($\Phi_r = \emptyset$, $\mathbb{Z} = \{\mathcal{O}_3\}$ and $\mathbb{O} = \emptyset$). Thus, s_{safe} is 0 from Equation 49 and the feasible speed for $J_{1,1}$ to execute $z_{1,1}$is 0.8 ($\frac{1}{\frac{1}{0.8}+0}$) from Equation 35. Similarly, the amount of available slack on core $\mathcal{O}_1$ is 0.375 units at time 1.125.

Following above steps, the execution of the first job for each task and the variations of wrapper-jobs on each core at each invocation time are shown in Figure 3.

5.8 Analysis of SA-DVFS

In what follows, we analyze SA-DVFS with respects to its time complexity and schedulability.

Complexity of SA-DVFS: At each invocation time, at most nt jobs arrive concurrently due to the periodic task arrival patterns. Recall that the slack time of each wrapper-job must be depleted at or before this wrapper-job's deadline by the wrapper-job scheme. Thus, for each core, at most nt wrapper-jobs stay in the *WJQ* at any given time, such that building the wrapper-job queue in order of incrementing deadlines requires $O(nt \cdot \log(nt))$ time.

Each invocation implies a possible DVFS synchronization. For each DVFS synchronization, updating the remaining WCETs of all current running jobs takes $O(nc)$ time. Determining the amount of reclaimable slack by a job's non-critical section can be done in $O(nt)$ time. It takes our slack release and preservation policies $O(1)$ time to calculate the amount of slack released, borrowed and returned by a job's non-critical section. Also, it takes our dynamic scheme $O(1)$ time to obtain the feasible speed of a job's non-critical section. After that, the adjustment of corresponding wrapper-jobs can be done in $O(nt)$ time.

For each slack stealing for pre-execution, it requires $O(1)$ time calculating the amount of stolen slack and $O(nt)$ time handling such slack in the *WJQ*. When a job intends to access a resource, the calculation of s^{c1}_{limit} requires $O(nt)$ time and that of s^{c2}_{limit} for jobs in set Φ_r requires $O(nc)$ time. Finding out the core sets $\mathbb{O}$ and $\mathbb{Z}$ can be done in $O(nt)$ time. Thus, our constrained slack allocation policy requires $O(nt)$ time for slowing down a critical section assuming that $nc \leq nt$. With

the VFI limitation being considered, the shared frequency of a VFI can be determined in $O(nc)$ time.

Therefore, the time complexity of our SA-DVFS can be found as $O(nt \cdot \log(nt))$ at each invocation time, which is the same as that of the original wrapper-job scheme.

Schedulability of SA-DVFS with Uniform Frequency: We first prove that our SA-DVFS with uniform initial scaled frequency satisfies the schedulability of all jobs. Note that, based on the extended wrapper-job slack management mechanism, the slack management policies of SA-DVFS discussed in the above sections do not introduce any additional workload on each core.

Theorem 1. *For a set of periodic tasks that access shared resources, when the tasks are executed on a VFI-based multi-core system under the partitioned-EDF scheduling with the suspension-based resource access protocol, SA-DVFS with uniform initial scaled frequency ensures that there is no deadline miss at runtime.*

Proof: Following the assumptions and notations in Lemma 1, we assume that $J_{m,n}$ executes its critical section $z_{m,y}$ during $[t', t]$ on $\mathcal{O}_a$. Suppose that the amount of slack generated by shorter time for $J_{m,n}$ to acquire resource $s(z_{m,y})$ is $\triangle_{m1}$. After $J_{m,n}$ completes accessing $s(z_{m,y})$, the amount of its reclaimed slack and released slack are denoted by ∇_m and $\triangle_{m2}$, respectively. Here *released slack* refers to those due to early completion, early released slack, the stolen slack for pre-execution and less than worst case synchronization overhead, but it excludes the returned slack due to borrowed slack. Thus, the maximum execution time of $J_{m,n}$ in $[t', t]$ is:

$$\frac{BW(z_{m,y}) + c_{cs}(z_{m,y})}{S_s} \cdot f_{max} - \triangle_{m1} + \nabla_m - \triangle_{m2}$$

By constraints C1 and C2 (see Section 5.6), we can obtain:

$$\frac{BW(z_{m,y}) + c_{cs}(z_{m,y})}{S_s} \cdot f_{max} - \triangle_{m1} + \nabla_m \le \frac{B_i \cdot f_{max}}{S_s} \tag{50}$$

We define two variables, ∇_x and $\triangle_x$, to denote the amount of reclaimed slack and released slack by task T_x's jobs pertaining to set $\mathbb{A}$ during $[t', t]$, respectively. From Hypothesis 1, we can have $\nabla_x \ge 0$ and $\triangle_x \ge 0$ for each task T_x. Moreover, for each job, its feasible speed is initialized to S_s when it arrives by our SA-DVFS. Since $J_{i,j}$ misses its deadline at time t, the total processor demand in $[t', t]$ exceeds X. Following the proof of Lemma 1, there is

$$\frac{BW(z_{m,y}) + c_{cs}(z_{m,y})}{S_s} \cdot f_{max} - \triangle_{m1} + \nabla_m - \triangle_{m2} + \sum_{\forall T_k \in \Psi_a}^{p_k \le p_i} \left(\nabla_k - \triangle_k + \frac{f_{max}}{S_s} \cdot (c_k + BW_k) \cdot (\lfloor \frac{X - p_k}{p_k} \rfloor + 1) \right) > X$$

Incorporating Equation 50 into above inequality, we have

$$\sum_{\forall T_k \in \Psi_a}^{p_k \le p_i} (\nabla_k - \triangle_k) > X + \triangle_{m2} - \frac{f_{max}}{S_s} \cdot \left(B_i + \sum_{\forall T_k \in \Psi_a}^{p_k \le p_i} (c_k + BW_k) \cdot (\lfloor \frac{X - p_k}{p_k} \rfloor + 1) \right) \tag{51}$$

Suppose that at time t', the amount of slack associated with those wrapper-jobs, whose deadlines are no later than t, is $\triangle'$ ($\triangle' \ge 0$). We indicate the amount of reclaimable slack for all jobs in set $\mathbb{A}$ by $\triangle''$. At time t', it is easy to get $\triangle'' = \triangle'$.

From the wrapper-job scheme, the wrapper-job corresponding with the slack generated by $J_{m,n}$ has a deadline later than t. This means that each job in set $\mathbb{A}$ cannot reclaim the slack that is released or returned by $J_{m,n}$. Note that, $J_{m,n}$ can reclaim and borrow the slack from the wrapper-jobs with deadlines sooner than or equal to time t. Therefore, when $J_{m,n}$ completes accessing resource $s(z_{m,y})$, we can get $\triangle'' \le \triangle'$ subject to the wrapper-job scheme.

For any job in set $\mathbb{A}$, it arrives at or after time t' and it is required to finish before or at time t. Therefore, all wrapper-jobs corresponding with the slack, which is reclaimed, released as well as returned by this job, have deadlines sooner than or equal to t from the wrapper-job scheme. Since core $\mathcal{O}_a$ is never idle (i.e., there exists ready jobs) during $[t', t]$, no slack is wasted subject to the wrapper-job scheme. Then, by incorporating the wrapper-job scheme and Lemmas 3-6, during the execution of jobs in set $\mathbb{A}$, the total reclaimable slack either has been reclaimed by them before time t or is being borrowed by $J_{i,j}$ at time t . The amount of such slack is $\triangle'' + \sum_{\forall T_k \in \Psi_a}^{p_k \le p_i} \triangle_k$ after $J_{m,n}$ releases resource $s(z_{m,y})$. Hence, we can have:

$$\triangle'' + \sum_{\forall T_k \in \Psi_a}^{p_k \le p_i} \triangle_k \ge \sum_{\forall T_k \in \Psi_a}^{p_k \le p_i} \nabla_k$$
$$\Rightarrow \sum_{\forall T_k \in \Psi_a}^{p_k \le p_i} (\nabla_k - \triangle_k) \le \triangle'' \le \triangle'$$

As $\triangle_{m2} \ge 0$ is true by Hypothesis 1 and constraints C1 and C2, combining above inequality with Equation 51, there is

$$\triangle' + \frac{f_{max}}{S_s} \cdot \left(B_i + \sum_{\forall T_k \in \Psi_a}^{p_k \le p_i} (c_k + BW_k) \cdot (\lfloor \frac{X - p_k}{p_k} \rfloor + 1) \right) > X$$

which contradicts with the feasibility of wrapper-job scheme subject to the definition of $\triangle'$ and the proof of Theorem 1. □

SA-DVFS with Different Initial VFI Frequencies: Note that, as the foundation of our dynamic energy management framework, the wrapper-job scheme enables to perform the slack management on a per-job and per-core basis. Therefore, with the different initial frequencies for VFIs being considered, our proposed policies (slack reclamation for non-critical sections, slack release and preservation, and slack stealing for pre-execution) can be applied with only some minor modifications.

That is, for a job $J_{i,j}$ on core $\mathcal{O}_k \in G_g$, its initial feasible speed (including non-critical and critical sections) is set as S_{G_g} (see Section 4.4) when it arrives. As these policies

together with dummy task, whose size is determined by the initial speed of the core that it is on, do not introduce any synchronization overhead to jobs, they can be *safely* performed without violating time constraints.

On the other hand, the constrained slack allocation policy for critical sections (see Section 5.6) is strongly dependent on the upper-bound of a job's synchronization overhead that is determined at off-line phase. Therefore, this policy can also be applicable with the following modifications.

Recall that, with different initial frequencies for VFIs, the worst-case synchronization overhead of a job $J_{i,j}$ (e.g., BW_i and B_i) has been scaled down (see Equations 21 to 25; Section 4.1). First, the two constraints C1 and C2 (see Section 5.6) are re-expressed as:

- **C1:** For any job $J_{a,b}$ on $\mathcal{O}_k$ that can be blocked by $J_{i,j}$, its actual blocking time needs to be bounded by B_a;
- **C2:** For any job $J_{e,l}$ that has a critical section $z_{e,y}$ to access the resource $\mathcal{R}_r$ on other cores, its actual waiting time for the execution of $J_{i,j}$'s critical section $z_{i,x}$ needs to be bounded by $BW(z_{e,y})$.

Recall that the VFI to which core $\mathcal{O}_k$ belongs is denoted by $G(\mathcal{O}_k)$ in Section 4.1. To satisfy the constraints C1 and C2, Equations 43 to 46 and 48 are respectively adjusted as:

$$BW^{act}(z_{i,x}) + \frac{c_{cs}(z_{i,x}) \cdot f_{max}}{S_{G_g}} + s^{c1}_{limit} \leq B_m \tag{52}$$

$$s^{c1}_{limit} = \min\{B_m - \frac{f_{max} \cdot c_{cs}(z_{i,x})}{S_{G_g}} - BW^{act}(z_{i,x})| \\ \forall T_m \in \Psi^i_k\} \tag{53}$$

$$BW^{exp}(z_{q,c}) = \left(\frac{c_{cs}(z_{i,x})}{S_{G_g}} + \sum_{J_{b,l} \in \Phi_r}^{t_b < t_q} \frac{c_{cs}(z_{b,e})}{S_{G(\mathcal{O}(T_b))}} \right) \cdot f_{max} \\ + t - t_q \tag{54}$$

$$s^{c2}_{limit} = \min\{BW(z_{q,c}) - BW^{exp}(z_{q,c}) | \forall J_{q,a} \in \Phi_r\} \tag{55}$$

$$\sum_{\mathcal{O}_m \in \mathbb{O}} tp^{max}_m(\mathcal{R}_r) + \sum_{\mathcal{O}_y \in \mathbb{Y}} \left(tp^{max}_y(\mathcal{R}_r) - \frac{c_{cs}(z_{q,c}) \cdot f_{max}}{S_{G(\mathcal{O}_y)}} \right) \tag{56}$$

Following the similar reasonings as Theorem 1, we can obtain the following theorem.

Theorem 2. *When VFIs have different initial frequencies, SA-DVFS scheme ensures that all jobs can finish in time at runtime.*

6 Simulation Results and Discussions

In this section, we evaluate the performance of our new synchronization-aware energy management schemes through extensive simulations. For such purpose, we designed a discrete time simulator in C++, where both static SA-WFD and dynamic SA-DVFS schemes were implemented.

6.1 Experimental Settings

For multicore processors, we consider the one with 65nm technology that has been adopted in other recent work [46]. Here, we assume that there are six supply voltage and frequency levels for the processor: (0.6V, 0.78GHz), (0.7V, 1.27GHz), (0.8V, 1.81GHz), (0.9V, 2.42GHz), (1.0V, 3.08GHz) and (1.1V, 3.8GHz). Regarding to the parameters in the power model, we use the same values as those in [46]: $L_g = 4 \times 10^6$, $V_{bs} = -0.7$, $K_3 = 5.38 \times 10^{-7}$, $K_4 = 1.83$ and $K_5 = 4.19$.

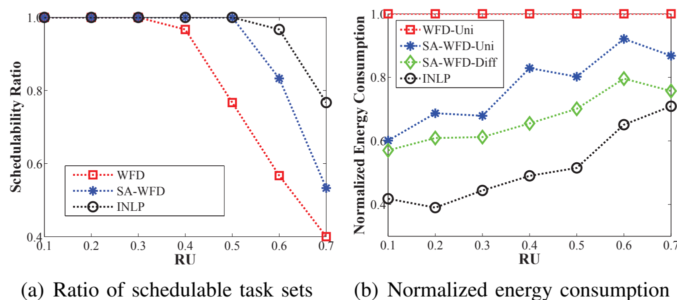
There are many factors that can affect the performance of the schemes under consideration. In this work, we vary the following parameters regarding to the system as shown in Table 4: the number of cores nc, the number of shared resources (nr), the number of cores per VFI (denoted as $NCPI$) and the number of discrete voltage/frequency levels ($DVFL$). For tasks, they are specified by the average system raw utilization (RU) that indicates the available static slack, the actual-to-worst case execution time ratio (AWR) that denotes the variability of tasks' actual workload (and thus the available dynamic slack), the number of critical sections in a task (N_{cs}) and the critical section ratio (CSR). Please be noted that, RU excludes the additional utilization due to task synchronization. Here, the degree of resource contention for the tasks in the systems is determined by N_{cs}, CSR as well as nr.

TABLE 4: System parameters for the simulations

Parameters	Values/ranges
Number of cores (nc)	4, 16, 32
Number of cores/VFI ($NCPI$)	1, 2, 4, 8, 16, 32
Number of V/F levels ($DVFL$)	3, 4, 5, 6
Number of resources (nr)	[1,10]
System raw utilization (RU)	$[0.1, 0.5]$
Number of tasks (nt)	[40, 120]
Period of tasks	[50, 200], [200, 500], [500, 2000]
AWR	$0.1 - 1.0$
CSR	$0.003 - 0.03$
Number of critical sections N_{cs}	[1, 8]

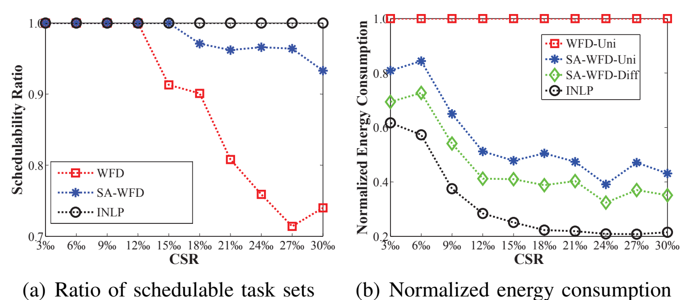
In the simulations, the synthetic task sets are generated from the above parameters as follows. For given nc, nt and RU, the utilization of a task T_i is set as $au_i = \frac{RU \cdot nc}{nt}$. Then, the task's period is randomly selected from one of the three types of periods in Table 4. Next, the value of c_i is obtained uniformly in the range of $[0.2 \cdot p_i \cdot au_i, 1.8 \cdot p_i \cdot au_i]$. The number of critical sections ($N_{cs}(T_i)$) is randomly obtained within the range of $[1, 8]$, followed by resource selection for each critical section. The execution time of a critical section is generated randomly within $[\frac{0.2 \cdot c_i \cdot CSR}{N_{cs}(T_i)}, \frac{1.8 \cdot c_i \cdot CSR}{N_{cs}(T_i)}]$. In the end, the execution time of non-critical section is obtained and the relative location of critical sections is randomly chosen. At run time, the actual execution time of non-critical and critical sections of tasks can be randomly determined in a similar way according to AWR.

Without specified otherwise, the default values for the parameters are: $nc = 16$, $NCPI = 2$, $DVFL = 6$, $RU = 0.25$, $AWR = 0.3$ and $CSR = 0.009$. For the results reported below, each data point corresponds to the average results of 1000 task sets.



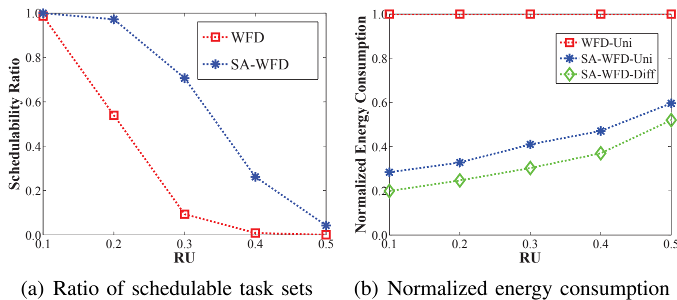
(a) Ratio of schedulable task sets (b) Normalized energy consumption

Fig. 4: Performance of static schemes with varying RU for small problems.



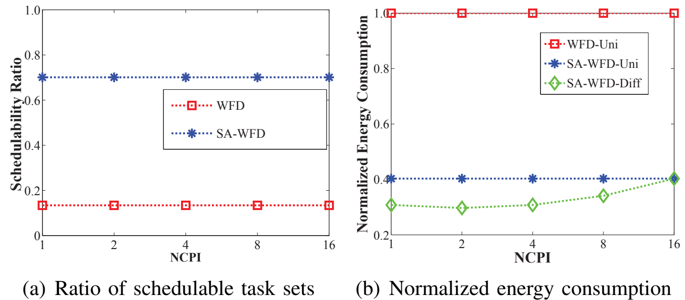
(a) Ratio of schedulable task sets (b) Normalized energy consumption

Fig. 5: Performance of static schemes with varying CSR for small problems.



(a) Ratio of schedulable task sets (b) Normalized energy consumption

Fig. 6: Performance of static schemes with varying RU for large problems.



(a) Ratio of schedulable task sets (b) Normalized energy consumption

Fig. 7: Performance of static schemes with varying $NCPI$ for large problems.

6.2 Performance of SA-WFD and Static schemes

Now we evaluate the performance of three static schemes (original WFD, SA-WFD and INLP) in terms of energy consumption and schedulability ratio. Here, schedulability ratio is defined as the rate of the number of schedulable task sets over the total number of task sets considered. Following the formalization of task mapping problem as described in Section 4.1, an INLP (Integer Non-Linear Programming) approach is implemented by Lingo tool [53] to find the optimal task-to-core mapping for small scale problems (with $nc = 4, NCPI = 2$ and $nt = [8, 15]$). Moreover, for comparison, we also implemented the conventional WFD task mapping scheme based only on task utilization.

Small scale problems: The results are shown in Figures 4 and 5, where each data point represents the average result of 100 task sets.

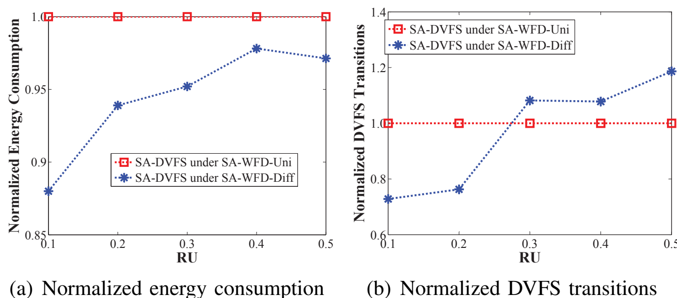
The results show that, although WFD can obtain well-balanced workload among the cores when task synchronization is ignored, the schedulability of tasks sharing resources also depends on tasks' blocking and waiting time and quickly deteriorates with increasing utilization RU. By assigning tasks that access similar resources to the same core, SA-WFD effectively incorporates the synchronization overhead during task mapping process, and thus leads to much better schedulability of tasks compared to that of WFD. Not surprisingly, INLP obtains the best schedulability result for all cases.

Moreover, for the schedulable task sets, Figure 4(b) further shows the normalized energy consumption, where the one of WFD with uniform frequency is used as the baseline (denoted as WFD-Uni). The static scheme with uniform scaled frequency for all VFIs based on SA-WFD mapping is denoted as SA-WFD-Uni, while the one with different frequencies is denoted as SA-WFD-Diff. We can see that, by assigning a different frequency for each VFI based on the workload of its cores, SA-WFD-Diff can obtain up to 40% energy savings when compared to that of WFD-Uni. Moreover, with each VFI having its individual lower frequency, SA-WFD-Diff can save up to 20% more energy compared to SA-WFD-Uni. Compared to the optimal solution of INLP, which also assigns individual frequency for each VFI, the energy savings obtained by SA-WFD-Diff is about 20% less. The difference generally becomes less when RU increases since there is less chance for energy management.

Figure 5 shows the impacts of CSR on the performance of the static schemes. Note that, both INLP and SA-WFD incorporate the task synchronization overhead into the task-to-core mapping. Therefore, when CSR becomes larger (i.e., stronger task synchronization requirements), they generally have better schedulability ratio (see Figure 5(a)) and better energy savings (see Figure 5(b)) compared to conventional WFD. As explained above, SA-WFD-Diff consumes less energy than SA-WFD-Uni due to lower static frequency setting.

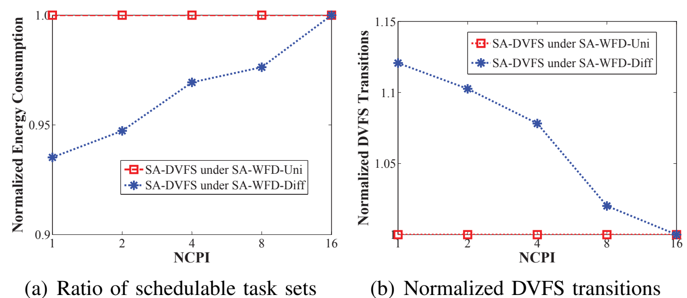
Large scale problems: Here, we set experimental parameters to default and evaluate the impacts of RU and $NCPI$ on the performance of SA-WFD and original WFD.

As shown in Figure 6, the schedulability of tasks under both WFD and SA-WFD becomes very low when RU is close to 0.5 due to increased synchronization overhead with more tasks and cores. Note that, the raw utilization RU does not include synchronization overhead. Moreover, compared to SA-WFD-



(a) Normalized energy consumption (b) Normalized DVFS transitions

Fig. 8: Performance of SA-DVFS under different static schemes with varying RU.



(a) Ratio of schedulable task sets (b) Normalized DVFS transitions

Fig. 9: Performance of SA-DVFS under different static schemes with varying $NCPI$.

Uni, the additional energy savings obtained by SA-WFD-Diff is similar to that of small scale problems.

Since both the conventional WFD and SA-WFD do not consider the VFI limitation during the task-to-core mapping, the parameter $NCPI$ plays no role in the schedulability rate of these two static schemes as depicted in Figure 7(a). Also, it does not affect the energy performance of original WFD and SA-WFD-Uni due to uniform scaled frequency setting as shown in Figure 7(b). When all cores constitute a single VFI (i.e., $NCPI = 16$), SA-WFD-Diff is exactly SA-WFD-Uni from Algorithm 2 and thus the same energy is consumed. While when each VFI owns only one processing core (i.e., $NCPI = 1$), SA-WFD-Diff obtains the best energy savings due to assigning independent frequency to each core.

6.3 Performance of online SA-DVFS

Next, we evaluate the performance of online SA-DVFS based on SA-WFD scheme. For comparison, we extended the following energy management schemes that have been designed for tasks with shared resource in uniprocessor systems to the multicore setting: EDF-based USFI [35], EDF-based DSDR [54] and RM-based FL-PCP [17]. Based on the feasible SA-WFD task-to-core mapping, each of the above schemes assumes the worst case synchronization cost of tasks and manages them on each core accordingly. The detailed modifications include:

- The common modification is incorporating the worst-case global waiting times, which are based on our adopted locking protocol, into the execution times of tasks to account for global resource contention. After the feasible speeds of tasks have been determined by these schemes on a per-core basis, for any resource, the highest speed among all tasks that intend to access this resource is calculated first; next, such speed is set as the speed for accessing this resource by any task. By this, for any task, both the global waiting time for any resource and local blocking time can be upper-bounded.
- Further, as for the DSDR, each processing core is associated with a high speed and a low one. The high speed is chosen on each core when blocking arises; otherwise, the low one is applied. Considering the global resource access, for each core, its high speed is normalized to the maximum of all cores' high speeds to meet deadlines.

By means of above modifications, these four schemes can be evaluated on the same baseline, i.e., the same task-to-core assignment and feasibility guarantee.

6.3.1 Impacts of different static frequency setting

Here, we evaluate the impacts of different static frequency setting (e.g., SA-WFD-Uni and SA-WFD-Diff) on the online energy savings. SA-WFD-Diff can be easily incorporated into our dynamic energy management framework (online SA-DVFS) as stated in Section 5.8.

Figure 8 shows the affects of RU on the performance of two static frequency setting schemes. Compared to Figure 6, although SA-WFD-Diff shows better energy saving than SA-WFD-Uni at off-line stage, the gap between them is significantly shrunk and almost disappears when RU becomes larger.

Recall that, using spare utilizations of cores under the wrapper-job scheme, the dummy task can generate static slack periodically (which is transformed to dynamic slack). Note that, SA-WFD-Diff can effectively reduce the processing frequencies of cores at off-line stage, resulting in higher utilization and thus less static slack than SA-WFD-Uni. This implies that SA-DVFS under SA-WFD-Uni scheme may reclaim more dynamic slack than that under SA-WFD-Diff as our SA-DVFS can effectively exploit dynamic slack to improve energy savings. Moreover, when the system utilization is very small, SA-DVFS can select the lowest frequency for the two schemes. This also helps SA-DVFS under SA-WFD-Uni reduce energy consumption even with high static frequency.

More importantly, the low processing frequency setting of SA-WFD-Diff can extend the execution time of a critical section (and thus local blocking time and global waiting time of a job). Our constrained slack allocation for critical sections are strongly dependent on the static frequency of a job, in order to allocate more slack to access resources under schedulability (see Section 5.6). Hence, SA-WFD-Diff may reclaim less slack than SA-WFD-Uni when accessing resources due to lower static frequency setting.

When the system utilization is high and the dynamic slack is medium (i.e., $AWR = 0.3$), a job may execute a critical section at a high speed under SA-WFD-Diff scheme. This can result in high shared processing frequency of a VFI due to DVFS synchronization and consequently relatively high energy consumption (and more DVFS transitions).

Figure 9 exhibits the affects of $NCPI$ on the performance of SA-DVFS under the two static schemes. When a VFI contains only one core, SA-DVFS under SA-WFD-Diff can obtain the best energy savings due to low static frequencies, no DVFS synchronization and our effective slack management policies as shown in Figure 9(a). As explained above, SA-WFD-Diff may lead to a high processing frequency when accessing a resource. Therefore, as depicted in Figure 9(b), it leads to more voltage/frequency switching than SA-WFD-Uni due to non-uniform frequencies on each core.

6.3.2 Impacts of RU

First, we evaluate the impacts of system utilization. Here, we consider the cases with $RU \leq 0.5$ to obtain schedulable task sets. Figure 10(a) shows the normalized energy consumption of the schemes with that of SA-DVFS being used as the baseline. Here, we can see that SA-DVFS always consumes less energy compared to the existing schemes as it can exploit more slack due to its synchronization-awareness. When system utilization is low (e.g., $RU = 0.1$), the energy savings obtained by SA-DVFS is relatively small. The reason comes from the fact that, due to excessive amount of available slack, all schemes can execute tasks at the lowest speed for most of the time. As system utilization increases, SA-DVFS consumes relatively less energy compared to that of existing schemes as it can effectively exploit both static and dynamic slack through its slack management policies.

In addition, Figure 10(b) shows the normalized number of DVFS transitions incurred by all the schemes. Again, that of SA-DVFS is used as the baseline. When system utilization is low, SA-DVFS is able to operate all cores at a relatively uniform frequency with proper slack management and results in less number of DVFS transitions. When RU becomes larger, FL-PCP results in less number of DVFS transitions compared to other schemes due to its fixed-priority setting and well-designed approaches to reduce DVFS overhead. Overall, we can see that the number of DVFS transitions incurred by SA-DVFS is comparable to those of existing schemes.

6.3.3 Impacts of AWR

To evaluate the impact of dynamic slack due to early completion of tasks, Figure 11(a) shows the normalized energy consumption with varying AWR and fixed system utilization $RU = 0.25$. In general, for smaller values of AWR, most slack can be effectively exploited by SA-DVFS, which leads to relatively better energy savings. However, when $AWR \leq 0.3$, the energy efficient frequency and common frequency constraint for cores on a VFI limit further slack reclamation by SA-DVFS and there is no change in energy savings.

Observe from Figure 11(b) that, compared to other schemes, the number of DVFS transitions by our SA-DVFS usually increases with augmenting AWR. For other schems, a small AWR will lead to too early completion of jobs and in turn frequent switching of processing cores (or VFIs) from sleep to active mode, and vice versa, which intuitively increases the number of DVFS transitions. In contrast, our scheme tries to extend the execution of jobs by fully exploiting available slack, thereby reducing the chance of voltage/frequency switching. Even when $AWR = 1$, our scheme still can reclaim static slack due to dummy tasks to conserve energy and thus results in higher DVFS overhead than other schemes. Moreover, the FL-PCP has the best performance in terms of DVFS overhead in this evaluation when the degree of both task and DVFS synchronization is medium or light.

6.3.4 Impacts of CSR

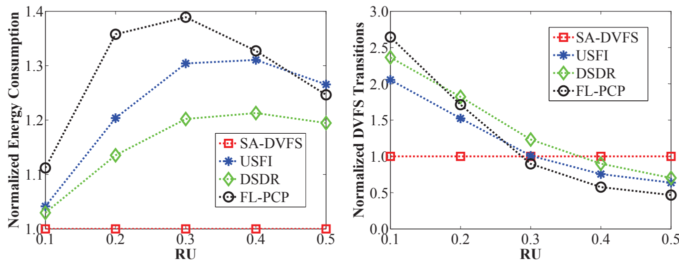
Figure 12(a) further shows the normalized energy consumption of the schemes with varying CSR, which is a key parameter to measure the synchronization requirements of tasks. Here, we can see that, when the value of CSR is in the middle range (i.e., $[0.009, 0.021]$), SA-DVFS performs better and consumes relatively less energy. The reason is that SA-DVFS steals extra slack and allocates appropriate slack to access resources by jobs. With such slack management policy, SA-DVFS can effectively reduce energy consumption under global resource contention and get better energy savings. However, when CSR becomes too large, task synchronization imposes more strict requirements for all schemes and the energy savings obtained by SA-DVFS becomes relatively less. Similarly, when CSR is too small, there is less task synchronization and the advantage of SA-DVFS becomes diminishing and other schemes consume relatively less energy.

As depicted in Figure 12(b), compared to other schemes, less number of DVFS transitions can be obtained by our SA-DVFS with an increasing CSR. Recall that the other schemes either change the speed to access resources (e.g., USFI and FL-PCP) or switch to a pre-determined high speed when blocking occurs (e.g., DSDR) in order to ensure deadlines. With the augment of CSR, the influence of task synchronization on the energy consumption reduction becomes stronger. Thus, the frequent speed adjustment of critical sections by these schemes results in more voltage/frequency switchings.

6.3.5 Impacts of Shared Resource Number (nr)

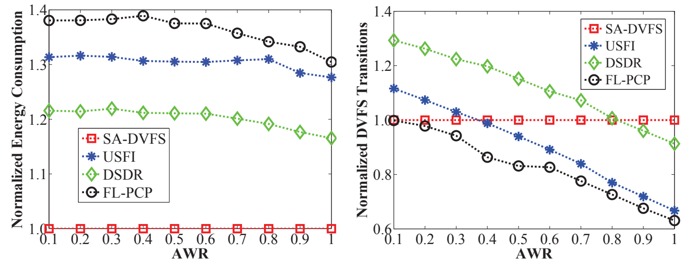
Note that, when the number of critical sections of tasks is fixed, having fewer number of shared resources leads to higher probability of more than one tasks accessing the same resource (i.e., stronger task synchronization requirements). To evaluate the impact of shared resource number (nr) on the energy performance of the schemes, Figure 13(a) shows the normalized energy consumption of the schemes with varying nr. Note that, when there is only one shared resource, all tasks with critical sections will access it, which brings in the strongest synchronization requirements. For this case, SA-DVFS obtains the best energy savings compared to other EDF-based schemes as it takes such synchronization requirements into consideration explicitly. However, for FL-PCP, its energy efficiency seems to be insensitive to the variance of nr. The reason is that FL-PCP strives to avoid speed adjustments when blocking or preemption occurs, which reduces the impacts of nr on its energy consumption with respect to other schemes.

Similarly, in contrast to our scheme, the number of voltage/frequency switchings yielded by USFI and DSDR generally increases with stronger task synchronization as shown in Figure 13(b). Furthermore, the FL-PCP shows its advantages in reducing the DVFS overhead as explained earlier.



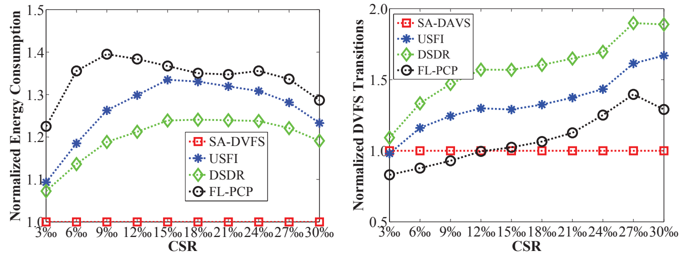
(a) Normalized energy consumption (b) Normalized DVFS transitions

Fig. 10: Performance of the schemes with varying RU.



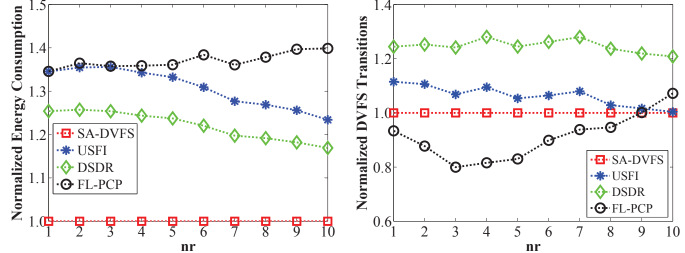
(a) Normalized energy consumption (b) Normalized DVFS transitions

Fig. 11: Performance of the schemes with varying AWR.



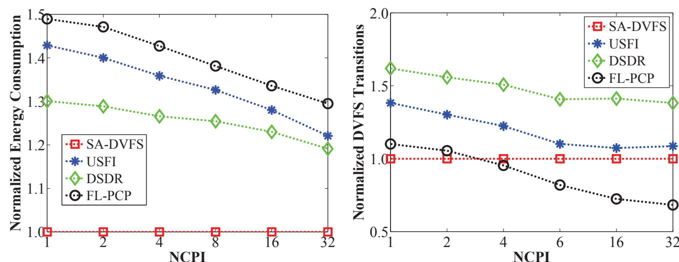
(a) Normalized energy consumption (b) Normalized DVFS transitions

Fig. 12: Performance of the schemes with varying CSR.



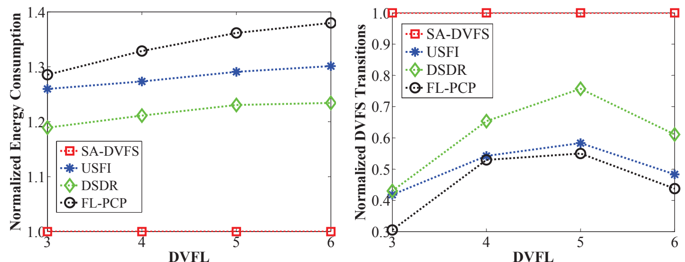
(a) Normalized energy consumption (b) Normalized DVFS transitions

Fig. 13: Performance of the schemes with varying nr.



(a) Normalized energy consumption (b) Normalized DVFS transitions

Fig. 14: Performance of the schemes with varying $NCPI$.



(a) Normalized energy consumption (b) Normalized DVFS transitions

Fig. 15: Performance of the schemes with varying $DVFL$.

6.3.6 Impacts of NCPI

For systems with a 32-core processor (i.e., $nc = 32$) and fixed $RU = 0.25$, Figure 14(a) further illustrates how the number of cores per a VFI affects the energy performance of the schemes. Recall that the cores on one VFI need to share a common voltage/frequency and $NCPI$ essentially reflects the degree of voltage/frequency synchronization in a multicore processor. When each VFI contains only a single core, SA-DVFS leads to the largest energy savings. The reason is that SA-DVFS can flexibly determine the tasks' speeds according to the runtime workload. When all cores are on the same VFI (i.e., global voltage/frequency synchronization), the energy savings obtained by SA-DVFS decreases, but it still outperforms other schemes due to its slack management policies that intend to operate the cores at similar frequencies for energy conservation.

As for the number of DVFS transitions, the similar trend under varying $NCPI$ is exhibited in Figure 14(b). The reasons have been explained earlier and thus we omit the analysis here.

6.3.7 Impacts of DVFL

Now, we test the impact of different voltage/frequency levels on the scheduling performance. Recall that the $DVFL$ accounts for the granularity of DVFS operations. A larger $DVFL$ indicates a finer-grained speed adjustment for jobs, which intuitively gives rise to better energy savings. In this evaluation, the tested three, four, five discrete voltage levels are set as: {0.6V, 0.9V, 1.1V}, {0.6V, 0.8V, 1.0V, 1.1V} and {0.6V, 0.7V, 0.9V, 1.0V, 1.1V}, respectively.

As can be seen from Figure 15(a), finer-grained voltage/frequency levels bring in better energy savings by our scheme. Moreover, this also helps SA-DVFS reduce the number of voltage/frequency switching. It is primarily because that more voltage/frequency options are offered to mitigate the negative affect of DVFS synchronization on both energy savings and DVFD overhead in this case.

Another observation is that undue fine-grained voltage/frequency levels will lead to more DVFS transitions due to more flexible and in turn more frequent DVFS operations by SA-DVFS. Thus, compared to the discrete voltage/frequency

levels, it can be deduced that our SA-DVFS can better energy savings under continuous ones but at the expense of more voltage/frequency switchings.

6.3.8 Impacts of Slack Management Policies

Finally, we evaluate the effects of different slack management policies adopted in SA-DVFS. Compared to SA-DVFS where all policies are deployed, we consider one scheme that adopts only the slack reclamation and preservation policies for non-critical section of tasks, denoted as *Basic Scheme (BS)*. Here, these two schemes are evaluated by varying RU, CSR and $NCPL$. The results are shown in Figures 16 to 18.

From the figures, we can see that SA-DVFS performs generally better than the basic scheme. By deploying the additional slack management policies (stealing and reclamation) for critical-sections of tasks, about 13% more energy can be saved by our SA-DVFS scheme.

As can be found from Figure 16, our propose slack management framework usually gains better energy savings than the BS when the average system utilization is medium. Further, SA-DVFS can achieve better improvement in reducing the number of DVFS transitions as the average utilization is light. The trend under different RU is very similar to that as exhibited in Figure 10 so that we ignore the detailed analysis here.

Figure 17 illustrates the performance comparison between these two schemes as CSR varies. Similar to the trend as shown in Figure 12, our scheme with fully deployed policies has better improvement in both energy conservation and DVFS overhead reduction when the task synchronization is medium.

Figure 18 further depicts the impacts of$NCPI$ on the performance of these two schemes, where $RU = 0.25$ and $nc = 32$. Recall that our synchronization-aware slack management policies are originated from the wrapper-job scheme, which targets uniprocessor systems and performs the DVFS on a per-job basis. Then, both two schemes enable to flexibly reclaim slack to better energy savings when the DVFS synchronization is weak. When the DVFS synchronization requirements become stronger, SA-DVFS with full policies shows better energy performance when comparing to BS. It mainly owes to the effectiveness of our slack stealing policy for pre-execution and constrained slack allocation policy for slowing down critical sections, but it costs these fully deployed policies more frequent speed adjustment as shown in Figure 18(b).

In summary, our SA-DVFS usually outperforms other schemes for uni-processor systems in terms of energy savings at the cost of comparable DVFS overhead. Further, it can be observed that, our scheme with fully deployed policies always achieves better performance than the BS that ignores the synchronization awareness, with regard to both energy consumption and DVFS overhead.

7 Conclusions

Focusing on VFI-based multicore systems with DVFS capability, we study both static and dynamic *synchronization-aware* energy management schemes for a set of periodic realtime tasks that access shared resources. First, we discuss a suspension-enabled extended MSRP resource access protocol and propose a synchronization-aware task mapping heuristic that allocates tasks accessing the same resources to the same core to effectively reduce synchronization overhead. Based on the result task-to-core mapping, static schemes with both uniform and different scale frequencies for VFIs are studied under the partitioned-EDF scheduling. To further exploit dynamic slack for more energy savings, we also propose a set of synchronization-aware slack management policies that can appropriately reclaim, preserve, release and steal slack at runtime to slow down the execution of both non-critical and critical sections of tasks. The scheme considers both common frequency limitation of VFIs and synchronization/timing constraints of tasks. The simulation results show that, the mapping scheme can significantly improve schedulability of tasks. Compared to the existing schemes, our new synchronization-aware schemes can obtain much better energy savings (up to 40% more) with comparable DVFS overhead.

Our future work will focus on more effective and efficient task-to-core mapping strategies by thoroughly analyzing task synchronization between tasks to improve the feasibility of task set. How to improve energy savings by further slowing down critical sections without violating the feasibility motivates our research in the future. Furthermore, we would ameliorate and extend our energy management framework to target other task patterns (e.g., sporadic task arrival) and/or other resource access models (e.g., nested critical sections).

Acknowledgments

This work is supported in part by the National Science Foundation of China under grant number 61173045 and by the US National Science Foundation under grants number CCF-1065448, CNS-1016974 and NSF CAREER Award CNS-0953005. Any opinions, findings, and conclusions as well as recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] IBM Power 7 Overview. http://www.redbooks.ibm.com/redpapers/pdfs/redp4638.pdf.

[2] Intel i7 Processor Specifications. http://www.intel.com/products/processor/corei7/specifications.htm.

[3] T.A. AlEnawy and H. Aydin. Energy-aware task allocation for rate monotonic scheduling. In *Proc. of the IEEE Real Time on Embedded Technology and Applications Symposium*, pages 213–223, 2005.

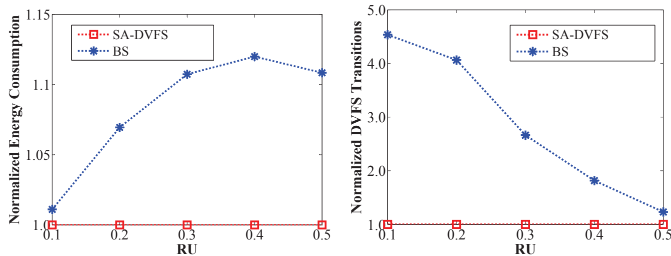
[4] J. Brodt (NEC Electronics America). Revving up with automotive multicore; available at http://www.edn.com/article/ca6528579.html, 2008.

[5] H. Aydin, V. Devadas, and D. Zhu. System-level energy management for periodic real-time tasks. In *Proc. of the $27th$ IEEE Real-Time Systems Symposium (RTSS)*, pages 313–322, 2006.

[6] H. Aydin, R. Melhem, D. Mossé, and P. Mejia-Alvarez. Power-aware scheduling for periodic real-time tasks. *IEEE Trans. on Computers*, 53(5):584–600, 2004.

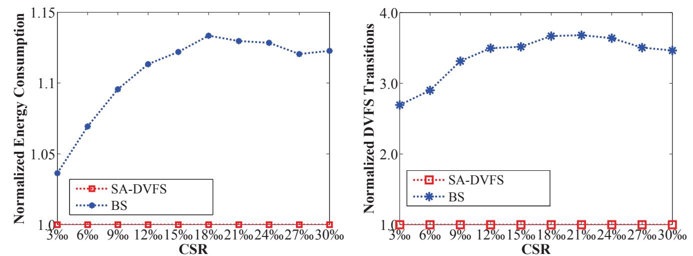
[7] H. Aydin and Q. Yang. Energy-aware partitioning for multiprocessor real-time systems. *Proc. Int'l Parallel and Distributed Processing Symp.*, 2003.

[8] T P. Baker. Stack-based scheduling of real-time processes. *Journal of Real-Time Systems*, 3(1):67–99, Jan. 1991.



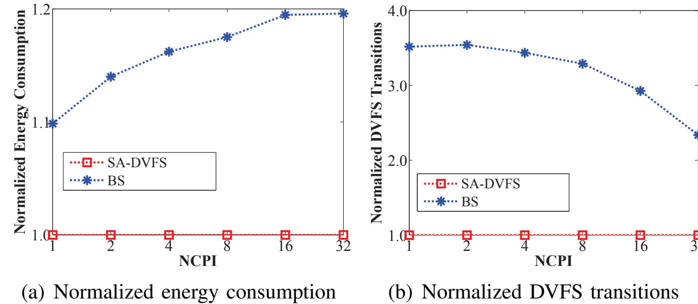
(a) Normalized energy consumption (b) Normalized DVFS transitions

Fig. 16: Effects of different policies with varying RU.



(a) Normalized energy consumption (b) Normalized DVFS transitions

Fig. 17: Effects of different policies with varying CSR.



(a) Normalized energy consumption (b) Normalized DVFS transitions

Fig. 18: EEffects of different policies with varying $NCPL$.

[9] D. Bautista, J. Sahuquillo, H. Hassan, S. Petit, and J. Duato. A simple power-aware scheduling for multicore systems when running real-time applications. In *IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–7, 2008.

[10] A. Block, H. Leontyev, B B. Brandenburg, and J H. Anderson. A flexible real-Time locking protocol for multiprocessors. *Proc. 13th IEEE Conf. Embedded and Real-Time Computing Systems and Applications*, pages 47–57, 2007.

[11] S. Borkar. Thousand core chips: a technology perspective. In *Proc. of the 44th Design Automation Conference (DAC)*, pages 746–749, 2007.

[12] B B. Brandenburg and J H. Anderson. Optimality results for multiprocessor real-time locking. *Proc. 31st IEEE Real-Time Systems Symp.*, pages 49–60, 2010.

[13] T D. Burd and R W. Brodersen. Energy efficient CMOS microprocessor design. *Proc. Hawaii Int'l Conf. System Sciences*, pages 288–297, 1995.

[14] J.-J. Chen and T.-W. Kuo. Multiprocessor energy-efficient scheduling for real-time tasks with different power characteristics. In *Proc. of the 2005 Int'l Conference on Parallel Processing (ICPP)*, pages 13–20, 2005.

[15] J.-J. Chen and T.-W. Kuo. Procrastination determination for periodic real-time tasks in leakage-aware dynamic voltage scaling systems. In *Proc. of the 2007 IEEE/ACM Int'l Conference on Computer-Aided Design (ICCAD)*, pages 289–294, 2007.

[16] J.-J. Chen, H.-R. Hsu, and T.-W. Kuo. Leakage-aware energy-efficient scheduling of real-time tasks in multiprocessor systems. In *Proc. of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 408–417, 2006.

[17] Y-S. Chen, C-Y. Yang, and T-W. Kuo. Energy-efficient task Synchronization for real-time systems. *IEEE Trans. Industrial Informatics*, 6(3):287–301, 2010.

[18] S. Cho and R. G. Melhem. On the interplay of parallelization, program performance, and energy consumption. *IEEE Transactions on Parallel and Distributed Systems*, 21(3):342–353, 2010.

[19] E T-H. Chu, T-Y. Huang, and Y-C. Tsai. An optimalsolution for the heterogeneous multiprocessor single-level voltage-setup problem. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 28(11):1705–1718, 2009.

[20] J.M. Cohn, D.W. Stout, P.S. Zuchowski, S.W. Gould, T.R. Bednar, and D.E. Lackey. Managing power and performance for system-on-chip designs using voltage islands. In *Proc. of the International Conference on Computer-Aided Design (ICCAD)*, pages 195–202, 2002.

[21] V. Devadas and H. Aydin. Coordinated power management of periodic real-time tasks on chip multiprocessors. In *Proc. of the First IEEE Int'l Green Computing Conference (IGCC)*, Aug. 2010.

[22] J. Donald and M. Martonosi. Techniques for multicore thermal management: classification and new exploration. In *Proc. of the 33rd Int'l Symposium on Computer Architecture (ISCA)*, pages 78–88, 2006.

[23] J. Dorsey, S. Searles, M. Ciraula, S. Johnson, N. Bujanos, D. Wu, M. Braganza, S. Meyers, E. Fang, and R. Kumar. An integrated quad-core opteron processor. In *IEEE International Solid-State Circuits Conference*, pages 102–103, Feb. 2007.

[24] A. Easwaran and B. Andersson. Resource sharing in global fixed-priority preemptive multiprocessor scheduling. *Proc. 30th IEEE Real-Time Systems Symp.*, pages 377–386, 2009.

[25] R. Ernst and W. Ye. Embedded program timing analysis based on path clustering and architecture classification. *Proc. Int'l Conf. Computer-Aided Design*, pages 598–604, 1997.

[26] P. Gai, M. D. Natale, G. Lipari, A. Ferrari, C. Gabellini, and P. Marceca. A comparison of MPCP and MSRP when sharing resources in the Janus multiple-processor on a chip platform. pages 189–198, 2003.

[27] L K. Goh, B. Veeravalli, and S. Viswanathan. Design of fast and efficient energy-aware gradient-based scheduling algorithms heterogeneous embedded multiprocessor systems. *IEEE Trans. Parallel and Distributed Systems*, 20(1):1–12, Jan. 2009.

[28] S. Herbert and D. Marculescu. Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. In *Proc. of the int'l symposium on Low power electronics and design (ISLPED)*, pages 38–43, 2007.

[29] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *IEEE Computer*, 41(7):33–38, 2008.

[30] http://public.itrs.net. International technology roadmap for semiconductors. 2008. S. R. Corporation.

[31] http://www.intel.com/products/processor/core2quad/, 2008.

[32] J. Hu, Y. Shin, N. Dhanwaday, and R. Marculescu. Architecting voltage islands in core-based system-on-a-chip designs. In *Proc. of the Int'l symposium on Low power electronics and design*, pages 180–185, 2004.

[33] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *Proc. of the 39th IEEE/ACM Int'l Symposium on Microarchitecture*, pp. 347–358, 2006.

[34] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *Proc. of the Int'l Symposium on Low Power Electronics and Design (ISLPED)*, pages 197–202, 1998.

[35] R. Jejurikar and R. Gupta. Energy-aware task scheduling with task synchronization for embedded real-time systems. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 25(6):1024–1037, 2006.

[36] R. Jejurikar, C. Pereira, and R. Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. In *Proc. of the* 41st *Design automation conference (DAC)*, pages 275–280, 2004.
[37] K. Lakshmanan, D. de Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. *Proc. 30th IEEE Real-Time Systems Symp.*, pages 469–478, 2009.
[38] W. Y. Lee. Energy-efficient scheduling of periodic real-time tasks on lightly loaded multi-core processors. *IEEE Trans. Parallel and Distributed Systems*, pages 344–357, 2011.
[39] L. Leung and C. Tsui. Energy-aware synthesis of networks-on-chip implemented with voltage islands. In *Proc. of the 44th ACM/IEEE Design Automation Conference (DAC)*, pages 128–131, 2007.
[40] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *Proc. of the Int'l Symp. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 2–11, Oct. 1996.
[41] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proc. of* $18th$ *ACM Symposium on Operating Systems Principles (SOSP'01)*, Oct. 2001.
[42] X. Qi and D. Zhu. Energy-efficient block-partitioned multicore processors for parallel applications. *Journal of Computer Science and Technology*, 26(3):418–433, 2011.
[43] G. Quan and X.S. Hu. Energy efficient DVS schedule for fixed-priority real-time systems. *ACM Trans. Embedded Comput. Syst.*, 6(4), 2007.
[44] R. Rajkumar, L. Sha, and J.P. Lehoczky. Real-time synchronization protocols for multiprocessors. *Proc. 9th IEEE Real-Time Systems Symp.*, pages 259–269, 1988.
[45] C. Scordino and G. Lipari. A resource reservation algorithm for power-aware scheduling of periodic and aperiodic real-time tasks. *IEEE Trans. on Computers*, 55(12):1509–1522, 2006.
[46] E. Seo, J. Jeong, S. Park, and J. Lee. Energy efficient scheduling of real-time tasks on multicore processors. *IEEE Trans. Parallel Distrib. Syst.*, 19(11):1540–1552, 2008.
[47] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Trans. on Computers*, 39(9):1175–1185, Aug. 1990.
[48] P. Leteinturier (Infineon Technologies). Multi-core processors: Driving the evolution of automotive electronics architectures; available at http://www.embedded.com/design/multicore/, 2007.
[49] Y. Wang, X. Wang, M. Chen, and X. Zhu. Partic: Power-aware response time control for virtualized web servers. *IEEE Trans. Parallel Distrib. Syst.*, 22(2):323–336, 2011.
[50] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. *Proc. First USENIX Symp. Operating system Design and Implementation*, pages 13–23, Nov. 1994.
[51] X. Wu, Y. Lin, J-J. Han, and J-L. Gaudiot. Energy-efficient scheduling of real-time periodic tasks in multicore systems. *Proc. 7th IFIP Int'l Network and Parallel Computing (NPC)*, pages 344–357, 2010.
[52] R. Xu, R. Melhem, and D. Mossé. Energy-aware scheduling for streaming applications on chip multiprocessors. *Proc. 28th IEEE Real-Time Systems Symp.*, pages 25–38, 2007.
[53] http://www.lindo.com/.
[54] F. Zhang and S.T. Chanson. Blocking-aware processor voltage scheduling for real-time tasks. *ACM Trans. Embedded Computing Systems*, 3(2):307–335, May 2004.
[55] D. Zhu and H. Aydin. Reliability-aware energy management for periodic real-time tasks. *IEEE Trans. on Computers*, 58(10):1382–1397, 2009.
[56] D. Zhu, R. Melhem, and B. R. Childers. Scheduling with dynamic voltage/speed adjustment using slack reclamation in multi-processor real-time systems. *IEEE Trans. on Parallel and Distributed Systems*, 14(7):686–700, 2003.
[57] J. Zhuo and C. Chakrabarti. Energy-efficient dynamic task scheduling algorithms for DVS Systems. *ACM Trans. Embedded Computing Systems*, 7(2):1–25, 2008.