CycleTandem: Energy-Saving Scheduling for Real-Time Systems with Hardware Accelerators

Sandeep D'souza and Ragunathan (Raj) Rajkumar
Carnegie Mellon University

***Abstract*—Cyber-physical systems such as autonomous vehicles need to process and analyze multiple simultaneous streams of sensor data in real-time. Therefore, these systems require powerful multi-core platforms with hardware accelerators such as GP-GPUs. These accelerators generally consume significant amounts of power. Therefore, power management is required to ensure that task deadlines are met while staying within the energy and thermal constraints of the system. In these systems, most tasks execute using a combination of CPU and accelerator resources. Hence, the power of the CPU and the accelerator needs to be managed in *tandem*. To reduce energy consumption, commercially-available accelerators such as GP-GPUs and DSPs expose interfaces to scale their operating voltage and frequency. Hence, we propose the *CycleTandem* static frequency-scaling technique to *co-optimize* the operating frequencies of both the CPU and the hardware accelerator. Based on practical considerations of real-world platforms, we consider various energy-management scenarios where the accelerator or CPU frequencies may or may not be adjustable, and propose the *CycleSolo* family of algorithms for such contexts. Furthermore, we also study partitioning techniques to reduce the operating frequency when multi-core processors are used in conjunction with hardware accelerators. Experimental evaluations indicate that our proposed techniques can yield significant energy savings. We also present a case-study on the NVIDIA TX2 embedded platform to illustrate the energy savings delivered by our proposed techniques.**

I. Introduction

Modern-day real-time systems combine high-computational demand with low-latency requirements. This is primarily driven by the emergence of application domains such as autonomous vehicles [1] and robotics, where multiple simultaneous streams of sensor data need to be processed in real-time to ensure that safety constraints are met. Analyzing this data relies on techniques ranging from signal processing, computer vision to machine learning. These workloads are compute intensive and highly parallel. Therefore, it is becoming increasingly common to find hardware accelerators such as General-Purpose Graphics Processing Units (GP-GPUs), Digital Signal Processors (DSPs), and Application-Specific ICs (ASICs) in the computing platforms used in such systems [2].

Hardware accelerators often consume significant amounts of power [3]. In addition, energy-constrained platforms such as smartphones, drones, robots and AR/VR headsets also contain one or more hardware accelerators [4][5][6]. Hence, it is necessary to focus on energy management for systems using hardware accelerators. Thermal constraints also make it necessary to reduce power consumption to avoid unexpected system shutdown or prevent bodily harm to the user [7].

To reduce energy consumption, processors are equipped with energy-management features such as Dynamic Voltage and Frequency Scaling (DVFS) [8], and the use of low-power sleep states. By using DVFS the processor can change its operating frequency and voltage, thereby reducing dynamic-switching power. On the other hand, low-power sleep states utilize power gating and/or clock gating [9] to reduce static-leakage power when the processor is idle. Multi-core processors generally expose these energy-management features in the form of P-states, which can be used to set the processor voltage and frequency; and C-states, which can be used to power and/or clock gate sections of the processor.

Hardware accelerators can also expose similar power-management interfaces. However, in commercial accelerators like GP-GPUs and DSPs, only P-states are exposed to the user [10][11]. Thus, in effect, they expose only voltage and frequency-scaling knobs for power management, and the job of reducing static power is done in firmware or hardware. Therefore, we focus on using frequency-scaling-based techniques to reduce the power consumption of systems using hardware accelerators. In particular, we propose techniques to statically set the processor and accelerator to a pre-computed taskset-specific frequency, such that the aggregate energy consumption is reduced, while ensuring that all deadlines are met. Therefore, as there are no dynamic frequency changes, the unpredictable latency involved in changing the oscillator frequency is avoided, leading to more deterministic operation.

The primary contributions of this paper are as follows:

- We introduce a novel search technique called *ratchet search*, and use it to jointly estimate the upper and lower bounds of the range containing the lowest feasible frequency, as additional tasks and resources are considered
- We propose the *CycleSolo* family of algorithms to calculate the energy-optimal frequency-scaling factor, when (i) the frequency of only the CPU or the accelerator can be scaled, or (ii) both the CPU and accelerator frequency must be scaled by the same common factor.
- We propose the *CycleTandem* algorithm to calculate low-power frequency-scaling factors for the CPU and the accelerator, when both the CPU and accelerator frequency can be independently scaled.
- We extend the *CycleSolo* and *CycleTandem* algorithms to the fully-partitioned multi-core context.

The rest of the paper is organized as follows. Section II discusses prior work. Section III provides the system model used in the paper. Section IV introduces the CycleSolo family of algorithms, and Section V discusses CycleTandem. Section VI extends CycleTandem and CycleSolo to the fully-partitioned multi-core context. Section VII provides experimental evaluations, and Section VIII concludes the paper.

II. Related Work

In this section, we first summarize the techniques proposed in literature to arbitrate access to hardware accelerators like GP-GPUs. Subsequently, we discuss the various real-time energy-saving scheduling techniques proposed in prior work.

Most commercially-available accelerators like GP-GPUs and DSPs do not typically support preemption. The large number of registers in these accelerators makes context switching an expensive proposition. Therefore, prior work [2][12] has focused on modeling accelerator access as a critical section arbitrated by a global lock. In particular, the work in [13][14] models GPUs as mutually-exclusive resources, whose access is governed by existing real-time synchronization protocols. The same authors also proposed GPUSync [12], which is a software framework for GPU management in multi-core real-time systems. Based on this synchronization-based approach, [2] extends the analysis proposed in [13][15][16][17], to propose a less pessimistic response-time analysis framework to decide the schedulability of tasksets which may use one or more accelerators. This analysis assumes the use of the Multiprocessor Priority Ceiling Protocol (MPCP) [15], while incorporating the effect of self suspensions [16][18].

In [19], Kim *et al.* propose the server-based approach, where a server task is created to access the GPU on behalf of the client applications. However, in this work, we focus on the synchronization-based approach.

In the context of real-time systems, a number of approaches have been proposed to reduce energy. In particular, the use of frequency-scaling-based techniques have been well-studied [20][21][22][23]. For fixed-priority uniprocessor scheduling, Saewong *et al.* [20] proposed the *SysClock* algorithm to analytically determine the lowest CPU frequency that allows all tasks to meet their deadlines. In [24], the authors extend *SysClock* to fully-partitioned multi-core processors which have a single frequency domain. In [21][22][23], frequency-scaling-based techniques for multi-core processors are proposed, where each core has its own power domain. However, none of the existing techniques in the literature consider the case where some task sections can be executed on a hardware accelerator.

At CMOS technology nodes smaller than 65nm [25], static-power dissipation is comparable to dynamic-power consumption. Thus, *a priori* task information can be utilized to cluster task execution so as to maximize the time spent in a low-power state [26][27][28][29][30]. The work in [29][30][7] introduced the family of *Energy-Saving* Schedulers, which utilize the processor deep-sleep state to maximize energy savings for both unicore and multi-core processors. However, most accelerators do not provide user-configurable sleep states. Thus, sleep-state-based techniques cannot be used in their context.

Apart from software-based approaches, the work in [31] proposes a hardware-based approach called MERLOT for GPU energy management in the context of real-time systems. MERLOT exploits the fact that, in the general case, most GPU kernels do not execute up to their worst-case execution time. In such situations, the slack can be dynamically used to reduce the voltage and frequency of the GPU. However, MERLOT considers individual job deadlines, and does not consider taskset schedulability, or the fact that most tasks execute using a combination of CPU and GPU segments.

III. Background and System Model

In this section, we present the assumptions and system model used in this paper. We also provide some background about the synchronization-based approach used to govern access to hardware accelerators [12], along with its suspension-based schedulability analysis introduced in prior work [2].

A. Assumptions and Task Model

Consider a taskset Γ consisting of *n* sporadic real-time tasks $\tau_1, \tau_2, ..., \tau_n$. The taskset is deployed on an m-core homogeneous multi-core processor M, with a single non-preemptive hardware accelerator A. This assumption is reasonable as most existing accelerators, including GP-GPUs and DSPs, do not support preemption [2]. We model the accelerator as a shared resource, and assume that access to the accelerator is treated as a *critical section* arbitrated by a global lock, L [12]. We also assume that, at any point of time, only a single task can utilize the accelerator. This avoids any unpredictability in execution time caused by accelerators which support concurrent execution [2]. Additionally, accelerators used in energy-constrained platforms may not support concurrent execution.

Based on the above assumptions, each task $\tau_i \in \Gamma$ can be characterized by $\{C_i, G_i, T_i, D_i\}$, where C_i is the worst-case execution time (WCET) on the CPU, G_i is the WCET on the accelerator, T_i is the period or minimum job inter-arrival time (sporadic tasks), and D_i is the relative deadline from the arrival time. The term G_i consists of: (i) G_i^e, the WCET of the task on the accelerator, and (ii) G_i^m, the worst-case CPU-intervention required to access the accelerator. Note that, $G_i \leq G_i^e + G_i^m$, as G_i^e and G_i^m may not occur on the same control path [2]. However, we assume that $G_i = G_i^e + G_i^m$. Therefore, for each task τ_i we define the total CPU time required as $E_i = C_i + G_i^m$. We also assume that each task can access the accelerator at most once every period. This assumption is reasonable since in practice, most tasks have a single accelerator-executed segment. Tasks which have multiple accelerator segments, can be split into separate tasks. We also assume that, while accessing the accelerator, each task suspends on the CPU.

In this paper, we consider fully-partitioned fixed-priority preemptive scheduling and assume deadlines are constrained, i.e., $D_i \leq T_i$. Task priorities are assumed to be unique with each task τ_i assigned the priority π_i. The taskset is listed in decreasing order of task priorities, i.e., $\pi_1 > \pi_2 > ... > \pi_n$.

MPCP-based Analysis: In the context of this work, we utilize the Multiprocessor Priority Ceiling Protocol (MPCP) [15] to govern access to the global lock L used to access the accelerator [2]. We consider the version of MPCP, where a task requesting access to a critical section locked by another task is suspended and inserted into a lock-specific priority queue [2]. When a task releases a lock, the task at the head of the priority queue is scheduled, and granted access to the critical section.

In doing so, the priority of the task is raised to the lock's priority ceiling. The priority ceiling of the critical section of task τ_i accessing lock L, is given by $\pi_i = \pi_k + \pi_B$, where π_B is a priority level greater than the base priority of any task in the system and π_k is the highest base priority of any task that uses L. On completion of the critical section, τ_i releases the lock, and its priority is returned to its original base priority.

To determine taskset schedulability, we use response-time-based analysis. Based on this technique, the worst-case response time for a task τ_i is given by the following recurrence:

$$W_i^0 = C_i + G_i + B_i, W_i^{k+1} = C_i + G_i + B_i + \sum_{h=1}^{i-1} I_{i,h} \quad (1)$$

where, W_i is the worst-case response time of the task τ_i, B_i provides an upper-bound on the worst-case blocking faced by τ_i in getting access to the accelerator, and $I_{i,h}$ denotes the worst-case CPU preemption τ_i faces due to a higher-priority task τ_h. If $W_i \leq D_i$, then τ_i will be schedulable.

The worst-case preemption $I_{i,h}$, faced by task τ_i due to a higher-priority task τ_h can be given by:

$$I_{i,h} = \alpha_{i,h} * E_h, \alpha_{i,h} = \lceil (W_i + W_h - E_h)/T_h \rceil \quad (2)$$

where, $\alpha_{i,h}$ represents an upper bound on the number of jobs of τ_h released during a single job of τ_i [2]. Note that, $\alpha_{i,j}$ considers the jitter, $W_h - E_h$, introduced by τ_h's self-suspension on the CPU, while accessing the accelerator [18].

The worst-case blocking B_i, faced by a task τ_i, in accessing the accelerator can be upper-bounded by multiple approaches described in prior work [2][13][17]. Three such approaches are (i) the job-driven analysis, (ii) the request-driven analysis, and (iii) the hybrid analysis. Neither of the first two analyses strictly dominates the other. In practice, the work in [2] observed that the job-driven analysis dominates the request-driven analysis when the number of critical sections a task executes on the accelerator increases. On the other hand, as C_i and W_i increase the job-driven analysis becomes more pessimistic. The hybrid analysis proposed in [2] uses a combination of the job-driven and request-driven analysis to provide a less-pessimistic worst-case response-time estimate.

In Section III-A, we assumed that each task has only one critical section which is executed on the accelerator. Under this assumption, we can easily prove the following theorem.

Theorem 1: If all tasks in Γ have *at most* one critical section executed on the accelerator, then the request-driven analysis *always* dominates the job-driven analysis.

Proof: The proof can be found in the Appendix. ∎

As a corollary, for the above case, it can also be proven that the request-driven analysis is equivalent to the hybrid analysis [2]. Therefore, in this work, we utilize the request-driven analysis. However, the algorithms we propose can easily be adapted to other analysis techniques.

Based on the request-driven analysis [2][17], the worst-case blocking B_i, faced by a task τ_i in accessing the accelerator, can be upper-bounded by the following recurrence [2]:

$$B_i^0 = \max_{\tau_l | l>i}(G_l), B_i^{k+1} = \max_{\tau_l | l>i}(G_l) + \sum_{h=1}^{i-1} \beta_{i,h} * G_h \quad (3)$$

where, $\beta_{i,h} = \lceil (B_i + W_h - E_h)/T_h \rceil$ upper bounds the number of accelerator requests made by a higher-priority task τ_h, while τ_i is being blocked. Note that, $\beta_{i,h}$ considers the self-suspension of a task on the CPU, while accessing the accelerator.

B. Power Model

The power consumption of modern CMOS-based processors is modeled as a combination of two major components:

(1) *Dynamic Power* is dependent on the processor operating frequency. Assuming that voltage is scaled with frequency, dynamic power consumption, P_D, can be modeled as a convex function of the operating frequency s as [25]: $P_D = Kf^\alpha$ where, α and K are technology-dependent system constants.

(2) *Static Power* is due to leakage current, which depends on the semiconductor technology. Static power, P_S, can be modeled as [25]: $P_S = VI_{leak}$ where, V is the operating voltage and I_{leak} is the technology-dependent leakage current.

Hence, power consumption $P = P_D + P_S$. Therefore, the total power consumed by the CPU-accelerator combination can be given by $P_{total} = P_{cpu} + P_{acc}$, where P_{cpu} and P_{acc} are the power consumption of the CPU and accelerator respectively.

As stated in Section I, while dynamic power is reduced using voltage and frequency scaling, static power is reduced using sleep states. However, as mentioned earlier, most accelerators do not provide user-configurable sleep states to reduce static power [32], and rely on firmware-based control to reduce static power. Therefore, we focus on using frequency-scaling-based power management, and assume that the processor/accelerator performs its own optimizations in parallel to reduce static power. In particular, we focus on statically choosing a single operating frequency for the CPU/accelerator. This is based on the well-known property that, for processors with a *non-decreasing* convex power-frequency function, the energy is minimized if the processor executes its workload at the lowest-possible constant frequency [20].

For the sake of simplicity, we assume a continuous processor/accelerator frequency range normalized to the range $[0, 1]$. In later sections, we discuss how discrete frequencies can be accommodated. We also assume that task worst-case execution times (WCET) are specified at the maximum frequency, $f_{max} = 1$, and the WCET is scaled in proportion to the operating frequency f, i.e, $WCET_f = WCET_{f_{max}}/f$. However, our proposed algorithms are independent of this scaling model, and any model where the execution-time monotonically-increases with decreasing frequency can be used.

IV. CycleSolo Algorithm

We now introduce the *CycleSolo* family of algorithms for uniprocessor systems with a single hardware accelerator. We propose three variants of *CycleSolo*:

1) *CycleSolo-CPU*: when the accelerator does not support frequency scaling, and only the CPU frequency can be scaled.

2) *CycleSolo-Accel*: when the accelerator supports frequency scaling, but the CPU frequency cannot be scaled.

3) *CycleSolo-ID*: when both the accelerator and the CPU frequencies must be scaled by a common scaling factor.

The two steps of the *CycleSolo* family of algorithms are as follows: 1) Compute a tight bound on the frequency range in which the optimal frequency lies. 2) Perform a binary search testing schedulability over the computed frequency range, to obtain the lowest-possible CPU/accelerator operating frequency which ensures that all deadlines are met.

We can easily prove that given a schedulability-analysis technique, a binary search (Algorithm 4 in the Appendix) will always converge to the lowest-possible operating frequency.

Lemma 1: Given a taskset Γ, and a response-time-based schedulability-analysis technique S, a binary search testing schedulability over the operating frequency range, converges to the lowest-possible operating frequency f_{min}, which guarantees that Γ is schedulable using technique S.

Proof: Consider an operating frequency $0 < f \leq f_{max}$. For every task τ_i, the worst-case execution time is inversely proportional to the frequency. Therefore, the response time of a task also increases monotonically as the frequency decreases. Thus, we can conclude that the taskset will be schedulable for all frequencies $f' \geq f$, if and only if the taskset is schedulable at frequency f, using technique S. Conversely, if a taskset is not schedulable at frequency f, it will not be schedulable for all frequencies $f' \leq f$. Given that Γ transitions from schedulable to unschedulable after frequency f_{min}, the estimated frequency range decreases with each iteration of the binary search (Algorithm 4), and given sufficient iterations converges to $f_c = f_{min}$, which is the lowest frequency guaranteeing Γ is schedulable according to analysis technique S. ■

However, performing the response-time-based schedulability test multiple times over the entire frequency range $[0, f_{max}]$ is not desirable, as the response-time-based analysis has pseudo-polynomial complexity. Therefore, we now explain how *CycleSolo* computes a tight bound on the range in which the optimal frequency lies, by proving various results in the context of *CycleSolo-CPU*. However, the same results can be easily extended to the entire *CycleSolo* family of algorithms.

CycleSolo-CPU: Consider a uniprocessor with a non-preemptive accelerator, whose frequency is not adjustable. Therefore, to minimize energy, we need to find the lowest CPU operating frequency at which all tasks meet their deadlines.

In [20], the *SysClock* algorithm was proposed for independent sporadic tasks using fixed-priority uniprocessor scheduling. SysClock calculates the lowest processor frequency at which all tasks meet their deadlines. For each task τ_i, SysClock calculates the slack at all scheduling points in the critical zone [33] to determine the minimum frequency, f_i, at which τ_i meets its deadline, in the presence of high-priority interference. SysClock finally chooses the lowest frequency f_{min}, as the maximum of these per-task minimum frequencies, i.e., $f_{min} = \max_{i|\tau_i \in \Gamma} f_i$. Thus, SysClock chooses the lowest frequency at which all tasks meet their deadlines.

In this work, we use the slack-calculation methodology from SysClock. However, unlike SysClock, the following issues are encountered while estimating the minimum frequency:

1) **Undefined Critical Instant**: In the presence of blocking and self suspensions, the critical instant *does not* occur when all high-priority jobs arrive together with the task τ_i, and is undefined [18]. Therefore, extra blocking and self-suspension terms are added to utilize the existing response-time analysis. In effect, this assumes the same critical instant, but adds extra pessimism by considering the worst-case blocking and modeling the self-suspensions as release jitter [18]. Therefore, like SysClock, the CycleSolo algorithms consider the critical-zone theorem [33] where, in the worst case, the requests of all tasks arrive simultaneously. In practice, the worst-case blocking, interference and self-suspension penalties never appear together. Due to this pessimism, all the known analysis techniques are safe, but none of them are exact [2]. This prevents us from finding the absolute minimum frequency. Instead, we can obtain the minimum frequency which allows a taskset to be schedulable *given* the analysis framework used.

2) **Frequency-Dependent Slack**: Due to self suspensions, the interference and blocking faced by each low-priority task depends on the response time and worst-case execution time of higher-priority tasks. However, the worst-case response time of higher-priority tasks depends on the operating frequency. Thus, different operating frequencies create different amounts of high-priority interference and blocking, which makes the slack calculation frequency-dependent. In SysClock, it is sufficient to calculate the slack at each scheduling point, i.e., a task's deadline or points in time when a new job of a task is released. However, in the presence of self-suspending tasks, the pessimism added to the response-time analysis changes the *effective* points in time at which new instances of tasks appear, and makes them dependent on the frequency.

Lemma 2: Consider response-time-based schedulability-analysis techniques which model self-suspension as release jitter. Then, for a task τ_h which self-suspends, the set of effective scheduling points, i.e., points in time where new instances of a task *effectively* arrive are given by:

$$S_h := \{j * T_h - (W_h - E_h) | j > 0\} \tag{4}$$

Proof: The definition of the response-time analysis stated in Equation 1, calculates the number of jobs, $\alpha_{i,h}$, of each higher-priority task τ_h which interfere with the execution of a low-priority job τ_i. For a high-priority task τ_h, which self-suspends on the CPU, at each time instant $t \in S_h$, the interference increases by the execution time of one job of τ_h. Therefore, they can be considered as effective scheduling points in the context of the response-time analysis being used. ■

Lemma 2 indicates that the interference calculation depends on the high-priority response time, which depends on the operating frequency. This dependence does not occur for tasks which do not self-suspend, as is the case in SysClock, and prevents us from calculating the lowest frequency in a single pass over all the tasks. We instead estimate a feasible range $[f^{low}, f^{high}]$ which contains the lowest frequency, f_{min}.

Algorithm 2 presents the pseudo-code for *CycleSolo-CPU*, which considers tasks in decreasing order of their priority. For each task $\tau_i \in \Gamma$, CycleSolo-CPU computes a range $[f_i^{low}, f_i^{high}]$, which contains the lowest frequency ensuring that τ_i and all higher-priority tasks $\tau_{h|h<i}$ are schedulable.

Algorithm 1 Calculating the CycleSolo Frequency Bounds

procedure RATCHETSEARCH-STEP($f_{high}, f_{low}, f_{est}$)
  /* f_{high} = upper bound, f_{low} = lower bound*/
  if $f_{est} > f_{high}$ **then** ▷ check estimate against bounds
    $f_{low} = f_{high}$; $f_{high} = f_{est}$ ▷ Case III
  elseif $f_{est} > f_{low}$
    $f_{low} = f_{est}$ ▷ Case II
  end if
  return f_{low}, f_{high} ▷ chosen frequency range
end procedure

Consider a task $\tau_i \in \Gamma$. Let us assume that the range $[f_{i-1}^{low}, f_{i-1}^{high}]$ has already been computed, and is known. To calculate the minimum frequency at which τ_i and all higher-priority tasks are schedulable, we need to estimate the available slack in the schedule. Based on Lemma 2, for a task τ_i, the workload β_i^t changes at every scheduling point $t \in \{S_h \mid h < i, t \leq D_i\} \cup D_i$. Therefore, CycleSolo-CPU determines the CPU workload β_i^t, that exists in the system up to each scheduling point t. However, for CycleSolo-CPU, the scheduling points depend on the frequency at which the high-priority workload is run. In particular, as the frequency decreases, the response time of high-priority tasks $W_{h|h<i}$, is monotonically non-decreasing. Thus, the estimated workload β_i^t also depends on the frequency f_h *chosen* for the higher-priority tasks, and is monotonically non-decreasing as the frequency f_h decreases. Assuming we choose a frequency f_h, CycleSolo-CPU's slack calculation would yield a frequency $f_i^{est} = \min_{t \in S_h, D_i} \beta_i^t/t$, where, $h < i$ and $t \leq D_i$.

We need to choose the frequency f_h, such that the obtained f_i^{est} can provide a safe range $[f_i^{low}, f_i^{high}]$ containing the lowest frequency f_i^{min} which can ensure that τ_i and all higher-priority tasks $\tau_{h|h<i}$ are schedulable.

Lemma 3: For a task $\tau_i \in \Gamma$, choosing the high-priority frequency f_h as f_{i-1}^{high}, yields a *correct* range $[f_i^{low}, f_i^{high}]$ in which lies the lowest frequency f_i^{min} guaranteeing that τ_i and all higher-priority tasks are schedulable using analysis S.

Proof: Let us choose $f_h = f_{i-1}^{high} - \delta$, where $\delta \in (0^+, f_{i-1}^{high} - f_{i-1}^{low}]$. Now, in the worst case, the minimum frequency required to schedule only the higher-priority tasks $\tau_{h|h<i}$ can be $f_{i-1}^{min} > f_h$. Therefore, at frequency f_h, at least one of the higher-priority tasks $\tau_{h|h<i}$ will miss their deadlines. As the slack calculation used to estimate f_i^{est} uses the higher-priority worst-case response time W_h, this deadline violation will lead to an incorrect estimate of f_i^{est}. Therefore, there is a contradiction. Thus, $\delta = 0$, which implies $f_h = f_{i-1}^{high}$. ■

Ratchet Search: A "ratchet search" is an incremental technique that refines earlier estimates based on additional parameters. Algorithm 1 presents the *RatchetSearch-Step* routine, which is a single step of RatchetSearch, in the context of estimating the bounds of the frequency range $[f_i^{low}, f_i^{high}]$.

Consider the frequency f_i^{est} obtained by choosing the high-priority frequency $f_h = f_{i-1}^{high}$. We can have three scenarios:

Case I. $f_i^{est} < f_{i-1}^{low}$ implies that $\tau_h \in \Gamma \mid h \leq i$ will be schedulable at f_h, as the available slack is sufficient to support an operating frequency $f_i^{est} \leq f_h$. Hence, no change is needed to the existing frequency range, as $f_i^{min} \in [f_{i-1}^{low}, f_{i-1}^{high}]$.

Case II. $f_i^{est} \in [f_{i-1}^{low}, f_{i-1}^{high}]$. Of all the frequencies in the previously-computed feasible range, choosing $f_h = f_{i-1}^{high}$, introduces the *least*-possible high-priority interference. This creates the *maximum*-possible slack for τ_i, and enables f_i^{est} to be minimized for the frequencies in the feasible range. Therefore, the minimum frequency f_i^{min} required to schedule tasks $\tau_h \in \Gamma \mid h \leq i$ is always greater than f_i^{est}. Thus, the lower bound of the range f_i^{low} can be safely updated to f_i^{est}.

Case III. $f_i^{est} > f_{i-1}^{high}$ implies that τ_i is not schedulable at $f_h = f_{i-1}^{high}$. Therefore, the lower bound of the range f_i^{low}

Algorithm 2 Minimizing CPU Frequency

procedure CYCLESOLO-CPU(Γ, ϵ_{conv})
  $f_{low} = f_{high} = U_{cpu}$ ▷ initial bounds from Lemma 4
  for $\tau_i \in \Gamma$ **do** ▷ from high to low priority
    f_{low}, f_{high} = EstimateFreqRange($\tau_i, \Gamma, f_{high}, f_{low}$)
  end for
  f_{min} = BinarySearch($\Gamma, \epsilon_{conv}, f_{high}, f_{low}$)
  return f_{min}
end procedure
procedure ESTIMATEFREQRANGE($\tau_i, \Gamma, f_{high}, f_{low}$)
  $f_h = f_{high}$ ▷ Lemma 3, f_h= high-priority frequency
  /* S = slack, ω = resp time, β = CPU workload */
  $S = I = \beta = \Delta = 0, f_{est} = 1$, BusyFlag=TRUE
  W = Calculate-HP-ResponseTime(f_h)
  B_i = CalculateBlocking(τ_i, Γ, f_h, W)
  $\omega_g = G_i + B_i - G_{cpu}$ ▷ Accelerator execution time
  $\omega = C_i + G_i + B_i$, $\omega' = 0$, $J_i = \omega$
  while $\omega < D_i$ **do**
    if BusyFlag == TRUE **then**
      $\Delta = D_i - \omega$
      while $\omega < D_i$ AND $\Delta > 0$ **do**
        $\omega' = \sum_{h=0}^{i-1} E_h * [\lfloor \frac{\omega + W_h - (E_h/f_h)}{T_h} \rfloor + 1]$
        $\omega' = \omega' + J_i + S$, $\Delta = \omega' - \omega, \omega = \omega'$
      end while
      BusyFlag = FALSE
    else ▷ Start of an idle period
      t = Find-EarliestSchedulingPoint(τ_i, Γ, ω)
      $S = S + (t - \omega), \omega = t, t' = \omega - \omega_g, \beta = \omega - S - \omega_g$
      if $\beta/t' < f_{est}$ **then**
        $f_{est} = \beta/t'$
      end if
      BusyFlag = TRUE
    end if
  end while
  f_{low}, f_{high} = RatchetSearch-Step($f_{high}, f_{low}, f_{est}$)
  return f_{low}, f_{high}
end procedure
procedure CALCULATEBLOCKING(τ_i, Γ, f_h, W)
  $G_{l,max} = max_{\tau_l \in lp(\tau_i)}(G_l)$, $B = G_{l,max}$, $B' = 0$
  while B != B' **do**
    $B' = B$, $B = G_{l,max} + \sum_{\tau_h \in hp(\tau_i)} \lceil \frac{B' + W_h - \frac{E_h}{f_h}}{T_h} \rceil * G_h$
  end while
  return B
end procedure

can be safely updated to f_h. The previous statement also implies that task τ_i would not be schedulable for any frequency $f < f_h$. Therefore, choosing $f_h = f_{i-1}^{high}$ also introduces the *maximum*-feasible high-priority interference, which minimizes the available slack and allows f_i^{est} to be maximized. Thus, f_i^{est} is a safe and tight upper bound which guarantees that tasks $\tau_h \in \Gamma \mid h \leq i$ are schedulable. Therefore, the upper bound of the range f_i^{high} can be safely updated to f_i^{est}.

However, the RatchetSearch routine requires an initial estimate of the upper and lower bounds of the range, f_{init}^{low} and f_{init}^{high}, in which the minimum frequency lies. As RatchetSearch always increases or "ratchets up" the value of the bounds, both bounds can be initialized to $f_{init}^{low} = f_{init}^{high} = 0$.

Lemma 4: The lowest-possible CPU frequency, f_{min}, at which a taskset Γ is schedulable is always greater than or equal to $U_{cpu} * f_{max}$, where U_{cpu} is the CPU utilization of the taskset at the maximum operating frequency f_{max}.

Proof: Let the taskset Γ be schedulable at frequency $f' = U_{cpu} * f_{max} - \epsilon$, for some $\epsilon > 0$. At this frequency f', the CPU utilization of the taskset will be $U_{cpu} * f_{max}/f' > 1$. ■

Therefore, as an optimization, the initial estimate of the bounds, f_{init}^{low} and f_{init}^{high}, containing the minimum frequency can be set to $f_{init}^{low} = f_{init}^{high} = U_{cpu} * f_{max}$.

Based on the final frequency range $[f_{low}, f_{high}]$ returned by RatchetSearch, a binary search over the estimated range converges to the lowest frequency which allows a taskset to be schedulable, for a given schedulability-analysis technique.

Theorem 2: CycleSolo-CPU converges to the lowest CPU frequency f_{min} at which Γ is schedulable using analysis S.

Proof: The proof follows from Lemmas 1, 2, 3 and 4. ■

The time-complexity of ratchet search is linear in the number of tasks. The CycleSolo slack-calculation step has pseudo-polynomial complexity due to the response-time test.

Example: Consider a taskset Γ, with two implicit-deadline tasks τ_1 = (C_1=10, G_1=8, T_1=50), and τ_2 = (C_2=20, G_2=5, T_2=80). For this example, assume that the CPU intervention required for accelerator access $G_m = 0$. To determine schedulability we consider the request-driven analysis. The initial range estimates, f_{init}^{low} and f_{init}^{high}, are set to the CPU utilization 0.45. For τ_1, the only effective scheduling point to consider is $t = 50$, which is τ_1's deadline. By calculating the CPU execution and slack up to time $t = 50$, we obtain the minimum CPU frequency estimate $f_1^{est} = C_1/(50 - G_1 - G_2) = 0.27$. However, as $f_1^{est} < f_{init}^{low}$ (Case I of RatchetSearch) the bounds are not updated. Subsequently, we need to determine the lowest frequency f_2^{est}, which ensures τ_2 is schedulable in the presence of interference from τ_1, running at CPU frequency $f_1^{high} = 0.45$ (Lemma 3). From Lemma 2, the effective scheduling points we need to consider are $t = 42$, corresponding to an *effective* scheduling point of task τ_1, and $t = 80$, corresponding to τ_2's deadline. However, the request-driven analysis indicates that there is no slack up to $t' = 53$. Hence, we only need to consider $t = 80$. Computing the CPU execution and slack up to $t = 80$ yields $f_2^{est} = (2 * C_1 + C_2)/(80 - G_1 - G_2) = 0.597$. As $f_2^{est} > f_1^{high}$, the upper bound is updated to $f_2^{high} = 0.597$ (Case III of RatchetSearch), and the lower bound f_2^{low} is set to $f_1^{high} = 0.45$. Therefore, the final CPU frequency range is $[f_2^{low} = 0.45, f_2^{high} = 0.597]$. Performing a binary search over this range yields the minimum CPU frequency $f_{min} = 0.597$, ensuring schedulability using the request-driven analysis.

CycleSolo-Accel: Consider a uniprocessor whose frequency is not adjustable, with a single accelerator which supports frequency scaling. To minimize energy, we need to find the lowest-possible accelerator frequency at which all tasks meet their deadlines. In practice, such a scenario rarely exists, but CycleSolo-Accel helps bootstrap our *CycleTandem* algorithm.

Algorithm 5 in the Appendix presents the pseudo-code for *CycleSolo-Accel*, which considers tasks in decreasing priority order. For each task $\tau_i \in \Gamma$, CycleSolo-Accel uses RatchetSearch to compute the range $[f_i^{low}, f_i^{high}]$, which contains the lowest accelerator frequency which ensures that tasks $\tau_h \in \Gamma \mid h \leq i$ are schedulable. CycleSolo-Accel is identical to CycleSolo-CPU, except that it (i) calculates the frequency using the accelerator workload in the critical zone, and (ii) only performs the frequency-range estimation for tasks with accelerator segments. Thus, the results proved for CycleSolo-CPU also hold in the context of CycleSolo-Accel.

CycleSolo-ID: Consider a uniprocessor coupled with a single non-preemptive accelerator, where their operating frequencies can only be scaled by the same factor. To minimize energy we need to find the lowest-possible *identical* frequency-scaling factor at which all tasks meet their deadlines. In practice, this scenario may exist when a CPU is combined with an on-chip accelerator, and both share the same oscillator.

Algorithm 6 in the Appendix presents the pseudo-code for *CycleSolo-ID*, which considers tasks in decreasing order of their priority. For each task $\tau_i \in \Gamma$, CycleSolo-ID uses RatchetSearch to compute the range $[f_i^{low}, f_i^{high}]$, which contains the lowest common frequency-scaling factor which can ensure that tasks $\tau_h \in \Gamma \mid h \leq i$ are schedulable. CycleSolo-ID is identical to CycleSolo-CPU, except that it calculates the frequency using both the CPU and accelerator workload in the critical zone. Thus, all the results proved for CycleSolo-CPU also hold in the context of CycleSolo-ID.

Like SysClock, in theory, the CycleSolo algorithms can also use a per-task binary search to converge to the minimum frequency f_i, which ensures that a task τ_i is schedulable in the presence of high-priority interference. This approach would entail multiple calls to the pseudo-polynomial response-time-analysis. Therefore, for n tasks and an operating frequency range f_{op}, the per-task-binary-search has $n * log(f_{op})$ pseudo-polynomial complexity. However, the ratchet-search-based-technique has $n + log(f_{rs})$ pseudo-polynomial complexity, where $f_{rs} < f_{op}$ is the range returned by RatchetSearch.

V. CycleTandem Algorithm

Consider a uniprocessor coupled with a single non-preemptive accelerator, where the CPU and accelerator operating frequencies can be set independently. Therefore, to minimize the energy consumption of the system, the CPU and accelerator frequencies need to be optimized in *tandem*.

Algorithm 3 CycleTandem Algorithm

procedure CYCLETANDEM(Γ, ϵ_{conv})
  f_{cpu}^{solo} = CycleSolo-CPU(Γ, ϵ_{conv})
  f_{acc}^{solo} = CycleSolo-Accel(Γ, ϵ_{conv})
  $f_{cpu}^{up}, f_{acc}^{up}$ = ComputeRange($\Gamma, f_{cpu}^{solo}, f_{acc}^{solo}, \epsilon_{conv}$)
  if $f_{cpu}^{up} - f_{cpu}^{solo} < f_{acc}^{up} - f_{acc}^{solo}$ **then**
    f_{cpu}, f_{acc} = SearchRange-CPU($\Gamma, f_{cpu}^{up}, f_{cpu}^{solo}$)
  else
    f_{cpu}, f_{acc} = SearchRange-Acc($\Gamma, f_{cpu}^{up}, f_{cpu}^{solo}$)
  end if
  return f_{cpu}, f_{acc}
end procedure

Though the CPU and accelerator frequencies can be set independently, the taskset-schedulability constraint introduces a dependency between the CPU and accelerator frequencies. We now prove the "*See-Saw* Theorem" which shows the relationship between the CPU and accelerator frequencies.

Theorem 3: For a taskset Γ to be schedulable according to analysis S, both the CPU frequency f_{cpu} and the accelerator frequency f_{acc} cannot be less than the minimum common frequency, f_{id}^{solo}, determined by CycleSolo-ID.

Proof: Suppose Γ is schedulable by S, when both the CPU and accelerator frequencies $f_{cpu}, f_{acc} < f_{id}^{solo}$. Therefore, Γ will be schedulable if both the accelerator and CPU frequencies are set to $f_{id} = max(f_{cpu}, f_{acc}) < f_{id}^{solo}$. However, given an analysis technique S, CycleSolo-ID returns the minimum common frequency, f_{id}^{solo}. Thus, there is a contradiction. ■

We now see that there is a *see-saw* relationship between f_{acc} and f_{cpu}. If one is reduced to be less than f_{id}^{solo}, the other will increase and *always* be greater than or equal to f_{id}^{solo}. Thus, we can also conclude that f_{acc} is monotonically non-increasing with f_{cpu}, and vice-versa. Therefore, given an analysis technique, for every *feasible* CPU frequency f_{cpu}, there exists a *unique* minimum accelerator frequency, f_{acc}, which guarantees schedulability. This accelerator frequency can be found by scaling the taskset to the CPU frequency, and then computing the corresponding minimum accelerator frequency using CycleSolo-Accel. Similarly, for every *feasible* accelerator frequency f_{acc}, CycleSolo-CPU can find the minimum CPU frequency, f_{cpu}, guaranteeing schedulability.

It is trivial to show that, for a fixed CPU (accelerator) frequency, the minimum corresponding accelerator (CPU) frequency minimizes energy. Therefore, given the one-to-one *see-saw* relationship between the CPU and accelerator frequencies, it is sufficient to find the *optimal* accelerator frequency in order to compute the *optimal* CPU frequency, and vice versa. Thus, an exhaustive search in the range of feasible accelerator frequencies will yield the optimal frequency pair $(f_{cpu}^{opt}, f_{acc}^{opt})$.

Theorem 4: The energy-optimal accelerator frequency f_{acc}^{opt} always lies in $[f_{acc}^{solo}, f_{acc}^{up}]$, where f_{acc}^{solo} is the frequency returned by CycleSolo-Accel, and f_{acc}^{up} is the minimum feasible accelerator frequency guaranteeing schedulability, when the CPU is set to the CycleSolo-CPU frequency f_{cpu}^{solo}.

Proof: If we set the CPU frequency to be f_{cpu}^{solo}, there may be some slack in the system which can be utilized to reduce the accelerator frequency below its maxima. As f_{cpu}^{solo} is the lowest feasible CPU frequency (Theorem 2), the corresponding accelerator frequency f_{acc}^{up}, can safely upper-bound the range which contains the optimal accelerator frequency, as any frequency above f_{acc}^{up} will not be energy-efficient. Additionally, CycleSolo-Accel yields the lowest feasible accelerator frequency f_{acc}^{solo} (corollary of Theorem 2). Therefore, the optimal accelerator frequency f_{acc}^{opt} must lie in $[f_{acc}^{solo}, f_{acc}^{up}]$. ■

As a corollary, if we choose the accelerator frequency to be f_{acc}^{solo}, we obtain the CPU frequency f_{cpu}^{up}. Therefore, the optimal CPU frequency f_{cpu}^{opt} will lie in $[f_{cpu}^{solo}, f_{cpu}^{up}]$.

For CycleSolo, assuming a convex energy function, the energy consumption is minimized at the minimum *feasible* frequency, which is independent of the power-model parameters. However, for CycleTandem, the *energy-optimal* frequency pair depends on the power-model parameters, due to the non-linear see-saw schedulability-analysis-dependent relationship between the accelerator and CPU frequencies which renders the energy function non-convex. This non-linearity is caused by an effect we refer to as *slack-squeezing*. Given some usable slack, the frequency of the CPU (accelerator) depends on the effective utilization of the CPU (accelerator) up to a scheduling point. Therefore, a $\delta > 0$ increase in the CPU (accelerator) frequency can cause a $\delta' > \delta$ decrease in the corresponding accelerator (CPU) frequency. In other words, if $\delta_{cpu} > \delta_{acc}$, the CPU *squeezes* the available slack more efficiently than the accelerator, and vice versa. The following theorem highlights the effect of slack squeezing on energy.

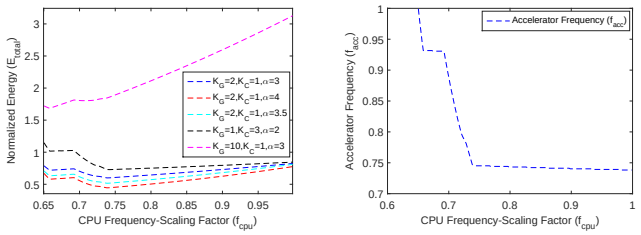
Theorem 5: Consider a feasible CPU frequency $f_{cpu} \in [f_{cpu}^{solo}, f_{max})$, and its corresponding minimum feasible accelerator frequency f_{acc}. If we increase the CPU frequency by δ, then the energy consumption decreases *if and only if*, the gradient of the accelerator frequency w.r.t the CPU frequency,

$$|\nabla_{f_{cpu}, f_{acc}}| = |\frac{\partial f_{acc}}{\partial f_{cpu}}| > \frac{K_{cpu} * U_{cpu} * f_{cpu}^{\alpha-1}}{K_{acc} * U_{acc} * f_{acc}^{\alpha-1}} \quad (5)$$

Proof: The proof can be found in the Appendix. ■

The corollary of the above theorem, corresponding to the accelerator frequency also holds.

Consider a taskset Γ with CPU and accelerator utilization U_{cpu} and U_{acc}. Let (f_{cpu}, f_{acc}) constitute a feasible frequency pair, ensuring Γ is schedulable. Therefore, based on the energy model described in Section III-B, the *normalized* energy consumption of the system in the hyperperiod is given by $E_{total} = E_{cpu} + E_{acc}$, where $E_{cpu} = K_{cpu} * U_{cpu} * f_{cpu}^{\alpha-1} + E_{static}^{cpu}$ is the CPU energy, and $E_{acc} = K_{acc} * U_{acc} * f_{acc}^{\alpha-1} + E_{static}^{acc}$ is the accelerator energy. Since we have no control on reducing the static power of the system, we conservatively assume that it also shares a direct dependency on the operating voltage and frequency. Thus, both E_{cpu} and E_{acc} are indeed convex non-decreasing functions in f_{cpu} and f_{acc} respectively, and their combination E_{total} will also be a convex function in f_{cpu} and f_{acc}. However, if the schedulability constraint is applied, then due to the *see-saw* theorem, f_{acc} can become a non-linear non-convex function of f_{cpu} and vice-versa, which in turn



(a) (b)

Fig. 1. (a) Energy and (b) Accelerator Frequency vs CPU Frequency

can render the energy function E_{total} non-convex. This non-convexity unfortunately makes it difficult to find the optimal frequency-pair which minimizes the total energy consumption.

By looking at the function for $E_{total} \forall (\alpha > 3)$, it may seem that if $K_{cpu} * U_{cpu} > K_{acc} * U_{acc}$ then to minimize E_{total}, $f_{cpu} < f_{acc}$, and vice versa. However, the non-linear relationship between f_{acc} and f_{cpu} causes this statement to not always hold, except under special conditions.

Theorem 6: If $K_{cpu} * U_{cpu} >> K_{acc} * U_{acc}$, then the energy-optimal CPU frequency for taskset Γ approaches f_{cpu}^{solo}.

Proof: In this case, $E_{total} \approx E_{cpu}$. As E_{cpu} is minimized at f_{cpu}^{solo}, therefore as $f_{cpu} \rightarrow f_{cpu}^{solo}$, E_{total} is also minimized. ■

As a corollary, if $K_{acc} * U_{acc} >> K_{cpu} * U_{cpu}$, then for taskset Γ the energy-optimal accelerator frequency $f_{acc} \rightarrow f_{acc}^{solo}$. Therefore, the energy-optimal frequency pair depends on (i) the ratio of the accelerator and CPU utilization, (ii) the power-model parameters, and (iii) the schedulability-analysis-dependent see-saw relationship between f_{acc} and f_{cpu}.

Example: Consider a taskset Γ consisting of 5 implicit-deadline tasks with their execution parameters (C, G_e, G_m, T): $\{(0.27, 15.11, 1, 84), (24.08, 0, 0, 221), (37.89, 0, 0, 231), (13.19, 0, 0, 330), (78.44, 45.18, 1, 427)\}$. Figure 1 illustrates the relationship of the CPU frequency, f_{cpu} with the energy consumption (Figure 1(a)) and the accelerator frequency (Figure 1(b)). In (Figure 1(a)), note the non-convexity of the energy function and the dependence of the energy-optimal frequency on the power-model parameters. Additionally, observe the non-linear relationship between the accelerator frequency and the CPU frequency (Figure 1(b)). Notice that, for $f_{cpu} \approx 0.7$ to 0.75, the accelerator frequency drops rapidly due to the slack-squeezing effect.

It appears that the energy-optimal frequency pair cannot be found analytically, and instead we need to search the *feasible* space to find the optimal solution. Of the CPU and accelerator frequency ranges, the smaller range can be chosen. In theory, a gradient descent over the feasible range can yield a local optima. Alternatively, if the range is small, an exhaustive search with a small step size can quickly yield a good solution.

Algorithm 3 presents the CycleTandem algorithm, which can utilize any reasonable search technique, *SearchRange*, in the feasible range. In our experiments, we use a greedy-search algorithm which first computes the energy at both ends of the feasible range. Subsequently, it chooses the endpoint with the lower energy, and increases/decreases the CPU (accelerator) frequency in small steps, until the first local minima is reached.

Accommodating Discrete Frequencies: In most systems, the frequency of the processor (accelerator) can only be set to discrete values. For CycleSolo, we can compute the minimum frequency, and pick the next-greater discrete frequency. Alternatively, the binary-search step of CycleSolo, can only consider the discrete frequencies. If no discrete frequencies lie in the feasible range, we pick the next-greater frequency than the upper bound of the range. Similarly, for CycleTandem, we can search over the discrete frequencies in the feasible range, and pick the frequency pair yielding the minimum energy.

VI. CycleSolo and CycleTandem in the Fully-Partitioned Multi-Core Setting

We now extend CycleSolo and CycleTandem for fully-partitioned multi-core processors, coupled with a single accelerator. We assume that if the CPU supports frequency scaling, then all its cores *must* be set to the same frequency.

Consider a taskset Γ, partitioned among m cores, such that each core has a subset of tasks $\Psi_j \subset \Gamma \mid j = 1, 2, .., m$. The context in which each algorithm can be used is as follows:

1) *CycleSolo-CPU*: all the CPU cores can be set to a common frequency, and the accelerator frequency is not adjustable.

2) *CycleSolo-Accel*: the frequency of the CPU cores cannot be scaled, and the accelerator frequency is adjustable.

3) *CycleSolo-ID*: all the CPU cores and the accelerator, can only be set to an identical frequency-scaling factor.

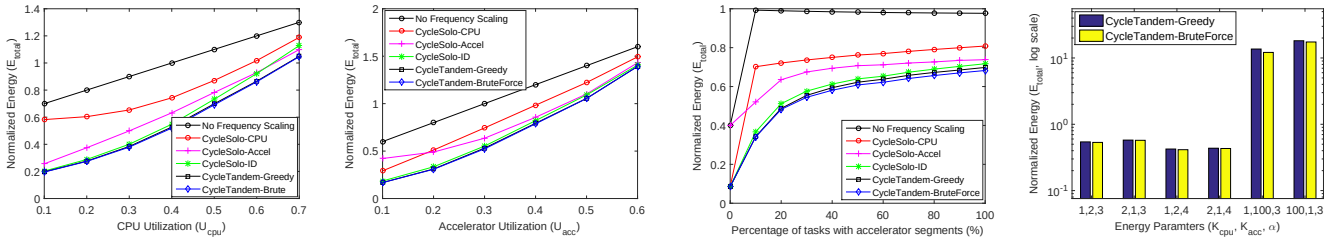
4) *CycleTandem*: all the CPU cores can be set to a common frequency, and the accelerator frequency is also adjustable.

Consider the CycleSolo algorithms. The best solution is still a *single* lowest frequency. Therefore, Lemma 3 holds, and we can use the RatchetSearch frequency-range estimation algorithm. The only differences from the uniprocessor case are as follows: (i) for each core j, interference is computed considering only the tasks $\tau_i \in \Psi_j$, and (ii) remote blocking by tasks on other cores is taken into account.

Theorem 7: Given a taskset Γ, and its schedulable partition Ψ onto an m-core processor, then the CycleSolo algorithms converge to the lowest CPU/accelerator/common frequency f_{min} at which partition Ψ is schedulable using analysis S.

Proof: The CycleSolo algorithms compute a single frequency range. Thus, Theorem 2 holds, and a *single* binary search, testing schedulability across m cores using analysis S, over the feasible range returned by RatchetSearch, can converge to the lowest frequency f_{min}. ■

Consider the CycleTandem algorithm. We share a single CPU frequency across all cores. Therefore, the *see-saw* theorem still holds, and a one to one mapping between every common CPU and accelerator frequency still exists. Thus, multi-core CycleTandem can use CycleSolo-CPU and CycleSolo-Accel to compute a lower bound on the common CPU frequency f_{cpu}^{solo}, and the accelerator frequency f_{acc}^{solo} respectively. Therefore, using f_{cpu}^{solo} and f_{acc}^{solo}, the safe upper bounds on the accelerator frequency f_{acc}^{up}, and the common CPU frequency f_{cpu}^{up} can be calculated. Subsequently, performing an exhaustive search over the feasible CPU or accelerator range can yield the *energy-optimal* frequency pair. Similarly, we can also use the greedy search proposed for uniprocessors.



(a) (b) (c) (d)

Fig. 2. Energy as a function of (a) CPU Utilization, (b) Accelerator Utilization, (c) % of tasks using the accelerator (d) power-model parameters

Note that, for CycleSolo-CPU and CycleTandem, if each CPU core can have its own frequency, we cannot perform a search over a *single* frequency range. Additionally, due to self-suspension, the response time of a task on one core, can depend on the frequency chosen for another core. This is due to the dependence of a task τ_i's response time on the response time of higher-priority tasks $\tau_h \mid h < i$, which may be allocated to other cores. Hence, more sophisticated search techniques are required, which are beyond the scope of this work.

Task Partitioning: Load balancing is often used to determine an energy-efficient partition in multi-core systems [24]. When the CPU frequency is common across all cores, the core with the highest *effective* load determines the CPU frequency. Thus, load balancing is useful as it tries to minimize the maximum load across cores [24]. Among task-partitioning heuristics studied in the literature, the *Worst-Fit Decreasing* (WFD) algorithm is known to yield a well-balanced partition [24]. WFD allocates tasks to cores in non-decreasing order of their utilization. Given a task to be allocated, WFD assigns it to the core with the least utilization. When WFD can allocate tasks to use only m cores, it is equivalent to *List Scheduling*.

However, the blocking and self-suspension penalties introduced by accessing the accelerator affects the frequency estimation. Therefore, motivated by the work in [34], we propose a modified version of WFD called *Sync-Aware* WFD or SA-WFD, which for a taskset Γ, first computes the fraction of CPU load belonging to tasks which utilize the accelerator, $\gamma_{acc} = U_{cpu}^{acc}/U_{cpu}^{total}$, and subsequently allocates $\psi = \lceil \gamma_{acc} * m \rceil$ cores for these tasks. Finally, our heuristic balances the load while constraining the tasks using the accelerator to ψ cores. Thus, restricting the self-suspension penalties to a few cores.

VII. Experimental Evaluation

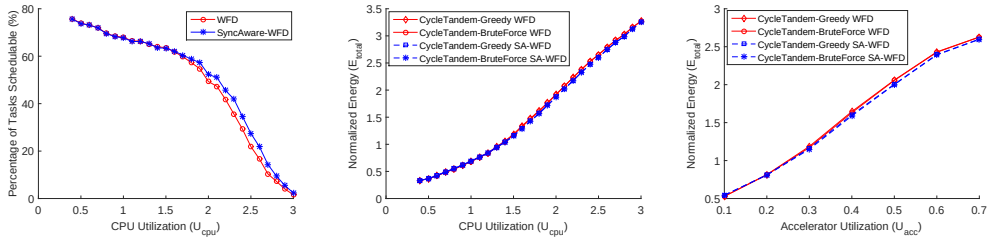
We now assess the energy-savings delivered by our proposed CycleSolo and CycleTandem algorithms. We first present analytical evaluations using the request-driven analysis [17] and the power model presented in Section III-B. Subsequently, we present experiments performed on the NVIDIA TX2 [35] embedded platform to demonstrate the practical applicability of our proposed techniques. We assume fully-partitioned fixed-priority scheduling, with task priorities assigned using the Rate-Monotonic policy [36]. To the best of our knowledge, no other energy-saving real-time scheduling techniques exist in the context of hardware accelerators. Therefore, we compare against the case without energy management.

We compare our proposed techniques on the basis of analytically-computed energy savings over the hyperperiod. Every data point plotted is an average of 5000 tasksets, randomly generated using the UUniFast-Discard [37] algorithm, such that no task has a CPU/accelerator utilization greater than 0.4. We consider sporadic tasks with the minimum inter-arrival time randomly assigned to be between 5 and 500 time units.

Uniprocessor Experiments: Figures 2 (a), (b) and (c), plot the average normalized energy as we vary (a) the CPU utilization keeping U_{acc} = 0.3, (b) the accelerator utilization keeping U_{cpu} = 0.4, (c) the fraction of tasks using the accelerator keeping U_{cpu} = 0.4 and U_{acc} = 0.3. The power-model parameters are set to: K_{cpu} = 1, K_{acc} = 2 and α = 3. Consider Figure 2 (a). As the CPU utilization is varied, our proposed CycleTandem greedy-search heuristic returns a solution that in the worst-case consumes 1.53% greater energy than the brute-force search. Compared to the case without energy management, i.e., executing all tasks at the maximum frequency, CycleTandem with the greedy-search heuristic on average delivers up to 71.88% lower energy consumption. Compared to the case without energy management, CycleSolo-CPU, CycleSolo-Accel and CycleSolo-ID deliver up to 27.42%, 63.41% and 71.03% lower energy respectively. Similar trends can be observed for Figures 2 (b) and (c), where the CycleTandem greedy-search heuristic yields a result with near-optimal energy savings. Figure 4 shows the computed CPU and accelerator frequencies for our proposed techniques. For all three sub-figures, (4 (a), (b) and (c)) the CycleSolo-Accel and CycleSolo-CPU frequencies are never greater than the CycleSolo-ID frequency. For CycleTandem, note the see-saw relationship between the accelerator and CPU frequencies around the CycleSolo-ID frequency.

We also performed experiments to determine the impact of varying the power-model parameters on the CycleTandem greedy-search heuristic. Figure 2(d) plots the average normalized energy for five sets of power-model parameters. Note that, when the power-model parameters are varied, the greedy-search heuristic yields solutions that in the worst-case consume only up to 12.48% more energy than the brute-force search.

Multicore Experiments: In Section VI, we proved that given a taskset Γ, and its schedulable partition P onto an m-core processor, we can find the optimal solution for all the CycleSolo algorithms. Therefore, we consider m = 4 cores, and focus on comparing our proposed Sync-Aware WFD (SA-WFD) heuristic against WFD, in the context of CycleTandem.



(a) (b) (c)

Fig. 3. WFD vs Sync-Aware WFD (SA-WFD) (a) schedulability comparisons, (b) Energy vs CPU utilization, and (c) Energy vs Accelerator utilization

TABLE I
EXPERIMENTAL TASKSETS DEPLOYED ON THE NVIDIA TX2

Taskset (Γ)	Tasks	U_{cpu}	U_{acc}	Description
Γ_1	8	1.75	0.39	High U_{cpu}, High U_{acc}
Γ_2	4	0.44	0.14	Low U_{cpu}, High U_{acc}
Γ_3	4	1.94	0.02	High U_{cpu}, Low U_{acc}
Γ_4	6	0.10	0.52	Low U_{cpu}, High U_{acc}

TABLE II
CPU,GPU FREQUENCIES USED ON THE NVIDIA TX2 (MHZ)

Γ	CS-CPU	CS-Accel	CS-ID	C-Tandem
Γ_1	**1728**,1135	2032,**1033**	1881,**1033**	**1728**,1135
Γ_2	**499**,1135	2032,**319**	806,421	499,523
Γ_3	**1267**,1135	2032,**115**	**1267**,727	**1267**,217
Γ_4	**346**,1135	2032,**931**	1728,**931**	653,**931**

Figure 3 (a) compares SA-WFD and WFD on the basis of schedulability. The CPU utilization is varied while keeping U_{acc} = 0.3. Observe that, as the CPU utilization increases, SA-WFD yields more schedulable partitions than WFD, and can schedule up to 6.3% more tasksets than WFD. We also illustrate the utility of our proposed greedy-search heuristic for CycleTandem. Figures 3 (b) and (c) plot the average normalized energy for CycleTandem as we vary (b) the CPU utilization keeping U_{acc} = 0.3, (c) the accelerator utilization keeping U_{cpu} = 1.5. Note that, for both WFD and SA-WFD, the CycleTandem greedy-search heuristic returns a solution that in the worst-case consumes 1.38% and 1.42% greater energy than the brute-force search, respectively. Comparing the CycleTandem greedy-search heuristic across the two partitioning techniques, indicates that SA-WFD on average yields a solution with up to 3.3% lower energy consumption than WFD. Thus, SA-WFD yields better schedulability and energy savings than WFD, with the same algorithmic complexity.

Experiments on the NVIDIA TX2: We examine the practical energy savings delivered by our algorithms, by performing experiments on the TX2 embedded platform [35]. The TX2 contains 4 ARM A57 CPU cores, 2 Denver CPU cores and an integrated 256-core Pascal GPU [35]. In our experiments we disable the 2 Denver CPU cores. The ARM cores can be set to 12 discrete frequencies ranging from 345.6 MHz to 2.03 GHz, and the GPU can be set to 12 discrete frequencies ranging from 114.75 MHz to 1.13 GHz. Note that all 4 ARM cores lie in the same voltage domain, and can *only* be set to the same frequency. However, the GPU lies in a separate power domain and its frequency can be independently set.

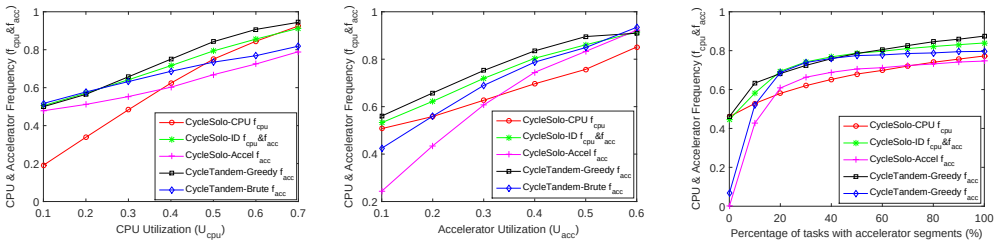
We consider four tasksets $\Gamma_{i|i=1,2,3,4}$, described in Tables I and IV. Each taskset consists of matrix-multiplication tasks accessing the GPU. The length of each task's CPU and GPU segments are configurable. An implementation of the MPCP-based synchronization approach was used to arbitrate GPU access. Tasks are allocated to cores using Sync-Aware WFD. For all our proposed techniques we compute the frequency(ies) using our theoretical model, and then choose the next larger available CPU/GPU frequency. Our algorithms depend on the ratio K_{cpu}/K_{gpu}. In our calculations the power-model parameters are set to: K_{cpu} = 1, K_{acc} = 1 and α = 3. This is based on the fact that both the CPU and the GPU lie on the same chip [35]. We also assume that the task WCET scales with the frequency. The frequencies we computed for the different tasksets can be found in Table II.

TABLE III
POWER MEASUREMENTS ON THE NVIDIA TX2 (MILLIWATTS)

Γ	Type	No VFS	CS-CPU	CS-Accel	CS-ID	C-Tandem
Γ_1	Total	1826.91	1435.20	1805.91	1601.79	**1425.93**
	CPU	1338.36	995.07	1363.23	1160.09	**985.45**
	GPU	488.03	440.12	**440.12**	441.69	440.48
Γ_2	Total	906.23	607.38	835.82	537.31	**530.12**
	CPU	607.14	**305.06**	611.41	312.01	305.59
	GPU	299.09	302.32	**224.41**	225.30	224.53
Γ_3	Total	1405.27	797.32	1363.00	784.31	**782.74**
	CPU	1211.90	636.12	1209.90	636.12	**629.53**
	GPU	193.37	161.20	**153.10**	161.20	153.20
Γ_4	Total	632.57	419.61	625.28	539.78	**404.65**
	CPU	469.41	**238.95**	467.97	381.29	244.86
	GPU	163.15	180.65	**157.30**	158.48	159.78

The energy consumption of each taskset is measured over multiple hyper-periods by using the on-board INA3221 power monitors [38]. To compute the energy consumption, we periodically read the monitors, which measure the power drawn by the 4 ARM cores and the GPU, every 10 ms. The possible sources of error in our measurement are: (i) the INA3221 has a measurement accuracy of 0.1%, and (ii) the overhead of reading the INA3221 on the CPU. However, we believe that these sources of error are small and do not affect our results.

To get a flavor of the realized energy savings, $\Gamma_{i|i=1,2,3,4}$ have differing amounts of CPU and GPU utilization. The results are described in Table III. Compared to the case without frequency scaling, CycleTandem (C-Tandem) delivers the most energy savings, and for taskset Γ_3, we observe 44.29% lower power, which translates to a 1.78x increase in the life of a battery-powered system. Similarly, when the CPU utilization is low (Γ_2 and Γ_4), using CycleSolo-CPU (CS-CPU) yields up to 32% reduction in power. Note that, as we reduce the operating



(a) (b) (c)

Fig. 4. CPU and Accelerator frequency as a function of (a) CPU Utilization, (b) Accelerator Utilization, and (c) Percentage of tasks using the accelerator

Algorithm 4 Binary Search Minimizing Frequency

```
1: procedure BINARYSEARCH(Γ, ε_conv, f_high, f_low)
2:     while f_high − f_low > ε_conv do
3:         f_est = (f_high + f_low)/2          ▷ chosen frequency
4:         Γ' = ScaleTaskset-Frequency(Γ, f_est)
5:         if Γ' is Schedulable then f_high = f_est
6:         else f_low = f_est
7:         end if
8:     end while
9:     return f_high
10: end procedure
```

frequency, we observe greater reduction in CPU energy as compared to GPU energy. This is likely because (i) the GPU energy also depends on the frequency of the GPU memory, and, (ii) since there are multiple CPU cores, there is more opportunity to reduce the CPU frequency.

VIII. Conclusion and Future Work

In this paper, we introduced energy-saving fixed-priority scheduling techniques for real-time systems with non-preemptive hardware accelerators. We first proposed the CycleSolo algorithms for systems where only the CPU or accelerator frequency can be set. CycleSolo utilizes our *ratchet search* to compute a tight range containing the lowest frequency, following which a binary search in this range yields the optimal frequency, for a given schedulability-analysis technique.

We also introduced the CycleTandem algorithm for systems where the processor and accelerator frequencies can be independently set. We compute the feasible ranges containing the energy-optimal CPU and accelerator frequency pair, and propose a greedy-search heuristic to find a good solution.

Finally, we extend the CycleSolo and CycleTandem algorithms to fully-partitioned multi-core processors, where all the CPU cores must be set to the same frequency. In this context, we also propose the *Sync-Aware* Worst-Fit Decreasing heuristic which yields better schedulability and energy savings, than Worst-Fit Decreasing. Analytical experiments show that our proposed techniques can deliver significant energy savings. In addition, practical experiments on the NVIDIA TX2 indicate significant energy savings and validate our theoretical findings.

In future work, we will focus on multi-core systems where the frequency of each processor core can be independently set. We will also consider systems which contain multiple accelerators, or accelerators which support software partitioning.

TABLE IV
NVIDIA TX2 Taskset Parameters (ms)

Γ	Implicit-deadline ($D = T$) Task Parameters (C,G_e,G_m,T)
Γ_1	(20,4,0.3,100),(10,6,0.3,100),(30,4,0.3,150),(50,18,0.3,200), (50,9,0.3,300),(100,30,0.3,300),(100,13,0.3,600),(400,30,0.3,1200)
Γ_2	(10,13,0.3,150),(50,4,0.3,300),(60,18,0.3,600),(125,9,0.3,1200)
Γ_3	(200,4,0.3,450),(300,4,0.3,600),(400,4,0.3,900),(1000,4,0.3,1800)
Γ_4	(2,18,0.3,250),(4,30,0.3,250),(10,47,0.3,500),(20,47,0.3,500), (10,89,0.3,750),(30,30,0.3,1500)

Appendix

Theorem 1: If all tasks in Γ have *at most* one critical section executed on the accelerator, then the request-driven analysis *always* dominates the job-driven analysis.

Proof: Consider the case where all tasks in Γ have at most one critical section executed on the accelerator. Both the request-driven and job-driven analyses determine schedulability using response-time calculations, and hence, use Equation 1. The only difference lies in the calculation of *high-priority* blocking faced by each task τ_i. In the request-driven analysis, the number of instances of each high-priority task τ_h, which contribute to blocking τ_i, is given by the term $\beta_{i,h}$, which upper bounds the number of accelerator requests made by a higher-priority task τ_h, while τ_i is being blocked. Thus, $\beta_{i,h} = \left\lceil \frac{B_i+W_h-E_h}{T_h} \right\rceil$, where B_i upper bounds the time for which τ_i is blocked [2][17]. In contrast, for the job-driven analysis, the number of instances of each high-priority task τ_h, which contribute to blocking τ_i, is given by the term $\alpha_{i,h}$, which upper bounds the number of accelerator requests made by a higher-priority task τ_h, during τ_i's response time. Thus, $\alpha_{i,h} = \left\lceil \frac{W_i+W_h-E_h}{T_h} \right\rceil$, where W_i upper bounds τ_i's worst-case response time [2]. From Equation 1, for every task τ_i, we can conclude that $B_i < W_i$. Therefore, for every feasible pair of τ_i and τ_h, $\beta_{i,h} \leq \alpha_{i,h}$. This implies that, for the given context, the request-driven analysis always computes a tighter blocking estimate, as compared to the job-driven analysis. Thus, from Equation 1, we can also conclude that the request-driven analysis always computes a tighter worst-case response-time estimate, as compared to the job-driven analysis. ■

Theorem 5: Consider a feasible CPU frequency $f_{cpu} \in [f_{cpu}^{solo}, f_{max})$, and its corresponding minimum feasible accelerator frequency f_{acc}. If we increase the CPU frequency by δ, then the energy consumption decreases *if and only if*, the gradient of the accelerator frequency w.r.t the CPU frequency,

$$|\nabla_{f_{cpu},f_{acc}}| = |\frac{\partial f_{acc}}{\partial f_{cpu}}| > \frac{K_{cpu} * U_{cpu} * f_{cpu}^{\alpha-1}}{K_{acc} * U_{acc} * f_{acc}^{\alpha-1}} \quad (6)$$

Algorithm 5 Minimizing Accelerator Frequency

procedure CYCLESOLO-ACCEL(Γ, ϵ_{conv})
  $f_{low} = f_{high} = U_{acc}$ ▷ initial bounds
  for $\tau_i \in \Gamma$ **do** ▷ from high to low priority
    f_{low}, f_{high} = EstimateFreqRange($\tau_i, \Gamma, f_{high}, f_{low}$)
  end for
  f_{min} = BinarySearch($\Gamma, \epsilon_{conv}, f_{high}, f_{low}$)
  return f_{min}
end procedure
procedure ESTIMATEFREQRANGE($\tau_i, \Gamma, f_{high}, f_{low}$)
  /*S = slack, ω = resp time, β = accelerator workload*/
  $f_h = f_{high}$, $S = \beta = \Delta = 0$, $f_{est} = 1$, BusyFlag=TRUE
  W = Calculate-HP-ResponseTime(f_h)
  B_i = CalculateBlocking(τ_i, Γ, f_h, W)
  $\omega_g = G_i - G_{cpu}$ ▷ Accelerator execution time
  $\omega = C_i + G_i + B_i$, $\omega' = 0$, $J_i = \omega$
  while $\omega < D_i$ **do**
    if BusyFlag == TRUE **then**
      $\Delta = D_i - \omega$
      while $\omega < D_i$ AND $\Delta > 0$ **do**
        $\omega' = \sum_{h=0}^{i-1} E_h * [\lfloor \frac{\omega + W_h - (E_h)}{T_h} \rfloor + 1]$
        $\omega' = \omega' + J_i + S$, $\Delta = \omega' - \omega, \omega = \omega'$
      end while
      BusyFlag = FALSE
    else ▷ Start of an idle period
      t=Find-EarliestSchedulingPoint(τ_i, Γ, ω)
      $S = S + (t - \omega), \omega = t, t' = \omega_g + S, \beta = \omega_g$
      if $\beta/t' < f_{est}$ **then**
        $f_{est} = \beta/t'$
      end if
      BusyFlag = TRUE
    end if
  end while
  f_{low}, f_{high} = RatchetSearch-Step($f_{high}, f_{low}, f_{est}$)
  return f_{low}, f_{high}
end procedure
procedure CALCULATEBLOCKING(τ_i, Γ, f_h, W)
  $G_{l,max} = max_{\tau_l \epsilon lp(\tau_i)}(G_l)$, $B = G_{l,max}/f_h$, $B' = 0$
  while B != B' **do**
    $B' = B$, $B = \frac{G_{l,max}}{f_h} + \sum_{\tau_h \epsilon hp(\tau_i)} \lceil \frac{B' + W_h - E_h}{T_h} \rceil * \frac{G_h}{f_h}$
  end while
  return B
end procedure

Proof: Given the power model from Section III-B, the gradients of the energy at any frequency pair (f_{cpu}, f_{acc}), are: $\partial E_{total}/\partial f_{cpu} = \alpha * K_{cpu} * U_{cpu} * f_{cpu}^{\alpha-1}$ and $\partial E_{total}/\partial f_{acc} = \alpha * K_{acc} * U_{acc} * f_{acc}^{\alpha-1}$. Now, for each frequency pair, assume that an increase in one frequency say f_{cpu}, causes a corresponding increase in energy (∂E_{total}), which can be compensated ($-\partial E_{total}$) by a decrease in the other frequency f_{acc}. In this case, we can divide one of the above gradients by the other, we get the ratio of the frequency changes which yields the same energy, i.e.,

$$|\nabla_0| = \frac{\partial E_{total}/\partial f_{cpu}}{\partial E_{total}/\partial f_{acc}} = \frac{\partial f_{acc}}{\partial f_{cpu}} = \frac{K_{cpu} * U_{cpu} * f_{cpu}^{\alpha-1}}{K_{acc} * U_{acc} * f_{acc}^{\alpha-1}}$$

Algorithm 6 Minimizing the Common Frequency

procedure CYCLESOLO-ID(Γ, ϵ_{conv})
  $f_{low} = f_{high} = U_{cpu}$ ▷ initial bounds
  for $\tau_i \in \Gamma$ **do** ▷ from high to low priority
    f_{low}, f_{high} = EstimateFreqRange($\tau_i, \Gamma, f_{high}, f_{low}$)
  end for
  f_{min} = BinarySearch($\Gamma, \epsilon_{conv}, f_{high}, f_{low}$)
  return f_{min}
end procedure
procedure ESTIMATEFREQRANGE($\tau_i, \Gamma, f_{high}, f_{low}$)
  /*S = slack, ω = resp time, β = total workload*/
  $f_h = f_{high}$, $S = \beta = \Delta = 0$, $f_{est} = 1$, BusyFlag=TRUE
  W = Calculate-HP-ResponseTime(f_h)
  B_i = CalculateBlocking(τ_i, Γ, f_h, W)
  $\omega = C_i + G_i + B_i$, $\omega' = 0$, $J_i = \omega$
  while $\omega < D_i$ **do**
    if BusyFlag == TRUE **then**
      $\Delta = D_i - \omega$
      while $\omega < D_i$ AND $\Delta > 0$ **do**
        $\omega' = \sum_{h=0}^{i-1} E_h * [\lfloor \frac{\omega + W_h - (E_h/f_h)}{T_h} \rfloor + 1]$
        $\omega' = \omega' + J_i + S$, $\Delta = \omega' - \omega, \omega = \omega'$
      end while
      BusyFlag = FALSE
    else ▷ Start of an idle period
      t=Find-EarliestSchedulingPoint(τ_i, Γ, ω)
      $S = S + (t - \omega), \omega = t, t' = \omega - B_i, \beta = \omega - B_i - S$
      if $\beta/t' < f_{est}$ **then**
        $f_{est} = \beta/t'$
      end if
      BusyFlag = TRUE
    end if
  end while
  f_{low}, f_{high} = RatchetSearch-Step($f_{high}, f_{low}, f_{est}$)
  return f_{low}, f_{high}
end procedure
procedure CALCULATEBLOCKING(τ_i, Γ, f_h, W)
  $G_{l,max} = max_{\tau_l \epsilon lp(\tau_i)}(G_l)$, $B = G_{l,max}/f_h$, $B' = 0$
  while B != B' **do**
    $B' = B$, $B = \frac{G_{l,max}}{f_h} + \sum_{\tau_h \epsilon hp(\tau_i)} \lceil \frac{B' + W_h - \frac{E_h}{f_h}}{T_h} \rceil * \frac{G_h}{f_h}$
  end while
  return B
end procedure

Now, for a feasible frequency pair (f_{cpu}, f_{acc}), if the absolute value of the gradient of the *see-saw* relationship at frequency f_{cpu}, $|\nabla_{f_{cpu}, f_{acc}}| > |\nabla_0|$, then the decrease in the accelerator frequency f_{acc} can more than compensate for the increase in energy caused by increasing the CPU frequency f_{cpu}, causing the overall energy to decrease. Conversely, if the energy decreases on increasing the CPU frequency f_{cpu} by a small value Δ, then the corresponding decrease in accelerator frequency f_{acc} by a small value Δ', is more than that required to compensate for the increase in energy caused by increasing f_{cpu}, leading to $|\nabla_{f_{cpu}, f_{acc}}| > |\nabla_0|$. ■

The source code for our algorithms can be found at *https://bitbucket.org/sandeepdsouza93/cycletandem*

ACKNOWLEDGEMENTS

The authors would like to thank Iljoo Baek and Anand Bhat for their suggestions and assistance with the experiments on the NVIDIA TX2 platform.

REFERENCES

[1] J. Wei, J. Snider, J. Kim, J. Dolan, R. Rajkumar and B. Litkouhi, "Towards a Viable Autonomous Driving Research Platform", in Proceedings of the *IEEE Intelligent Vehicles Symposium*, 2013.

[2] P. Patel, I. Baek, H. Kim and R. Rajkumar, "Analytical Enhancements and Practical Insights for MPCP with Self-Suspensions", in Proceedings of the *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2018

[3] S. Mittal, and J. Vetter, "A survey of methods for analyzing and improving GPU energy efficiency", in *ACM Computing Surveys*, 47(2), 2015

[4] Qualcomm Snapdragon, *https://www.qualcomm.com/snapdragon*

[5] NVIDIA tegra, *http://www.nvidia.com/object/tegra.html*

[6] Microsoft HoloLens, *https://www.microsoft.com/en-us/research/blog/second-version-hololens-hpu-will-incorporate-ai-coprocessor-implementing-dnns/*

[7] S. D'souza and R. Rajkumar, "Thermal Implications of Energy-Saving Schedulers", in Proceedings of the *Euromicro Conference on Real-Time Systems*, 2017

[8] F. Yao, A. Demers and S. Shenker, "A scheduling model for reduced CPU energy", in Proceedings of the *Annual Symposium on Foundations of Computer Science*, 1995

[9] K. Agarwal, K. Nowka, H. Deogun and D. Sylvester, "Power Gating with Multiple Sleep Modes", in Proceedings of the *International Symposium on Quality Electronic Design*, 2006

[10] NVIDIA GPU Performance State Interface, *https://docs.nvidia.com/gameworks/content/gameworkslibrary/coresdk/nvapi/group__gpupstate.html*

[11] ATI Dynamic Power Management, *https://wiki.archlinux.org/index.php/ATI#Dynamic_power_management*

[12] G. Elliott, B. Ward, and J. Anderson, "GPUSync: A framework for realtime GPU management", in Proceedings of the *IEEE Real-Time Systems Symposium*, 2013

[13] G. Elliott and J. Anderson, "Globally scheduled real-time multiprocessor systems with GPUs", in *Real-Time Systems*, 48(1):3474, 2012.

[14] G. Elliott and J. Anderson, "An optimal k-exclusion real-time locking protocol motivated by multi-GPU systems", in *Real-Time Systems*, 49(2):140170, 2013.

[15] R. Rajkumar, "Real-time synchronization protocols for shared-memory multiprocessors", in Proceesings of the *IEEE International Conference on Distributed Computing Systems*, 1990.

[16] J.-J. Chen et al., "Many suspensions, many problems: A review of self-suspending tasks in real-time systems", in *Technical Report 854*, Department of Computer Science, TU Dortmund, 2016

[17] K. Lakshmanan, D. de Niz, and R. Rajkumar, "Coordinated task scheduling, allocation and synchronization on multiprocessors", in Proceedings of the *IEEE Real-Time Systems Symposium*, 2009

[18] K. Bletsas, N. Audsley, W.-H. Huang, J.-J. Chen, and G. Nelissen, "Errata for three papers (2004-05) on fixed-priority scheduling with self suspensions", in *Technical Report CISTER-TR-150713*, CISTER, 2015.

[19] H. Kim, P. Patel, S. Wang and R. Rajkumar, "A Server-based Approach for Predictable GPU Access Control", in Proceedings of *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2017

[20] S. Saewong and R. Rajkumar, "Practical voltage-scaling for fixed-priority RT-systems," in Proceedings of the *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2003

[21] A. Hakan and Q. Yang, "Energy-aware partitioning for multiprocessor real-time systems," in Proceedings of the *International Symposium on Parallel and Distributed Processing*, 2003

[22] T. AlEnawy and H. Aydin, "Energy-aware task allocation for rate monotonic scheduling", in Proceedings of the *IEEE Real Time and Embedded Technology and Applications Symposium*, 2005

[23] J. J. Chen, C. Y. Yang, H. I. Lu and T. W. Kuo, "Approximation algorithms for multiprocessor energy-efficient scheduling of periodic real-time tasks with uncertain task execution time," in Proceedings of the *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2008

[24] A. Kandhalu, J. Kim, K. Lakshmanan and R. Rajkumar, "Energy-Aware Partitioned Fixed-Priority Scheduling for Chip Multi-processors," in Proceedings of the *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2011

[25] N.S. Kim, T. Austin, D. Blaauw, T. Mudge, K. Flautner, J. S. Hu, M.J. Irwin, M. Kandemir and V. Narayanan, "Leakage current: Moore's law meets static power," in *Computer*, vol.36, no.12, pp.68-75, December 2003

[26] S. Irani, S. Shukla, and R. Gupta, "Algorithms for power savings," in Proceedings of *ACM-SIAM Symp. Discrete Algorithms*, 2003

[27] R. Jejurikar and R. K. Gupta, "Procrastination scheduling in fixed priority real-time systems", in Proceedings of *CM SIGPLAN/SIGBED Conf. Languages, Compilers, and Tools for Embedded Systems*, 2004

[28] J. J. Chen and T. W. Kuo, "Procrastination for leakage-aware rate-monotonic scheduling on a dynamic voltage scaling processor," in Proceedings of *ACM SIGPLAN/SIGBED LCTES*, 2006

[29] A. Rowe, K. Lakshmanan, H. Zhu and R. Rajkumar, "Rate-Harmonized Scheduling and Its Applicability to Energy Management," in *IEEE Transactions on Industrial Informatics*, vol. 6, no. 3, 2010

[30] S. D'souza, A. Bhat and R. Rajkumar, "Sleep Scheduling for Energy-Savings in Multi-Core Processors", in Proceedings of the Euromicro Conference on Real-Time Systems, 2016

[31] M. Santriaji and H. Hoffmann, "MERLOT: Architectural Support for Energy-Efficient Real-Time Processing in GPUs", in Proceedings of the *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2018

[32] M. Peres, "Reverse engineering power management on NVIDIA GPUs - A detailed overview", in Proceedings of *X.org Developer's Conference*, 2013

[33] J. Lehoczky and L. Sha and Y. Ding, "The rate monotonic scheduling algorithm: exact characterization and average case behavior", in Proceedings of the *IEEE Real-Time Systems Symposium*, 1989

[34] K. Lakshmanan, "Scheduling and Synchronization for Multi-core Real-time Systems", *Ph.D. Thesis*, Carnegie Mellon University, 2011

[35] NVIDIA Jetson TX2, *https://devblogs.nvidia.com/jetson-tx2-delivers-twice-intelligence-edge/*

[36] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," in *J. ACM*, vol. 20, no. 1, pp. 4661, 1973.

[37] P. Emberson, R. Stafford and R. Davis, "Techniques For The Synthesis Of Multiprocessor Tasksets", in *Intl. Workshop on Analysis Tools and Methodologies for Embedded Real-Time Systems*, 2010

[38] Jetson TX2 - INA226 (Power Monitor with i2C Interface), *https://devtalk.nvidia.com/default/topic/1000830/jetson-tx2/jetson-tx2-ina226-power-monitor-with-i2c-interface-/*