HEART: Hybrid Memory and Energy-Aware Real-Time Scheduling for Multi-Processor Systems

MARIO GÜNZEL, CHRISTIAN HAKERT, KUAN-HSUN CHEN, and JIAN-JIA CHEN,
TU Dortmund University, Germany

Dynamic power management (DPM) reduces the power consumption of a computing system when it idles, by switching the system into a low power state for hibernation. When all processors in the system share the same component, e.g., a shared memory, powering off this component during hibernation is only possible when all processors idle at the same time. For a real-time system, the schedulability property has to be guaranteed on every processor, especially if idle intervals are considered to be actively introduced.

In this work, we consider real-time systems with hybrid shared-memory architectures, which consist of shared volatile memory (VM) and non-volatile memory (NVM). Energy-efficient execution is achieved by applying DPM to turn off all memories during the hibernation mode. Towards this, we first explore the hybrid memory architectures and suggest a task model, which features configurable hibernation overheads. We propose a multi-processor procrastination algorithm (HEART), based on partitioned earliest-deadline-first (pEDF) scheduling. Our algorithm facilitates reducing the energy consumption by actively enlarging the hibernation time. It enforces all processors to idle simultaneously without violating the schedulability condition, such that the system can enter the hibernation state, where shared memories are turned off. Throughout extensive evaluation of HEART, we demonstrate (1) the increase in potential hibernation time, respectively the decrease in energy consumption, and (2) that our algorithm is not only more general but also has better performance than the state of the art with respect to energy efficiency in most cases.

CCS Concepts: • **Computer systems organization → Embedded software**; **Real-time systems**; • **Hardware → Memory and dense storage;**

Additional Key Words and Phrases: Multi-processor, dynamic power management, non-volatile memory, hybrid memory architecture, normally-off computing

ACM Reference format:
Mario Günzel, Christian Hakert, Kuan-Hsun Chen, and Jian-Jia Chen. 2021. HEART: Hybrid Memory and Energy-Aware Real-Time Scheduling for Multi-Processor Systems. *ACM Trans. Embedd. Comput. Syst.* 20, 5s, Article 88 (September 2021), 23 pages.
https://doi.org/10.1145/3477019

This article appears as part of the ESWEEK-TECS special issue and was presented in the International Conference on Embedded Software (EMSOFT).
This work has been supported by Deutsche Forschungsgemeinschaft (DFG), as part of Sus-Aware (Project No. 398602212) and as part of OneMemory (Project No. 405422836). This result is part of a project (PropRT) that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 865170).
Authors' address: M. Günzel, C. Hakert, K.-H. Chen, and J.-J. Chen, TU Dortmund University, Dortmund, Germany; emails: {mario.guenzel, christian.hakert, kuan-hsun.chen}@tu-dortmund.de, jian-jia.chen@cs.tu-dortmund.de.

1539-9087/2021/09-ART88 $15.00
https://doi.org/10.1145/3477019

1 INTRODUCTION

Emerging technologies for non-volatile main memories (NVMs) allow a computing system to turn off the main memory without the need of storing and restoring any data in the main memory. This feature allows the system to apply advanced dynamic power management (DPM) to switch to low power states (hibernation) almost immediately. For certain *normally-off* embedded systems which spend most of their lifetime idle, it is crucial to reduce the energy consumption and thus to extend, for instance, battery lifetimes in a resource constrained environment [43, 44]. Such embedded systems may serve real-time sensitive applications, where the execution of the system and the internal tasks have to meet deadlines. Due to dynamics of the environment, input-dependent tasks on such systems might often finish earlier than the pessimistic worst-case execution time (WCET), which is known as *early completion*. However, the schedulability of all tasks considering their WCET has to be ensured at all times.

When the main memory is shared by all processors in a multi-processor system, the memory cannot be turned off as long as at least one processor still executes. Thus, to enter the hibernation state with disabled memories, *all processors have to idle at the same time*, i.e., no task is executing on the processors. Consequently, common idle intervals across all processors are necessary or should be enforced without violating schedulability of a given task set.

Furthermore, the transition between the hibernation state and the active (execute or idle) state requires an additional energy and time overhead to turn on/off the memory and to copy relevant data to/from possibly existing volatile memory (backup and restore). For energy efficiency, the *break-even time* [48] defines the break-even point for deciding whether the system should be switched to the hibernation state or kept in the idle (but active) state, when all processors idle. That is, when the idle time is shorter than the break-even time, it is more energy-efficient to keep the system in the idle state; otherwise, it is more energy-efficient to turn the system to the hibernation state and schedule a *hibernation interval* instead of *idle interval*.

Towards energy-efficient scheduling with DPM, several techniques are proposed for single-processor systems [1–3, 11, 17, 24, 34, 35, 39]. In particular, Jejurikar et al. [35] provide an on-line procrastination algorithm[1] to enlarge consecutive idle intervals for single-processor systems while preserving schedulability guarantees under earliest-deadline-first (EDF) scheduling. To the best of our knowledge, the latest procrastination algorithm under partitioned earliest-deadline-first (pEDF) that maximizes common idle time among multiple processors is proposed by Fu et al. [21]. However, their approach is only applicable when the real-time jobs have concrete arrival times, absolute deadlines, and fixed execution times known apriori off-line. Their algorithm can be used to handle (strictly) periodic tasks in an off-line manner.

The pessimism in terms of early completion and the restricted applicability of the proposed approach in [21] motivate us to develop another pEDF-based scheduling algorithm for sporadic tasks on a multi-processor real-time system. This algorithm enforces all processors in the system to pause their current jobs once a certain number of idle processors is reached. We call this artificial introduction of idle time *forced procrastination*, as inherited from the literature. Afterwards, the maximal possible common idle interval across all processors is determined, while maintaining the schedulability of all task-sets on all processors. Please note that the proposed *procrastination algorithm* in this work can be applied on-line to sporadic task sets and benefits from early completion.

Giving schedulability guarantees when considering our proposed procrastination algorithm becomes much more complex since forced procrastination introduces *self-suspension* to the system. Whereas in the single-processor case jobs are only delayed, with multiple processors we even pause

[1]In this work, we only focus on DPM. While DPM and dynamic voltage scaling (DVS) are combined, the procrastination algorithm in [35] still works well when all tasks execute at maximum speed (see Theorem 2 in [35]).

job execution and resume it later on after the forced procrastination state. To ensure that all jobs meet their deadlines, we need to apply a schedulability test, which can handle this self-suspending behavior. However, the research topic of self-suspending task sets turns out to be challenging and can easily lead to unsound timing analyses, as recently reported by Chen et al. [13], and Günzel and Chen [22]. The multi-processor procrastination algorithm proposed in this work does not only analyze, but even controls the self-suspending behavior without any impact on the schedulability. Such control mechanisms have been under analysis for several years. As reported in [13] there are yet three types of such mechanisms proposed in the literature, i.e., *period enforcement* [46], *release guard* or *release enforcement* [28, 47], and *slack enforcement* [38]. However, concerns about the applicability of the period enforcer [46] are reported in [10] and the slack enforcement mechanism [38] has been recently disproved in [23]. Therefore, this work on the multi-processor procrastination algorithm may provide valuable insight for ongoing research in the field of self-suspension enforcement mechanisms.

In this work, we further explore the technical requirement of real-time task scheduling on hybrid (non-)volatile memory systems. Such systems can be utilized in ultra low power devices [14, 42], especially if they use unreliable energy harvesting to power themselves. To this end, we propose a design of real-time tasks with a configurable allocation of task's memory to the volatile and non-volatile memory, which leads to varying worst-case execution times (WCETs) due to different access speeds and different hibernation overheads. This yields an optimization potential where a trade-off between lower WCET and decreased hibernation overhead can be made. In order to derive a *hibernation optimized schedule*, we integrate the varying WCETs and overheads into our scheduling algorithm.

Our Contributions:

- A real-time task design for hybrid NVM systems, making the hibernation and wake-up overhead configurable.
- A pEDF-based procrastination scheduling algorithm, which enforces concurrent idle intervals among all processors with maximal length, while maintaining the schedulability in multi-processor systems.
- Experimental evaluation of the proposed scheduling algorithm.

Section 3 describes the extended task implementation, the hibernation procedure in the operating system and the power consumption characteristics. In Section 4, the problem definition studied in this paper is presented. In Section 5, we develop a procrastination algorithm based on our schedulability analyses. In Section 6, we discuss implementation aspects that require particular attention. We present our evaluation results in Section 7 by showing that our method significantly saves energy in different scenarios and that we outperform the state-of-the-art method in most cases. Section 8 concludes the paper.

2 RELATED WORK

Several techniques for energy-efficient scheduling with dynamic power management (DPM) and dynamic voltage scaling (DVS) have been proposed in the literature. Baptiste [2] proposes the first polynomial-time algorithm to compute an (off-line) optimal strategy to minimize the number of idle intervals. Baptiste et al. [3] further enhance the previous result to jobs with arbitrary execution times and reduce the time complexity. Augustine et al. [1] provide on-line scheduling algorithms considering systems with multiple low-power states, while the system is not allowed to enter low-power states when the ready queue is not empty. A *procrastinating principle* is initially proposed by Irani et al. [33], which postpones job execution as much as possible in order to bundle workload for batch execution. For periodic tasks, Chen and Kuo [37] and Jejurikar et al. [34] propose procrastination algorithms under fixed-priority scheduling. However, these two algorithms are not applicable to multi-processor systems. Additionally, several works [11, 35, 39, 49] consider DPM for dynamic priority scheduling. Devadas et al. [16] discuss the interaction of DVS and DPM for frame-based systems. Recently, Ha et al. [24] adopt DPM while considering the property of non-volatile memory.

Bingham and Greenstreet [5] consider job migration and show that the optimal schedule can be obtained by linear programming. The approximation algorithm by Demaine et al. [15] based on dynamic programming minimizes the number of idle intervals and the total active time over all processors at the same time. However, it does not address for the studied problem in this work, that all processors have to idle concurrently for turning off shared memories. Chen et al. [36] propose on-line EDF-based scheduling algorithms to procrastinate the arrival time of upcoming jobs, but this algorithm is limited to periodic tasks while assuming that the processors go into hibernate mode independently.

While considering only DPM on multi-processor systems as the focus of this work, Fu et al. [21] provide several off-line scheduling algorithms by checking over the busy intervals of each processor to maximize common idle time at which all processors are idle. However, their approaches are limited to a set of real-time jobs with fixed arrival times, absolute deadlines, and execution times. It is possible to apply their approaches to periodic real-time tasks by unrolling the job releases in a hyper-period, i.e., least common multiple of the periods, which requires high time/space complexity. Moreover, their algorithms do not benefit from early completion since the schedule is created off-line. Especially for systems with high *worst-case* utilization but significant early completion behavior, i.e., low *actual* utilization, such an off-line hibernation decision is very pessimistic since it neglects the potential hibernation time enabled by early completion. In 2018 Calinescu et al. [8] present strategies to decide whether tasks should be executed on a local or on shared memory to optimize energy utilization. However, to minimize the energy consumption if the assignment of tasks to memory is specified, results like [21] need to be utilized on top of that.

Our proposed control mechanism enforces self-suspending behavior, i.e., jobs pause their execution and resume it at a later time point. Control mechanisms to manage self-suspension intervals have been studied in the literature for several years [38, 46, 47]. They inject additional suspension into the schedule to reduce the impact of self-suspending behavior or even ignore it completely. Although this property is desirable, the correctness of [38] and [46] is an issue, as detailed below. The *period enforcer* is a control mechanisms proposed by Rajkumar [46]. The author presents a runtime rule and claims that it forces tasks to behave like ideal periodic tasks without any scheduling penalty. It was shown by Chen and Brandenburg [10] that this rule may cause deadline misses. The *slack enforcement* mechanism by Lakshmanan and Rajkumar [38] moves job segments to the latest time, such that they meet their worst-case response time, by calculating the available *slack*. They state that this makes tasks behave like ideal periodic tasks. However, the correctness of that claim has been challenged in [13] and recently disproven in [23]. *Release guard* [47] or *release enforcement* [28] mechanisms release the single computation segments of a job with a guaranteed minimum inter-arrival time. Such enforcement removes the impact of self-suspension from the system. In the survey paper [13] this mechanism is termed *static period enforcement*.

Independent from the work area of scheduling on DPM and DVFS systems, energy consumption and timing characteristics especially of NVM based systems are a widely studied field. Since NVMs can be used in various configurations and at different points in the memory hierarchy, determining latencies and power/energy consumption os not trivial. Dong et al. propose NVSim [20], which is a circuit level simulator for various NVM technologies. NVSim simulates the architecture of a memory device and provides according device characteristics. Inci et al. propose simulators to specifically analyze different NVM architectures and technologies in the use of deep learning applications [29, 30]. Despite simulations of the memory device itself, the executed software and the interplay of the full system with the memory device have a huge impact on the allover energy consumption and memory latency and execution time, respectively. To this end, Poremba et al. propose NVMain2.0, a simulation framework which takes the memory accesses of a full system as an input and simulates latencies within the simulation process [45]. NVMain2.0 further allows simulation of the energy and power consumption of the entire memory hardware. Hakert et al. build a precise analysis framework on top of NVMain2.0 [27], which considers side effects from an underlying operating system and allows separated analysis of single software components within NVMain2.0 [25].

3 SYSTEM MODEL

Since byte addressable NVMs have become popular in embedded systems, the proposed and available architectures usually provide hybrid memories [14, 19, 42]. These architectures are equipped with a volatile memory (VM) (DRAM or SRAM in most cases), which is populated from a ROM during the startup procedure. In addition to the VM, the NVM is usually connected with a separate memory controller and is available at a dedicated address range in the system's address space, for instance in the TI MSP430FR6xx family [31]. After configuring the controllers and starting up the system, a volatile memory is available in one certain address range, while the NVM is available in another dedicated address range. Thus, the program contents can be placed arbitrarily in either VM or NVM. Several systems may also include local memories (VM and NVM), which are not shared among all processors. However, the most common configuration in available systems is shared memory. Therefore, in this paper we focus on systems with one volatile and one non-volatile shared memory, which are used simultaneously by all processors.

3.1 Power States and Initial Exploration

The power supply of the memories is usually managed by system power states. We utilize three different power states: (1) **execution-i** where memories are turned on, $i \in \mathbb{N}$ processor are actively executing tasks and all other processors are halted, (2) **idling** where all memories are turned on but all processors are halted and (3) **hibernation** where all processors are halted and all memories are turned off. We denote by P_{ex}^{i}, P_{id} and P_{hi} the momentary power consumption during execution-i, idling and hibernate.

Our proposed method aims to reduce the total energy consumption of a certain task set by scheduling it in such a way, that the hibernation time is maximized. Many factors despite the hibernation time, as for instance the momentary power consumption in execution-i and hibernation state, determine the allover energy consumption of a concrete system and thus also the energy saving potential. We note that any target hardware needs to fulfill requirements in order to implement the above described power states. The system must be able to turn off single processors, at least by disabling the clock signal. In addition, the system must implement power gating for the memories such that the hibernation state can be realized. Independently from the specific implementation of the target platform, the three power states can be constructed and they feature a certain power consumption. Our proposed algorithm only relies on the existence of these power states and integrates the power consumption as parameters.

To provide an intuitive demonstration of these effects, we implement the above three power states in the system and emulate the execution of a randomly generated schedule. We do not implement the full scheduler, but rather simulate the schedule offline and deploy a sequence of execution phases to the hardware. Furthermore, we measure the power consumption of the system to obtain an exemplary set of parameters. As an exemplary system for this paper, we choose

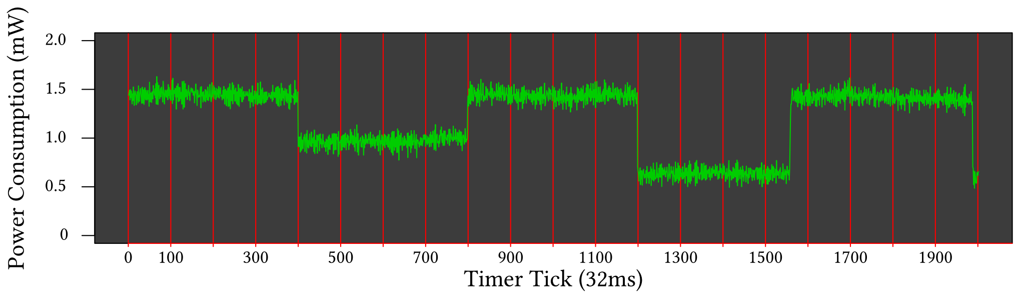


Fig. 1. Demonstrative schedule with power consumption.

the Texas Instruments MSP430FR6989 microcontroller [31]. This device features a single computing core, integrated on-chip SRAM and on-chip FRAM as the non-volatile memory. Although the objective of our study is on multi-processor systems, the power consumption characteristics and the realization of power levels are mostly independent from single-processor or multi-processor systems. Embedded multi-processor systems with on-chip NVM are currently rarely available. Nevertheless, such systems might be an upcoming trend, as multi-processor embedded systems with off-chip flash-based NVM are already available.[2]

For the realization of our proposed power levels, we directly utilize the *low power modes* (LPM), which are implemented by the MSP430FR6989 system. Since the system has only one processor, only the execution-1 state exists. When the scheduler decides to idle, low power mode 1 (LPM1) is entered, where only the CPU clock is deactivated, and all other peripherals (including all memories and the memory controllers) remain running. When the scheduler decides to hibernate, low power mode 3 (LPM3) is entered, which disables all clocks, except the auxiliary clock (required for the wake-up interrupt). Furthermore, SRAM and FRAM are powered off in LPM3. To acquire realistic power consumption measurements, a schedule of a task set is emulated on the system, together with accesses to SRAM and FRAM, and overheads for entering and leaving idle or hibernation states, depending on how much memory has to be backed up from SRAM to FRAM and vice versa. We connect the micro-controller to a high precision power analyzer, which samples at 50kHz. The samples are averaged for each timer tick (32ms) to reduce the noise and extract the mean power consumption. Please note that the timer tick length of 32ms is rather high for a real world system, however our simulation is scaled with the tick length. The choice of the timer tick length is motivated by achieving more accurate measurement results.

Figure 1 depicts the recorded momentary power consumption on an exemplary schedule. The power consumption of the three power levels can be clearly distinguished. Investigating the measured power consumption further, average power consumption values can be provided: The System consumes $P^1_{ex} = 1.43mW$ during execution-1, $P_{id} = 0.97mW$ during idle and $P_{hi} = 0.63mW$ during hibernation. We observe that the transition between the power states does not introduce any peak power consumption. In this example, the transition is performed within a single clock tick and thus can be approximated with an additional overhead of one tick. If the transition takes longer, the hibernation overhead has to be considered accordingly. Considering an overhead of 15 timer ticks for backup and restore when hibernating the system, in this example a *break-even* time of 35.29 timer ticks is given. This means that, whenever the system becomes idle for more than

[2]We do not investigate such off-chip NVM systems like [32], because the energy consumption and access speed are not comparable to the aforementioned FRAM based system.

36 timer ticks, hibernating consumes less energy than idling. Therefore it is crucial to precisely know the length of idle intervals in advance to minimize the energy consumption.

3.2 NVM-Aware Task Implementation

Independent from a specific system, we investigate hybrid memory architectures and provide classifications of different implementations. Because the access latency of the NVM is typically higher than of the VM, e.g., FeRAM is slower by a factor of 5-8 compared to DRAM [7], the task execution time is usually increased when more memory accesses target the NVM. Contrarily, when the system turns off the memories, if a large portion of the memory footprint is allocated in the NVM, the overhead for saving the task memory to the NVM is significantly reduced. We consider three performance classes as follows:

*100% Persistent (**1P**).* A task in this class allocates its entire memory (text, data and stack) to the NVM. Thus, during execution, every memory access from the task targets the NVM. When the system hibernates, no further content needs to be saved. After waking up the system, contents in NVM are untouched and 1P tasks are resumed.

*X% Persistent (**XP**).* A task in this class allocates some memory, e.g., its stack, to the VM and all the other memory contents to the NVM. This decision can be application dependent, e.g., rarely used memory could remain in the NVM. Compared to **1P**, this speeds up the execution, since many memory accesses during execution target the stack memory [26] in the faster VM. When the system turns off the memories, the VM content of every task in **XP** has to be saved to the NVM.

*0% Persistent (**0P**).* A task in this class allocates its entire memory to the VM. Compared to **1P** and **XP**, this leads to the fastest execution time. When the system turns off the memories, the entire memory footprint of the task has to be stored to the NVM, which causes an overhead that is dependent on the task implementation. In addition, the wake up procedure has to restore the full task memory footprint from NVM to VM.

Depending on the system properties and implementation details, the overhead for storing and restoring data to and from the NVM might not be symmetric. In the following we consider that storing and restoring is completely done during the hibernation and wake up procedures, such that the overhead can be considered to happen within these intervals. We quantify the total overhead for storing and restoring then as a single number to consider it in our algorithm.

We note that the operating system has to persist several data structures in addition to the volatile parts of the task memory. Moreover, the overhead for putting a task into hibernation and the overhead for resuming a task are not necessarily symmetric. However, throughout this work we focus on the *total* overhead, independent from its distribution.

The aforementioned performance classes allow a high degree of configurability, which can be exploited to optimize the schedule of a task set further regarding the increase of idle times and reduction of hibernation overheads. Performance classes with lower worst-case execution times enable the procrastination algorithm to make more optimistic decisions. Depending on the scenario, the related higher hibernation overhead and thus the higher break-even time might (or might not) undo this optimism in terms of hibernation time as in the following example:

Example 1. Consider one task τ_1 with period 10, i.e., a job is released every 10 time units. Assume that the system has a constant hibernation overhead of 1 and that for the two performance classes 1P and 0P each job executes for 5 and 4 time units. Whereas for 1P the hibernation overhead is 0, the hibernation overhead and hence also break-even time for 0P is higher: If τ_1 has the performance

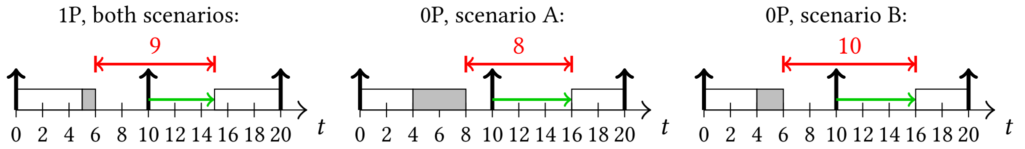

1P, both scenarios:
9
0 2 4 6 8 10 12 14 16 18 20 t
0P, scenario A:
8
0 2 4 6 8 10 12 14 16 18 20 t
0P, scenario B:
10
0 2 4 6 8 10 12 14 16 18 20 t

Fig. 2. Hibernation optimized schedule from Example 1. Green arrows show job delay and gray areas indicate overhead. Depending on persistence class and scenario we obtain a hibernation interval of length 9, 8 or 10.

class 1P, we obtain a total overhead of 1. If τ_1 is in the class 0P, we assume two scenarios A and B for the overhead. In scenario A the total overhead is 4 and in scenario B it is 2. Figure 2 illustrates the resulting schedules where each second job is delayed as much as possible. In scenario A, an allocation of τ_1 to 1P guarantees the longest hibernate-interval (1P: 9 vs 0P: 8 time units), while for scenario B, an allocation of τ_1 to 0P yields a better result (1P: 9 vs 0P: 10 time units).

3.3 Real-Time Task and Platform Model

Next to the implementation details, we also extend the classical sporadic real-time task model, where the timing effects from the implemented performance classes are encountered. We denote the set of n_j independent sporadic implicit-deadline real-time tasks on processor P_j by $\mathbb{T}_j = \{\tau_{1,j}, \dots, \tau_{n_j,j}\}$. Each task $\tau_{i,j} = (C^{1P}_{i,j}, C^{XP}_{i,j}, C^{0P}_{i,j}, T_{i,j})$ releases jobs sporadically with minimum inter-arrival time $T_{i,j} > 0$. We assume that the tasks have implicit deadline, i.e., the relative deadline of a task $\tau_{i,j}$ is $T_{i,j}$. In particular, each job (task instance) of task $\tau_{i,j}$ has to be completed $T_{i,j}$ time units after its release. Moreover, each job of $\tau_{i,j}$ has a worst-case execution time (WCET) $C^{PC_{i,j}}_{i,j} \geq 0$ and utilization $U_{i,j} = \frac{C^{PC_{i,j}}_{i,j}}{T_{i,j}}$ depending on the assignment of $\tau_{i,j}$ to a specific performance class $PC_{i,j} \in \{1P, XP, 0P\}$. The different execution times are caused by different access speeds of the memory systems, i.e., $C^{1P}_{i,j} \geq C^{XP}_{i,j} \geq C^{0P}_{i,j}$. We assume that the hibernation of each task is associated with an overhead $O^{PC_{i,j}}_{i,j} \geq 0$ depending on the underlying performance class $PC_{i,j}$. For 1P tasks, we assume that there is no task specific overhead at all since no preparation for the hibernation state is needed, i.e., $O^{1P}_{i,j} = 0$. The hibernation overhead for XP and 0P tasks is higher, as detailed in the Section 3.2. We assume that $0 \leq O^{XP}_{i,j} \leq O^{0P}_{i,j}$. In addition to the task specific hibernation overheads, a constant overhead O_{const} is assumed for persisting management data structures and bringing the hardware into the desired hibernation state.

Throughout this paper, we consider a platform with M identical processors with synchronized clocks, i.e., execution and timing properties coincide on each processor. Tasks are statically assigned to a processor, i.e., on each processor P_j, $j = 1, \dots, M$ we have one task set $\mathbb{T}_j$ with $\bigcup_{j=1}^{M} \mathbb{T}_j =: \mathbb{T}$ and $\mathbb{T}_j \cap \mathbb{T}_{j'} = \emptyset$ for any $j \neq j'$. The utilization of each task set is defined by $U_j = \sum_{i=1}^{n_j} U_{i,j}$. We consider partitioned multi-processor scheduling where on each processor *preemptive earliest-deadline-first* (EDF) [40] scheduling is utilized, i.e., the job in the ready queue with the earliest absolute deadline is always executed first. If a new job with a shorter absolute deadline arrives, the current job execution is preempted.

The system can be in one of the following three states: *execution-i*, *idling*, and *hibernation*, where we define the corresponding power consumption as P^i_{ex}, P_{id} and P_{hi}. We assume that whenever a processor becomes idle, it is immediately halted, i.e., the clock from the specific processor is disabled. This implies, that every actively executing processor causes an additional power consumption, which adds to the idle power consumption P_{id}, i.e., when all processors are halted. We denote this additional, per processor power consumption by P_{ac}. This assumption leads to $P^i_{ex} = P_{id} + i \cdot P_{ac}$, i.e., the power consumption with i active processors is the sum of the power consumption if all processors idle plus i times the additional power consumption of an active processor. Under the assumption of the hybrid shared memory architecture, the hibernation state is only reachable if all of the M processors are halted, such that the NVM and VM can be both turned off.

4 PROBLEM DEFINITION

In this paper, we focus on the design of hibernation scheduling, provided that the performance classes and task partitions based on partitioned EDF (pEDF) are determined beforehand, i.e., $PC_{i,j}$ is given for each $\tau_{i,j} \in \mathbb{T}$, and the task set is already partitioned onto the multi-processor systems. Although the choice of performance classes for concrete tasks may influence their worst-case execution time and the hibernation overhead, our algorithm has to deal with given instances of these parameters. We therefore do not focus on choosing the performance classes to achieve optimized execution times and overhead, but rather on handling a given configuration well. However, many straight forward optimization strategies seem reasonable for choosing the mapping of tasks to performance classes, when, for instance, a cost function on the worst-case execution time and overhead is constructed and fed to a generic optimization framework. Existing task synthesis algorithms, e.g., by Chen in [9], and task partitioning algorithms, e.g., by standard bin packing or the algorithm by Baruah and Fisher [4] can be applied. We assume that the resulting EDF schedule on each of the processors is feasible. The **problem** studied in this paper is defined as follows:

- **Input:** A given set $\mathbb{T}$ of tasks partitioned onto M processors, with subsets $\mathbb{T}_1, \mathbb{T}_2, \ldots, \mathbb{T}_M$ and an assignment of tasks to performance classes.
- **Output:** An on-line scheduling strategy to optimize energy consumption under partitioned EDF scheduling on a homogeneous multi-processor system, in which hibernation is only possible when all M processors are in the hibernation state, while maintaining the feasibility of any task set which is feasible under partitioned EDF.

Let $O_{total} = \sum_{\tau_{i,j} \in \mathbb{T}} O_{i,j}^{PC_{i,j}} + O_{const}$ denote the total hibernation overhead, which is composed of a constant system-specific overhead O_{const} and a task-specific overhead $O_{i,j}^{PC_{i,j}}$. We assume that during the backup and restore operations, the system has the power consumption $P_{ex}^1 = P_{id} + 1 \cdot P_{ac}$ as the memory has to be turned on and one processor is active.[3] Moreover, we assume that the transition between different power states can be approximated pessimistically with a momentary power consumption of P_{ex}^1.

Assume we have an idle interval of length t and need to decide if it is efficient to hibernate. If we decide to let the processors hibernate, the total energy consumption during the overhead is $O_{total} \cdot (P_{ac} + P_{id})$. Afterwards the system is in the hibernation mode for $t - O_{total}$ time units, where it consumes $(t - O_{total}) \cdot P_{hi}$ amount of energy. If we instead decide to let the processors idle, the total energy consumption is $t \cdot P_{id}$. Therefore, the break-even time $\mathcal{B}$ to decide whether it is more energy-efficient to put all processors to hibernation is to solve the equation $\mathcal{B} \cdot P_{id} = O_{total} \cdot (P_{ac} + P_{id}) + (\mathcal{B} - O_{total}) \cdot P_{hi}$, where the left-hand side is the energy consumption for the processors to idle for $\mathcal{B}$ time units and the right-hand side is the energy consumption due to the hibernation overhead and the energy consumption of the processors when they are in the hibernation state. Therefore, the break-even time can be computed by

$$\mathcal{B} = O_{total} \cdot \frac{P_{ac} + P_{id} - P_{hi}}{P_{id} - P_{hi}}. \tag{1}$$

[3]We assume that the entire memory bandwidth can be served by a single processor when only copying from one to another memory. Therefore, the overhead (which only includes backup and restore operations) only needs one active processor.

Our objective in this paper is to develop a procrastination algorithm in Section 5, which aims to preempt current and delay further job executions to maximize the length of common idle intervals across all processors which exceed the break-even time $\mathcal{B}$, while minimize the number of these idle intervals for hibernation state. With the proposed algorithm a current idle interval is enlarged and allows longer hibernation, while future idle intervals are shortened.

We state our analysis to be applicable to any assignment of tasks to persistence classes since throughout this whole section we assume that the assignment of tasks to persistence classes is arbitrary but fixed. However, we note that the determination of a good assignment of tasks to persistence classes, which further minimizes the power consumption, remains a complex optimization problem and is considered out of the scope.

5 FORCED PROCRASTINATION ANALYSIS

In this section, we first discuss the design of a scheduling algorithm in which individual processors are allowed to procrastinate independently (see Section 5.1). Such a special case helps to understand how to force all processors to procrastinate at the same time and create sufficiently long idle time for hibernation of the system (see Section 5.2).

5.1 Independent Procrastination

Although the considered system can only hibernate when all processors idle, here we present a scenario where each processor is allowed to procrastinate independently, e.g., if they do not have shared memory. Please note that the method presented in this subsection does not create common idle intervals among the processors, and hence cannot be used for hibernation in our system model.

Consider one processor P_j with task set $\mathbb{T}_j$ with n_j tasks. Under the assumption that the task set is an implicit-deadline task set and is schedulable by EDF on each processor, we can apply the utilization bound of EDF developed by Liu and Layland [40]. That is, $\sum_{i=1}^{n_j} \frac{C_{i,j}^{PC,j}}{T_{i,j}} \leq 1$. Jejurikar et al. [35] modify this schedulability test to compute a *procrastination interval* $Z_{i,j}$ for each task $\tau_{i,j}$. If an idle instance occurs during runtime, the *Procrastination Algorithm* [35] delays the next upcoming job releases $r_{i,j},\ i = 1, \ldots, n_j$ until some procrastination interval expires, i.e., until $\min_i r_{i,j} + Z_{i,j}$. By this means, still all deadlines can be met and on each processor long idle intervals are obtained independently. Originally in [35], the maximal procrastination intervals are computed off-line using the following theorem:

Theorem 2 (Procrastination Based on [35]). *Consider a task set* $\mathbb{T}_j = \{\tau_{1,j}, \ldots, \tau_{n_j,j}\}$ *with* $T_{1,j} \leq T_{2,j} \leq \ldots \leq T_{n_j,j}$. *If for each* $\tau_{i,j} \in \mathbb{T}_j$ *we have a* procrastination interval $Z_{i,j} \geq 0$ *with*

$$\frac{Z_{i,j}}{T_{i,j}} + \sum_{k=1}^{i} \frac{C_{k,j}^{PC_{k,j}}}{T_{k,j}} \leq 1, \text{ for all } i \in \{1, \ldots, n_j\} \tag{2}$$

$$\text{and } Z_{k,j} \leq Z_{l,j}, \text{ for all } k \leq l, \tag{3}$$

then the Procrastination Algorithm from [35] on processor P_j *guarantees that all jobs of all tasks* $\tau_{i,j} \in \mathbb{T}_j$ *still meet their deadline.*

5.2 Synchronized Procrastination

To enable hibernation and turn off the shared memory, all processors have to idle at the same time. Theorem 2 is based on the classical analysis of real-time scheduling theory by introducing $Z_{i,j}$ as the blocking time of task $\tau_{i,j}$. This derivation of $Z_{i,j}$ is valid since the procrastination decision[4] is

[4]That is, applying Theorem 2 on each of the M processors.

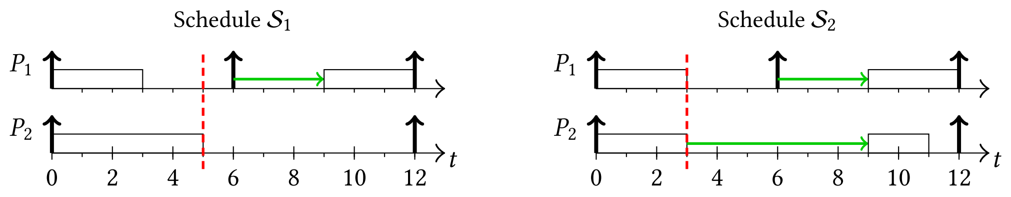


Fig. 3. Schedules of one task on each processor P_j obtained by procrastination forced at time $t = 5$ (left) and $t = 3$ (right). Interrupting the job execution on P_2 enlarges the interval of forced procrastination.

done when the processor idles. In particular, there is no unfinished job before the procrastination interval starts. However, when a job is released but unfinished before the procrastination of all processors starts, the blocking-based analysis in the literature becomes invalid. Hence, if we trivially utilize the job delay analysis from [35] as in Theorem 2 then we would have to wait until all processors idle at the same time to delay the upcoming job releases. This does not always yield the best results as shown in the following example, which is depicted in Figure 3.

Example 3. Assume we have a system of two processors P_1 and P_2 which execute one periodic task each, $\tau_{1,1}$ and $\tau_{1,2}$ with period $T_{1,1} = 6, T_{1,2} = 12$ and phase 0, i.e., the first job release is at time 0. We further assume that the performance class $PC_{1,i}$ of $\tau_{1,i}, i = 1, 2$ is fixed and that each job of task $\tau_{1,i}$ executes $C_{1,1}^{PC_{1,1}} = 3$ and $C_{1,2}^{PC_{1,2}} = 5$ time units. For simplicity, we do not consider hibernation overhead in this example. If we wait until all processors idle at the same time naturally ($t = 5$), then we can delay the second job of $\tau_{1,1}$ to 9 while still preserving the feasibility of the schedule, i.e., all jobs meet their deadline. The algorithm procrastinates for $9 - 5 = 4$ time units, as depicted in the schedule $\mathcal{S}_1$ in Figure 3. Alternatively, forcing to procrastinate as soon as one processor in the system idles ($t = 3$), and continuing the execution on both processors at time 9 yields an idle time of $9 - 3 = 6$ as in $\mathcal{S}_2$.

Whereas for independent procrastination, delaying upcoming job releases can be modeled as blocking behavior and tested as those, with synchronized procrastination, we might observe self-suspending behavior as in schedule $\mathcal{S}_2$ of Figure 3. That is, when a processor is forced to procrastinate while there are still unfinished jobs in the ready queue, these jobs give up their unfinished execution and suspend themselves. When tasks suspend, the schedulability condition is different from classical blocking-time analysis.[5] For suspension-aware analysis, it has been demonstrated that modeling suspension as blocking is possible for single-processor fixed-priority scheduling by Liu [41, pages 164-165] (proved by Chen et al. [12]) but incorrect for single-processor EDF scheduling (conjectured by Devi in [18] in 2003 and disproved recently by Günzel and Chen [22]).

Therefore, if the forced procrastination introduces arbitrary suspensions, suspension-aware analysis must be provided. A priori, it is not clear whether Equation (2) remains a valid schedulability analysis for such cases. To make self-suspending behavior still controllable, we ensure that the ready queue of each processor had been empty since the last procrastination.

Algorithm 1 presents our multi-processor procrastination algorithm. Beforehand, the maximal values for $Z_{i,j}$ have to be computed according to properties in Equations (2) and (3). Whenever the ready queue of some processor becomes empty at time t, the algorithm has to make a ***forced procrastination decision*** in line 6, i.e., it checks whether

[5]For detailed survey of scheduling and schedulability analyses for self-suspending task systems, we refer to the review by Chen et al. [13].

- a certain amount $F \in \{1, \dots, M\}$, also called the *forced procrastination threshold*, of processors have empty ready queues at time t and
- the other processors had empty ready queues since the last forced procrastination state.

In that case the algorithm initializes the timer with the minimal $Z_{i,j}$ among the tasks under execution and starts the *forced procrastination state* on each processor, described by Algorithm 2. During procrastination the job execution is suspended, and the processors wait for the wake-up signal from Algorithm 2. The timer reduces each clock cycle and is further reduced each time a new job arrives at one of the processors. When the timer expires, all processors are waken up and continue/begin to schedule the highest-priority job in the ready queue.

If the forced procrastination threshold F is high, then many processors need to have empty ready queues at the same time to enable the forced procrastination state. This might happen very rarely, or even not take place at all. However, if all processors have empty ready queues at the same time, there is high potential for long forced procrastination intervals, as the idle time until new job releases can be fully exploited. On the other hand, if the forced procrastination threshold F is low, only few processors need to have empty ready queues at the same time to enable the forced procrastination state. In such a case more intervals of forced procrastination are enabled. However, there is low potential for long idle intervals since all suspended jobs have to be resumed before their deadlines. Additionally, a short interval of forced procrastination might prevent a subsequent long interval of forced procrastination, since all processors need to have empty ready queues between forced procrastination. In general, the trade-off between different forced procrastination thresholds is dependent on the number and the distribution of time-instances, at which many processors have empty ready queues at the same time. This is again dependent on several factors such as utilization and release pattern of the tasks or their early completion behavior.

The following theorem provides the guarantee that the multi-processor procrastination Algorithm 1 preserves feasibility of the system.

Theorem 4 (Multi-Processor Procrastination). *Suppose the task sets $\mathbb{T}_1, \dots, \mathbb{T}_M$ are scheduled by partitioned EDF on M processors. Further, for each task set $\mathbb{T}_j = \{\tau_{1,j}, \dots, \tau_{n_j,j}\}$ let $Z_{1,j}, \dots, Z_{n_j,j} \geq 0$ be procrastination intervals with properties in Equations (2) and (3) as in the independent procrastination case. In the schedule obtained by modifying pEDF with Algorithm 1, all tasks still meet their deadline.*

The proof of this Theorem is two-folded. First, we prove that by the *forced procrastination decision* during each busy-interval (where the ready queue is not empty) the schedule cannot be affected by hibernation-induced self-suspension more than once. Second, we show that in such a case Theorem 2 can still be applied to ensure schedulability.

Proof. We check feasibility of the modified EDF schedule on each processor individually. Assume that there is a deadline miss at time t_1 on processor P_j. We define t_0 to be the last time instant before t_1 where the ready queue on P_j is empty or contains only jobs with deadline after t_1. Then we derive the following two properties for the interval $[t_0, t_1]$:

(a) During $[t_0, t_1]$ the algorithm forces procrastination at most once.
(b) During $[t_0, t_1]$ the processor executes at all times jobs with release $\geq t_0$ and deadline $\leq t_1$.

If (a) would not hold then there are two forced procrastinations in $[t_0, t_1]$. Due to the forced procrastination decision (line 6 of Alg. 1) there has to be a time between both procrastinations where the ready queue of P_j is empty. This contradicts the maximality of t_0.

If the processor executes a job with deadline $> t_1$ during $[t_0, t_1]$, i.e., b) does not hold, then at that time there are only jobs with deadline after t_1 in the ready queue, which contradicts the

ALGORITHM 1: Multi-Processor Procrastination Algorithm.

```
1:  **Preparation**:
2:  for $j = 1, \dots, N$ do
3:      Compute max. $Z_{i,j}, i = 1, \dots, n_j$ with Eq. (2) and (3)
4:  end for

5:  **On Emptying of Ready Queue of some Processors**:
6:  if Forced procrastination decision then
7:      if All proc. idle then
8:          Do not initialize $timer$
9:      else
10:         $timer \leftarrow \min\{Z_{i,j} \mid \tau_{i,j}$ is exec.$\}$ {Initialize timer}
11:     end if
12:     START Forced Procrastination State (Algorithm 2)
13: end if
```

maximality of t_0. On the other hand, if a job with release $t_0' < t_0$ and deadline at most t_1 would be executed during $[t_0, t_1]$, then the processor would be busy with executing jobs with deadline at most t_1 during $[t_0', t_1]$ which contradicts the definition of t_0.

In the following, we basically argue that Theorem 2 can still be applied since during $[t_0, t_1]$ there is at most one forced procrastination. More specifically, we follow a similar proof-strategy as in [35] to show that this assumption is sufficient to reproduce the schedulability decision. Let $\tau_{i,j}$ be the task with the highest index i, i.e., the one with the shortest minimum inter-arrival time, whose jobs are executed during $[t_0, t_1]$. To estimate the time where the system is forced to procrastinate during $[t_0, t_1]$ we first note that due to the algorithm, as long as the processor is forced to procrastinate while the ready queue is not empty, a timer is active and minimized each clock cycle. Hence, the maximal time in forced procrastination state during $[t_0, t_1]$ can be estimated by the maximal timer value, which is $Z_{i,j}$ due to the property in Equation (3). Furthermore, the number of jobs that are being executed by task $\tau_{k,j}$ with $k \leq i$ during $[t_0, t_1]$ is estimated by $\lfloor \frac{t_1 - t_0}{T_{k,j}} \rfloor$ due to b). Since there is a deadline miss on processor P_j at time t_1, there is more workload and forced procrastination during $[t_0, t_1]$ than could be handled by the processor, i.e., $Z_{i,j} + \sum_{k=1}^{i} \lfloor \frac{t_1 - t_0}{T_{k,j}} \rfloor C_{k,j} > t_1 - t_0$. Since $t_1 - t_0 \geq T_{i,j}$, we obtain $\frac{Z_{i,j}}{T_{i,j}} + \sum_{k=1}^{i} \frac{C_{k,j}}{T_{k,j}} > 1$ which contradicts the property in Equation (2). □

Since we exploit procrastination for hibernation, we additionally check along with the procrastination decision (line 6) whether it is guaranteed that switching to hibernation mode pays off, i.e., the idle time exceeds the break-even time. The time, where the system actually benefits from the hibernation is called *power saving time* and has to be maximized. For a schedule $\mathcal{S}$ the power saving time is defined by

$$pst(\mathcal{S}) = t_{pr} - n_{pr} \cdot \mathcal{B}, \tag{4}$$

where n_{pr} and t_{pr} are the total number of forced procrastinations and the sum of the lengths of all forced procrastinations in schedule $\mathcal{S}$.

As the break-even time $\mathcal{B}$ is known for the system, we can use an estimation of the next upcoming job releases to guarantee that the break-even time is exceeded, as follows. Let t be a time where F many processors begin to have empty ready queue and the other processors had empty ready queue since the last forced procrastination. For simplicity of notation let $P_1, \dots P_F$ be those processors with empty ready queue at time t. Since the processors wake up as soon as the timer expires and the timer counts down every clock cycle, we estimate the expiration of the timer by collecting events where the timer is initialized or minimized through releases. Let $r_{1,j}, \dots, r_{n_j,j}$ be

ALGORITHM 2: Forced Procrastination State.

```
1:  **On arrival of a new job of** $\tau_{i,j}$:
2:  **if** (Timer is not active) **then**
3:      $timer \leftarrow Z_{i,j}$ {Initialize timer}
4:  **else**
5:      $timer \leftarrow \min(timer, Z_{i,j})$
6:  **end if**

7:  **On expiration of Timer** ($timer = 0$):
8:  Wake-up all processors
9:  Scheduler schedules highest-priority job on each processor
10: Deactivate timer
11: End Procrastination State

12: **Timer Operation**:
13: $timer--$ {Counts down every clock cycle}
```

the (minimal) upcoming job releases not before t for each task on all processors $P_j,\ j = 1, \ldots, N$ and let further $\tau_{a_j,j},\ j = F+1, \ldots, M$ be the task at which P_j is working at time t. Then the timer for forced procrastination at time t does not expire before

$$E(t) := \min\left(\{t + Z_{a_j,j} \mid j = F+1, \ldots, M\} \ \cup \bigcup_{j \in \{1,\ldots,N\}} \{r_{i,j} + Z_{i,j} \mid i \in \{1, \ldots, n_j\}\}\right). \tag{5}$$

We conclude that it is guaranteed that the forced procrastination starting at time t exceeds the break-even time $\mathcal{B}$, if $\mathcal{B}$ is less than $E(t)$.

We note that the concept of Algorithm 2 to control self-suspension can be extended to global and semi-partitioned scheduling. In this case the calculation of $Z_{i,j}$ has to be based on a global or semi-partitioned schedulability test, i.e., Theorem 4 has to be revised.

6 IMPLEMENTATION CONSIDERATIONS

In order to realize the aforementioned algorithm in practice, tracking of idling processors and those who idled after the last procrastination requires careful considerations. Furthermore, the system timers, which control the wake up of the system from hibernation are not trivial to realize as well:

To decide for procrastination, the number of idling processors needs to be known. In this regard we use a counter, which is called *idle processors counter*. Furthermore the number of processors needs to be known, which own empty ready queues (ERQ) after the last hibernation. For that we use another counter, called *ERQ counter*. The implementation of the *forced procrastination decision* can be done by using shared counters to achieve a constant overhead for the implementation of this decision. Initially, each processor can be assumed to have an empty ready queue. Therefore, the *ERQ counter* is set to the number of processors during startup of the system. The *idle processors counter* is set to the total number of processors as well and is decreased when one processor becomes busy. Whenever one processor has no job in its ready queue, it increases the *idle processors counter*. At the moment a job arrives and the processor is not idle any longer, the counter is decreased. By using atomic operations, the counter is maintained and used to check the *forced procrastination threshold*. After each procrastination, the *ERQ counter* is reset to 0. Whenever one processor becomes idle for the first time after a procrastination, the counter is increased. Each processor therefore needs to store a flag which indicates that the processor already idled after the last procrastination, because if it idles again, the *ERQ counter* is not modified again. The system only forces procrastination, if the *ERQ counter* is equal to the total number of processors in the system, because only in that case all processors idled at least once since the last procrastination.

The HEART algorithm assumes programmable timers existing in the considered system, which are capable to wake up processors on expiration and also to wake up the system from the hibernation state. Such timers are provided by the hardware in most processors. However, during system hibernation or idling of all processors, the hardware only supports monotonic decrease of the timers. In order to handle arriving job releases during hibernation or idle properly according to Algorithm 2, the timer has to be set to a new value. Jejurikar et al. propose an additional circuit to handle this timer update [35], which is also applicable to our proposed algorithm. If such special hardware is not integrated, the system has to be waken up on job releases and at least once processor has to enter the execution state in order to set the timer value accordingly. This may increase the total power consumption during the hibernation interval then.

7 EVALUATION AND DISCUSSION

In the following, we evaluate the multi-processor procrastination algorithm (HEART) (Algorithm 1) in comparison to the state of the art [21] (OSPAL). We compare HEART with OSPAL for a wide range of scenarios using synthesized tasks. We consider different parameters for utilization, forced procrastination threshold and early completion, and demonstrate how these affect the quality of derived results. Since complexity for the OSPAL algorithm for recurrently released jobs can only be controlled for periodic tasks with low hyper-period, we stick to those task sets for comparison with OSPAL. Please note that a more generous setup does not affect the complexity or quality of our result significantly. In some cases, a sporadic setup does even increase the amount of time in the hibernation state achieved by HEART.

7.1 Task Sets Synthesis

For synthetic generation of M periodic task sets each with a utilization U [%] within a range of $\pm p$ [%], we rely on the UUniFast algorithm [6]. Please note that the period of the executed tasks has a significant influence on the potential for procrastination intervals. For instance, if there is a task with period below $\frac{1}{2}$ of the break-even time in the system, HEART and OSPAL can still schedule the system, but never have a chance to enforce a hibernation state. Therefore, we respect a minimal period, which is significantly greater than the break-even time, in our task set synthesis.

In this work, we use three different specifications for generating task sets:

(1) Semi-harmonic task sets with periods in $[10, 1000]$ [ms]
(2) Task sets with periods in $[10, 1000]$ [ms]
(3) Semi-harmonic task sets with periods in $[10, 100]$ [ms]

For the semi-harmonic task sets in (1) or (3) we draw the tasks periods log-uniformly from the interval $[10, 200]$ or $[10, 2000]$ and round them down to the next value in the semi-harmonic set $\{10, 20, 50, 100\}$ [ms] or $\{10, 20, 50, 100, 200, 500, 1000\}$ [ms]. The periods of task sets in (2) are directly drawn log-uniformly from the interval $[10, 1000]$ without rounding. We then derive the worst-case execution time (WCET) by multiplying the period of each task with the task utilization from UUniFast. Moreover, we draw task specific hibernation overheads from the interval $[0, 0.08]$ [ms] by a normal distribution with mean 0.04 and scale 0.02. This stems from the assumption that application dependent amount of memory has to be backed up during the overhead. We consider a few kilobytes for the memory footprint of our tasks. Please note that only semi-harmonic task sets, as in (1) and (3), ensure a low hyper-period for the complexity of the OSPAL algorithm.

Table 1. Average Power Saving time [%] for The Multi-Processor Case

	U = 5%	U = 40%	U = 80%
EC=0.05, H1	49.24	40.14	29.10
EC=0.05, H3	52.14	42.38	30.59
EC=0.05, H5	72.63	58.02	43.37
EC=0.5, H1	37.55	26.94	14.25
EC=0.5, H3	39.32	28.10	14.72
EC=0.5, H5	70.47	44.11	20.95
EC=1.0, H1	36.71	23.34	7.10
EC=1.0, H3	37.30	23.67	7.14
EC=1.0, H5	68.87	34.59	9.11
OSPAL	70.28	42.93	15.87

Cases where HEART outperforms OSPAL in the multi-processor case are marked green.

Performance classes. After task creation, each task is statically assigned to one performance class 1P, XP or 0P. Each performance class is described by two values, one to determine the actual WCET of the task and one to determine the actual hibernation overhead. For evaluation, we use the specification $1P = (1.0, 0.0), XP = (0.9, 0.5)$ and $0P = (0.75, 1.0)$, where the first tuple entry is a scaling coefficient for the WCET and the second entry is a scaling coefficient for the overhead, e.g., a task $\tau_{i,j}$ with period $T_{i,j} = 100$, WCET $C_{i,j} = 10$ and overhead 0.04 which is assigned to XP has then WCET $C_{i,j}^{XP} = 9$ and a hibernation overhead of $O_{i,j}^{XP} = 0.02$. The choice of these values implies that for XP, approximately 50% of the memory can remain in the NVM. Further, a speedup of the execution time by a factor of 1.33 is assumed when only using the VM instead of the NVM.

According to our description in Section 3.3, for a system with M processors we calculate the total overhead by $O_{total} = \sum_{i,j} O_{i,j}^{PC_{i,j}} + O_{const}$, i.e., summing up all task-specific overheads $O_{i,j}^{PC_{i,j}}$ plus a constant overhead O_{const} of 0.1 [ms]. To generate the schedules, we make the (simplifying) assumption that during hibernation the energy consumption is 0 and the energy consumption when all processors are active is twice as much as when all processors idle, i.e., $P_{ex}^{M} = 2 \cdot P_{id}$. Therefore, we set $P_{ac} = \frac{P_{id}}{M}$. We calculate the break-even time by $\mathcal{B} = O_{total} \cdot \frac{M+1}{M}$ according to Equation (1).

Early completion. Under specification of a global early completion bound (EC-bound), the time a job of $\tau_{i,j}$ has to be processed until it is finished is computed by a multiplication $\gamma \cdot C_{i,j}^{PC_{i,j}}$ where γ is drawn log-uniformly from the interval [EC-bound, 1] for each job individually. Early completion is introduced on runtime and can not be considered in off-line hibernation decision.

7.2 Evaluation Results

The results of HEART are dependent on many parameters as the utilization, amount of early completion and forced procrastination threshold. However, we compare with OSPAL only when considering periodic task sets with restricted hyperperiod. In this section we address these two issues: First, we compare HEART with OSPAL for a wide range of scenarios. Second, we show that HEART still provides good results in scenarios where OSPAL can not be applied.

Given an utilization of $U = 5\%, 40\%, 80\%$ with a precision of $p = 0.01\%$, we synthesize task sets according to (1) and (2) in Section 7.1. Period and WCET are processed with a minimal tick of 10^{-6} ms. For the evaluation, we schedule for $5000 \cdot 10^6$ ticks which is a multiple of the hyper-period

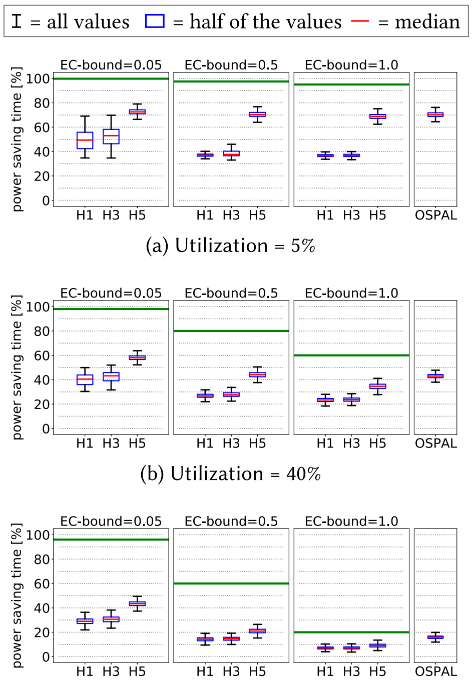


Fig. 4. Results for numerical evaluation for periodic tasks with predefined semi-harmonic periods.

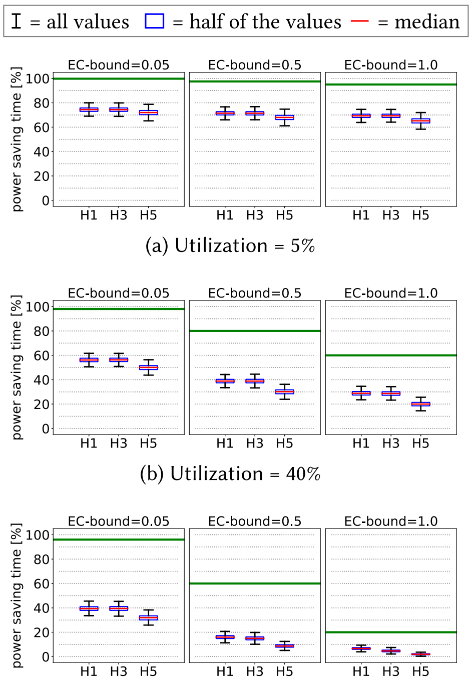
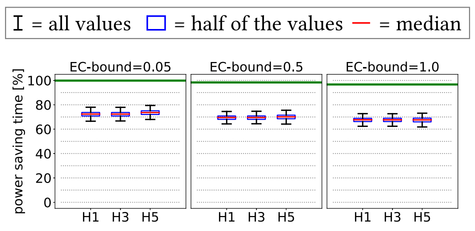


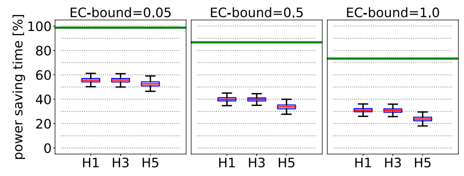
Fig. 5. Results for numerical evaluation for periodic tasks with standard log-uniform periods.

($1000 \cdot 10^6$ ticks) of 1). We draw the performance classes 1P, XP or 0P with equal probability and set the constant overhead to $O_{const} = 0.1$ ms.

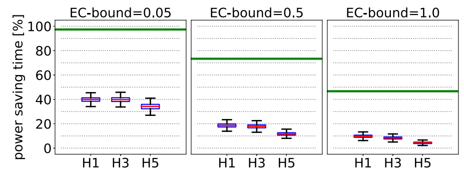
Evaluation for Periodic Tasks: For comparison of OSPAL and HEART, only the semi-harmonic, periodic task sets obtained by 1) are utilized. We run HEART with a fixed *forced procrastination threshold* of $F = 1, 3$ and 5 (resulting in scheduling policies H1, H3, H5), on a system with $M = 5$ processors, each with 20 tasks after partitioning. Figure 4 shows the *power saving time* from Equation (4), in % of the whole schedule length for 1000 runs. Each boxplot marks the median with a red line and $\frac{1}{2}$ of the results with a blue box. The power saving time cannot be higher than the baseline ($1 - U \cdot$ EC-bound), where $U \cdot$ EC-bound is a lower bound on the time that some processor executes a job. The baseline is marked by a green horizontal line and provides intuition for the optimal solution. Please note that the break-even time is not considered in the baseline. In addition to the boxplots, we present the average power saving time in Table 1. As depicted, the more early completion occurs, i.e., lower EC-bound, the better gets the result by the on-line hibernation decision of HEART compared to OSPAL. We note that for higher utilization this difference is more significant due to the increased impact of early completion on the system. Throughout all cases, H5 shows the best results among the HEART-algorithms. However, we emphasize that H5 does not dominate H3 and H1, as cases similar to Figure 3 confirm. The task sets under consideration share the same semi-harmonic periods. Hence, there are many time points where the releases of jobs coincide among several processors. In general we observe that in such cases it is better to procrastinate at these time instances, which is achieved the best with H5. When we consider high worst-case utilization (U = 80%) and a low early completion bound (EC = 0.05), HEART



(a) Utilization = 5%

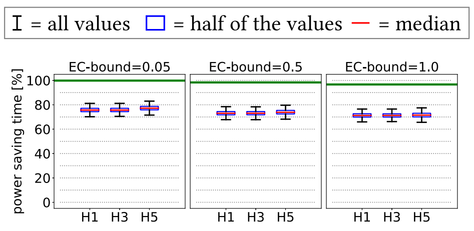


(b) Utilization = 40%

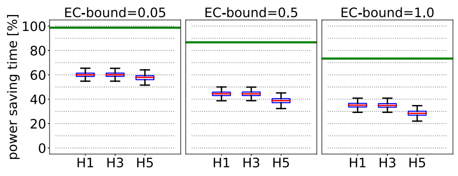


(c) Utilization = 80%

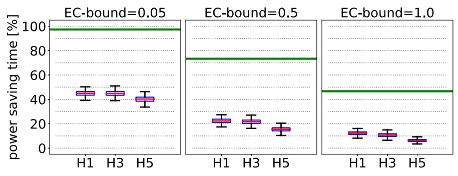
Fig. 6. Results for numerical evaluation for sporadic tasks with predefined semi-harmonic inter-arrival times.



(a) Utilization = 5%



(b) Utilization = 40%



(c) Utilization = 80%

Fig. 7. Results for numerical evaluation for sporadic tasks with standard log-uniform minimal inter-arrival times.

outperforms the state-of-the-art method significantly. In this case the average power saving time can be increased from 15.87% to 43.37%, which is an improvement of 173%.

In addition, we evaluate HEART without constraining hyper-periods, i.e., using directly the values drawn from the interval [10, 1000] [ms] log-uniformly as periods, according to 2) in Section 7.1. The results are presented in Figure 5. We note that OSPAL is not applicable in general for such a scenario, due to high complexity as a result of high hyperperiod. As a result, we do not to report any information regarding OSPAL in Figure 5. As shown in Figure 5, H1 and H3 outperform H5 in most cases. Since there are much less coinciding releases of jobs among several processors comparing to the previous evaluation setup, H1 and H3 gain benefits from earlier procrastination in general. We observe that HEART still performs almost as good as in the case with semi-harmonic periods.

Evaluation for Sporadic Tasks: Since HEART is also applicable to task sets with sporadic task behavior, i.e., job arrivals are further delayed after the period, we examine the impact of sporadicity on the power saving time. As in the periodic case, we run HEART with a *forced procrastination threshold* of $F = 1$, 3 and 5 (resulting in H1, H3, H5), on a system with $M = 5$ processors, each with 20 tasks. Figure 6 and Figure 7 show the power saving time with task sets generated with predefined semi-harmonic periods and with standard log-uniform periods similar to the beforementioned cases. During runtime, for each job $J \in \tau_i$ an additional release-delay is drawn uniformly from the interval $[0, 0.5T_i]$. The power saving time cannot be higher than the baseline $(1 - \frac{1}{1.5}U \cdot \text{EC-bound})$, where $\frac{1}{1.5}U \cdot \text{EC-bound}$ is a lower bound on the time that some processor executes a job. The baseline is marked by a green horizontal line. Please note that OSPAL can not be utilized for sporadic task sets, since the job arrival pattern becomes unpredictable for off-line determination of hibernation times. We observe that the average power saving time with standard log-uniformly drawn periods in Figure 7 behaves almost like in Figure 5. In fact, HEART even slightly benefits from sporadic behavior injected into the schedule since it results in lower utilization of the processor. If we compare the results in the case with semi-harmonic periods, i.e., Figure 6 and Figure 4, we observe that HEART performs differently in the sporadic case. In contrast to the periodic case, H1 and H3 perform better than H5 in the sporadic case since due to the random assignment of sporadic delays there are less coinciding job releases among the processors. The scheduling policies H1 and H3 are the superior choice to handle this scenario in almost all cases. For the sporadic setup, HEART still produces a large power saving time comparable to the results in Figure 4.

Case Study on a Concrete Exemplary System. Although the increase of the power saving time leads to a reduction of energy consumption, the relation of the actual energy consumption strongly depends on the concrete hardware platform. More specifically it depends on the power consumption during the implemented power states and of the required time to perform transitions between them. We provide the evaluation of energy saving of a concrete schedule $\mathcal{S}$ for a duration t_{all} as follows: Let t_{pr} be the total forced procrastination time and n_{pr} the number of forced procrastinations. Moreover, we denote by $t_{ac,j}$ the total time that the j-th processor is active, i.e., executing a task, in the schedule $\mathcal{S}$. The energy consumption of the schedule $\mathcal{S}$ without utilizing hibernation can be *calculated* by

$$\overline{E_{\mathcal{S}}} = \sum_j t_{ac,j} \cdot P_{ac} + t_{all} \cdot P_{id}. \tag{6}$$

Furthermore, the energy consumption utilizing hibernation can be *calculated* by

$$\overline{E_{\mathcal{S}}^{hib}} = \sum_j t_{ac,j} \cdot P_{ac} + (t_{all} - t_{pr}) \cdot P_{id} + (t_{pr} - n_{pr} \cdot O_{total}) \cdot P_{hi} + n_{pr} \cdot O_{total} \cdot (P_{id} + P_{ac}). \tag{7}$$

The difference between $\overline{E_{\mathcal{S}}^{hib}}$ and $\overline{E_{\mathcal{S}}}$ is the amount of energy that is saved when going to hibernate mode instead of idling. Combining Equations (1) and (4) yields $pst(\mathcal{S}) = t_{pr} - n_{pr}(O_{total} \cdot \frac{P_{ac}+P_{id}-P_{hi}}{P_{id}-P_{hi}})$. We use that to obtain

$$\overline{E_{\mathcal{S}}} - \overline{E_{\mathcal{S}}^{hib}} = t_{pr} \cdot (P_{id} - P_{hi}) - n_{pr} \cdot O_{total} \cdot (P_{ac} + P_{id} - P_{hi}) = (P_{id} - P_{hi}) \cdot pst(\mathcal{S}). \tag{8}$$

This shows that the reduction of power consumption when considering hibernation is linearly related to the power saving time. Thus, given a concrete system with known power consumption, the power saving time can be used to assess the energy savings of different schedules. The power saving time itself, however, depends on the break-even time and therefore on the power consumption properties of the system.

In the following, we compare the above calculated energy consumption to a real measurement to support our examination. Due to the limited availability of suitable hardware, we reuse our exemplary system (Section 3.1), i.e., a Texas Instruments MSP430FR6989 Launchpad development board (16 Bit RISC Microcontroller), to conduct the following evaluation. Since this system only has a single processor, we limit the task set to the single-processor case, use the generated schedules from the compared schedulers and measure the total energy consumption throughout the execution of the schedule. We note that we do not run an OS or the full scheduler on the hardware, we simulate the schedule off-line and flash a sequence of execution states to the system, which are then simulated from a bare-metal application. We simulate memory accesses and backup and

Table 2. Power Saving Time [%] for The Single-Processor Case

Utilization / PMCs	**OSPAL**	**HEART**
$U = 5.0\%$, **All 1P**	93.84	95.99
$U = 5.0\%$, **Mixed**	87.68	89.72
$U = 40.0\%$, **All 1P**	58.96	78.48
$U = 40.0\%$, **Mixed**	54.48	70.73
$U = 80.0\%$, **All 1P**	19.98	59.72
$U = 80.0\%$, **Mixed**	27.26	59.03

Table 3. Energy Consumption of the Different Schedulers

Utilization / PMCs	$\overline{\textbf{EDF}}$	$\overline{\textbf{OSPAL}}$	$\overline{\textbf{HEART}}$	$\overline{\%}$	**EDF**	**OSPAL**	**HEART**	**%**
$U = 5.0\%$, **All 1P**	$0.4700J$	$0.3175J$	$0.3140J$	-01.1%	$0.4303J$	$0.3064J$	$0.2742J$	-10.5%
$U = 5.0\%$, **Mixed**	$0.4698J$	$0.3291J$	$0.3257J$	-01.0%	$0.4913J$	$0.2950J$	$0.2863J$	-02.9%
$U = 40.0\%$, **All 1P**	$0.5087J$	$0.4130J$	$0.3812J$	-07.7%	$0.4911J$	$0.3451J$	$0.3444J$	-00.2%
$U = 40.0\%$, **Mixed**	$0.5037J$	$0.4179J$	$0.3917J$	-06.3%	$0.5107J$	$0.3887J$	$0.3542J$	-08.9%
$U = 80.0\%$, **All 1P**	$0.5507J$	$0.5183J$	$0.4537J$	-12.5%	$0.4753J$	$0.4683J$	$0.3821J$	-18.4%
$U = 80.0\%$, **Mixed**	$0.5402J$	$0.4968J$	$0.4459J$	-10.2%	$0.5096J$	$0.4727J$	$0.4412J$	-06.6%

Our scheduler (HEART) saves more energy than OSPAL in all 6 cases.

restore procedures while entering and leaving the hibernation states. The target evaluation platform implements dynamic power management in the form of system power states. The system features a selection of multiple clocks (up to 16 MHz), where different components as the CPU and the memories can be configured to be driven by a certain clock. Within the power states offered by the system, clocks and entire system components are turned off. We use the low power mode 3 (LPM3) to implement the hibernation state. Within LPM3, most clocks are turned off and SRAM and FRAM are power gated. An auxiliary clock still provides a system wide timer, which we use to implement the wake up time. On expiration, this timer triggers an interrupt, which releases the system from LPM3. We generate 6 task sets according to 3) in Section 7.1 with a precision of $p = 1\%$, fix the early completion bound to 0.2, and modify the utilization and the mapping of tasks to performance classes. The constant overhead is set to $O_{const} = 0.1$ ms for this case study. The results of the power saving time are depicted in Table 2. The rows with *All 1P* each indicate a task set, where all tasks are mapped to 1P. The rows with *Mixed* each indicate a task set, where each one third of the tasks is mapped to 1P, XP and 0P. Whereas for lower utilization the power saving time is almost the same, for higher utilization the power saving time for HEART is more than twice as much as for OSPAL. For the measurements we use a minimal tick of 0.02 ms and stop after 15000 ticks. To increase the measurement precision, we scale each tick for 32 ms instead of 0.02 ms. The results of the energy measurements are provided in Table 3. The overlined columns describe the calculated energy consumption $\overline{E_S^{hib}}$, according to Equation (7), the remaining columns describe the actually measured energy consumption. The % columns describes the difference between OSPAL and our proposed scheduler. A negative number means that the power consumption is reduced with our scheduler. We observe that in a concrete system, considering side effects in the measurement, the total energy consumption slightly differs from the calculated value. This mainly stems from the fact, that considering an average momentary power consumption for execution-i, idle and hibernate is not necessarily accurate. Moreover, the task properties take influence on the momentary power consumption as well. However, our scheduler reduces the total energy consumption by several percent in most cases compared to OSPAL in the physical measurement, as predicted by the calculated energy consumption.

8 CONCLUSION AND OUTLOOK

Emerging technologies for non-volatile main memories allow a system to switch to the hibernation state, where the memory is turned off, without the need of storing and restoring data in the main memory. However, in a multi-processor system such hibernation is only possible when all processors idle at the same time.

In this paper we introduce an NVM-aware task model to describe a system which uses volatile memory and non-volatile memory side by side, realized by different performance classes. We develop a multi-processor procrastination algorithm (HEART) based on partitioned earliest-deadline-first scheduling, forcing certain processors into the idle state at which the shared memory can be turned off to save energy. To evaluate HEART, we compare the impact of different forced procrastination thresholds in terms of (1) time instants to force procrastination and (2) allocation of tasks to performance classes and show that HEART outperforms the state of the art (OSPAL) in terms of power saving time significantly in most cases. Moreover, our approach is more applicable than the state of the art in terms of task periodicity.

In future work we plan to further investigate different forced procrastination thresholds. In fact, HEART can be improved by dynamic forced procrastination thresholds, i.e., the policy is chosen on the fly. Furthermore, we plan to generalize the method to constrained and arbitrary-deadline task sets. This work provides the theoretical baseline for future work, especially on normally-off embedded systems. In addition, we plan to extend the considered platforms and relax the assumed model on the power levels. We plan to investigate the case that the VM is turned off, but the NVM and the processor stay executing. This basically introduces, in addition to changing task execution time and migration overhead, a second power level for execution. Additionally, although the memories only can be turned off when no processor uses them, it can be useful to power off some processors even if others are running meanwhile. With such additional options, schedules can be even further optimized.

REFERENCES

[1] John Augustine, Sandy Irani, and Chaitanya Swamy. 2008. Optimal power-down strategies. *SIAM J. Comput.* 37, 5 (2008), 1499–1516. https://doi.org/10.1137/05063787X

[2] Philippe Baptiste. 2006. Scheduling unit tasks to minimize the number of idle periods: A polynomial time algorithm for offline dynamic power management. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*.

[3] Philippe Baptiste, Marek Chrobak, and Christoph Dürr. 2007. Polynomial time algorithms for minimum energy scheduling. In *Algorithms – ESA 2007*, Lars Arge, Michael Hoffmann, and Emo Welzl (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 136–150.

[4] Sanjoy K. Baruah and Nathan Fisher. 2005. The partitioned multiprocessor scheduling of sporadic task systems. In *RTSS*. 321–329.

[5] Brad D. Bingham and Mark R. Greenstreet. 2008. Energy optimal scheduling on multiprocessors with migration. In *2008 IEEE International Symposium on Parallel and Distributed Processing with Applications*. 153–161.

[6] Enrico Bini and Giorgio C. Buttazzo. 2005. Measuring the performance of schedulability tests. *Real-Time Systems* 30, 1–2 (2005), 129–154. https://doi.org/10.1007/s11241-005-0507-9

[7] Jalil Boukhobza, Stéphane Rubini, Renhai Chen, and Zili Shao. 2018. Emerging NVM: A survey on architectural integration and research challenges. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* (2018).

[8] Gruia Călinescu, Chenchen Fu, Minming Li, Kai Wang, and Chun Jason Xue. 2018. Energy optimal task scheduling with normally-off local memory and sleep-aware shared memory with access conflict. *IEEE Trans. Computers* 67, 8 (2018), 1121–1135.

[9] Jian-Jia Chen. 2013. Task set synthesis with cost minimization for sporadic real-time tasks. In *2013 IEEE 34th Real-Time Systems Symposium*. 350–359.

[10] Jian-Jia Chen and Björn Brandenburg. 2017. A note on the period enforcer algorithm for self-suspending tasks. *Leibniz Transactions on Embedded Systems (LITES)* 4, 1 (2017), 01:1–01:22. https://doi.org/10.4230/LITES-v004-i001-a001

[11] Jian-Jia Chen, Mong-Jen Kao, D. Lee, Ignaz Rutter, and Dorothea Wagner. 2015. Online dynamic power management with hard real-time guarantees. *Theoretical Computer Science* 595 (06 2015). https://doi.org/10.1016/j.tcs.2015.06.014

[12] Jian-Jia Chen, Geoffrey Nelissen, and Wen-Hung Huang. 2016. A unifying response time analysis framework for dynamic self-suspending tasks. In *Euromicro Conference on Real-Time Systems (ECRTS)*. 61–71.

[13] Jian-Jia Chen, Geoffrey Nelissen, Wen-Hung Huang, Maolin Yang, Björn Brandenburg, Konstantinos Bletsas, Cong Liu, Pascal Richard, Frédéric Ridouard, Neil Audsley, Raj Rajkumar, Dionisio de Niz, and Georg von der Brüggen. 2019. Many suspensions, many problems: A review of self-suspending tasks in real-time systems. *Real-Time Systems* 55, 1 (2019), 144–207. https://doi.org/10.1007/s11241-018-9316-9

[14] Jongouk Choi, Hyunwoo Joe, Yongjoo Kim, and Changhee Jung. 2019. Achieving stagnation-free intermittent computation with boundary-free adaptive execution. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 331–344.

[15] Erik D. Demaine, Mohammad Ghodsi, Mohammad Taghi Hajiaghayi, Amin S. Sayedi-Roshkhar, and Morteza Zadimoghaddam. 2013. Scheduling to minimize gaps and power consumption. *J. Scheduling* (2013). https://doi.org/10.1007/s10951-012-0309-6

[16] Vinay Devadas and Hakan Aydin. 2008. On the interplay of dynamic voltage scaling and dynamic power management in real-time embedded applications. In *EMSOFT*. ACM, 99–108.

[17] Vinay Devadas and Hakan Aydin. 2008. Real-time dynamic power management through device forbidden regions. In *IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE Computer Society, 34–44.

[18] UmaMaheswari C. Devi. 2003. An improved schedulability test for uniprocessor periodic task systems. In *15th Euromicro Conference on Real-Time Systems (ECRTS)*. 23–32.

[19] Gaurav Dhiman, Raid Ayoub, and Tajana Rosing. 2009. PDRAM: A hybrid PRAM and DRAM main memory system. In *2009 46th ACM/IEEE Design Automation Conference*. 664–669.

[20] Xiangyu Dong, Cong Xu, Yuan Xie, and Norman P. Jouppi. 2012. Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 31, 7 (2012), 994–1007.

[21] Chenchen Fu, Yingchao Zhao, Minming Li, and Chun Jason Xue. 2017. Maximizing common idle time on multicore processors with shared memory. *IEEE Trans. VLSI Syst.* (2017).

[22] Mario Günzel and Jian-Jia Chen. 2020. Correspondence Article: Counterexample for suspension-aware schedulability analysis of EDF scheduling. *Real-Time Systems Journal* (2020).

[23] Mario Günzel and Jian-Jia Chen. 2021. A note on slack enforcement mechanisms for self-suspending tasks. *Real-Time Systems Journal* (2021).

[24] Chi-Yuan Ha, Yo-Xian Wang, and Che-Wei Chang. 2017. Dynamic power management for wearable devices with non-volatile memory. In *2017 International Conference on Applied System Innovation (ICASI)*.

[25] Christian Hakert, Kuan-Hsun Chen, Simon Kuenzer, Sharan Santhanam, Shuo-Han Chen, Yuan-Hao Chang, Felipe Huici, and Jian-Jia Chen. 2020. Split'n Trace NVM: Leveraging library Oses for semantic memory tracing. In *2020 9th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. IEEE, 1–6.

[26] Christian Hakert, Kuan-Hsun Chen, Mikail Yayla, Georg von der Brüggen, Sebastian Bloemeke, and Jian-Jia Chen. 2020. Software-based memory analysis environments for in-memory wear-leveling. In *25th Asia and South Pacific Design Automation Conference ASP-DAC 2020*. Beijing, China.

[27] Christian Hakert, Kuan-Hsun Chen, Mikail Yayla, Georg von der Brüggen, Sebastian Blömeke, and Jian-Jia Chen. 2020. Software-based memory analysis environments for in-memory wear-leveling. In *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 651–658.

[28] Wen-Hung Huang and Jian-Jia Chen. 2016. Self-suspension real-time tasks under fixed-relative-deadline fixed-priority scheduling. In *Design, Automation, and Test in Europe (DATE)*. 1078–1083.

[29] Ahmet Inci, Mehmet Meric Isgenc, and Diana Marculescu. 2020. DeepNVM++: Cross-layer modeling and optimization framework of non-volatile memories for deep learning. *arXiv preprint arXiv:2012.04559* (2020).

[30] Ahmet Fatih Inci, Mehmet Meric Isgenc, and Diana Marculescu. 2020. DeepNVM: A framework for modeling and analysis of non-volatile memory technologies for deep learning applications. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1295–1298.

[31] Texas Instruments. 2017. MSP430FR58xx, MSP430FR59xx, and MSP430FR6xx Family User's Guide. (2017).

[32] Texas Instruments. 2018. AM654x EVM User's Guide. (2018).

[33] Sandy Irani, Sandeep Shukla, and Rajesh Gupta. 2007. Algorithms for power savings. *ACM Trans. Algorithms* (2007), 23. https://doi.org/10.1145/1290672.1290678

[34] Ravindra Jejurikar and Rajesh Gupta. 2004. Procrastination scheduling in fixed priority real-time systems. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. Association for Computing Machinery. https://doi.org/10.1145/997163.997173

[35] Ravindra Jejurikar, Cristiano Pereira, and Rajesh K. Gupta. 2004. Leakage aware dynamic voltage scaling for real-time embedded systems. In *Design Automation Conference (DAC)*. ACM.

[36] Jian-Jia Chen, Heng-Ruey Hsu, and Tei-Wei Kuo. 2006. Leakage-aware energy-efficient scheduling of real-time tasks in multiprocessor systems. In *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*. 408–417.

[37] Jian-Jia Chen and Tei-Wei Kuo. 2007. Procrastination determination for periodic real-time tasks in leakage-aware dynamic voltage scaling systems.. In *2007 IEEE/ACM International Conference on Computer-Aided Design*. 289–294.

[38] Karthik Lakshmanan and Ragunathan Rajkumar. 2010. Scheduling self-suspending real-time tasks with rate-monotonic priorities. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 3–12. https://doi.org/10.1109/RTAS.2010.38

[39] Yann-Hang Lee, Krishna P. Reddy, and C. Mani Krishna. 2003. Scheduling techniques for reducing leakage power in hard real-time systems. In *15th Euromicro Conference on Real-Time Systems (ECRTS'03)*. https://doi.org/10.1109/EMRTS.2003.1212733

[40] C. L. Liu and James W. Layland. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM* 20, 1 (Jan. 1973), 46–61. https://doi.org/10.1145/321738.321743

[41] Jane W. S. Liu. 2000. *Real-Time Systems* (1st ed.). Prentice Hall PTR.

[42] Songran Liu, Wei Zhang, Mingsong Lv, Qiulin Chen, and Nan Guan. 2020. LATICS: A Low-overhead adaptive task-based intermittent computing system. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2020), 1–1.

[43] Joseph A. Paradiso and Thad Starner. 2005. Energy scavenging for mobile and wireless electronics. *IEEE Pervasive Computing* 4, 1 (2005), 18–27.

[44] Shwetak N. Patel and Joshua R. Smith. 2017. Powering pervasive computing systems. *IEEE Pervasive Computing* 16, 3 (2017), 32–38.

[45] Matthew Poremba, Tao Zhang, and Yuan Xie. 2015. Nvmain 2.0: A user-friendly memory simulator to model (non-) volatile memory systems. *IEEE Computer Architecture Letters* 14, 2 (2015), 140–143.

[46] Ragunathan Rajkumar. 1991. *Dealing with Suspending Periodic Tasks*. Technical Report. IBM T. J. Watson Research Center. http://www.cs.cmu.edu/afs/cs/project/rtmach/public/papers/period-enforcer.ps.

[47] Jun Sun and Jane W.-S. Liu. 1996. Synchronization protocols in distributed real-time systems. In *Proceedings of the 16th International Conference on Distributed Computing Systems*. 38–45. https://doi.org/10.1109/ICDCS.1996.507899

[48] Xiaobo Fan, Carla S. Ellis, and Alvin R. Lebeck. 2001. Memory controller policies for DRAM power management. In *ISLPED'01: Proceedings of the 2001 International Symposium on Low Power Electronics and Design (IEEE Cat. No.01TH8581)*. https://doi.org/10.1109/LPE.2001.945388

[49] Yifan Zhu and Frank Mueller. 2005. Feedback EDF scheduling of real-time tasks exploiting dynamic voltage scaling. *Real Time Syst.* 31, 1–3 (2005), 33–63.

Received April 2021; revised June 2021; accepted July 2021