Received December 20, 2021, accepted December 31, 2021, date of publication January 4, 2022, date of current version January 12, 2022.

Digital Object Identifier 10.1109/ACCESS.2021.3140166

Supporting Swap in Real-Time Task Scheduling for Unified Power-Saving in CPU and Memory

SUJI YOON, HEEJIN PARK, KYUNGWOON CHO, AND HYOKYUNG BAHN, (Member, IEEE)
Department of Computer Engineering, Ewha Womans University, Seoul 120-750, South Korea

Corresponding author: Hyokyung Bahn (bahn@ewha.ac.kr)

This work was supported in part by the ICT Research and Development Program of IITP (Institute of Information & communications Technology Planning & Evaluation) grant funded by the Korea government (MSIT) (2019-0-00074, developing system software technologies for emerging new memory that adaptively learn workload characteristics) and (2020-0-00121, development of data improvement and dataset correction technology based on data quality assessment).

ABSTRACT As the size of data grows rapidly in modern IoT (Internet-of-Things) and CPS (Cyber-Physical System) applications, the memory power consumption of real-time embedded systems increases dramatically. Unlike general-purpose systems where memory consumes about 10% of the CPU power consumption, modern real-time systems have the memory power of 20–50% of CPU power. This is because the memory size of a real-time system should be large enough to accommodate the entire task set, and thus DRAM refresh operations become a major source of power consumption. In this article, we present a new swap scheme for real-time systems, which aims at reducing memory power consumption. To support swap with real-time constraints, we adopt high-speed NVM storage and co-optimize power-savings in CPU and memory. Unlike traditional real-time task models that only consider the executions in CPU, we define an extended task model that characterizes memory and storage paths of tasks as well, and tightly evaluate the worst-case execution time by formulating the overlapped latency between CPU and memory. By optimizing the CPU supply voltage and the memory swap ratio of given task set, our scheme reduces the energy consumption of real-time systems by 31.1% on average under various workload conditions.

INDEX TERMS Real-time task scheduling, partial swap, genetic algorithm, power saving, voltage scaling, deadline, high-speed storage, NVM.

I. INTRODUCTION

With the recent advances in IoT (Internet-of-Things) and CPS (Cyber-Physical System) technologies, reducing the power consumption in battery-based real-time systems is becoming increasingly important [1], [2]. Dynamic voltage scaling is a widely used technique for CPU power-saving by lowering the supply voltage of a processor when the computational load of tasks is less than the processing capacity of CPU [3]–[5]. If we lower the voltage supplied to CPU, the computational speed of the processor becomes slow, which increases the execution time of tasks. However, as CPU power consumption is proportional to the square of the supply voltage, although the execution time is increased, the overall energy can be saved. Thus, CPU voltage scaling offers flexibilities in real-time task scheduling by considering the computational load of tasks and energy-saving effects.

Meanwhile, as the size of data grows rapidly in modern embedded systems, memory power consumption of the system increases dramatically [6], [7]. Due to its volatile medium characteristics, DRAM needs continuous refresh of all cells in order to maintain its contents although no read/write operation is performed. As the memory size increases, the power consumption by refresh also increases, which accounts for a significant portion of total power consumption in real-time embedded systems [8].

Unlike general purpose systems, real-time systems keep the entire footprint of all tasks in memory to guarantee deadlines, so it is not possible to use virtual memory swap that loads data from storage on demand [9]. This is because predicting the time of accessing code or data in storage is not feasible upon the execution of tasks in CPU. Thus, the memory size of a real-time system should be large enough to accommodate the entire task set, which makes DRAM refresh operations the major source of power consumption. Note that this is not the case for general-purpose systems like laptops, where the two main sources of power consumption are CPU

The associate editor coordinating the review of this manuscript and approving it for publication was Christian Pilato.

and display, while DRAM accounts for only 3% of total power consumption [45]. Specifically, when comparing CPU and memory power in laptops, memory consumes only 10% of CPU power consumption [45]. However, this large gap has been narrowed in modern real-time systems as CPUs adopt power-saving techniques like voltage scaling but the size of real-time tasks that should reside in memory continues to grow. For this reason, the memory power of mobile embedded systems and real-time systems has increased to 20-50% of CPU power [4], [6].

In this article, we present a new swap scheme for real-time systems, which aims at reducing the DRAM size and memory power consumption. To support swap with real-time constraints, we adopt high-speed NVM (non-volatile memory) storage and accurately estimate the swap latency. NVM technologies have recently been caught attention and some commercial products like Intel's Optane^{TM} are already available on the market [12]. As high-speed NVM storage has low-variance of access time [10], [13], our idea is to place a certain part of a task in NVM storage rather than shadowing in memory. This can reduce the size of DRAM and memory energy consumption in real-time embedded systems.

We, then, integrate our swap scheme with CPU voltage scaling and formulate the effect of the two techniques as a unified measure. Although CPU voltage scaling and memory swap reduce the energy consumption, they increase the execution time of tasks, possibly resulting in the deadline misses of real-time tasks. Thus, we define an extended task model that characterizes memory and storage latency as well as CPU executions, and accurately estimate the worst-case execution time when adopting these energy saving techniques. In our model, we tightly evaluate the worst-case execution time by formulating the overlapped latency between CPU, memory, and swap storage. As co-optimizing the power-saving configurations of CPU and memory with real-time constraints is a complex optimization problem, we use genetic algorithms to determine CPU voltage level and memory swap ratio of given task set.

To assess the effectiveness of the proposed scheme, we perform simulation experiments for a wide range of workload conditions. Our experimental results show that the proposed scheme significantly reduces the power consumption of real-time systems. Specifically, the energy-saving effect is 31.1% on average and up to 45.6% without deadline misses. The main contributions of this article can be summarized as follows.

- Unlike traditional real-time task models that only consider executions in CPU, we define an extended task model that also characterizes the memory and storage paths of tasks.
- We propose a swap scheme for real-time systems, which partially swaps out a certain portion of a task and restores it before the task activates, thereby preventing page faults.
- Our model tightly evaluates the scaled worst-case execution time of a task, considering the overlapped latency between CPU, memory, and swap storage, to minimize overall energy consumption.
- We design a steady-state genetic algorithm to co-optimize the energy consumption in CPU and memory without deadline misses by defining appropriate cost functions and genetic operators.

The remainder of this article is organized as follows. Section II briefly summarizes previous works related to this article. In Section III, we explain the partial swap scheme and integrate it with CPU voltage scaling for energy efficient real-time task scheduling. Section IV describes the optimization of our problem with genetic algorithms. In Section V, we present experimental results to validate the effectiveness of the proposed scheme. Finally, we conclude this article in Section VI.

II. RELATED WORK

A. SWAP IN REAL-TIME SYSTEMS

Traditional memory swap used in general purpose systems determines the part of a task to be swapped based on the prediction of re-reference likelihood by the replacement algorithm [19], [28]. For example, the CLOCK algorithm evicts a page not used recently as it is not likely to be used again in the near future. However, page faults may occur in these systems as an evicted page can be used again. Unlike such types of systems, real-time systems do not allow page faults since unpredictable I/O latency may incur deadline misses [9]. Thus, the full address space of a task is pinned on the physical memory once a task starts its execution.

In order to satisfy this semantic, swap in real-time systems should work in a completely different way from that in general-purpose systems. To this end, our scheme swaps out a certain portion of a task during its inactive period, but restores the swapped part to physical memory before its activation. Thus, it is guaranteed that an entire footprint of a task resides in physical memory while the task is active. This indicates that we do not need to consider the target of swap. Thus, the main focus of our swap is to determine how much a task's memory should be involved in swap rather than considering the replacement algorithm.

Previous studies on paging systems have suggested some replacement algorithms for soft real-time tasks. Kim *et al.* adopt flash memory as a code storage and present a new page replacement algorithm in portable media players [21]. Lee *et al.* present MRT-PLRU (multitasking real-time constrained combination of pinning and LRU) that combines pinning and LRU (least recently used) policies to reduce the memory size of real-time systems [22]. However, these studies probabilistically guarantee the real-time task's deadlines without fully satisfying the constraints of hard real-time systems. Thus, they are different from our approach that meets the complete real-time constraints with memory swap.

B. REAL-TIME TASK SCHEDULING

Real-time task scheduling has been widely studied for decades. For periodic tasks, EDF (Earliest Deadline First) is known to find a schedule that does not miss the deadlines of all tasks in the task set if there exists any feasible schedule. However, EDF cannot be used if there are multiple processors or cores to execute the task set. Baruah *et al.* present the Pfair (Proportionate-fair) scheduling that optimally and efficiently schedules periodic tasks on symmetric multiprocessors [24]. Pfair scheduling differs from traditional real-time scheduling principles in that tasks are explicitly required to proceed at a steady rate.

Anderson and Srinivasan present a work-conserving version of Pfair scheduling called ER-fair (Early-Release fair) [30]. ER-fair differs from original Pfair scheduling in that it allows the execution of the latter part of a task as soon as the former part of the same task is completed. Anderson and Srinivasan also define the notion of intra-sporadic task, in which subtasks of a task may be released late, and present variants of Pfair and ER-fair scheduling [31]. They prove the feasibility condition for scheduling intra-sporadic tasks, and present a polynomial-time algorithm that can be used to optimally schedule intra-sporadic tasks on 1 or 2 core processors.

In real-time systems, the utilization test with the worst case execution time of tasks should be done beforehand as we need to know if the fixed resources can accommodate the given real-time task set. However, actual executions may be completed much earlier than the worst case, which will lead to the waste of resources significantly. To cope with this situation, some reactive schemes have been presented. Chen *et al.* decide the baseline schedule of the task set in advance, but while the tasks are actually executed, new proactive schedules are generated by considering the completion of the tasks or arrival of new tasks, leading to efficient resource management [32]. This can be adopted in cloud computing environments where resources can be scaled as the workload evolves.

Dehnavi *et al.* utilize the hybrid cloud infrastructure for scheduling of real-time tasks in industrial systems [33]. They propose resource provisioning policies to partition a given workload among different computing tiers, including local private clouds, edge nodes, fog nodes, and public cloud data centers. Zhou *et al.* propose a theoretical model for real-time scheduling problems in dynamic cloud manufacturing services [34]. To improve performances, they also propose a scheduling policy based on dynamic data-driven simulations.

C. DYNAMIC VOLTAGE SCALING

Dynamic voltage scaling has been studied extensively for the power-saving of processors in various industrial systems [1], [2]. Pillai and Shin propose three techniques to find the lowest voltage level to meet the deadlines of given real-time task set [1]. They are static, cycle-conserving, and look-ahead voltage scaling techniques. Static voltage scaling selects the voltage level of a processor statically, whereas cycle-conserving voltage scaling makes use of the reclaimed cycles for decreasing the voltage level of a processor if the execution of a real-time task finishes earlier than its worst case execution time. Look-ahead voltage scaling tries to lower the voltage level of a processor even more by analyzing the required amount of computation in the near future and postpones the scheduling of the task based on the result of the analysis.

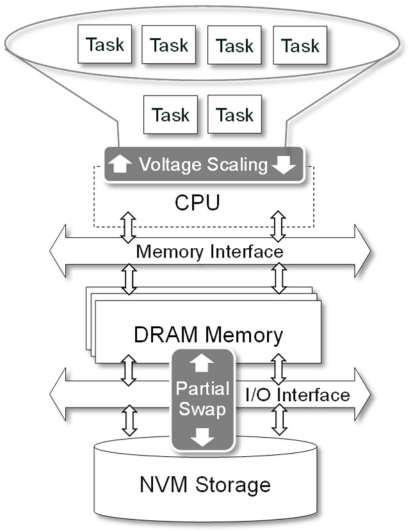


FIGURE 1. The architecture of the memory swap and CPU voltage scaling proposed in this article.

Lee *et al.* make use of the slack time to lower the voltage level of a processor [16]. Specifically, the voltage level of a processor is lowered if some clock cycles are reclaimed by completing a task before its deadline reaches. Ghor and Aggoune aim to find the schedules with the least voltage level of a processor for real-time tasks by utilizing the slack time [2]. In particular, their algorithm aims at stretching the worst case execution time of real-time tasks as much as possible without violating deadlines. Nam *et al.* present a task scheduling policy for real-time systems that aims at reducing the energy consumption of CPU and memory selectively by considering the relative energy-saving effect of the two layers [4]. To reduce the time overhead of scheduling and maximize the power saving effect, their policy adopts dynamic programming with the constraint of resource utilization. Bahn *et al.* present a new task model for hybrid memory placement in hard real-time systems [15]. Specifically, they re-evaluate the worst-case execution time of a task by considering the memory location of each task in heterogeneous memory environments.

III. THE PROPOSED SCHEME

In our task model, a real-time task set is defined as $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$, and the target system has CPU with a voltage scaling function and main memory supporting swap as shown in Figure 1. A task τ_i is characterized by $<C_i, T_i, M_i>$, where C_i is the worst case execution time of τ_i with the default CPU voltage and no memory swap, T_i is the period of τ_i, and M_i is the memory footprint of τ_i. We consider periodic real-time tasks, and thus the deadlines are implicitly determined by the period.

By following the common assumptions of real-time task models in previous work [4], [16], and considering our partial swap model, we make the following four assumptions.

Assumption 1: All tasks are independent, and thus the result of a task does not affect others.

There may be some dependent tasks in real-world task set, but most real-time scheduling studies make this assumption to simplify the problem without loss of generality. That is, dependent tasks can be merged into a single task as they should be performed sequentially by using the result of the preceding task as the input to the following task.

Assumption 2: Tasks can be preempted and the overhead of context switch from one task to another is negligible.

In computer systems, CPU is a representative resource that allows the preemption of a task during its execution. This is because the context of a task can be easily saved and restored without incurring large overhead. Specifically, the context switch of a processor usually takes 5-10 microseconds, which is less than 0.01% of the minimum time quantum between context switch, and thus we can hide it by including in the actual execution time of a task [35].

Assumption 3: When the target clock frequency is determined, the supply voltage of CPU can be adjusted accordingly.

When the clock frequency of a processor increases, the supply voltage of a processor also becomes higher. Although they do not have exact linear relations, it is known that the supply voltage of real processors is adjusted according to clock frequency based on a linear-like function. For example, in Transmeta Crusoe processors, when the clock frequency is changed from 500 MHz to 1 GHz, the supply voltage is adjusted from 1.35 V to 2.80 V [36].

Assumption 4: The access time of swap storage is predictable.

This was not possible in HDD or flash storage, where the access time depends heavily on the internal state of storage [20]. In HDD storage, the access time varies depending on the disk scheduling and the head movement. In flash memory, the access time fluctuates greatly when garbage collection is performed [18], [19]. However, NVM storage has predictable access time [11], [14], and thus we can estimate the worst case access time to load storage data to memory if we know the size of data to be swapped.

A. BASIC MODEL

In our model, the worst case execution time C_i of a task should be adjusted as we use CPU voltage scaling and memory swap. That is, C_i should be recalculated based on the longest time path between CPU and memory considering the increased latency due to the lowered supply voltage and swap I/O. Actually, the scaled worst case execution time is determined by the slower time component of executing instructions in CPU and accessing memory since executions in CPU and memory can be overlapped. That is, we tightly estimate the latency that may overlap between CPU and memory by defining the function f for scaling C_i as follows.

$$f(C_i) = \max\{f_{\text{CPU}}(C_i), f_{\text{SWAP}}(C_i)\} + \varepsilon_i \tag{1}$$

$f_{\text{CPU}}(C_i)$ and $f_{\text{SWAP}}(C_i)$ are the scaled worst case execution time of τ_i by applying CPU voltage scaling and swap, respectively, and ε_i is the stall factor for executing swap I/O commands in CPU. $f_{\text{CPU}}(C_i)$ and $f_{\text{SWAP}}(C_i)$ can be defined as

$$f_{\text{CPU}}(C_i) = C_i/\mu_i \tag{2}$$

$$f_{\text{SWAP}}(C_i) = C_i + 2 * latency_{\text{SWAP}}(r_i * M_i) \tag{3}$$

where μ_i is the relative clock frequency of CPU compared to the default frequency to execute task τ_i, r_i is the swap ratio of τ_i, and $latency_{\text{SWAP}}$ is the time required to access the swap storage as a function of the swap I/O size.

Based on this model, the schedulability test of a real-time task set can be performed by the following utilization test, implying that the scaled worst case execution time after adopting CPU voltage scaling and memory swap should satisfy this inequality.

$$U = \sum_{\tau_i \in \Gamma} \frac{f(C_i)}{T_i} \leq 1 \tag{4}$$

If a real-time task set passes this schedulability test, we can obtain a feasible schedule for the given set of tasks by the earliest deadline first (EDF) algorithm [16], [17]. Note that EDF schedules the task with the nearest deadline first.

Let us look at TABLE 1 to see an example situation of the utilization test. There are two tasks, τ_1 and τ_2, whose worst case execution times C_1 and C_2 are 4 and 8, respectively, and their periods are equally 25. The schedulability of the task set can be tested by calculating the utilization of the tasks, i.e., $U = 4/25 + 8/25 = 0.48$. As U is less than 1, the task set is schedulable. Figure 2(a) shows the scheduling result for the example in TABLE 1. As shown in the figure, all the tasks can be executed within their deadlines, but the scheduling generates a large portion of idle intervals.

This idle slot can be reduced by lowering the supply voltage of CPU, thereby increasing the system utilization. For example, if a low clock frequency of 0.5 is applied for both tasks τ_1 and τ_2, the scaled worst case execution time $f(C_1)$ and $f(C_2)$ will be 8 and 16, respectively. As a result, the CPU utilization increases to $U = 8/25 + 16/25 = 0.96$, where $U < 1$ is still satisfied, so it is schedulable. Figure 2(b) shows the scheduling result after voltage scaling is adopted. As we see, the idle slot is greatly reduced compared to Figure 2(a), which will eventually lead to reduced power consumption.

Now, let us see how the power consumption can be further reduced by considering the memory system. In real-time systems, storage I/O does not occur as all tasks reside in memory and virtual memory swap is not allowed. This is because storage I/O increases the memory access time excessively and it makes the prediction of the execution time difficult. However, we focus on the fact that recently emerged NVM storage has fast and predictable access time, and thus we suggest partial swap that places some part of a task in NVM storage and loads the swapped part to memory before the task

TABLE 1. An example of a task set for schedulability test.

Task	Worst case execution time	Period	Memory footprint
τ_1	4	25	0.5
τ_2	8	25	0.5

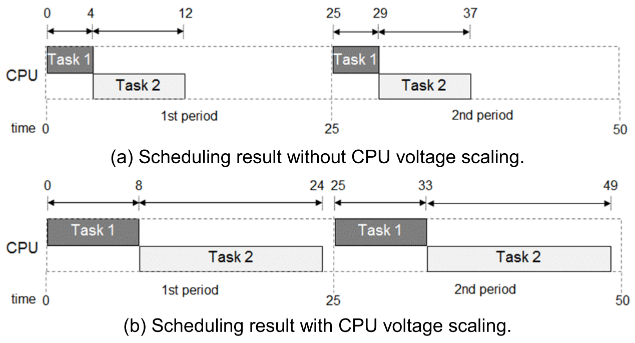


FIGURE 2. Comparison of the scheduled results.

is executed in CPU. This eventually leads to the reduction of the DRAM capacity, contributing to power-saving of the system.

As swap-out and swap-in need I/O time, the worst-case execution time of tasks should be re-evaluated by considering this latency. If we increase the swap ratio of a task, memory power can be saved more by reducing the DRAM capacity, but the worst-case execution time of the task may increase due to the handling of swap I/O. Conversely, if we lower the swap ratio, the worst-case execution time of a task may increase less, but the power-saving effect would be reduced.

Meanwhile, there is a trade-off between memory swap and CPU voltage scaling. For example, if we increase the swap ratio, the possibility of lowering the CPU voltage is decreased. Thus, it is necessary to maximize power-savings by combining these two techniques appropriately. In addition, as CPU voltage scaling and memory swap can be overlapped, tight evaluation of the worst case execution time is necessary.

Let us see the situation of our partial swap with the example in TABLE 1. Figure 3 shows the result of swap in conjunction with the CPU voltage scaling of Figure 2(b) with this example. The figure shows when the swap ratio of tasks 1 and 2 are equally 0.5 and a swap command takes 0.5 time unit in CPU. Note that the exact locations of the swap command may be varied in real situations, but we have marked them at the end of each task for simplicity. Remind that in the schedule of Figure 2(b), 1 time unit per period remains after adopting voltage scaling. Thus, swap overhead should not exceed this remaining time slot. Although the I/O time for swap-out and swap-in increases as the swap ratio of a task becomes higher, actual swap I/Os can be overlapped with CPU executions. Thus, in an ideal case, I/O latency can be hidden and the worst case execution time will be increased only by the swap I/O command in CPU. However, as the swap ratio increases, the swap I/O time of a task may exceed the inactive period of the task. Due to this reason, the swap ratio of a task is generally set to less than 1.0. As shown in Figure 3, the swap ratio in our example is set to 0.5, and the memory size is reduced to 75% compared to the system without swap. Figure 3 also shows how the design of the system can be changed by adopting our swap with this example.

B. EXTENDING THE BASIC MODEL

From now on, we will discuss how the basic model can be extended for multi-core systems. In our CPU model, we assume that all cores have the same computing powers (i.e., symmetric multi-core processor) as in most server and embedded processor architectures. In this architecture, if a task executed on one core is interrupted, it can be resumed later on another core. However, it is not allowed that a task is executed in multiple cores concurrently as a single task should be executed in a sequential manner.

Based on this multi-core architecture, the feasibility test in our basic model can be simply extended by replacing the right side of Equation (4) by K, where K is the number of cores in the processor.

$$U = \sum_{\tau_i \in \Gamma} \frac{f(C_i)}{T_i} \leq K \tag{4$'$}$$

In a single-core processor system, if the schedulability test is passed, we can use the EDF algorithm to determine a schedule that does not miss the deadlines of all tasks. However, this does not work in multi-core processors as a single task is not allowed to be executed in multiple cores concurrently. To cope with this situation, Pfair (Proportionate-fair) scheduling has been introduced as a way of scheduling periodic tasks on multi-core/multi-processor systems [24], [30]. The basic philosophy of Pfair is similar to EDF, but it performs scheduling based on time quantum to make progress of each task at steady rates. We determine the schedule of a task set through Pfair scheduling when it passes the utilization test of Equation (4)$'$.

C. ENERGY POWER MODEL

The CPU energy E_{CPU} of a CMOS processor is dominated by charging and discharging gates in circuits, and can be formulated as a function of supply voltage and operating frequency [37], [38], that is

$$E_{\text{CPU}} = \sum_{\tau_i \in \Gamma} c V_i^2 f_i t_i \tag{5}$$

where c is the effective switch capacitance, V_i is the supply voltage for task τ_i, f_i is the CPU clock frequency for executing task τ_i, and t_i is the time to execute task τ_i under this CPU mode.

In our model, supply voltage V_i is adjusted according as the clock frequency f_i is varied. It is known that the clock frequency and the supply voltage have some linear-like relations, although they do not have the exact linear relation. The function to model this relation depends on processors, and we use the ARM Cortex-R52 processor model [39].

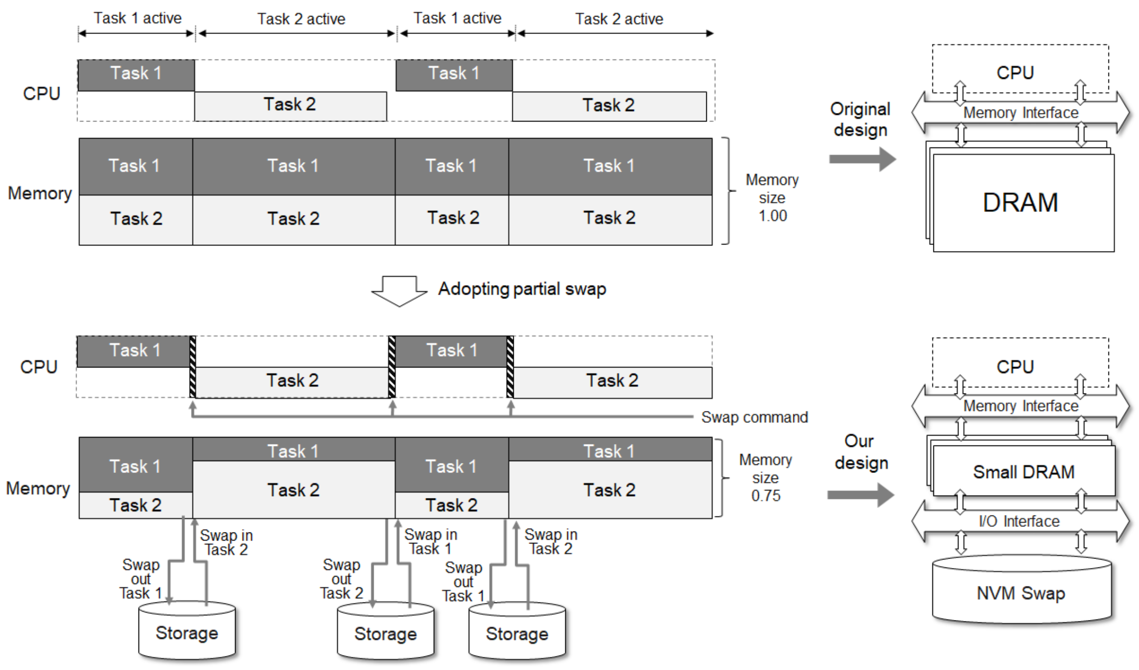


FIGURE 3. **The effect of adopting partial swap.**

The memory energy E_{MEM} is the sum of dynamic energy $E_{\mathrm{M\_dyn}}$ and static energy $E_{\mathrm{M\_stat}}$ [40], that is

$$E_{\mathrm{MEM}} = E_{\mathrm{M\_dyn}} + E_{\mathrm{M\_stat}} \tag{6}$$

The dynamic energy $E_{\mathrm{M\_dyn}}$ is energy consumed while a read or a write operation is performed [41], which can be modeled as

$$E_{\mathrm{M\_dyn}} = \sum_{\tau_i \in \Gamma} (read_i * E_{\mathrm{M}\_read} + write_i * E_{\mathrm{M}\_write}) \tag{7}$$

where $read_i$ and $write_i$ are the number of memory read and write operations on task τ_i, respectively, and E_{M_read} and E_{M_write} are the read and write energy for the default access size of DRAM, respectively. The static energy $E_{\mathrm{M\_stat}}$ is the energy consumed consistently irrespective of any operations in DRAM memory, which can be calculated as

$$E_{\mathrm{M\_stat}} = P_{\mathrm{M}} \sum_{\tau_i \in \Gamma} M_i \left(\frac{f(C_i)}{T_i} + (1 - r_i) \left(1 - \frac{f(C_i)}{T_i} \right) \right) T \tag{8}$$

where P_{M} is the static power of DRAM per capacity, r_i is the swap ratio of task τ_i, and T is the total running time of the system.

The storage energy E_{STR} of our swap can be calculated as

$$E_{\mathrm{STR}} = \sum_{\tau_i \in \Gamma} M_i r_i (E_{\mathrm{STR}\_read} + E_{\mathrm{STR}\_write}) n_i \tag{9}$$

where r_i is the swap ratio of task τ_i, n_i is the total number of swaps performed on task τ_i, and E_{STR_read} and E_{STR_write} are the read and write energy for the unit access size of NVM, respectively. Note that we do not consider the static power of storage as it is non-volatile, and thus does not spend refresh power.

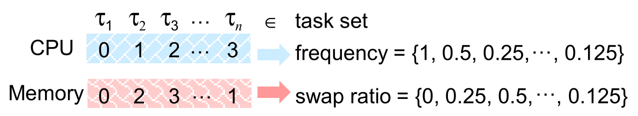
IV. OPTIMIZATIONS WITH GENETIC ALGORITHMS

Our problem is to select the CPU voltage level and the memory swap ratio of all real-time tasks in the task set, which aims at minimizing the energy consumption of the system with deadline constraints. This is a kind of combinatorial optimization problem known as NP-hard. For example, if there are 4 CPU voltage levels and 5 memory swap ratios, the number of possible states for each task is 20. When the number of tasks is N, there are 20^N cases, and searching all of these is not feasible even with high-end server systems.

To cope with this situation, we explore our search space by genetic algorithms [23]. Specifically, we maintain a small number of candidate solutions that represent the voltage level and the swap ratio of the tasks, and evolves the solution set until it converges. Typical genetic algorithms evolve the solution set by completely replacing an old set with a new one at each iteration. However, this usually loses some superior solutions maintained, making convergence difficult. Specifically, if there are multiple domains to optimize together like our problem (i.e., CPU and memory), it takes much time to converge, and in some cases, the solution set does not converge even after a large number of iterations [9]. To resolve this issue, we replace only a few solutions per iteration. This type of genetic algorithms is called steady-state GA, which has the ability of fast convergence [25].

A. ENCODING

In genetic algorithms, a solution is typically represented by a linear string. As our problem needs to determine the memory



τ1 τ2 τ3 ⋯ τn ∈ task set
CPU
0 1 2 ⋯ 3
frequency = {1, 0.5, 0.25,⋯, 0.125}
Memory
0 2 3 ⋯ 1
swap ratio = {0, 0.25, 0.5,⋯, 0.125}

FIGURE 4. **Encoding of the problem.**

swap ratio and the CPU voltage level of all tasks in the task set, we use two strings and the length of the string is equal to the total number of tasks as shown in Figure 4.

In a theoretical aspect, we can set various levels of CPU voltage and memory swap ratio, but practically we need to set a certain limited number of levels. The default setting of our configuration consists of 4 CPU voltage levels and 4 swap ratios. Specifically, in our encoding, each entry in the CPU string is represented by a 2 bit value {0, 1, 2, 3}, which represents the clock frequency of CPU {1, 0.5 0.25, 0.125}, respectively. Similarly, entries in the memory string can have a 2 bit value {0, 1, 2, 3} representing the swap ratio of {0, 0.125, 0.25, 0.5}, respectively. Note that task τ_1 in Figure 4 is executed under the CPU clock frequency of 1 and the swap ratio of τ_1 is 0. Based on this encoding method, we randomly generate 100 solutions as an initial population.

In genetic algorithms, a cost function is needed to evaluate the quality of a solution. We define our cost function as the energy consumption of the system when the scheduling is performed with the given resource configurations the solution represents. If the CPU utilization exceeds 1 and thus the scheduling is not feasible with the given solution, a penalty value is added to the cost function. That is,

$$Cost(i) = Energy(i) + \alpha Penalty(i) \tag{10}$$

where $Energy(i)$ is the energy consumption of the tasks scheduled by solution i, α is the weight factor, and $Penalty(i)$ is the penalty function of the solution i in case it does not pass the schedulability test, that is

$$Penalty\ (i) = \sum_{\tau_i \in \Gamma} \frac{f(C_i)}{T_i} - K \tag{11}$$

B. SELECTION OF PARENT SOLUTIONS

A selection operation chooses two parent solutions in the current population for generating one or two offspring solutions to evolve the population. This is usually based on a probabilistic rule that assigns higher probabilities to better solutions in order to improve the solution set. In our problem, the goodness of a solution is evaluated based on the cost of the solution. However, if the selection probability is excessively biased, the result of selection may be limited to a small number of extremely superior solutions. This has a risk of premature convergence to a local optimum as the characteristics of a few solutions may rapidly dominate the entire population. To cope with this situation, instead of assigning a selection probability based on the cost of a solution as it is, we rank the solutions by their cost order and then assign selection probabilities based on their ranks. Specifically, we normalize the selection probability such that the best solution in the population is four times more probable to be selected than the worst one [23].

C. CROSSOVER AND MUTATION

The crossover operation merges a certain part of the string from two parents to generate offspring. We use 1-point crossover that randomly selects a cut point within a string, and the offspring is generated by copying the left segment of the cut point from one parent and the right segment from the other. As we have two strings that represent CPU and memory configurations, we select the cut point of each string to perform 1-point crossover independently.

After crossover, a mutation operation is performed for the offspring generated, which perturbs a certain location of the strings in order to widely search the problem space not to stay in a local optimum. Our mutation is performed by selecting a certain random location of the strings and changes it to another random value. The mutation probability of our genetic algorithm is set to 0.01.

D. REPLACEMENT

After an offspring is generated by crossover and mutation, a new population is produced by substituting a solution in the current generation by the offspring. In this article, we discard the worst solution, i.e. a solution that incurs the highest cost, in the current population and insert the offspring generated. Note that this is the most commonly used replacement method in steady-state genetic algorithms [25].

E. STOPPING CRITERIA AND CONVERGENCE

It is not an easy matter to determine the number of iterations repeated for the evolution of genetic algorithms as it is sensitive to the experimental configurations and the problem domain. Instead of setting the constant number of iterations, we repeat the evolution until the population converges [23]. In order to ensure the convergence of our genetic algorithm, we monitor the cost of each solution and the utilization of the system when the task set is scheduled by the solution. Our monitoring results showed that the population converges within 10,000 generations in all cases, and the average running time of our genetic algorithm is 8.7 seconds for its convergence. We also confirmed that our genetic algorithm does not converge to a local optimum since the utilization of the final solution approaches the full capacity of the resources unless the task set is too small to fully utilize the resources. This proves that our policy have the ability of finding sufficiently good solutions that satisfy real-time constraints as well as energy efficiency.

The complexity of genetic algorithms is not easy to prove, but in empirical aspects, we found that our genetic algorithm converges with a constant number of iterations regardless of the number of tasks (up to 1,000 tasks we tested), and hence the complexity of our genetic algorithm can be considered as $O(1)$. Also, as we consider hard real-time systems where resource configurations and scheduling possibility should be determined at the design phase, the running time of our genetic algorithm does not affect the actual execution of tasks in target systems.

V. EXPERIMENTAL RESULTS

In this section, we conduct simulation experiments to assess the effectiveness of the proposed scheme called PSVS-GA (Partial Swap with Voltage Scaling using Genetic Algorithms). We developed our in-house simulator to evaluate the effectiveness of PSVS-GA [42]. We compare PSVS-GA with three schemes, VS-GA (Voltage Scaling using Genetic Algorithms), PS-GA (Partial Swap using Genetic Algorithms), and Baseline. Baseline does not use either CPU voltage scaling or memory swap. PS-GA uses partial swap for memory power-saving similar to PSVS-GA, but does not use CPU voltage scaling, and VS-GA optimizes the CPU voltage level for each task, but does not consider memory swap. Similar to the proposed PSVS-GA, PS-GA and VS-GA make use of genetic algorithms for their optimizations. This implies that the improvement of our scheme against PS-GA and VS-GA is obtained through the tight modeling of worst-case execution time by hiding the overlapped latency rather than just thorough optimizations by genetic algorithms.

The experimental configuration of our simulation consists of 1.6GHz 4-core ARM cortex-R52 real-time processor [39]. For simulating NVM storage, we use PCM (Phase-Change Memory), which is considered as a type of fast storage media in many previous studies [27], [29]. The read and write latency of PCM is set to 100 (ns) and 350 (ns), respectively, and the read and write energy of PCM is set to 0.2 (nJ/bit) and 1.0 (nJ/bit), respectively [28]. For simulating DRAM memory, the read and write latencies are equally set to 50 (ns) and the read/write energy is set to 0.1 (nJ/bit) following previous studies [26], [28]. The static power of DRAM is set to 1 (W/GB) and the default size of DRAM is set to the entire footprint of workloads in order not to incur any page faults.

Our experiments were conducted under a wide range of workload conditions. We vary the workload density from 0.1 to 0.8, where the workload density of 1.0 indicates the saturation of full CPU resources in the system. The number of real-time tasks is set to 100 and the worst-case execution time of the tasks is randomly generated between 1ms and 500 ms. The period of a task is determined based on the target workload density of 0.1 to 0.8. To assess the effectiveness of the proposed scheme in more realistic target systems, we performed additional experiments under two realistic workload conditions, Robotic Highway Safety Marker (RSM) workload [43] and IoT workload [44]. RSM is a task set for the actions of a mobile robot that carries safety markers in a highway for road construction safety. IoT is a task set for the actions of a real-time controller in an industry machine hand. Tables 2 and 3 list the task configurations of the RSM and IoT workloads, respectively.

Figure 5 depicts the energy consumption of PSVS-GA in comparison with PS-GA, VS-GA, and Baseline as the workload density is varied. The values on the *y*-axis represent the energy consumption of each scheme normalized to that of

TABLE 2. Task configurations of the IoT workload.

Task	WCET	PERIOD
Temperature sensing	10us	100ms
Data delivery to server	6ms	1min
Vibration sensing	600us	10ms
Compression & sending	7.5ms	1s
Get info. & calculating	1ms	10ms
Machine controlling	1ms	10ms
GUI updating	20ms	1s

TABLE 3. Task configurations of the RSM workload.

Task	WCET	PERIOD
Serial	100us	7.81ms
Length	1ms	7.81ms
Way Point	2.5ms	23.44ms
Encoder	350us	23.44ms
PID	1.06ms	23.44ms
Motor	250us	23.44ms

Baseline. That is, the energy consumption of Baseline is set to 1.0 and the relative value of each scheme scaled to Baseline is plotted. As shown in the figure, PSVS-GA exhibits the best results in all cases. Specifically, the energy-saving effect of PSVS-GA is large when the density of workloads is not high. The reason is that there are more chances of resource optimizations with respect to energy-saving when the load of tasks is low. We can make use of energy-saving techniques such as voltage scaling and swap more aggressively without deadline misses in such situations. On the other hand, as the density of the workload increases, idle time slots are reduced and hence power-saving techniques become less effective. For example, lowering the CPU supply voltage is difficult in these cases as it may incur deadline misses of real-time tasks.

Now, let us compare the results in detail. The reduced energy consumption of PSVS-GA is 31.1% on average and up to 45.6% compared to Baseline. When compared to PS-GA and VS-GA, the energy-saving effect of PSVS-GA is 23.1% and 14.1%, on average, respectively. When comparing the relative effect of CPU voltage scaling and memory swap, VS-GA performs better than PS-GA in most cases, implying that voltage scaling is more effective than swap in terms of energy-saving. The only case where PS-GA outperforms VS-GA occurs when the workload density is 0.1. In that case, the scheduling incurs too much empty time slot, which cannot be filled even if the CPU voltage is lowered as much as possible. This will be further discussed in Figure 8, which shows that the CPU utilization is still low despite applying CPU voltage scaling maximally. However, in any case, PSVS-GA outperforms PS-GA and VS-GA, which implies that combining the voltage scaling and swap techniques makes even better results.

Figures 6 and 7, respectively, show the energy consumption in CPU and memory for the four schemes as the workload density is varied. As shown in Figure 6, schemes using CPU voltage scaling, i.e., PSVS-GA and VS-GA, reduce the CPU energy consumption significantly compared to those that do not use it. The power-saving effect of voltage scaling is small

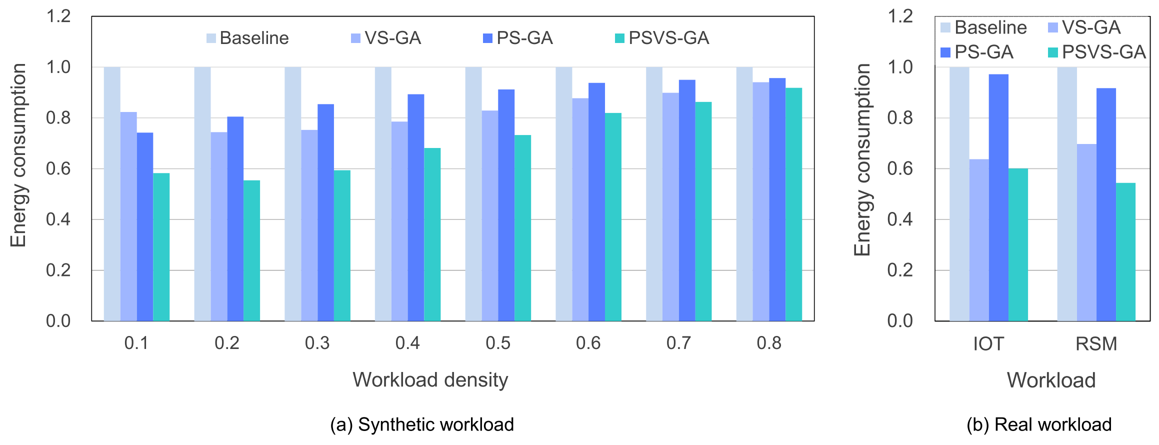


FIGURE 5. Total energy consumptions of baseline, VS-GA, PS-GA, and PSVS-GA as a function of the workload density.

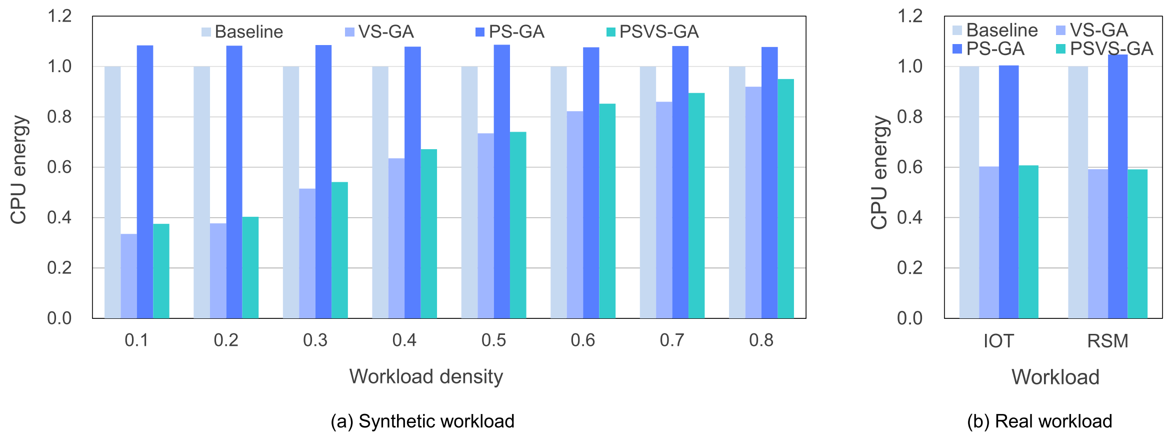


FIGURE 6. CPU energy consumptions of baseline, VS-GA, PS-GA, and PSVS-GA as a function of the workload density.

when the workload becomes heavy because the possibility of utilizing idle time slots of CPU by lowering the supply voltage is reduced. When we compare PSVS-GA and VS-GA, VS-GA performs slightly better than PSVS-GA with respect to CPU energy consumption. This is because PSVS-GA needs additional time to access swap storage, which also increases the execution time in CPU. However, as shown in the figure, the effect of swap on CPU energy consumption is very small.

Now, let us see the energy consumption in memory. As shown in Figure 7, PSVS-GA and PS-GA that support swap consume less memory energy than VS-GA and Baseline, which use memory shadowing. This is because supporting swap has the effect of reducing the DRAM capacity of the system, which can save the refresh power of DRAM significantly. Meanwhile, when using swap, additional CPU time is required for the swap-in and swap-out process, which may increase the CPU energy consumption. This is shown in Figure 6 that PS-GA spends more CPU energy than Baseline. However, such overhead is compensated by the energy-saving effect in memory as shown in Figure 7. Moreover, although PSVS-GA also adopts swap, the CPU energy is not increased compared to VS-GA as shown in Figure 6. This implies that using CPU voltage scaling along with memory swap and co-optimizing them can hide the swap overhead in CPU by overlapping the execution in CPU and I/O. That is, CPU issues I/O commands and executes other tasks while the actual I/O is performed. Another notable phenomenon is that the effectiveness of swap is less influenced by the workload density. That is, swap is still effective in power-saving even when the workload becomes heavy as shown in Figure 7, which is different from the effectiveness of CPU voltage scaling in Figure 6.

Figure 8 shows the CPU utilization of the four schemes as a function of the workload density. As we see in the figure, PSVS-GA exhibits high utilization of almost 1.0 except for the case of the workload density 0.1. Note that PSVS-GA makes use of the power-saving techniques as much as possible to maximize the resource utilization. However, as discussed previously, when the workload density is 0.1, there are too much empty time slot, which cannot be filled even with the full usage of CPU voltage scaling and memory swap. The utilization of VS-GA is also near 1.0 as we optimize it by genetic algorithms. However, the energy-saving effect of VS-GA is lower than PSVS-GA as it does not use swap. PS-GA also raises the CPU utilization compared to Baseline,

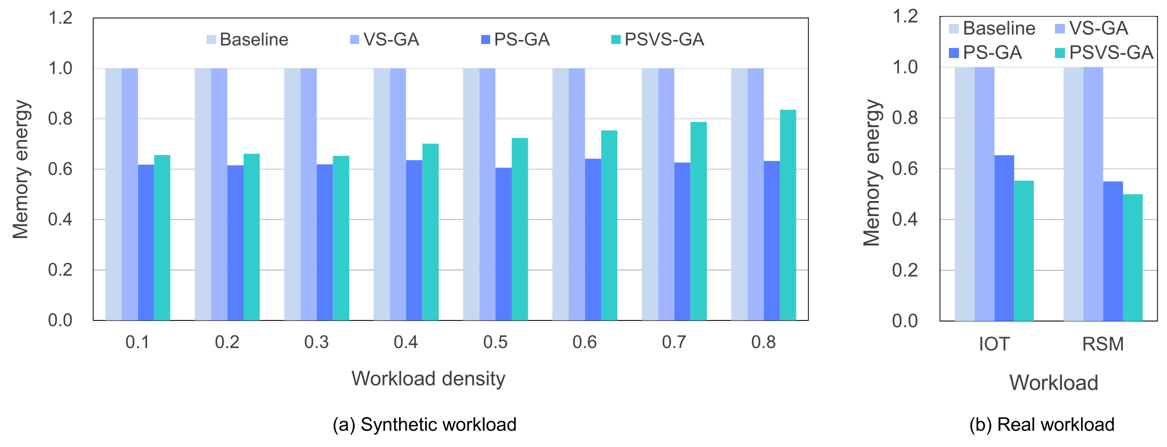


FIGURE 7. Memory energy consumptions of baseline, VS-GA, PS-GA, and PSVS-GA as a function of the workload density.

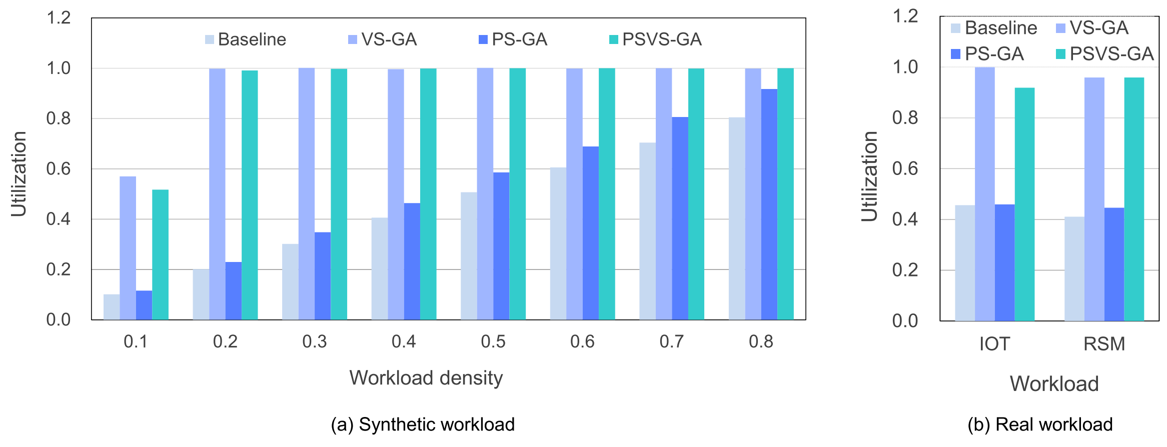


FIGURE 8. CPU utilizations of baseline, VS-GA, PS-GA, and PSVS-GA as a function of the workload density.

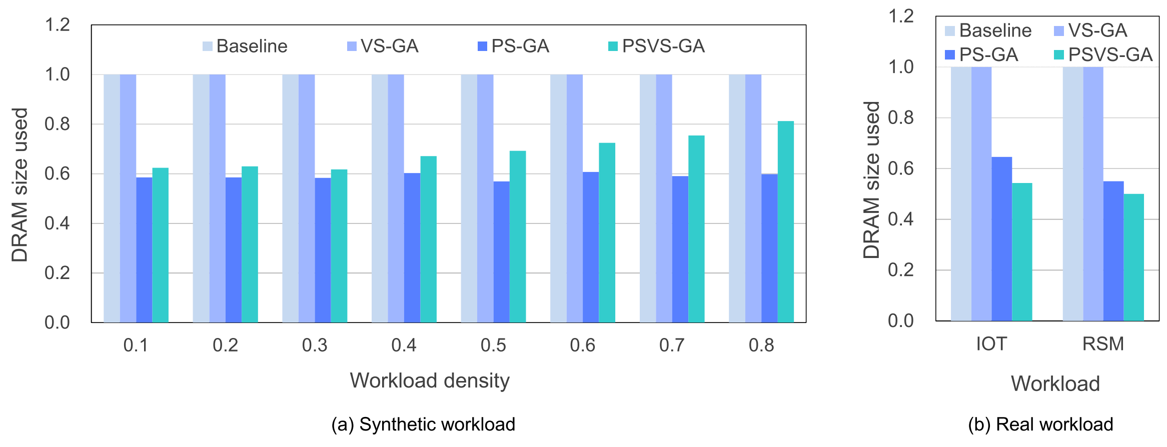


FIGURE 9. DRAM size of baseline, VS-GA, PS-GA, and PSVS-GA as a function of the workload density.

but the effect is limited as it does not make use of CPU power-saving techniques. The utilization of Baseline is consistently lower than PSVS-GA and VS-GA, and the gap becomes wider as the workload density is decreased.

Figure 9 compares the DRAM size used in the four schemes. As we see in the figure, PSVS-GA reduces the DRAM size significantly compared to VS-GA and Baseline. Specifically, the DRAM size used in PSVS-GA is smaller than these two schemes by 34.3% on average. The effect of reducing the DRAM size is large when the workload density becomes low. This is because the inactive period of a task becomes longer when the workload is not heavy, implying that the swap ratio can be further increased without deadline misses in this case. When we compare PS-GA and PSVS-GA, PS-GA reduces more DRAM capacity than PSVS-GA in synthetic workloads, especially when the workload is heavy. As our optimization goal has focused on the overall energy-saving rather than the DRAM size reduction, this implies that CPU voltage scaling is more effective than reducing the DRAM size in an energy-saving effect under heavy workloads. In real workloads, however, PSVS-GA performs better than PS-GA as shown in Figure 9(b). This implies that the density of real workloads we simulated is not high compared to the synthetic workloads.

VI. CONCLUSION

As the size of data grows rapidly in modern embedded systems, the DRAM memory of the system keeps increasing, which accounts for a large portion of the power consumption. In this article, we presented a new real-time task scheduling scheme that supports partial swap in order to reduce the DRAM size of the system. To enable swap functions, we adopted high-speed NVM storage with predictable access latency, which allows for the feasible estimation of real-time task's worst case execution time. Unlike typical real-time systems that maintain entire footprint of tasks in memory, we place a certain part of real-time tasks in NVM storage and perform swapping. The ratio of swap for each task is determined based on the schedulability and the power-saving effect.

We also combined our swap scheme with CPU voltage scaling by formulating the effect of the two techniques as a unified measure, and co-optimized the supply voltage of CPU and the swap ratio of memory for each task with respect to energy consumption. Our experimental results under a wide range of workload conditions showed that the energy-saving effect of the proposed scheme is 31.1% on average without any deadline misses.

ACKNOWLEDGMENT

(Suji Yoon and Heejin Park contributed equally to this work.)

REFERENCES

[1] P. Pillai and K. G. Shin, "Real-time dynamic voltage scaling for low-power embedded operating systems," in *Proc. SOSP*, Oct. 2001, pp. 89–102.

[2] H. E. Ghor and E. M. Aggoune, "Energy saving EDF scheduling for wireless sensors on variable voltage processors," *Int. J. Adv. Comput. Sci. Appl.*, vol. 5, no. 2, pp. 158–167, 2014.

[3] K. Choi, W. Lee, R. Soma, and M. Pedram, "Dynamic voltage and frequency scaling under a precise energy model considering variable and fixed components of the system power dissipation," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Design*, Nov. 2005, pp. 29–34.

[4] S. Nam, K. Cho, and H. Bahn, "Tight evaluation of real-time task schedulability for processor's DVS and nonvolatile memory allocation," *Micromachines*, vol. 10, no. 6, Jun. 2019, Art. no. 371.

[5] Y. Wang, M. Sheng, X. Wang, L. Wang, and J. Li, "Mobile-edge computing: Partial computation offloading using dynamic voltage scaling," *IEEE Trans. Commun.*, vol. 64, no. 10, pp. 4268–4282, Aug. 2016.

[6] A. Carroll and G. Heiser, "An analysis of power consumption in a smartphone," in *Proc. USENIX Annu. Tech. Conf.*, 2010, p. 21.

[7] J. Kim and H. Bahn, "Analysis of smartphone I/O characteristics—Toward efficient swap in a smartphone," *IEEE Access*, vol. 7, pp. 129930–129941, 2019.

[8] S. Liu, K. Pattabiraman, T. Moscibroda, and B. Zorn, "Flikker: Saving DRAM refresh-power through critical data partitioning," in *Proc. ACM ASPLOS*, 2011, pp. 213–224.

[9] S. Yoo, Y. Jo, and H. Bahn, "Integrated scheduling of real-time and interactive tasks for configurable industrial systems," *IEEE Trans. Ind. Informat.*, vol. 18, no. 1, pp. 631–641, Jan. 2022.

[10] E. Lee, H. Bahn, S. Yoo, and S. H. Noh, "Empirical study of NVM storage: An operating system's perspective and implications," in *Proc. IEEE MASCOTS Conf.*, Sep. 2014, pp. 405–410.

[11] E. Lee and H. Bahn, "Caching strategies for high-performance storage media," *ACM Trans. Storage*, vol. 10, no. 3, pp. 1–22, Jul. 2014.

[12] *Intel Optane^{TM}*. Accessed: Dec. 20, 2021. [Online]. Available: https://www.intel.com/content/www/us/en/products/details/memory-storage/consumer-ssds/optane-ssd-9-series.html

[13] D. Kim, E. Lee, S. Ahn, and H. Bahn, "Improving the storage performance of smartphones through journaling in non-volatile memory," *IEEE Trans. Consum. Electron.*, vol. 59, no. 3, pp. 556–561, Aug. 2013.

[14] Y. Park and H. Bahn, "Modeling and analysis of the page sizing problem for NVM storage in virtualized systems," *IEEE Access*, vol. 9, pp. 52839–52850, 2021.

[15] H. Bahn and K. Cho, "Evolution-based real-time job scheduling for co-optimizing processor and memory power savings," *IEEE Access*, vol. 8, pp. 152805–152819, 2020.

[16] Y.-H. Lee, Y. Doh, and C. M. Krishna, "EDF scheduling using two-mode voltage-clock-scaling for hard real-time systems," in *Proc. CASES*, 2001, pp. 221–228.

[17] H. Chetto and M. Chetto, "Some results of the earliest deadline scheduling algorithm," *IEEE Trans. Softw. Eng.*, vol. 15, no. 10, pp. 1261–1269, Oct. 1989.

[18] S. Hyun, H. Bahn, and K. Koh, "LeCramFS: An efficient compressed file system for flash-based portable consumer devices," *IEEE Trans. Consum. Electron.*, vol. 53, no. 2, pp. 481–488, May 2007.

[19] J. Park, H. Lee, S. Hyun, K. Koh, and H. Bahn, "A cost-aware page replacement algorithm for NAND flash based mobile embedded systems," in *Proc. ACM EMSOFT Conf.*, 2009, pp. 315–324.

[20] O. Kwon, K. Koh, J. Lee, and H. Bahn, "FeGC: An efficient garbage collection scheme for flash memory based storage systems," *J. Syst. Softw.*, vol. 84, no. 9, pp. 1507–1523, Sep. 2011.

[21] J.-C. Kim, D. Lee, C.-G. Lee, and K. Kim, "RT-PLRU: A new paging scheme for real-time execution of program codes on NAND flash memory for portable media players," *IEEE Trans. Comput.*, vol. 60, no. 8, pp. 1126–1141, Aug. 2011.

[22] D. Lee, J.-C. Kim, C.-G. Lee, and K. Kim, "MRT-PLRU: A general framework for real-time multitask executions on NAND flash memory," *IEEE Trans. Comput.*, vol. 62, no. 4, pp. 758–771, Apr. 2013.

[23] D. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Boston, MA, USA: Addison-Wesley, 1989.

[24] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate progress: A notion of fairness in resource allocation," *Algorithmica*, vol. 15, no. 6, pp. 600–625, 1996.

[25] D. Whitley and J. Kauth, "GENITOR: A different genetic algorithm," in *Proc. Rocky Mountain Conf. Artif. Intell.*, 1988, pp. 118–130.

[26] E. Lee, J. E. Jang, T. Kim, and H. Bahn, "On-demand snapshot: An efficient versioning file system for phase-change memory," *IEEE Trans. Knowl. Data Eng.*, vol. 25, no. 12, pp. 2841–2853, Dec. 2013.

[27] S. Eilert, M. Leinwander, and G. Crisenza, "Phase change memory: A new memory technology to enable new memory usage models," Micron, Boise, ID, USA, White Paper 06/23/11 EN.L, 2011, pp. 1–4.

[28] S. Lee, H. Bahn, and S. H. Noh, "CLOCK-DWF: A write-history-aware page replacement algorithm for hybrid PCM and DRAM memory architectures," *IEEE Trans. Comput.*, vol. 63, no. 9, pp. 2187–2200, Sep. 2014.

[29] E. Lee, S. H. Yoo, and H. Bahn, "Design and implementation of a journaling file system for phase-change memory," *IEEE Trans. Comput.*, vol. 64, no. 5, pp. 1349–1360, May 2015.

[30] J. H. Anderson and A. Srinivasan, "Mixed Pfair/ERfair scheduling of asynchronous periodic tasks," *J. Comput. Syst. Sci.*, vol. 68, no. 1, pp. 157–204, Feb. 2004.

[31] J. H. Anderson and A. Srinivasan, "Pfair scheduling: Beyond periodic task systems," in *Proc. IEEE RTCSA*, Dec. 2000, pp. 297–306.

[32] H. Chen, X. Zhu, G. Liu, and W. Pedrycz, "Uncertainty-aware online scheduling for real-time workflows in cloud service environment," *IEEE Trans. Services Comput.*, vol. 14, no. 4, pp. 1167–1178, Jul. 2021.

[33] S. Dehnavi, H. R. Faragardi, M. Kargahi, and T. Fahringer, "A reliability-aware resource provisioning scheme for real-time industrial applications in a fog-integrated smart factory," *Microprocessors Microsyst.*, vol. 70, pp. 1–14, Oct. 2019.

[34] L. Zhou, L. Zhang, L. Ren, and J. Wang, "Real-time scheduling of cloud manufacturing services based on dynamic data-driven simulation," *IEEE Trans. Ind. Informat.*, vol. 15, no. 9, pp. 5042–5051, Sep. 2019.

[35] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 9th ed. Hoboken, NJ, USA: Wiley, 2014.

[36] Transmeta Crusoe. *Operating Modes for New Generation Processors*. Accessed: Dec. 20, 2021. [Online]. Available: https://en.wikipedia.org/wiki/Transmeta_Crusoe

[37] D. Zhu, D. Mosse, and R. Melhem, "Power-aware scheduling for AND/OR graphs in real-time systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 9, pp. 849–864, Sep. 2004.

[38] J. Zhou, T. Wei, M. Chen, J. Yan, X. S. Hu, and Y. Ma, "Thermal-aware task scheduling for energy minimization in heterogeneous real-time MPSoC systems," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 8, pp. 1269–1282, Aug. 2016.

[39] *ARM Cortex-R52 Processor Model*. Accessed: Dec. 20, 2021. [Online]. Available: https://developer.arm.com/ip-products/processors/cortex-r/cortex-r52

[40] R. Salkhordeh and H. Asadi, "An operating system level data migration scheme in hybrid DRAM-NVM memory architecture," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, 2016, pp. 936–941.

[41] J. Zhan, Y. Zhang, W. Jiang, J. Yang, L. Li, and Y. Li, "Energy-aware page replacement and consistency guarantee for hybrid NVM–DRAM memory systems," *J. Syst. Archit.*, vol. 89, pp. 60–72, Sep. 2018.

[42] *PSVS-GA Simulator*. [Online]. Available: https://github.com/oslab-ewha/PSVS-GA

[43] A. Qadi, S. Goddard, and S. Farritor, "A dynamic voltage scaling algorithm for sporadic tasks," in *Proc. 24th IEEE Real-Time Syst. Symp. (RTSS)*, Dec. 2003, pp. 52–62.

[44] Z. Wang, Y. Liu, Y. Sun, Y. Li, D. Zhang, and H. Yang, "An energy-efficient heterogeneous dual-core processor for Internet of Things," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2015, pp. 2301–2304.

[45] A. Mahesri and V. Vardhan, "Power consumption breakdown on a modern laptop," in *Proc. Int. Workshop Power-Aware Comput. Syst.*, in Lecture Notes in Computer Science, vol. 347, 2005, pp. 165–180.

SUJI YOON is currently a senior student with the Department of Computer Science and Engineering, Ewha Womans University, Seoul, Republic of Korea. Her research interests include operating systems, storage systems, embedded systems, systems optimizations, process scheduling, and real-time systems.

HEEJIN PARK is currently a senior student with the Department of Computer Science and Engineering, Ewha Womans University, Seoul, Republic of Korea. Her research interests include operating systems, storage systems, embedded systems, systems optimizations, real-time systems, and process scheduling.

KYUNGWOON CHO received the B.S., M.S., and Ph.D. degrees in computer science and engineering from Seoul National University, in 1995, 1997, and 2012, respectively. He is currently a Senior Researcher at the Embedded Software Research Center, Ewha Womans University, Seoul, Republic of Korea. Before joining Ewha, he was a Chief Officer at the Clunix Research and Development Center, Seoul. His research interests include multimedia systems, cloud computing, real-time systems, embedded systems, and operating systems.

HYOKYUNG BAHN (Member, IEEE) received the B.S., M.S., and Ph.D. degrees in computer science and engineering from Seoul National University, in 1997, 1999, and 2002, respectively. He is currently a Full Professor of computer science and engineering at Ewha Womans University, Seoul, Republic of Korea. He has published more than 100 papers in leading conferences and journals, including USENIX FAST, IEEE TRANSACTIONS ON COMPUTERS, IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS, and *ACM Transactions on Storage*. His research interests include operating systems, caching algorithms, storage systems, embedded systems, systems optimizations, and real-time systems. He also received the Best Paper Awards at the USENIX Conference on File and Storage Technologies, in 2013.