J Sign Process Syst (2016) 84:69–89
DOI 10.1007/s11265-015-0995-3

Energy Efficient Real-Time Task Scheduling for Embedded Systems with Hybrid Main Memory

Zhiyong Zhang[1] · Zhiping Jia[1] · Peng Liu[1] · Lei Ju[1]

Received: 12 October 2014 / Revised: 6 February 2015 / Accepted: 13 March 2015 / Published online: 8 April 2015

Abstract Available energy becomes a critical design issue for the increasingly complex real-time embedded systems. Phase Change Memory (PCM), with high density and low idle power, has recently been extensively studied as a promising alternative of DRAM. Hybrid PCM-DRAM main memory architecture has been proposed to leverage the low power of PCM and high speed of DRAM. In this paper, we propose energy-aware real-time task scheduling strategies for hybrid PCM-DRAM based embedded systems. Given the execution time variation when a task is loaded into PCM or DRAM, we re-design the static table-driven scheduling for a set of fixed tasks, as well as the Rate-Monotonic (RM) and Earliest Deadline First (EDF) scheduling policies for periodic task sets. Furthermore, since the actual execution time can be much shorter than the worst-case execution time in the actual execution, we propose online schedulers which migrates the tasks between PCM and DRAM to optimize the energy consumption by utilizing the slack time resulted from the completed tasks. All the proposed algorithms minimize the number of task migrations from PCM to DRAM by ensuring that aperiodic tasks are not migrated while each periodic task instance can be migrated at most once. Experimental results show our proposed scheduling algorithms satisfy the real-time constraints and significantly reduce the energy consumption.

Keywords Hybrid main memory · Energy · PCM · Real-time task scheduling

1 Introduction

An embedded system is a computer system with dedicated functions to specific applications. In recent years, there has been a rapid and wide spread of embedded devices, especially mobile and smart devices. As functions and applications become increasingly sophisticated, the battery life is the most serious limitation on these devices [14]. In modern systems, main memory becomes the major energy consumer to the total energy [3, 19, 24]. Therefore, reducing the energy consumption of main memory is an effective way to prolong the available battery life.

Phase-change memory (PCM) has been intensively studied as a promising candidate main memory. Comparing with dynamic random access memory (DRAM), which has been widely used as the main memory for decades, PCM needs no refresh energy and consumes much lower leakage energy. In addition, PCM provides DRAM-like byte-addressable access and has the characteristics of non-volatility, low power, better scalability and higher density. As such, it has been regarded as potential alternative to replace DRAM to build main memory for energy optimization.

✉ Lei Ju
julei@sdu.edu.cn

Zhiyong Zhang
zhangzhiyongschool@163.com

Zhiping Jia
jzp@sdu.edu.cn

Peng Liu
zationlue@mail.sdu.edu.cn

[1] School of Computer Science and Technology, Shandong University, Jinan, China

Unfortunately, directly replace DRAM with PCM as main memory encounters the challenge of limited lifetime of PCM. For example, state-of-the-art process technology has demonstrated that the maximum writes number of PCM is around 10^8 to 10^9. Besides, access latency of PCM is longer than DRAM which significantly impedes the high performance of main memory. Hybrid PCM-DRAM main memory, as an ideal architecture, has been proposed to utilize the respective strengths of DRAM and PCM, namely to achieve high performance using DRAM (shorter read/write latency) and to save energy using PCM (less energy consumption).

However, the hybrid main memory system complicates the real-time task scheduling: as an intelligent device, it should provide high performance which consumes more energy, but as an embedded system, it should maximize the battery life which prolongs the execution time, violating the real-time constraints. Therefore, the tradeoff between energy conservation and time overhead is a critical issue to be addressed.

Our goal is to propose a task schedule and allocation scheme for the hybrid main memory and thereby to utmostly save energy consumption while guaranteeing the real-time task constraints. We consider this problem for both aperiodic tasks and periodic tasks. We propose static table-driven scheduling for for a set of fixed tasks, as well as the Rate-Monotonic (RM) and Earliest Deadline First (EDF) scheduling policies for periodic task sets. Furthermore, we propose online schedulers which migrates the tasks between PCM and DRAM to optimize the energy consumption by utilizing the slack time resulted from the completed tasks. The main contributions of this paper are as follows:

1) We propose static table-driven scheduling for a set of fixed tasks (aperiodic tasks). Meanwhile, for periodic tasks, we propose static scheduling algorithms under the Rate-Monotonic (RM) and Earliest Deadline First (EDF) scheduling policies. The static schedulers minimize the energy consumption while reducing the number of writes in PCM for the hybrid main memory system.
2) As the actual execution time is usually much smaller than the Worst-case Execution Time ($WCET$), we propose online schedulers (dynamic schedulers) which migrates the tasks between PCM and DRAM to optimize the energy consumption of static table-driven schedulers by utilizing the slack time resulted from the completed tasks.
3) All parameters used in the evaluations are got from the hybrid main memory platform we designed. The scheduling algorithms minimize the number of task migrations between PCM and DRAM. The mechanisms of our algorithms ensure any aperiodic task will not be migrated while each periodic task instance can be migrated at most once.

The remainder of this paper is organized as follows. Section 2 describes the background of PCM and DRAM, real-time issues and problem description of our strategies. For a given aperiodic task set, Section 3 details our static and dynamic aperiodic scheduling algorithms. For a periodic task set, Section 4 explains our static scheduling algorithms based on the RM and EDF policies. Section 5 presents the dynamic scheduling mechanisms to optimize the results of static solutions. Experimental results and correlation analysis are presented in Section 6. Section 7 describes the previous work and finally, this paper is concluded in Section 8.

2 Problem Analysis

In this section, we compare the characteristics of PCM and DRAM, which is based on to present our energy model. Then, we introduce the real-time issues. Finally, we give the problem assumptions and describe the problem to be addressed in this paper.

2.1 Comparisons of PCM and DRAM

PCM is superior to DRAM in terms of nonvolatile characteristics and energy consumption. DRAM stores a bit in the form of charge in a small capacitor which loses the data when power is off, while PCM stores the data in a special phase change material which can maintain the data without power supply. Since the charge can get exhausted over time, DRAM needs to be refreshed to sustain its data. Different from DRAM, PCM needs no refresh energy and consumes much lower leakage energy.

In the aspects of access latency and endurance, DRAM shows its advantages over PCM. A PCM cell has longer read/write latency than that of a DRAM cell. Additionally, PCM has limited write numbers which significantly limits its lifetime.

Considering all the factors, PCM and DRAM have their respective advantages and disadvantages. In this paper, we employ a hybrid PCM-DRAM main memory architecture which enables exploitation of positive aspects of the respective memories to achieve higher energy efficiency and performance.

The main memory architecture used in this paper is shown in Fig. 1. In our architecture, PCM and DRAM are addressed consecutively, which makes both PCM and DRAM can be accessed directly by CPU. Operating sys-

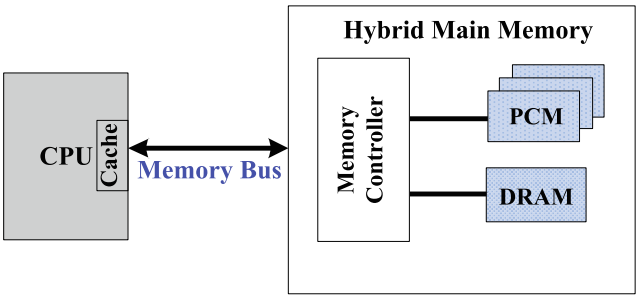


Figure 1 The hybrid main memory architecture.

tem distinguishes the address space and determines which part to communicate with. Meanwhile, effective scheduling algorithms will aid the OS in making the determination to guarantee the hybrid main memory's advantage of low energy consumption and short access latency.

2.2 Energy Model

In order to measure the parameters of hybrid main memory in real environment, we have built a testbed of the proposed hybrid PCM-DRAM memory system by Microns P8P PCM chips connected to Xilinx ZC702 board via FMC interface, which is shown in Fig. 2. The PCM controller is implemented with FPGA. We connected the YOKOGAWA WT310 digital power meter to measure the (average) working current of the memory system. Other electrical characteristics (e.g., leakage current of DRAM) are obtained from the corresponding data sheets of the chips.

The energy consumption of DRAM can be broken into activation/precharge, refresh, leakage and read/write energy. In order to accurately calculate the energy, we measured DRAM's average current of various operations. The read/write current we got is about 200mA and the refresh current is about 300mA. It is worth noting that the leakage current (about 4μA according to the data sheets) is less than 0.01 % of the DRAM working current (including the read/write and the refresh current). As such, the power consumption of the leakage has little effect on our evaluation results. Therefore, we didn't take the leakage current into account in our experiment, similarly as in the previous work of [12, 16, 20]. We calculate the DRAM energy according to the following equation:

$$E_{DRAM} = U_D \cdot I_{rw} \cdot T_{rw} + U_D \cdot I_{refresh} \cdot T_{refresh} \quad (1)$$

where, I_{rw} and $I_{refresh}$ are the read/write and refresh current respectively. Similarly, T_{rw} and $T_{refresh}$ are the working and refresh time. U_D is DRAM's working voltage which is set to 1.8V (DDR2).

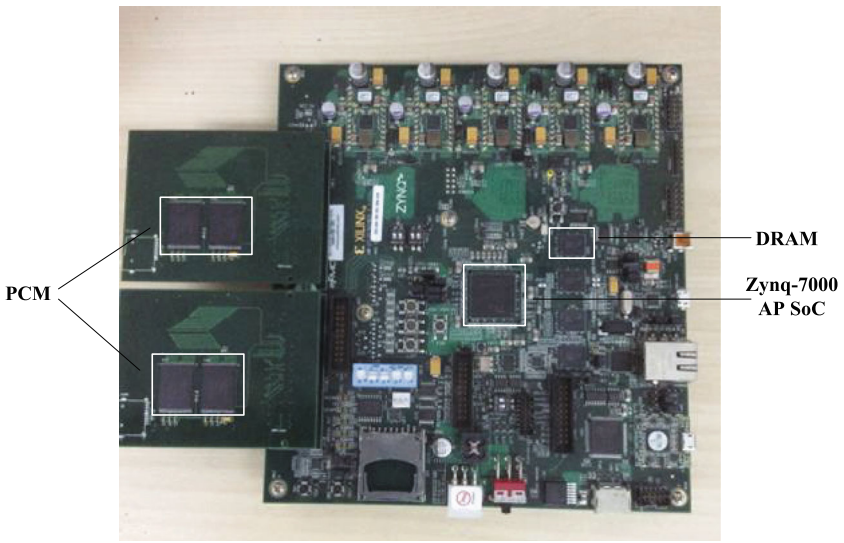
In our hybrid main memory architecture, when tasks are not executed in DRAM, we set DRAM to standby state during which DRAM deactives its clocking circuitry and stops all operations except self-refresh to save power. The standby current is about 30mA.

For PCM, the energy calculation is relatively simple. Only read and write energy is considered. According to the measurement results, we unify the measured read/write current to 40mA which is the upper bound of the working current. We calculate the PCM energy in this manner:

$$E_{PCM} = U_P \cdot I_{rw} \cdot T_{rw} \quad (2)$$

where, I_{rw} and T_{rw} are the working current and working time, respectively. U_P is PCM's working voltage which is set to 2.7V.

Figure 2 The designed hybrid main memory test-board.



2.3 Real-time Issues

In real-time embedded systems, tasks must be completed by some specified deadlines. During the last decades, many studies have been conducted on the problem and a host of algorithms have been proposed [8, 10, 11], among which the two most-studied real-time scheduling schemes are Rate Monotonic (RM) scheduler and Earliest Deadline First (EDF) scheduler [11].

RM is a static priority scheduling algorithm which always selects task with the shortest period to execute. For a specific task set, the priority of each task will not change once assigned. Unlike RM scheduler, EDF is a dynamic priority scheduling algorithm. Whenever a scheduling event occurs, the EDF scheduler chooses the task closest to its deadline to execute.

It goes without saying that the hybrid PCM-DRAM main memory system makes the real-time task scheduling more complicated since tasks have different execution time in PCM and DRAM. In hybrid main memory system, allocating a task to PCM brings energy saving but increases the time overhead resulting in potential violation of the real-time constraints which can be met in DRAM originally. Therefore, the scheduling algorithms in hybrid main memory must guarantee that if a task set is schedulable in pure DRAM main memory system, it remains schedulable in hybrid PCM-DRAM memory system.

2.4 Problem Description

In the classic model of real-time systems, both RM and EDF scheduling algorithms assume a task must complete before its next invocation and all periodic tasks are ready at the same time. In our schemes, we maintain the same assumptions. Moreover, the tasks in this paper are independent and don't have any un-preemption parts. For simplicity, we assume the overheads of scheduling and preemptions are negligible.

For a periodic task set $T = \{T_1, T_2, \cdots, T_n\}$, we characterize each task T_i by a five-tuple $< W_{di}, W_{pi}, P_i, N_{wi}, S_i >$, where W_{di} and W_{pi} are the task's worst-case execution time in DRAM and PCM, P_i is the associated task period, N_{wi} represents the number of writes and S_i denotes the migration size. It is worth noting that when task T_i needs to be migrated from PCM to DRAM, S_i is the maximum data size needed to be migrated, including code segment, data segment and stack segment, etc. In this paper, we assume that all tasks needn't interact with users, which is reasonable for real-time embedded systems. Furthermore, in our real hybrid main memory system, the migration rate between PCM and DRAM we measured is 380Mb/s which can be used to compute the migration time of each task. Since the actual migration data size is much less than S_i, the use of S_i can estimate the most conservative performance of our scheduling algorithms.

Similarly, for an aperiodic task set $T = \{T_1, T_2, \cdots, T_n\}$, we also characterize each task T_i by a five-tuple $< W_{di}, W_{pi}, D_i, N_{wi}, S_i >$, where D_i is the deadline of task T_i.

Our proposed algorithms allocate each task (or task instance) to DRAM, PCM or part in PCM and part in DRAM. The scheduling sequence generated by our algorithms minimizes the energy consumption and reduces the number of writes in PCM while guaranteeing the real-time constraints of all tasks.

3 Aperiodic Scheduling Solutions

In embedded systems, aperiodic task set is executed only once, so the scheduler just provides service according to the best-effort principle. In this section, we perform our algorithms by the following rule: the earlier the deadline, the higher the priority. For aperiodic task set, this approach has been proven to be optimal and our proposed aperiodic scheduling algorithms are based on this.

3.1 Static-Aperiodic Scheduling Algorithm

In static scheduling algorithm, we assume each task requires its $WCET$. Apart from all the $WCET$, if there is still free time, the scheduler can exploit it for energy efficiency. To

Table 1 Notations used in aperiodic scheduling algorithms.

Tasks	T_1	T_2	$\cdots$	T_i	$\cdots$	T_n
WCET	C_1	C_2	$\cdots$	C_i	$\cdots$	C_n
cumulative_time	C_1	$C_1 + C_2$	$\cdots$	$\sum_{k=1}^{i} C_k$	$\cdots$	$\sum_{k=1}^{n} C_k$
deadline	D_1	D_2	$\cdots$	D_i	$\cdots$	D_n
elastic_time	$D_1 - C_1$	$D_2 - (C_1 + C_2)$	$\cdots$	$D_i - \sum_{k=1}^{i} C_k$	$\cdots$	$D_n - \sum_{k=1}^{n} C_k$

Table 2 Example aperiodic task set.

Tasks	T_1	T_2	T_3
WCET	50	20	30
cumulative_time	50	70	100
deadline	100	110	120
elastic_time	50	40	20

this aim, we sort the task set by the ascending order of deadline. If $i < j$, T_i has higher priority than T_j. we introduce $cumulative\_time_i$ to represent all the needed worst case execution time of higher priority tasks than T_i (including task T_i), which is calculated by the following equation:

$$cumulative\_time_i = \sum_{k=1}^{i} C_k \tag{3}$$

where, C_k is the worst case execution time of task T_k.

Besides, we introduce $elastic\_time_i$ to represent the cumulative free time of higher priority tasks than T_i (including task T_i), which is calculated by the following equation:

$$elastic\_time_i = D_i - \sum_{k=1}^{i} C_k \tag{4}$$

where, D_i is the deadline of task T_i. All the *elastic_time* cannot be less than 0, or the task set is not schedulable. Meanwhile, $elastic\_time_i$ is the maximum available free time from time 0 to time $\sum_{k=1}^{i} C_k$.

The relationship of these concepts is shown in Table 1.

However, $elastic\_time_i$ doesn't mean all the free time can be used by task T_i for its potentially execution in PCM. Blindly using the free time will affect the execution of lower priority tasks, causing them to miss the deadlines. Therefore, the elastic time isn't the real *free time can be allocated*. To understand this, consider an aperiodic task set containing three tasks with following parameters:

$W_{d1} = 50,\ W_{p1} = 80,\ D_1 = 100;$

$W_{d2} = 20,\ D_2 = 110;$

$W_{d3} = 30,\ D_3 = 120.$

Table 3 Allocating T_1 to PCM.

Tasks	T_1	T_2	T_3
WCET	80	20	30
cumulative_time	80	100	130
deadline	100	110	120
elastic_time	20	10	-10

Table 4 Example aperiodic task set.

Tasks	T_1	T_2	T_3
WCET	50	20	30
cumulative_time	50	70	100
deadline	100	110	120
elastic_time	50	40	20
revised_elastic_time	20	20	20

All tasks are initialized as D-task, namly executed in DRAM, and the task set has been sorted by the ascending order of deadline. The corresponding values are shown in Table 2.

According to the value of $elastic\_time_1$, the free time is sufficient to execute the whole T_1 in PCM. Once we allocate T_1 to PCM, the situation is shown in Table 3. Obviously, this allocation affects the execution of lower priority tasks and leads to $elastic\_time_3$ being less than 0 which means T_3 cannot complete before its deadline.

Fortunately, for an aperiodic task set, this question isn't hard to address. We introduce another concept, called *revised_elastic_time*, to revise the elastic time to be *safe available free time*. Utilizing *revised_elastic_time*, the scheduler can schedule tasks safely without worrying about whether the allocation will violate the real-time constrains of other tasks.

In order to calculate the *revised_elastic_time*, we just need to reverse scan the sorted task set. For a higher priority task, if its elastic time is bigger than the adjacent lower priority task, we set the elastic time to the value of lower priority task, or it remains unchanged. For the above example task set, all the values used in aperiodic scheduling algorithms is shown in Table 4.

From the last row of Table 4, we can see that task T_1 can be allocated at most 20 additional time units while guaranteeing all tasks' real-time constraints.

We are now ready to present our static-aperiodic scheduling algorithm. First, static-aperiodic algorithm initially allocates all tasks to DRAM and calculates *cumulative_time*, *elastic_time* and *revised_elastic_time*. If a task has a larger *revised_elastic_time* than the additional time when executed in PCM rather than DRAM, it is inserted to task queue *TQ*. Therefore, *TQ* contains tasks which MAY be arranged to execute in PCM. To the aim to save energy as much as possible with the least writes to PCM, we adopt the metric of $(W_{pi} - W_{di})/N_{wi}$ to evaluate the comprehensive gains of tasks in *TQ* when allocated to PCM. Once a task with the most gains is allocated to PCM, *cumulative_time* and *elastic_time* of lower priority tasks will be updated. Also, the *revised_elastic_time* of all tasks need to be recalculated. Then we update the elements in *TQ*. This process is repeated until the task queue *TQ* is empty. The

static-aperiodic scheduling algorithm is shown in Algorithm 1.

Algorithm 1 Static-Aperiodic Scheduling Algorithm

Input: Aperiodic Task set $T = \{T_1, T_2, \cdots, T_n\}$, each task T_i is characterized by $< W_{di}, W_{pi}, D_i, N_{wi}, S_i >$
Output: The scheduling and allocation scheme for task set T

```
Sort the task set T according to the ascending order of D_i;
Initialize all tasks to D-tasks, set computing time C_i = W_di;
Calculate each task's cumulative_time and elastic_time, then
    reverse scan the sorted task set T to get revised_elastic_time;
For each task, calculate ΔC_i = W_pi − W_di and insert
    tasks with ΔC_i ≤ revised_elastic_time_i to task queue TQ;
Sort the task queue TQ according to the descending
    order of (W_pi − W_di)/N_wi;
while TQ is not empty do
    Remove the first task T_i from TQ;
    C_i = W_pi;
    Label the task T_i as P-task;
    for each T_j ∈ {T_1, T_2, ··· , T_n | i ≤ j} do
        cumulative_time_j = cumulative_time_j + ΔC_i;
        elastic_time_j = elastic_time_j − ΔC_i;
    end for
    Reverse scan the task set T to recalculate
    revised_elastic_time of all tasks;
    for each task T_j in task queue TQ do
        if ΔC_j > revised_elastic_time_j then
            TQ = TQ − {T_j};
        end if
    end for
end while
Schedule the task set according to the order in task
    set T while the P-tasks are executed in PCM and
    the D-tasks in DRAM
```

3.2 Dynamic-Aperiodic Scheduling Algorithm

In static algorithm, we reserve the worst case execution time for each task. In fact, when a task completes ahead of its $WCET$, the unused time, i.e. *slack time*, is produced. This part of time is generally considerable for the reason that the actual execution time is usually much smaller than the worst case execution time. In this section, in order to optimize the static solution, our dynamic aperiodic scheduling algorithm utilizes this part of time to try to allocate more tasks to PCM for energy efficiency.

The dynamic-aperiodic scheduling algorithm is based on the static algorithm. Compared with static-aperiodic algorithm, the dynamic solution introduces two extra counters, *ACET* and *slack_time*, to record a task's actual case execution time and the early completion, respectively. When a task is being executed, its *ACET* increases over time. When it completes, we can calculate the slack time produced by the early completion by the following equation:

$$slack\_time_i = C_i - ACET_i \tag{5}$$

where, C_i is the computing time of task T_i, if T_i a P-task, $C_i = W_{pi}$, or $C_i = W_{di}$.

Since a higher priority task completes ahead of its $WCET$, the lower priority tasks gain less *cumulative_time* and more *elastic_time*, and then get more *revised_elastic_time*. Additional *revised_elastic_time* means more tasks can be allocated to PCM. Similarly, we employ the same approach as static-aperiodic algorithm to examine and allocate tasks to PCM. The dynamic-aperiodic scheduling algorithm is detailed in Algorithm 2.

Algorithm 2 Dynamic-Aperiodic Scheduling Algorithm

Input: Aperiodic Task set $T = \{T_1, T_2, \cdots, T_n\}$, each task T_i is characterized by $< W_{di}, W_{pi}, D_i, N_{wi}, S_i >$
Output: The scheduling and allocation scheme for task set T

```
Execute the static-aperiodic algorithm and initialize all the
    ACET and slack_time to 0;
Upon task T_i completion:
    set slack_time_i = C_i − ACET_i;
    remove T_i from task set T;
    for each task T_j in task set T do
        cumulative_time_j −= slack_time_i;
        elastic_time_j += slack_time_i;
        revised_elastic_time_j += slack_time_i;
    end for
    For each D-task in T, calculate ΔC_i and insert tasks
        with ΔC_i ≤ revised_elastic_time_i to queue TQ;
    Sort the task queue TQ according to the descending
        order of (W_pi − W_di)/N_wi;
    According to the rules in static-aperiodic algorithm,
        examine tasks in TQ check whether they can be
        converted to P-tasks until TQ is empty.
When task T_i is executing:
    increase the ACET_i over time;
```

Actually, when a task completes, although the slack time is insufficient to allocate a whole task to PCM, the scheduler can also allocate it to PCM to execute a while and migrate it back to DRAM when necessary to keep its real-time constraint. However, for an aperiodic task set, we have a clear understanding of the entire task set, so we focus on the whole task set rather than a specific task. Meanwhile, task migrations lead to extra time, control, and energy overheads. Therefore, our dynamic algorithm schedules the task set from a holistic perspective while eliminating the overheads of task migrations.

4 Static Periodic Scheduling Solutions

4.1 Static Scheduling Algorithms for Periodic Tasks

In our hybrid main memory architecture, energy and performance are conflict issues. The new scheduling algorithms should make the greatest efforts to save energy, while guaranteeing all tasks will always receive enough time to

complete each invocation in time. To provide such guarantees, the new algorithms have to satisfy some conditions, for a periodic task set, often expressed in the form of schedulability tests [14].

We employ the well-known schedulability tests for EDF and RM algorithms to generate our static assignment schemes. For EDF algorithms, the necessary and sufficient schedulability condition requires the total CPU utilization no more than 100 % [11]. But for the RM policy, the case is slightly more complicated.

For RM scheduler, the response time of a specific task is the time span between the request and the end of the response to that request. When a task is requested simultaneously with all higher priority tasks, the task will have the largest response time and the instant is called critical instant. According to [11], if all tasks are fulfilled at their critical instants without violating respective real-time constraints, the scheduling algorithm is feasible. In this paper, we adopt this sufficient condition to test the schedulability under the RM algorithm.

We propose simple mechanisms for energy efficiency while guaranteeing the real-time constraints. The algorithms used to schedule a specific task set not only consider the energy saving but also take the number of writes in PCM into account. We allocate different tasks to different memory mediums. In our static scheduling solutions, once a task is allocated to PCM or DRAM, the allocation scheme will not change for a given task set.

Algorithm 3 Static-RM Scheduling Algorithm

Input: Periodic Task set $T = \{T_1, T_2, \cdots, T_n\}$, each task T_i is characterized by $< W_{di}, W_{pi}, P_i, N_{wi}, S_i >$
Output: The scheduling and allocation scheme for task set T

Sort the task set T according to the descending order of $(W_{pi} - W_{di})/N_{wi}$;
Initialize each task T_i's computing time $C_i = W_{di}$;
while T is not empty **do**
  Find the first task T_i from T;
  $C_i = W_{pi}$;
  if $\forall T_j \in \{T_1, T_2, \cdots, T_n | i \leq j\}$
  $\lceil P_j/P_1 \rceil \cdot C_1 + \cdots + \lceil P_j/P_j \rceil \cdot C_j \leq P_j$ **then**
    Label the task T_i as P-task;
  else
    Label the task T_i as D-task;
    $C_i = W_{di}$;
  end if
  $T = T - \{T_i\}$;
end while
Schedule the task set according to the RM algorithm while the P-tasks are executed in PCM and the D-tasks in DRAM

We propose two static scheduling algorithms, termed as static-RM and static-EDF, under RM and EDF policies respectively, in which we allocate tasks one by one to PCM according to the descending order of $(W_{pi} - W_{di})/N_{wi}$ to check whether the task set remains schedulable. If all the constraints remain unaffected, we allocate the task to PCM and label the task as a P-task, or the task is a D-task. The static scheduling algorithms repeat the procedure until all tasks are examined. It is worth noting that, for the RM policy, allocating the lower priority tasks to PCM won't result in the violation of the real-time constraints of higher priority tasks. So when we check whether T_i is a P-task, we just need to guarantee the schedulability of tasks with longer period than P_i, as shown in Algorithm 3. The static-EDF algorithm is presented in Algorithm 4.

Algorithm 4 Static-EDF Scheduling Algorithm

Input: Periodic Task set $T = \{T_1, T_2, \cdots, T_n\}$, each task T_i is characterized by $< W_{di}, W_{pi}, P_i, N_{wi}, S_i >$
Output: The scheduling and allocation scheme for task set T

Sort the task set T according to the descending order of $(W_{pi} - W_{di})/N_{wi}$;
Initialize each task T_i's computing time $C_i = W_{di}$;
while T is not empty **do**
  Find the first task T_i from T;
  $C_i = W_{pi}$;
  if $\sum_{i=1}^{n} C_i/P_i \leq 1$ **then**
    Label the task T_i as P-task;
  else
    Label the task T_i as D-task;
    $C_i = W_{di}$;
  end if
  $T = T - \{T_i\}$;
end while
Schedule the task set according to the EDF algorithm while the P-tasks are executed in PCM and the D-tasks in DRAM

4.2 An Example of Static-EDF Algorithm

As an example of static-EDF algorithm, we consider the task set in Table 5. All tasks are ready at time 0. In order to guarantee all tasks meet the real-time constrains, we assume tasks always consume their $WCET$. For the task set in pure DRAM main memory architecture, the total CPU

Table 5 Example task set.

Task	$WCET$ in DRAM	$WCET$ in PCM	Period	Number of writes
T_1	100	120	350	5
T_2	100	150	400	10
T_3	150	200	550	15

utilization is $100/350+100/400+150/550 = 0.808$. Obviously, the task set passes the schedulability test for EDF scheduler.

First, we sort the the task set according to the descending order of $(W_{pi}-W_{di})/N_{wi}$. For the task set, the result of T_1 is $(120-100)/5 = 4$, T_2 is 5, and T_3 is 3.3. So we allocate T_2 to PCM and check whether the task set remains schedulable. Allocating T_2 to PCM means T_2 consumes 150 time units. Now the CPU utilization is $100/350+150/400+150/550 = 0.933 < 100\ \%$. We can see that allocating T_2 to PCM will not violate the schedulability, so we label T_2 as a P-task.

Secondly, we allocate T_1 to PCM. Similarly, we calculate the CPU utilization: $120/350 + 150/400 + 150/550 = 0.99 < 100\ \%$ and label T_1 as a P-task.

Finally, we examine the last task T_3. If we allocate T_3 to PCM, the CPU utilization is $120/350 + 150/400 + 200/550 = 1.08 > 100\ \%$ and the task set is no longer schedulable, which means T_3 is a D-task and cannot be executed in PCM.

The schematic diagram of EDF and our static-EDF scheduler is shown in Fig. 3a and b, respectively. Obviously, our static scheduling algorithms do not violate the schedulability of a given task set, i.e. if a task set is schedulable in pure DRAM main memory system, it remains schedulable in hybrid PCM-DRAM memory system.

5 Dynamic Periodic Scheduling Solutions

In the real scheduling environment, the actual execution time is usually much smaller than the worst-case execution time. In order to guarantee all tasks meet the real-time constrains, we conservatively reserve the worst-case execution time for all tasks. Consequently, much unused execution time, namely slack time, is produced as tasks execute, which can be allocated to other unexecuted tasks for energy saving. We present two dynamic scheduling algorithms, dynamic-RM and dynamic-EDF, to dynamically reclaim the slack time to enable the ready tasks executed partially in low-power main memory, i.e. PCM.

Additionally, the reclaimed time may be not enough for allocating a whole D-task to PCM, which brings the troublesome migration issue from PCM to DRAM to maintain the real-time constrains. Task migrations lead to extra time, control, and energy overheads. In this section, our proposed dynamic scheduling algorithms minimize the number of task migrations between PCM and DRAM. The mechanisms of our algorithms ensure each task instance can be migrated at most once.

5.1 Dynamic-RM Scheduling Algorithm

In static-RM algorithm, all tasks satisfy the real-time constrains even we assume tasks always require the worst-case execution time. Actually, each task consumes far less time than the $WCET$ which results in much slack time. If we allocate all the slack time to other tasks in a reasonable way, the real-time constrains can still hold. Accordingly, in the dynamic-RM algorithm, we attempt to apply the reduction of the execution time to other unexecuted tasks.

Our basic principle is based on the view that any reduction in the actual execution time can be used by other tasks to potentially change their execution media. However, the RM schedulability test is significantly complex. It's time-consuming to determine whether the re-allocation will affect the schedulability of the entire task set from the over-

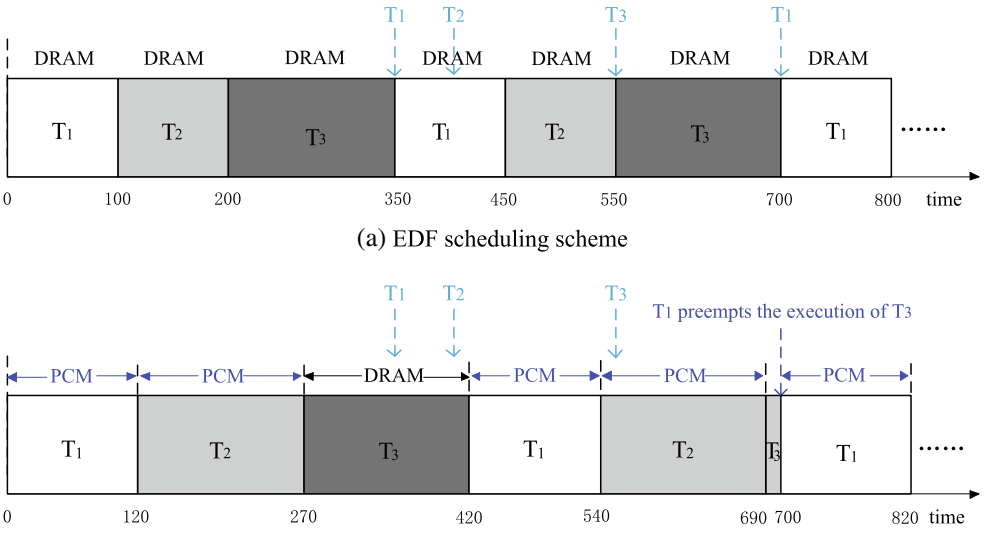


Figure 3 Schematic diagram of static-EDF algorithm.

all point of view. Moreover, when a task is released for its next invocation, we cannot know the exact time it will actually require, so we must reserve the $WCET$ time for all the future task instances. In fact, the scheduling algorithm will be very inefficient if we perform such complex re-allocation and schedulability test each time a task completes. To solve this problem, we take a novel way to avoid the complex RM schedulability test. We propose dynamic-RM algorithm to keep monitoring the closest deadlines to optimize the static-RM solution. The dynamic-RM algorithm only focuses on the time interval from current time to the closest deadline.

Figure 4 illustrates the mechanism of dynamic-RM algorithm. Consider three periodic tasks with following parameters:

$$W_{d1} = 100,\ P_1 = 350;$$
$$W_{d2} = 100,\ W_{p2} = 150,\ P_2 = 440;$$
$$W_{d3} = 150,\ P_3 = 550.$$

where, T_1 has the minimum task period of 350 and all tasks are D-tasks. Static-RM algorithm schedules the task set assuming all tasks consume the $WCET$, as shown in Fig. 4a. In actual execution process, task T_1 only spends 50 time units which leads to 50 slack time before the closest deadline, i.e. D_1. Now we focus on the time span between current time and the closest deadline, i.e. the interval from time 50 to time 350. Since $W_{d2} = 100$ and $W_{p2} = 150$, all the slack time produced by task T_1 can be allocated to task T_2 to convert it to a P-task. After executing T_2 in PCM, repeat this process. Finally, the scheduling result of dynamic-RM is shown in Fig. 4b, c and d.

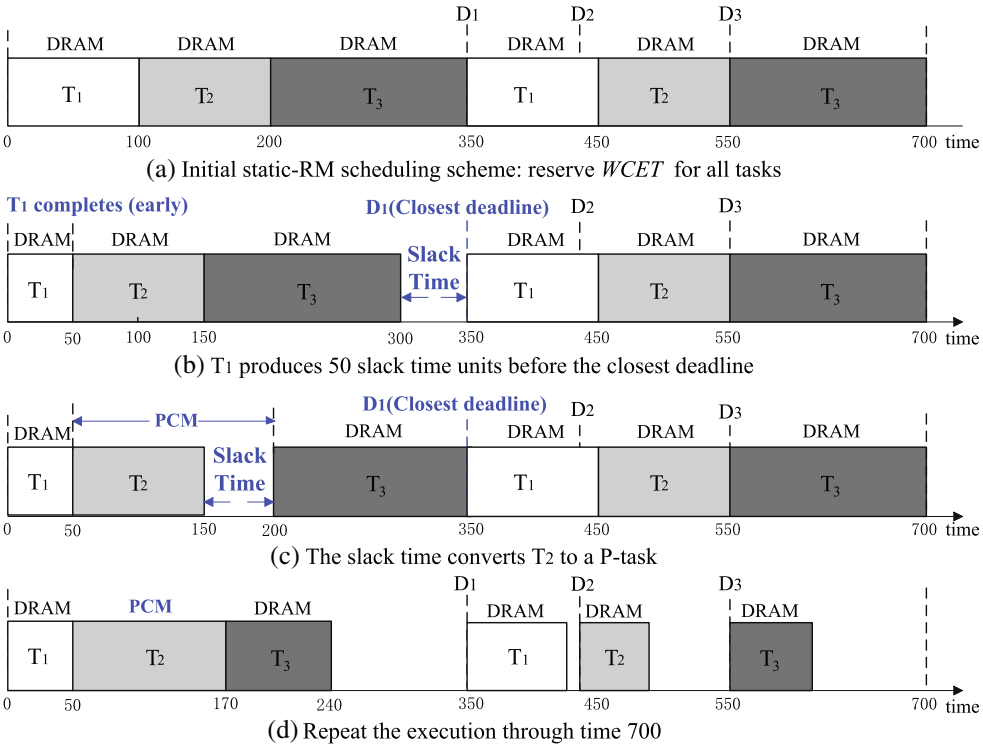
Table 6 defines the notations used in the dynamic-RM algorithm. Unlike static algorithms, dynamic algorithms schedule the task set with the instances of each periodic task.

Algorithm 5 TimeAllocation($T_{closedeadline}$)

Notes: The ready task instances are sorted by the ascending order of period. If $i < j$, t_i has higher priority than t_j

1: **if** $slack = T_{closedeadline} - \sum_{i=1}^{n} allocation_i > 0$ and the current highest priority task instance t_i is a D-task and $allocation_i == C_i$ **then**
2: **if** $slack + allocation_i \geq W_{pi}$ **then**
3: $allocation_i = W_{pi}$;
4: **else** $\{(slack - migTime_i)/W_{pi} \geq threshold\%\}$
5: $allocation_i = slack + C_i$;
6: $migFlag_i = 1$;
7: **end if**
8: **end if**

Procedure TimeAllocation($T_{closedeadline}$), shown in Algorithm 5, is used to allocate extra time to a task. Only a D-task instance with current highest priority can be reallocated. In our mechanism, if the slack time is sufficient to convert a D-task t_i to a P-task, scheduler allocates W_{pi} time to it.



(a) Initial static-RM scheduling scheme: reserve $WCET$ for all tasks

(b) T1 produces 50 slack time units before the closest deadline

(c) The slack time converts T2 to a P-task

(d) Repeat the execution through time 700

Figure 4 Schematic diagram of dynamic-RM algorithm.

Table 6 Notations used in dynamic-RM algorithm.

Notation	Definition
t_i	a task instance of T_i
$allocation_i$	the time allocated to the task instance t_i
$migTime_i$	migration time of task T_i which is the ratio of S_i and the migration rate
$migFlag_i$	migration flag of task instance t_i, 1 means task t_i needs to be migrated from PCM to DRAM
$T_{closedeadline}$	the time to the closest deadline
C_i	the $WCET$ of task instance t_i, if T_i is a P-task, $C_i = W_{pi}$, or $C_i = W_{di}$

However, if the slack time is not adequate, the situation is slightly more complicated. As a task may not be uniform, we cannot make any assumptions about the remaining execution time in different kinds of memory media. So before allocating a task to PCM, we have to reserve the whole W_{di} in case the task will be migrated to DRAM in the future. In addition to this guarantee, we check if the slack time can ensure t_i executed at least *threshold* % (e.g. 50 %) of W_{pi} in PCM. The main rationale behind the rule is that the task's actual execution time is usually much smaller than the $WCET$. The threshold may be enough for the task's actual execution without migration between PCM and DRAM.

Algorithm 6 Dynamic-RM Scheduling Algorithm

Input: Periodic Task set $T = \{T_1, T_2, \cdots, T_n\}$, each task T_i is characterized by $< W_{di}, W_{pi}, P_i, N_{wi}, S_i >$
Output: The scheduling and allocation scheme

for all task instances

1: Execute the static-RM algorithm and initialize all the $migFlag_i$ and $allocation_i$ to 0;
2: Upon task arrival/preemption:
  set $allocation_i = C_i$;
  update $T_{closedeadline}$ to the current time interval to the closest deadline;
  TimeAllocation($T_{closedeadline}$);
3: Upon task completion:
  set $allocation_i = 0$;
  set $migFlag_i = 0$;
  update $T_{closedeadline}$ to the current time interval to the closest deadline;
  TimeAllocation($T_{closedeadline}$);
4: When task is executing:
  decrease the $allocation_i$ over time;
5: Upon task migration:
  if $migFlag_i > 0$ and $allocation_i == C_i + migTime_i$ **then**
    task migration from PCM to DRAM;
    $allocation_i = allocation_i - migTime_i$;
    $migFlag_i = 0$;
  end if

The dynamic-RM scheduling algorithm is presented in Algorithm 6. The dynamic-RM algorithm initially allocates at least the $WCET$ time to the ready tasks according to the task type, D-task or P-task, classified by the static-RM scheduler. If a task is about to execute (upon task arrival or preemption) and there exists slack time before the closest deadline, scheduler reallocates time to it according to Algorithm 5. When a task completes its execution ahead of the allocated time, dynamic-RM scheduler reclaims the unused time by setting $allocation_i = 0$ and slack time is produced which can be used for other tasks.

If the allocated time isn't enough for a task's whole execution in PCM, the task have to be migrated from PCM to DRAM. However, as we reserve the whole W_{di} in case of the situation, the task is still able to hold its real-time constraints.

Additionally, dynamic-RM scheduler is a preemptive scheduler. However, we focus on the closest deadline and we assume a task's deadline is the same as its period, so the preemption will not happen before the closest deadline, which means task preemption will not affect our time allocation process. To better understand the dynamic-RM algorithm, Fig. 5 shows the flowchart of a D-task before the closest deadline.

Theorem 1 *If a task set is schedulable under static-RM scheduler, it remains schedulable under the dynamic-RM scheduler.*

Proof Firstly, we prove that if a task t_i can complete before the closest deadline under static-RM scheduler, the dynamic-RM scheduler will not violate its real-time constraints.

According to Algorithm 5, dynamic-RM algorithm reallocates slack time to the task with current highest priority before the closest deadline. So before the arrival of the deadline, the task t_i keeps the highest priority and will not be preempted by other tasks. If we allocate W_{pi} time to t_i (line 2, 3 in Algorithm 5), the reallocated time is even enough for task t_i's completion in PCM before the closeset deadline. If the slack time is not enough to convert t_i to a P-task (line 4, 5, 6 in Algorithm 5), Algorithm 6 migrates the task to DRAM in a conservative way, which reserves W_{di} time for t_i's execution in DRAM (line 5 in Algorithm 6). Therefore, task t_i's completion time under dynamic-RM is no greater than that of static-RM.

Secondly, we assert that if t_i cannot complete before the closest deadline under static-RM scheduler, the dynamic-RM scheduler will not violate its real-time constraints. We can see from Algorithm 5 that the reallocated time only relies on the slack time, it will not occupy the time allocated to other uncompleted tasks. Therefore,

the influence of dynamic-RM will not cross the closest deadline.

In summary, if a task set is schedulable under static-RM scheduler, it remains schedulable under the dynamic-RM scheduler. □

Theorem 2 *The dynamic-RM scheduler ensures that each task instance can be migrated at most once.*

Proof Algorithm 5 allocates slack time to the task with current highest priority. If the allocated time is enough to convert the task to a P-task, migration will not happen. However, if the time is not sufficient for the task's entire execution in PCM, Algorithm 6 will migrate it from PCM to DRAM. After the migration, the task instance will be executed in DRAM until its completion and will not be reallocated slack time (line 1 in Algorithm 5), which ensures the task can be migrated at most once. □

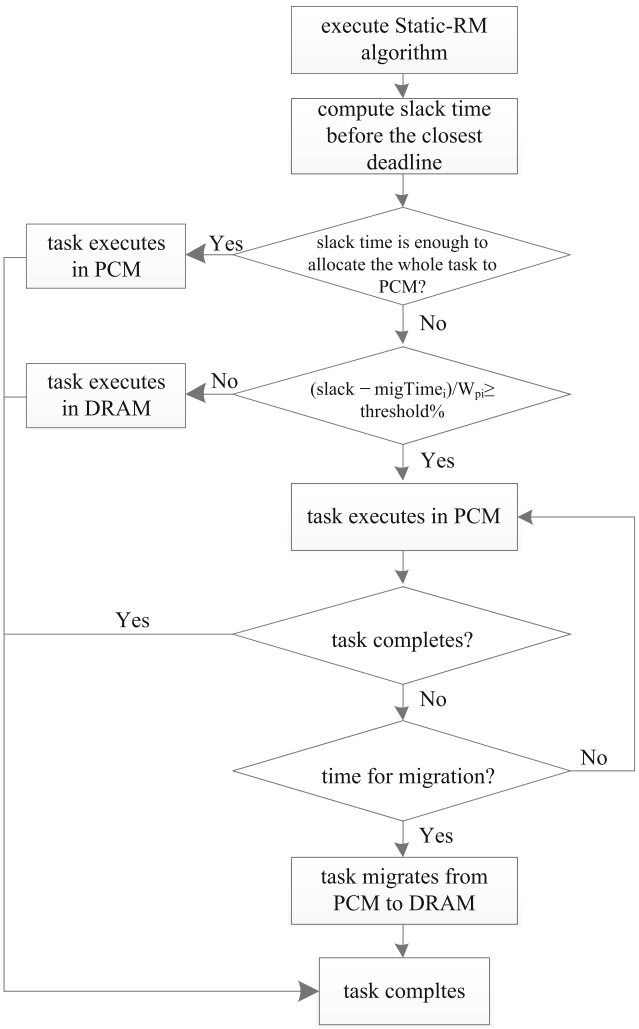


Figure 5 The flowchart of a D-task before the closest deadline in dynamic-RM algorithm.

5.2 Dynamic-EDF Scheduling Algorithm

Since tasks generally consume much less time than the worst case on most invocations, our dynamic algorithms detect early completions and apply them to other tasks on-the-fly for energy saving while ensuring all deadlines still hold. In dynamic-RM scheduler, in order to simplify the complexity of the algorithm, we only focus on the closest deadline. In dynamic-EDF algorithm, we can use the same approach to optimize the static-EDF solution. However, unlike RM scheduler, as a dynamic priority scheduler, EDF schedules tasks more flexibly. Thus, our dynamic-EDF scheduling algorithm employs a different mechanism to reclaim the slack time.

In the ideal case, all slack time produced by early completions can be allocated to other tasks to achieve maximum energy saving. However, in practice, considering real-time constrains and task migration issues, part of slack time has to be wasted. In dynamic-EDF algorithm, we try the best to reclaim the slack time. To this aim, we calculate the slack time from a holistic perspective rather than merely based on the closest deadline.

Unfortunately, before presenting our algorithm, we have to emphasize the fact that it's not a safe way to blindly use all the slack time produced by the completed tasks. To understand this, consider three periodic tasks with following parameters:

$$W_{d1} = 100,\ W_{p1} = 140,\ P_1 = 300;$$
$$W_{d2} = 180,\ P_2 = 300;$$
$$W_{d3} = 60,\ P_3 = 900.$$

All the three tasks are D-tasks (Fig. 6a). Suppose task T_3 completes at $t = 300$, leaving 40 slack time units before its deadline, as shown in Fig. 6b. If all the 40 time units are allocated to T_1, T_1 will be converted to a P-task which apparently leads to T_2 missing its deadline, if both tasks consume their $WCET$ (Fig. 6c).

Obviously, computing the available slack time is not a trivial task. To manage all the allocated time by static-EDF, we introduce a data structure, called *Ready-Queue*, to maintain the remaining time of all the ready tasks in the static-EDF schedule. For a periodic task set $T = \{T_1, T_2, \cdots, T_n\}$, the Ready-Queue contains at most n elements since our task model ensures that a task must be completed before its next arrival. We sort the Ready-Queue according to the EDF priority. The task t_i with minimum deadline, denoted by d_i, is at the head of the queue.

We use C_i to record the remaining time of task t_i. In static-EDF algorithm, if the task is a P-task, we initialize $C_i = W_{pi}$, or $C_i = W_{di}$. A task t_i can be removed from the Ready-Queue if and only if the remaining time $C_i = 0$.

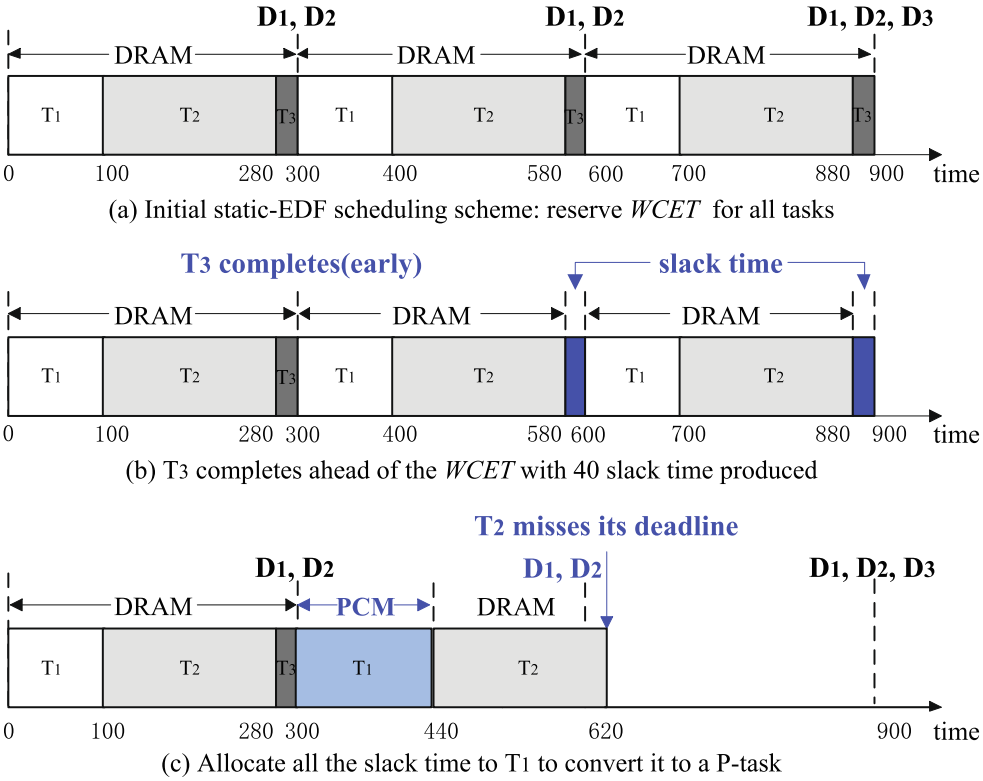


Figure 6 Insecure allocation of slack time.

Therefore, the Ready-Queue records all the remaining time allocated by static-EDF scheduler.

Utilizing the Ready-Queue, we can compute the available slack time for the task needed to be allocated extra time. For task t_i which is about to execute, the available slack time is the sum of remaining time of all completed tasks with higher priority than t_i. We calculate the available slack time of t_i in this manner:

$$Slack(i) = \sum_{x|d_x<d_i} C_x \tag{6}$$

It is worth noting that when t_i is about to execute, tasks with higher priority that are still in the Ready-Queue have completed in the actual schedule, but they don't complete in the static-EDF.

As tasks execute, the Ready-Queue is updated dynamically. The C_i of the first element in the queue decreases with the time. When C_i reaches to 0, the first element is removed from the queue and the updating process continues with the next element. For efficiency, we perform the update process of the Ready-Queue only when the task arrives, completes and migrates between PCM and DRAM. The Algorithm 7 presents the procedure of UpdateQueue(t), where the notation t is the elapsed-time since the last event.

Algorithm 7 UpdateQueue(t)

```
1:  while Ready-Queue is not empty and t > 0 do
2:      t_i is the head of the queue;
3:      if C_i > t then
4:          C_i = C_i − t;
5:          t = 0;
6:      else
7:          t = t − C_i;
8:          C_i = 0;
9:          Remove t_i from the queue;
10:     end if
11: end while
12: if Ready-Queue is empty then
13:     t = 0;
14: end if
```

Table 7 shows the update of Ready-Queue for the example task set in Fig. 6a. At time 600, three tasks have the same deadlines. As t_3 has the earliest arrival time, here we consider t_3 as the highest priority task.

Table 7 Update of ready-queue.

Time	Event	*Ready-Queue*={(t_i, d_i, C_i)}
0	t_1, t_2, t_3 arrive	{(t_1, 300, 100), (t_2, 300, 180), (t_3, 900, 60)}
100	t_1 completes	{(t_2, 300, 180), (t_3, 900, 60)}
280	t_2 completes	{(t_3, 900, 60)}
300	t_1, t_2 arrive	{(t_1, 600, 100), (t_2, 600, 180), (t_3, 900, 40)}
400	t_1 completes	{(t_2, 600, 180), (t_3, 900, 40)}
580	t_2 completes	{(t_3, 900, 40)}
600	t_1, t_2 arrive	{(t_3, 900, 20), (t_1, 900, 100), (t_2, 900, 180)}
620	t_3 completes	{(t_1, 900, 100), (t_2, 900, 180)}
720	t_1 completes	{(t_2, 900, 180)}
900	t_2 completes	∅

Let's reconsider the situation shown in Fig. 6b. At time 300, t_3 completes its execution, leaving 40 slack time units before its deadline. From Table 7, when time is 300, the Ready-Queue contains three elements, namely {(t_1, 600, 100), (t_2, 600, 180), (t_3, 900, 40)}. According to the calculation rule of available slack time, available slack time of t_i is the sum of remaining time of all completed tasks with higher priority than t_i. At this time, although t_3 is in the Ready-Queue with 40 remaining time, t_1 cannot use this time for its deadline is smaller than that of t_3, which avoids the unsafe allocation of slack time.

However, although we exploit the Ready-Queue to make it possible to compute the available slack time, there are still problems in the actual schedule. Since the nature of EDF scheduler, an executing task can be preempted at any time. Consider the scenario that task t_i is preempted by task t_j. If we allocate the slack time to t_j, which has already been allocated to t_i, the task t_i will potentially violate the real-time constraints since it has been executed for a while in the slower memory, i.e. PCM. Therefore, to address the problem, we should prevent task t_j preempting the allocated slack time of t_i. That is, higher priority task t_j can preempt the execution of lower priority task t_i, but it cannot preempt the additional allocated time of t_i. For this purpose, we introduce another data structure, *Preempt-Queue*, to maintain the allocated slack time of the preempted task. Whenever the Preempt-Queue is not empty, we will not allocate slack time to any tasks.

Before presenting the dynamic-EDF algorithm, Table 8 defines the notations used in the algorithm. The notations $exedTime_i$ and $PTime_i$ are used for tasks migrating from PCM to DRAM.

The details of our proposed dynamic-EDF algorithm are presented in Algorithm 8. The procedure UpdateQueue(t), as shown in Algorithm 7, is used to update the Ready-Queue.

As shown in Algorithm 8, if a task is about to execute and the Preempt-Queue is empty, we calculate the available slack time and allocate to it. If the slack time is enough to convert a D-task to a P-task, we allocate W_{pi} to it. Otherwise, we allocate all the slack time to it and set the migration flag $migFlag_i$ to 1 since the task *may* have to migrate from PCM to DRAM in the future to meet the real-time constraint. Similarly, we also use the *threshold%* in the allocation process to potentially avoid the migration.

When a D-task reaches the migration condition (line 5 in Algorithm 8), dynamic-EDF migrates it to PCM and updates corresponding counters. At this time, the task is still in Ready-Queue and its C_i ensures the entire execution in DRAM, so the real-time constraint holds.

In dynamic-EDF algorithm, the preemption could occur at any time. When task preemption occurs, we protect the allocated time of lower priority tasks by inserting them to Preempt-Queue.

Table 8 Notations used in dynamic-EDF algorithm.

Notation	Definition
t_i	a task instance of T_i
$allocation_i$	the time allocated to the task instance t_i
$migTime_i$	migration time of task T_i which is the ratio of S_i and the migration rate
$migFlag_i$	migration flag of task instance t_i, 1 means task t_i needs to be migrated from PCM to DRAM
$exedTime_i$	the executed time of a D-task in PCM
$PTime_i$	the allocated time of a D-task in PCM
C_i	the remaining time of task t_i; in static-EDF algorithm, if T_i is a P-task, initialize $C_i = W_{pi}$, or $C_i = W_{di}$

Algorithm 8 Dynamic-EDF Scheduling Algorithm

1: Execute the static-EDF algorithm and initialize all the $migFlag_i$, $exedTime_i$ and $PTime_i$ to 0, the Ready-Queue and the Preempt-Queue to empty queue;
2: Upon task arrival:
  set $allocation_i = C_i$;
  insert t_i to Ready-Queue according to the EDF priority;
  if task t_j is executing, set $allocation_j = allocation_j - t^{'}$, where $t^{'}$ is the elapsed time since the last event;
  UpdateQueue($t^{'}$);
3: Upon task completion:
  set $allocation_i$, $migFlag_i$, $exedTime_i$, $PTime_i$ to 0;
  UpdateQueue($t^{'}$);
4: Upon task preemption:
  insert the task with $allocation_i > C_i$ to the Preempt-Queue;
5: Task migration:
  if $migFlag_i > 0$ and $exedTime_i == PTime_i - migTime_i$ **then**
    $allocation_i = allocation_i - migTime_i - t^{'}$;
    $migFlag_i = 0$;
    UpdateQueue($t^{'} + migTime_i$);
  end if
6: Upon task restoration from preemption:
  if t_i is only element in the Preempt-Queue **then**
    remove t_i from the Preempt-Queue;
    if $Slcak(i) + C_i \geq W_{pi} - exedTime_i$ **then**
      $allocation_i = W_{pi} - exedTime_i$;
      $migFlag_i = 0$;
    else
      $PTime_i = Slcak(i) + exedTime_i$;
      $allocation_i = Slcak(i) + C_i$;
    end if
  end if
7: Upon an unexecuted task is about to execute:
  if t_i is a D-task and Preempt-Queue is empty **then**
    if $Slcak(i) + C_i \geq W_{pi}$ **then**
      $allocation_i = W_{pi}$;
    else $\{(Slcak(i) - migTime_i)/W_{pi} \geq threshold\%\}$
      $PTime_i = Slcak(i)$;
      $allocation_i = Slcak(i) + C_i$;
      $migFlag_i = 1$;
    end if
  end if
8: For the task with $migFlag_i = 1$, increase $exedTime_i$ with its execution;

If a D-task restores from the preemption before migrating to DRAM, the situation is complicated. If the task is in the Preempt-Queue, we remove it. If the removal leads to an empty Preempt-Queue, the dynamic-EDF scheduler recalculates the available time and attempts to allocate more slack time to it. If the extra slack time is adequate for the remaining execution, the migration flag $migFlag_i$ is reset to avoid migration and the task can complete in PCM. Or, we allocate all the extra slack time to it to potentially avoid the migration.

Similar to dynamic-RM scheduler, the allocation rules of dynamic-EDF scheduler guarantee tasks can be only migrated from PCM to DRAM and cannot be migrated back. This is the key point to ensure each task can be migrated at most once. The flowchart of a D-task in dynamic-EDF algorithm is shown in Fig. 7.

Theorem 3 *If a task set is schedulable under static-EDF scheduler, it remains schedulable under the dynamic-EDF scheduler.*

Proof Similarly to the dynamic-RM scheduler, dynamic-EDF only allocates slack time to uncompleted tasks and will not occupy other tasks' original allocated time (i.e. C_i). The Ready-Queue guarantees the secure use of slack time generated by higher priority tasks. Meanwhile, we introduce the

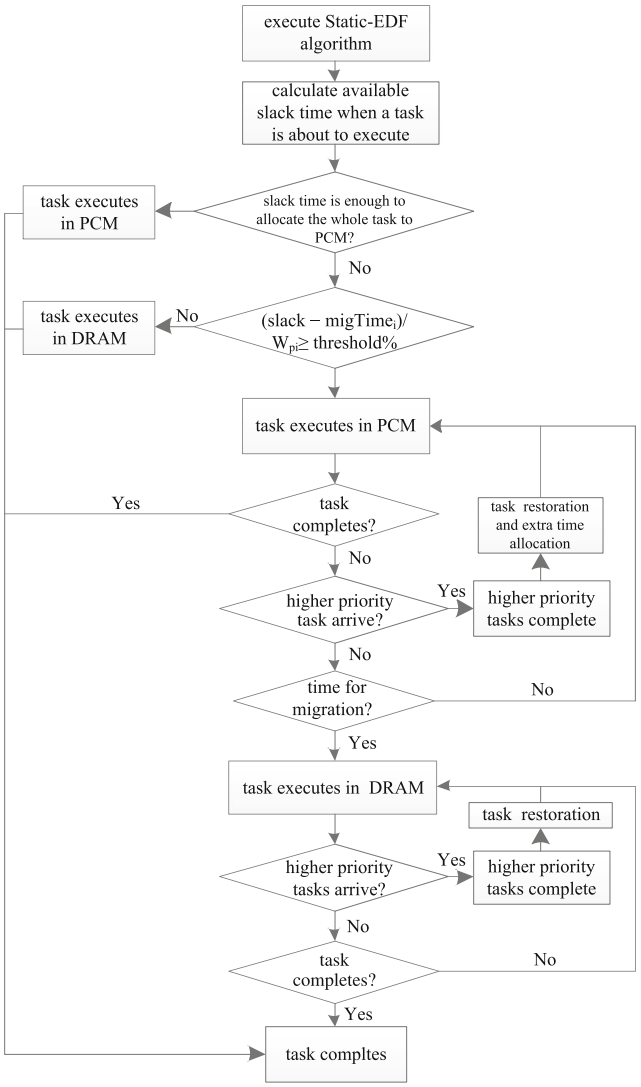


Figure 7 The flowchart of a D-task in dynamic-EDF algorithm.

Preempt-Queue to avoid the higher priority tasks preempting the allocated slack time of lower priority tasks. All these mechanisms ensure that, at any time t during the execution of dynamic-EDF, no task instance completes later than the completion time in static-EDF. The similar theorem has been detailed in [1, 2]. Accordingly, if a task set is schedulable under static-EDF scheduler, it remains schedulable under the dynamic-EDF scheduler. □

Theorem 4 *Under the dynamic-EDF scheduler, each task instance can be migrated at most once.*

Proof In Algorithm 8, if the slack time is not sufficient for a task's entire execution in PCM, it will be migrated from PCM to DRAM. After the migration, if the task is preempted by a higher priority task, it will not be inserted to the Preempt-Queue for its $allocation_i \leq C_i$ (line 4 in Algorithm 8). When the task restores from the preemption, it will not be reallocated time again for the algorithm 8 requires that only the unique task of the Preempt-Queue can be reallocated slack time (line 6 in Algorithm 8). Therefore, the migrated task will be executed in DRAM until its completion which ensures each task instance can be migrated at most once. □

6 Experiments

In this section, we evaluate the performance of our scheduling algorithms according to the parameters measured in our own hybrid main memory platform. To show the superiorities of our algorithms, we compare the traditional scheduler in pure DRAM memory (256MB DRAM) with our proposed algorithms in hybrid main memory (128MB DRAM and 128MB PCM).

We developed a task scheduling simulator to evaluate the performance of various real-time scheduling algorithms. For different evaluation scenarios, we adopted different experimental parameters. In order to evaluate the energy saving of proposed aperiodic scheduling algorithms, we test the average energy consumption of 20 aperiodic task sets, each containing 10 aperiodic tasks. We evaluate our algorithms with variable actual workloads by altering the ratio of $BCET$ (the Best-Case Execution Time) and $WCET$. When a task executes, it selects a random actual execution time which is with uniform distribution between $BCET$ and $WCET$. Since periodic algorithms are much more complex than aperiodic algorithms, we test our periodic algorithms with 30 periodic task set, each containing 10 periodic tasks, and evaluate our algorithms under two cases. First, we assume tasks require a fixed computation time range from 10 % to 100 % of their $WCET$ for each invocation. Then we assume tasks consume different computation time for each invocation. Similarly, when a task instance is released, it selects a random actual execution time which is with uniform distribution between $BCET$ and $WCET$.

Particularly, we adopt the energy model presented in Section 2 to focus on the average energy consumption. For our algorithms, a hybrid PCM-DRAM main memory system is introduced. In hybrid memory, whenever a task is executed in PCM, we set the DRAM to standby state which brings extra energy consumption, called standby energy. In our work, we keep PCM in the active state due to its very low static power. So there is no switch overhead for PCM. As shown in [7], the timing and energy overhead of switching the DRAM between standby and active states is very small. Furthermore, given our migration policy, the number of switches (corresponding to each migration) is far smaller than that of the read/write operations. Therefore, the switching overhead can be ignored in the overall experimental results. We calculate the energy consumption of the whole simulation time until all tasks complete.

For traditional scheduler in pure DRAM memory, the energy includes working and refresh consumption of DRAM. But for our algorithms in hybrid memory, the situation is more complicated. Whenever tasks are not executed in DRAM, DRAM maintains the standby state. Additionally, when tasks are migrated from PCM to DRAM, we need to read the data needed to be migrated from PCM and write to DRAM. We assume that we need to migrate the maximum data size, that is S_i. Therefore, the performance of our algorithms presented below is more conservative.

6.1 Aperiodic Scheduling Algorithms

For an aperiodic task set, we schedule it with the principle: the earlier the deadline, the higher the priority. We compare the traditional scheduler in pure DRAM memory, denoted as EDF-Aperiodic, with our proposed aperiodic scheduling algorithms in hybrid main memory, hereafter referred to as Static-Aperiodic and Dynamic-Aperiodic. We focus on the average energy consumption of 20 aperiodic task sets, each containing 10 aperiodic tasks. In the following simulations, we set the *threshold* % to 50 % to potentially avoid the migrations from PCM to DRAM.

Since the aperiodic task set in this paper is the execution of all tasks, the simulation time depends on the latest deadline. In the simulation, we assume tasks don't require their $WCET$. We evaluate our aperiodic scheduling algorithms with variable actual workloads by altering the ratio of $BCET$ and $WCET$. When a task executes, it selects a random actual execution time which is with uniform distribution between $BCET$ and $WCET$.

Compared with EDF-Aperiodic scheduler in pure DRAM memory, our aperiodic scheduling algorithms gain different energy saving with different $BCET/WCET$.

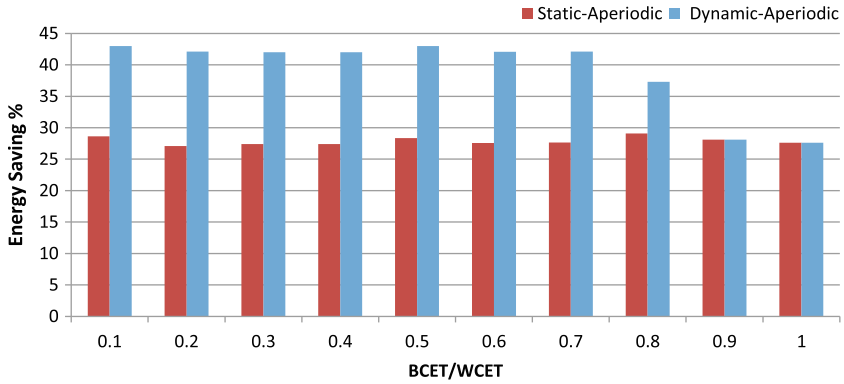


Figure 8 The energy saving of aperiodic scheduling algorithms with different BCET/WCET.

When the ratio of $BCET$ and $WCET$ changes from 0.1 to 1, the energy saving of our proposed Static-Aperiodic and Dynamic-Aperiodic algorithms is shown in Fig. 8, in which the value of EDF-Aperiodic scheduler is zero. We can see from the figure, both our proposed aperiodic algorithms reduce the energy consumption while the Dynamic-Aperiodic algorithm outperforms the Static-Aperiodic algorithm for dynamic solution continuously updates the elastic time to enable more tasks to be allocated to PCM.

When $BCET/WCET$ increases, the probability of actual execution time falling in higher time interval increases which means higher workloads. If the workloads are high, dynamic and static solutions achieve similar results because dynamic algorithm is fail to allocate more tasks to PCM under higher workload environments, which is verified in Fig. 8 when $BCET/WCET$ is equal to 0.9 and 1. But for lower workloads, the dynamic solution significantly reduces more energy consumption than static solution for it allocates more tasks to PCM.

6.2 Periodic Scheduling Algorithms

In this section, we show the performance of our proposed periodic scheduling algorithms. We compare the EDF scheduler in pure DRAM memory, denoted as EDF, with our proposed periodic scheduling algorithms in hybrid main memory, hereafter termed as Static-RM, Static-EDF, Dynamic-RM, and Dynamic-EDF. We test the average energy consumption of 30 periodic task sets, each containing 10 periodic tasks. Similarly, in the following simulations, we also set the *threshold* % to 50 % to potentially avoid the migrations from PCM to DRAM.

6.2.1 Fixed Computation Time

Since all periodic algorithms proposed in this paper assume tasks consume their $WCET$, we evaluate the performance when tasks do not consume their $WCET$. We define the $utilization$ as the ratio of actual execution time and the $WCET$. In the preceding simulation experiments, we assume tasks require a fixed computation time range from 10 % to 100 % of their $WCET$ for each invocation. We normalized the results with the baseline of EDF scheduler in pure DRAM memory.

Firstly, we observe the execution time and the number of tasks executed in PCM. As shown in Table 9, the execution time increases with the increment of the number of tasks in PCM. Among all our proposed algorithms, Dynamic-EDF costs the maximum time while Static-RM costs least. The reason is that Dynamic-EDF algorithm reclaims the slack time as much as possible to allocate more tasks to PCM.

Table 9 The normalized comparisons on execution time and the number of tasks in PCM.

Utilization	EDF		Static-RM		Dynamic-RM		Static-EDF		Dynamic-EDF	
	$Time$	$TasksinPCM$	$Time$	$TasksinPCM$	$Time$	$TasksinPCM$	$Time$	$TasksinPCM$	$Time$	$TasksinPCM$
U=0.1	1	0 %	1.09569	9.56938 %	1.26794	61.244 %	1.22967	49.7608 %	1.29665	69.8565 %
U=0.3	1	0 %	1.06182	12.3648 %	1.22875	62.442 %	1.19165	51.3138 %	1.25657	70.7883 %
U=0.5	1	0 %	1.0553	12.9032 %	1.2212	62.6728 %	1.18433	51.6129 %	1.24885	70.9677 %
U=0.7	1	0 %	1.05253	13.132 %	1.19041	54.4977 %	1.18122	51.74 %	1.24557	71.044 %
U=0.9	1	0 %	1.05099	13.2585 %	1.1795	51.8103 %	1.1795	51.8103 %	1.18409	53.1871 %
		Average	1.063266	12.245576 %	1.21756	58.53336 %	1.193274	51.24756 %	1.246346	67.16872 %

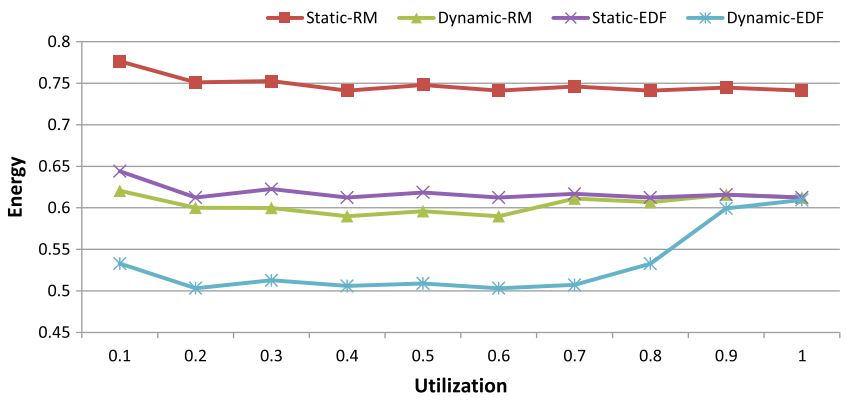


Figure 9 The normalized energy consumption with different utilization.

When the utilization is low, the execution time of Dynamic-EDF is up to 130 % and decreases with the increment of utilization for the reason that in low workloads Dynamic-EDF algorithm can allocate almost all tasks to PCM which leads to higher time overheads.

The energy consumption with different utilizations is shown in Fig. 9. The EDF scheduler in pure DRAM consumes the most energy (Energy=1) while Dynamic-EDF costs least. The energy consumption of all our proposed algorithms is stable regardless of the utilizations which shows better adaptability of our algorithms to various workloads. In higher workloads ($U > 0.8$), Static-EDF, Dynamic-RM and Dynamic-EDF schedulers show the similar performance since in higher workloads, less slack time is produced by completed tasks. Among all our proposed algorithms, the dynamic scheduling algorithms gain better energy efficiency than the static algorithms. Compared with the EDF scheduler, the Dynamic-EDF achieves an average of 52% energy saving.

6.2.2 Variable Computation Time

In the real environment, tasks consume different computation time for each invocation. In order to evaluate the real performance of our algorithms, we test our algorithms with variable actual workloads by altering the ratio of $BCET$ and $WCET$. When a task instance is released, it selects a random actual execution time which is with uniform distribution between $BCET$ and $WCET$.

We show the execution time and number of tasks in PCM as a function of $BCET/WCET$, as shown in Figs. 10 and 11, respectively. For the EDF scheduler in pure DRAM, all tasks are executed in DRAM (Task in PCM %=0), so it consumes the minimum time (Time=1). When $BCET/WCET$ increases, the probability of actual execution time falling in higher time interval increases which means higher workloads. As shown in Fig. 10, when $BCET/WCET = 0.1$, all our algorithms have maximum execution time since in the low workloads, all our scheduling algorithms allocate more tasks to PCM, which is verified by Fig. 11. When $BCET/WCET = 1$, which means each task instance consumes its $WCET$, the workloads are maximum. We can see from Fig. 10 that when $BCET/WCET = 1$, except Static-RM, the other three algorithms consume almost the same time for the reason that in high workloads the three algorithms are fail to allocate more tasks to PCM with less slack time. In addition, all our proposed algorithms show better adaptabilities to different workloads from $BCET/WCET = 0.1$ to $BCET/WCET = 1$.

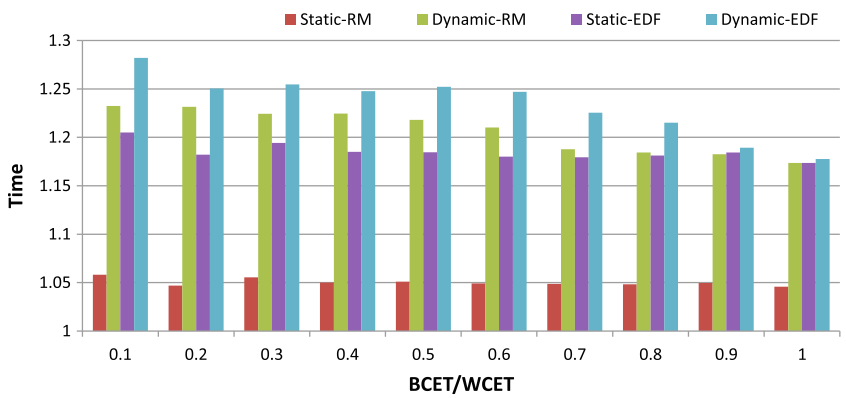


Figure 10 The normalized execution time with different BCET/WCET.

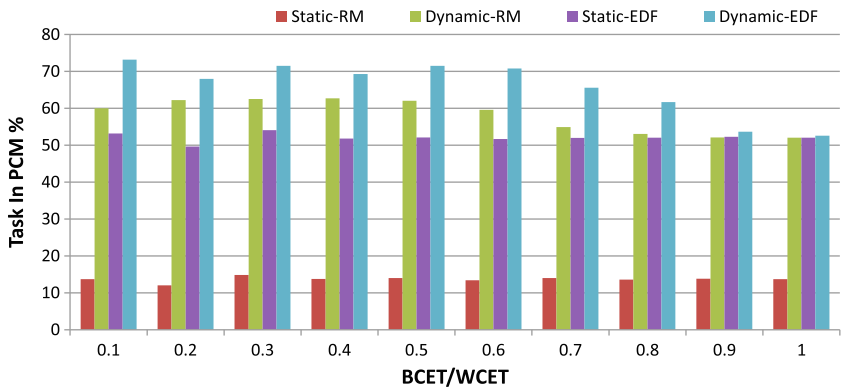


Figure 11 The percentage of tasks in PCM with different BCET/WCET.

Figure 12 shows the energy saving with different $BCET/WCET$. We can see that among our proposed algorithms, the Dynamic-EDF algorithm achieves the best energy saving for any workloads since it dynamically reclaims slack time to allocate more D-tasks to PCM. The Dynamic-RM is superior than Static-EDF in energy efficiency. Figure 12 shows the similar result with Fig. 9: compared with the EDF scheduler, all our proposed algorithms show better adaptability for various workloads.

6.3 Summary

In this section, we focused on demonstrating the effectiveness of our proposed real-time scheduling algorithms. The experimental results show that our proposed algorithms significantly reduce energy consumption. All the proposed algorithms are adaptive for different workloads. Overall, our aperiodic scheduling algorithms reduce the energy consumption range from 28.8 % to 39.1 % on average. Meanwhile, compared with EDF scheduler in pure DRAM, the proposed periodic scheduling algorithms Static-RM, Static-EDF, Dynamic-RM and Dynamic-EDF achieve the average energy saving of 25.7 %, 38 %, 39.4 % and 47.2 % while bringing the time overheads of 5.02 %, 17.5 %, 18.7 % and 23.4 %, respectively.

7 Related Work

PCM, as an emerging nonvolatile memory, has been intensively studied as a promising candidate main memory. Comparing with DRAM, PCM shows the advantages of nonvolatility, low power and high density. Considering the drawbacks of limited write endurance and longer access latency, the architectures integrating PCM and DRAM into the main memory have been proposed [4, 13, 15, 21, 22]. In these studies [4, 13, 22], operating system has been studied to support the hybrid main memory. The software and hardware design approaches were detailed in [13]. Pages between PCM and DRAM can be transferred to prolong the lifetime of PCM [4, 22]. A small DRAM as the buffer/cache of PCM has been exploited to improve the performance and reduce the writes on PCM [15]. These previous studies were designed in page level. They focus on the issues

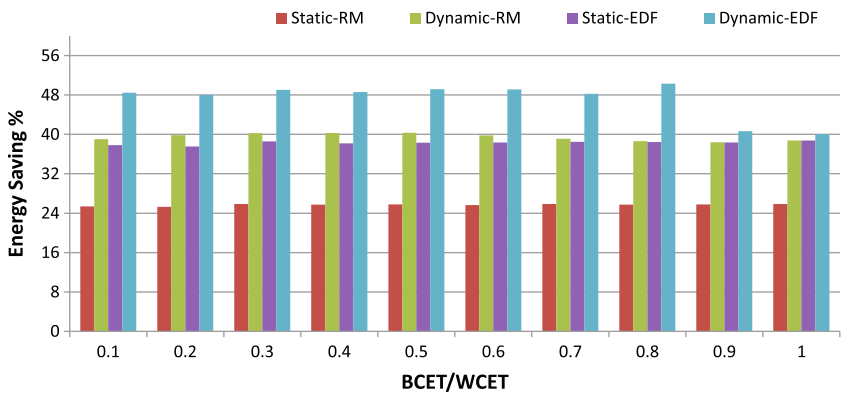


Figure 12 The energy saving with different BCET/WCET.

of page placement and migration between DRAM and PCM for energy optimization.

Different schemes have been proposed to optimize the energy consumption of hybrid main memory system [12, 16, 20]. Focusing on the variable partitioning problem, ILP and heuristic algorithms were proposed to maximize the energy saving [12]. At the same time, multiple optimization objectives, such as power consumption, schedule length and the number of writes, were considered. In addition, Zili Shao et al. proposed an energy cost model for hybrid main memory and designed an optimal static data allocation scheme in [16]. However, these work aimed to improve the lifetime of PCM and reduce energy consumption in variable level. They considered the instruction scheduling rather than task scheduling with real-time constraints.

By leveraging the respective superiorities, the task allocation problem on hybrid memory has been studied to reduce the energy consumption in [20]. The ILP and two sets of heuristic algorithms were proposed for different objectives. But this work focused on the problem of task allocation instead of task scheduling. Besides, the issue of task migration has not been addressed well.

Miao Zhou et al. added a real-time scheduler for prioritizing requests at the bottleneck resource, the PCM controller [23]. The external priorities, critical read boosting and read over write are studied to reduce the number of deadline misses. However, all the schemes focus on the memory controller, rather than the real-time scheduling algorithms. Hybrid main memory makes the real-time task scheduling problem more complicated, but the problem has not been studied extensively yet.

Real-time task scheduling issues have been widely studied for decades. Typical real-time task schedulers of RM and EDF were proposed in [11]. A host of algorithms on dynamic voltage scaling [2, 9, 14, 17] and dynamic power management [5, 6, 18] have been presented along with the development of CPU and power technology. However, in modern embedded systems, CPU is no longer the main energy contributor to the total system. Although a growing amount of energy is consumed by the main memory, few real-time task scheduling studies have been conducted to reduce the energy consumption of the main memory.

Therefore, we focus on the real-time task scheduling on low-energy embedded systems with hybrid PCM-DRAM main memory. For different task sets, aperiodic and periodic task set, we propose respective low complexity static and dynamic algorithms to maximize energy saving while minimizing the task migration between PCM and DRAM.

8 Concluding Remarks

In this paper, we studied the real-time task scheduling problem. For an aperiodic task set, we proposed static table-driven and dynamic (online) scheduling algorithms for energy efficiency. Meanwhile, for a periodic task set, we proposed four real-time scheduling algorithms based on the RM and EDF schedulers in real-time embedded systems with hybrid PCM-DRAM main memory. Based on static schedulability analysis, the static-RM and static-EDF algorithms are detailed to utmostly allocate tasks to PCM for energy saving. Besides, we presented a dynamic-RM algorithm exploiting the slack time of closest deadline and a dynamic- EDF algorithm reclaiming all available slack time to optimize the results of static solutions. The experimental results show that our scheduling algorithms outperform the EDF scheduler in pure DRAM main memory in terms of energy consumption. Furthermore, all the proposed algorithms are capable of adapting to the varying workloads.

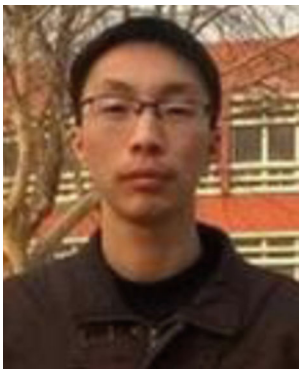
Nonetheless, the current dynamic scheduling algorithms can be further optimized. The optimal size ratio of PCM and DRAM is another important issue in hybrid memory system. Moreover, we do not consider the hybrid scheduling of aperiodic and periodic tasks. Furthermore, the problem of periodic tasks with jitter is also not addressed for the hybrid main memory system. All of these will be addressed in the future.

Acknowledgments This research is sponsored by the Natural Science Foundation of China (NSFC) under Grant No. 61070022 and 61202015, Shandong Provincial Natural Science Foundation under Grant No. ZR2011FQ036 and ZR2013FM028, National 863 Program 2013AA013202, and Chongqing cstc2012ggC40005.

References

1. Aydın, H. (2001). *Enhancing performance and fault tolerance in reward-based scheduling*. University of Pittsburgh: Ph.D. thesis.
2. Aydin, H., Melhem, R., Mossé, D., & Mejia-Alvarez, P. (2001). Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *Proceedings 22nd IEEE Real-Time Systems Symposium, 2001.(RTSS 2001)* (pp. 95–105): IEEE.
3. Barroso, L.A., & Holzle, U. (2007). The case for energy-proportional computing. *Computer*, *40*(12), 33–37.
4. Dhiman, G., Ayoub, R., & Rosing, T. (2009). Pdram: a hybrid pram and dram main memory system. In *46th ACM/IEEE Design Automation Conference, 2009. DAC'09* (pp. 664–669): IEEE.
5. Gary, S., Ippolito, P., Gerosa, G., Dietz, C., Eno, J., & Sanchez, H. (1994). Powerpc 603, a microprocessor for portable computers. *Design & Test of Computers, IEEE*, *11*(4), 14–23.

6. Harris, E.P., Depp, S.W., Pence, W.E., Kirkpatrick, S., Sri-Jayantha, M., & Troutman, R.R. (1995). Technology directions for portable computers. *Proceedings of the IEEE*, *4*, 636–658.
7. Huang, H., Pillai, P., & Shin, K.G. (2003). Design and implementation of power-aware virtual memory. *Ann Arbor*, *1001*, 48,109–2122.
8. Ka, A., & Mok, L. (1983). Fundamental design problems of distributed systems for the hard-real-time environment, vol. 1. MIT Thesis.
9. Kim, W., Kim, J., & Min, S. (2002). A dynamic voltage scaling algorithm for dynamic-priority hard real-time systems using slack time analysis. In *Proceedings of the Conference on Design, Automation and Test in Europe* (p. 788): IEEE Computer Society.
10. Leung, J.Y.T., & Whitehead, J. (1982). On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance evaluation*, *2*(4), 237–250.
11. Liu, C.L., & Layland, J.W. (1973). Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, *20*(1), 46–61.
12. Liu, T., Zhao, Y., Xue, C.J., & Li, M. (2011). Power-aware variable partitioning for dsps with hybrid pram and dram main memory. In *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)* (pp. 405–410): IEEE.
13. Mogul, J.C., Argollo, E., Shah, M.A., & Faraboschi, P. (2009). Operating system support for nvm+ dram hybrid main memory. In *HotOS*.
14. Pillai, P., & Shin, K.G. (2001). Real-time dynamic voltage scaling for low-power embedded operating systems. In *ACM SIGOPS Operating Systems Review*, (Vol. 35 pp. 89–102): ACM.
15. Qureshi, M.K., Srinivasan, V., & Rivers, J.A. (2009). Scalable high performance main memory system using phase-change memory technology. *ACM SIGARCH Computer Architecture News*, *37*(3), 24–33.
16. Shao, Z., Liu, Y., Chen, Y., & Li, T. (2012). Utilizing pcm for energy optimization in embedded systems. In *2012 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)* (pp. 398–403): IEEE.
17. Shin, Y., Choi, K., & Sakurai, T. (2000). Power optimization of real-time embedded systems on variable speed processors. In *Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design* (pp. 365–368): IEEE Press.
18. Stemm, M., & et al (1997). Measuring and reducing energy consumption of network interfaces in hand-held devices. *IEICE Transactions on Communications*, *80*(8), 1125–1131.
19. Tian, W., Li, J., Zhao, Y., Xue, C.J., Li, M., & Chen, E. (2011). Optimal task allocation on non-volatile memory based hybrid main memory. In *Proceedings of the 2011 ACM Symposium on Research in Applied Computation* (pp. 1–6): ACM.
20. Tian, W., Zhao, Y., Shi, L., Li, Q., Li, J., Xue, C.J., Li, M., & Chen, E. (2013). Task allocation on nonvolatile-memory-based hybrid main memory. Very Large Scale Integration (VLSI) Systems. *IEEE Transactions on*, *21*(7), 1271–1284.
21. Xue, C.J., Zhang, Y., Chen, Y., Sun, G., Yang, J.J., & Li, H. (2011). Emerging non-volatile memories: opportunities and challenges. In *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis* (pp. 325–334).
22. Zhang, W., & Li, T. (2009). Exploring phase change memory and 3d die-stacking for power/thermal friendly, fast and durable memory architectures. In *18th International Conference on Parallel Architectures and Compilation Techniques, 2009. PACT'09.* (pp. 101–112): IEEE.
23. Zhou, M., Bock, S., Ferreira, A.P., Childers, B., Melhem, R., & Mossé, D. (2011). Real-time scheduling for phase change main memory systems. In *2011 IEEE 10th International Conference on trust, Security and Privacy in Computing and Communications (TrustCom)* (pp. 991–998): IEEE.
24. Zhou, P., Zhao, B., Yang, J., & Zhang, Y. (2009). A durable and energy efficient main memory using phase change memory technology. In *ACM SIGARCH Computer Architecture News*, (Vol. 37 pp. 14–23): ACM.



Zhiyong Zhang received the M.E. and B.E. degree in the School of Computer Science and Technology at Shandong University, in 2013 and 2010, respectively. Now he is pursuing the Ph.D. degree. His main research interests include real-time and embedded systems, emerging nonvolatile memory, trust computing and mobile network.

Zhiping Jia received the Master and Ph.D. degree from the School of Computer Science and the School of Control Science, Shandong University, Jinan, China, in 1989 and 2007, respectively. From July 1989, he was with the Department of Computer Science and Technology at Shandong University. Since 2002, he has been a professor in the Department of Computer Science and technology at the Shandong University. He has published more than 70 research papers in refereed conferences and journals, and served as program committee members in numerous international conferences. He received Shandong Province Award, and Teaching Award.

Peng Liu received the M.E. degree in the School of Software Engineering, Shandong University, in 2013, and B.S. degree in the School of Information Science and Engineering, Shandong University, in 2011. He has worked in Qingdao Institute of Marine Engineering, Tianjin University since 2014, and he has been focusing on embedded systems, solid state storage and hardware design for years.

Lei Ju received his Ph.D. in 2010 from School of Computing, National University of Singapore. In 2011, He started working as an associate professor in School of Computer Science and Technology, Shandong University. His research interests focus on design, analysis and optimization of real-time systems and embedded networks. He has authored a number of referred publications (including DAC, RTAS, DATE, and CODES+ISSS), and served as the technical program committee of several international conferences.