# NPAM: NVM-Aware Page Allocation for Multi-Core Embedded Systems

Farimah R. Poursafaei, Mostafa Bazzaz, and Alireza Ejlali

**Abstract**—Energy consumption is one of the prominent design constraints of multi-core embedded systems. Since the memory subsystem is responsible for a considerable portion of energy consumption of embedded systems, Non-Volatile Memories (NVMs) have been proposed as a candidate for replacing conventional memories such as SRAM and DRAM. The advantages of NVMs compared to conventional memories are that they consume less leakage power and provide higher density. However, these memories suffer from increased overhead of write operations and limited lifetime. In order to address these issues, researchers have proposed NVM-aware memory management techniques that consider the characteristics of the memories of the system when deciding on the placement of the application data. In systems equipped with memory management unit (MMU), the application data is partitioned into pages during the compile phase and the data is managed at page level during the runtime phase. In this paper we present an NVM-aware data partitioning and mapping technique for multi-core embedded systems equipped with MMU that specifies the placement of the application data based on access pattern of the data and characteristics of the memories. The experimental results show that the proposed technique improves the energy consumption of the system by 28.10 percent on average.

**Index Terms**—Energy-efficient embedded systems, emerging technologies, hierarchical data placement, memory management

## 1 Introduction

Many embedded systems rely on batteries as the source of power which makes energy consumption an important aspect of embedded system design [1]. Considering that a large portion of energy is consumed by the memory subsystem [2], [3] memory architecture and memory management can have a significant effect on the energy consumption of embedded systems.

One of the promising approaches for reducing the energy consumption of embedded systems is using NVMs instead of conventional SRAM memory [2], [3], [4]. The leakage power dissipation of NVMs such as STT-RAM and PCM is much less than that of SRAM. Furthermore, they provide more capacity given the same die area as SRAM, which increases the on-chip memory capacity and could reduce the number of off-chip memory accesses if managed efficiently. However, these benefits come at the cost of increased write operation overhead and limited maximum number of write operations to a single cell (about $10^{13}$ for STT-RAM [5]).

To overcome the disadvantages of NVMs, many previous studies have proposed a hybrid memory configuration comprised of both conventional and non-volatile memories. For example, Bathen et al. [4] have presented a multi-core embedded system which is equipped with both SRAM and non-voltile memory. One of the most important design decisions in such systems is to find the best possible data allocation algorithm. Since each data is better suited for a specific memory type, the memory management must be aware of the characteristics of each memory and manage the memory space accordingly. For instance, write-intensive data are better suited for SRAM memories and read-intensive data are better suited for non-volatile memories.

Managing the memory could be performed at different granularity levels such as variable level [6], page level [4], [7], and task level [8]. In the variable-level management, each variable is mapped to one the memories. The page-level management performs the same operation for each page, and the task-level allocates the whole data of each task to one of the memories. The variable-level management provides the finest level of granularity but it suffers from problems such as pointer invalidation [9]. This approach is more suited for less complex systems where the memory management is directly implemented in the applications using compiler optimization or post-processor tools.

However, high-end systems with more complex operating systems such as Linux usually perform the allocation at the page level using MMU which allows for implementing data allocation techniques without requiring application recompilation. Furthermore, in this architecture the memory address translation (logical to physical) is transparent to the application which means that the application does not face pointer invalidation problem even if the page allocation is changed during the runtime.

In the page-level allocation, the memory management system, usually as a part of the operating system, allocates the pages to the memories based on the number of read and write accesses to each page. However, the number of read and write accesses to each page depend on variable-to-page mapping algorithm which decides on the placement of variables

• *The authors are with the Department of Computer Engineering, Sharif University of Technology, Tehran 1136511155, Iran. E-mail: {poursafaei, bazzaz}@ce.sharif.edu, ejlali@sharif.edu.*

*Manuscript received 30 Jan. 2017; revised 19 Apr. 2017; accepted 20 Apr. 2017. Date of publication 10 May 2017; date of current version 14 Sept. 2017. (Corresponding author: Alireza Ejlali.)*
*Recommended for acceptance by Z. Shao.*
*For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below*
*Digital Object Identifier no. 10.1109/TC.2017.2703824*

in each memory page. Several different parameters should be considered for the mapping of the variables to pages, including the size of the variables, and the number of read and write accesses to each variable. Neglecting any of these parameters may lead to an inefficient variable mapping in a hybrid memory configuration environment. For instance, one approach for variable mapping may only consider the size of the variables. In this approach, if write-intensive variables are scattered across pages, a considerable number of write accesses are directed toward the non-volatile memory which is not suitable for this kind of access and results in increased energy consumption and reduced lifetime of the system. Therefore, in a system with hybrid memory configuration, the variable-to-page mapping algorithm plays an important role.

In this paper we present an energy-efficient NVM-aware variable-to-page mapping technique for multi-core embedded systems with hybrid memory configuration that assigns data with similar access patterns to the same memory. Our proposed technique, called *NPAM* (**N**vm-aware **P**age **A**llocation for **M**ulticore embedded systems), comprises different phases. First, each application is profiled separately using a profiler that keeps track of the memory accesses of each variable. Then for each application, a clustering algorithm partitions variables to different sets based on the memory access statistics of each variable. Afterward, the variable-to-page mapping algorithm tries to assign the variables to pages such that variables from the same set do not scatter across different pages. Finally, the placement of the pages in each type of memory is determined.

To the best of our knowledge, this is the first research that presents an energy-efficient NVM-aware variable-to-page mapping technique for MMU-equipped embedded systems with hybrid memory configuration. Our main contributions are:

- Exploring the effects of variable-to-page allocation on the energy efficiency of MMU-equipped embedded systems with hybrid memory configurations.
- Presenting and evaluating an efficient variable-to-page mapping technique in terms of energy consumption, execution time, and lifetime of the system.
- Presenting an MILP-based oracle page mapping simulator for evaluating the proposed technique.

The rest of the paper is organized as follows. The related work is presented in Section 2. Section 3 introduces the system model. The proposed allocation technique is described in Section 4. The experiments and the evaluation results are presented in Section 5. Finally, Section 6 concludes the paper.

## 2 Related Work

Many previous studies have proposed program code or data allocation techniques for pure SRAM SPM in order to improve the energy consumption or performance of applications. The SPM allocation techniques can be roughly classified into static and dynamic techniques. Static allocation techniques [10], [11] determine the location of code or data at the compile time and the placement of the objects is not changed during the execution of the workload. Although these techniques incur low overhead and are easier to implement, they cannot adapt to the dynamic changes of the access pattern of the workload efficiently.

Dynamic allocation techniques [7], [12], [13], [14], [15], on the other hand, change the allocated locations of the code or data during the workload execution. The decision to load which object into what type of memory is usually made with the help of profiling techniques. Dynamic allocation is more efficient than static allocation, since it can better utilize the on-chip memory of the system. However, compared to static allocation techniques, dynamic allocation techniques require more complex analysis. In this paper, we propose a dynamic approach for allocating task-level data in a multi-core system.

Dynamic allocation of program code to SPM at page-level has been considered in [14] and [15]. Egger et al. [14] propose heuristic algorithms for dynamic placement of program code to SPM for single-core systems working with virtual memory that are equipped with MMU. In their approach, the program code executing from SPM is grouped into pages and during the runtime, MMU handles the transfer of the pages between SPM and the main memory. Cho et al. [7] present a page-level dynamic SPM allocation for function-level program data that is implemented by means of MMU's address translation schemes. [7] considers single-core embedded systems without any real-time constraints. The memory hierarchy of [7], [14], [15] is composed of volatile memory (i.e., SRAM and DRAM), so their allocation schemes are not applicable to NVM-based systems. In these researches the minimal transfer unit between the off-chip and the on-chip memory is an MMU page; therefore, the appropriate mapping of data to each page of the memory is of great importance.

Due to the attractive features of NVMs such as low leakage power, high density, and non-volatility [6], several studies have exploited them in the memory hierarchy [8], [16], [17], [18]. The problem of applying the pure SRAM SPM allocation techniques to an NVM-based memory system is that they do not consider the higher cost and limited admissible number of write activities on NVMs. Recently, it has been observed that hybrid memory configuration consisting of volatile and non-volatile memory helps in taking advantage of the low leakage power and high density of NVMs, while exploiting the symmetric read/write accesses and long lifetime of conventional memories [17], [18], [19].

Having in mind the benefits of combining the conventional memories and NVMs, the exploitation of NVMs as part of a hybrid main memory has been investigated in several researches [8], [16], [17], [18]. These studies propose data allocation techniques for improving the system efficacy in terms of energy consumption or lifetime of NVM. For example, Tian et al. [8], [16] propose task-level allocation schemes for systems with hybrid main memory consisting of DRAM and PCM with optimization objectives such as minimizing the energy consumption of the system and extending the lifetime of NVM. The task-level allocation schemes of [8], [16] assume that the data of each task should be allocated to contiguous memory address space. So, such schemes may result in underutilization of the memory.

In addition to exploiting NVM as main memory, it has been also employed as SPM and several different allocation techniques have been proposed to improve the energy or performance [6], [20], [21], [22], [23]. What distinguishes the allocation techniques of these studies from the ones proposed for pure SRAM SPM is that they are conscious of expensive write accesses of NVMs. Although these

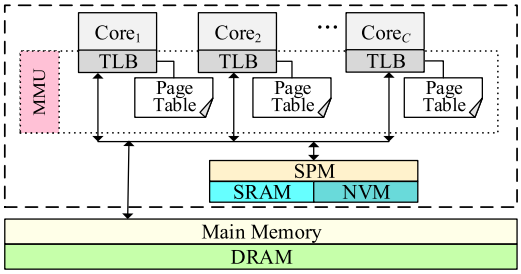


Fig. 1. System architecture.

techniques are effective in improving energy or performance of the system, they may suffer from some important challenges such as pointer aliasing, pointer invalidation, and object sizing, since they perform SPM allocation at variable-level [24]. In an MMU-equipped system, the transparent address translation mechanism together with the automatic memory allocation can resolve such problems.

In this paper we tackle the problem of allocating memory in multi-core embedded systems equipped with MMU which employ hybrid memory configuration. Our proposed technique allocates the tasks' variables at two distinct levels. First, the mapping of the variables of each task to virtual memory pages at variable-level is specified. Then, the placement of each virtual page in each available type of memory is determined at page-level.

## 3 System Model

In this section the system model is presented. First, the hardware model of the system is described in Section 3.1; then, the software model of the workload executing on the system is introduced in Section 3.2. Finally, in Section 3.3, the performance and energy model are presented.

### 3.1 Hardware Model

We target a multi-core embedded system consisting of $C$ processing cores whose on-chip memory is implemented as SPM. The architectural model of the system is shown in Fig. 1. Compared to the hardware-managed cache, the software-managed SPM is more efficient in terms of area, performance, and energy consumption [25]. We consider a hybrid SPM consisting of SRAM and NVM, while the main memory is pure DRAM. As is shown in Fig. 1, the hybrid SPM is shared among the cores. It should be noted that our allocation technique can be extended to the systems with hybrid main memory as well.

Since ARM processors are widely used in many embedded systems, we consider an ARM-based processor. ARM processors adopt a RISC architecture where only load/store instructions are allowed to access the memory [26]. In MMU-equipped systems, the virtual to physical address translation is managed by MMU and virtual addresses can be mapped to any one of the memories. Each memory type is characterized by its leakage power consumption, capacity, and cost of read and write accesses in terms of energy consumption and latency.

### 3.2 Software Model

We consider application specific resource-constrained embedded systems where the application's memory access pattern can be characterized at compile time. The application includes multiple tasks where each task's data variables can be classified into global, stack, or heap variables. The dynamic allocation of memory to heap variables in hard real-time systems is usually avoided, since it may cause predictability issues and may lead to program crashes [27], [28]. Therefore, in line with [7], [10], [11], [27], [28], [29], we consider global and stack variables in our proposed allocation technique.

The application workload comprises a set of $N$ independent hard real-time tasks $T=\{t_1, t_2, \ldots, t_N\}$ with a specific deadline $D$. The tasks are frame-based (similar to the task model in [30], [31]), and are scheduled non-preemptively. In a non-preemptive multitasking environment, tasks can share a common space for their stack variables [28], [29], [32]. Consequently, we can treat the global and stack variables in the same way. Each task $t_i$ from $T$ utilizes a set of $M_i$ variables: $Vars_i=\{V_{i(1)}, V_{i(2)}, \ldots, V_{i(M_i)}\}$. A variable $V_{i(j)}$ from $Vars_i$ is characterized by $V_{i(j)}=(R_{i(j)}, W_{i(j)}, S_{i(j)})$, where $R_{i(j)}$ represents the number of read accesses to $V_{i(j)}$, $W_{i(j)}$ denotes the number of write accesses to $V_{i(j)}$, and $S_{i(j)}$ represents the size of $V_{i(j)}$.

The objective of our proposed allocation technique could be either the minimization of the total execution time or total energy consumption of the task set. As the proposed memory allocation technique is independent of the tasks scheduling, it is assumed that the mapping of tasks to processing cores and the scheduling of tasks on each core are specified prior to finding the suitable memory allocation.

### 3.3 Performance and Energy Model

Considering the hardware and software model of the system, for evaluating the performance and energy consumption of the workload, we use the same models as the previous studies [20], [21], [23], [29]. In line with previous studies such as [20], [21], [23], [29], we focus our attention on energy consumption and execution time of accessing the memory which are affected by the data allocation. The execution time of task $t$, $T_{exe}(t)$, is calculated according to

$$T_{exe}(t) = CPU_{time}(t) + \sum_{u \in Data_t} (R_u \times L_R(m) + W_u \times L_W(m)), \quad (1)$$

where $Data_t$ is the set of data units (e.g., variable, page, etc.) of task $t$, $R_u$ and $W_u$ represent the number of read and write accesses of unit $u$, respectively. $L_R(m)$ and $L_W(m)$ denote the latency of read and write accesses of memory $m$ where $u$ is mapped. Finally, $CPU_{time}(t)$ is the number of cycles required for executing non-memory-related instructions of task $t$.

The dynamic energy consumption of task $t$ when accessing memory, $T_{DE}(t)$, is the sum of energies consumed by the memory accesses of $t$. Considering that $E_R(m)$ and $E_W(m)$ represent the energy of reading from or writing to a memory of type $m$, the dynamic energy consumption of $t$ is calculated according to

$$T_{DE}(t) = \sum_{u \in Data_t} (R_u \times E_R(m) + W_u \times E_W(m)). \quad (2)$$

The total energy consumption of the memory subsystem consists of dynamic and static energy. The dynamic energy of the memory, $TotalDE$, is the sum of the dynamic energy consumptions of all tasks running on the system ($T$)

TABLE 1
The Characteristics of Task $T_{13}$ of the Motivational Example

| Task Name | Variable Name | Number of Accesses | | Size of Variable (Bytes) |
|---|---|---|---|---|
| | | Read | Write | |
| $T_{13}$ | $V_1$ | 19,834 | 10,395 | 293 |
| | $V_2$ | 2,807 | 4,839 | 1,024 |
| | $V_3$ | 5,623 | 12,005 | 213 |
| | $V_4$ | 13,224 | 7,919 | 340 |
| | $V_5$ | 12,705 | 7,231 | 129 |
| | $V_6$ | 4,213 | 7 | 242 |
| | $V_7$ | 5,736 | 365 | 1,024 |

$$TotalDE = \sum_{t \in T} T_{DE}(t). \quad (3)$$

The static energy of the memory, $TotalSE$, depends on the memory leakage power consumption, $P_{Leak}(m)$, and the time duration which the memory module $m$ is being used

$$TotalSE = \sum_{m \in M} (CP \times P_{Leak}(m) \times ExeCycles), \quad (4)$$

where $CP$ denotes the clock cycle period in seconds and $ExeCycles$ specifies the total execution time of the system in cycles. The total energy consumption, $TotalEnergy$, can be calculated according to

$$TotalEnergy = TotalDE + TotalSE. \quad (5)$$

## 4 The Memory Allocation Technique

In this section our proposed memory allocation technique is presented. We first define the problem of allocating memory to tasks' data variables. Then, a motivational example is presented. Finally, our proposed solution to this problem is introduced in Section 4.3.

### 4.1 Problem Statement

Given the hardware configuration and the software model of the system, we aim to propose a memory allocation technique for the tasks' data variables in order to minimize either the energy consumption or total execution time of the task set. As the system is equipped with MMU, the minimum transfer unit between different memories is an MMU page. Therefore, the allocation problem is mainly divided into two subproblems. First, we should specify the allocation of the tasks' variables to virtual pages. Then, the mapping of virtual pages to each memory should be determined. Considering the different characteristics of each type of memory, various allocations of tasks' variables to virtual pages result in distinguished configurations for each task. From the set of all

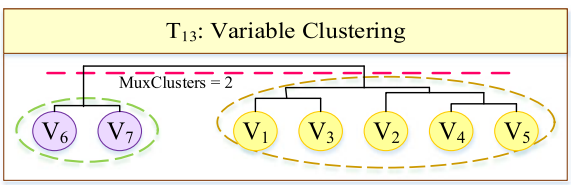


Fig. 2. Clustering the variables of $T_{13}$.

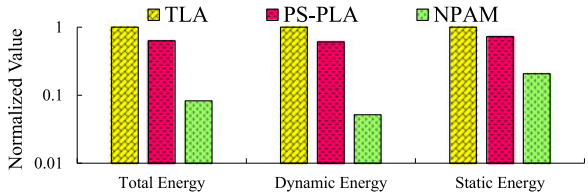


Fig. 3. The energy consumption of different memory allocation techniques. The results are normalized to TLA [8].

available configurations for a task, the best one that yields the best results should be selected, and the mapping of its pages to each type of memory should be determined with respect to different characteristics of each memory.

### 4.2 Motivational Example

This section provides a motivational example to illustrate the impact of the proposed technique on the tasks' energy consumption. In this example, we consider a task set consisting of six tasks executing on a dual-core system with hybrid SPM and a pure DRAM main memory. The characteristics of memories are according to $MemConfig_1$ which is demonstrated in Table 4. The memory pages are considered to be 1 KB. We assume that task $T_{ij}$ is the $j$th task on the $i$th core. In this example, NPAM is applied to minimize the energy consumption of the task set.

For evaluation purpose, we compare our technique to two different memory allocations: a task-level [8], and a page-level [7] technique. Hereafter, we refer to the **t**ask-**l**evel **a**llocation [8] and the **p**age-**l**evel **a**llocation for **p**ure **S**RAM SPM [7] as *TLA* and *PS-PLA*, respectively. The results of different memory allocation techniques are normalized to TLA [8].

Considering the characteristics of different memories, it may be more efficient to allocate the variables with similar access patterns to the same pages. Since the write accesses to NVMs are challenging, it is preferable to cluster variables with similar number of write accesses to the same groups; and then, map the variables in each group to virtual pages in order to map write-intensive variables to SRAM, while the read-intensive variables are mapped to NVM.

The characteristics of task $T_{13}$ are given in Table 1, and the clustering of its variables is shown in Fig. 2. This figure shows the first part of applying *Variable Clustering* phase of NPAM to $T_{13}$. Considering the average number of write accesses of each variable, a hierarchical tree of the variables is constructed. In this tree, the variables with more similar number of write accesses are closer to each other. The details of constructing the hierarchical tree are clarified in Section 4.3.1. The procedures of allocating the variables of each task to virtual pages are performed for each of the task.

The results of NPAM, PS-PLA [7], and TLA [8] are shown in Fig. 3. For each of the techniques, the execution time and dynamic energy of each task normalized to the results of TLA [8] are represented in Fig. 4. It can be observed that by grouping the variables of each task and considering the page-level allocation of a hybrid SPM, NPAM is able to exploit the memory space more efficiently. In other words, applying NPAM results in significant energy savings because of assigning the variables with more similar characteristics to the same pages, and considering the different characteristics of different parts of a hybrid SPM. This helps

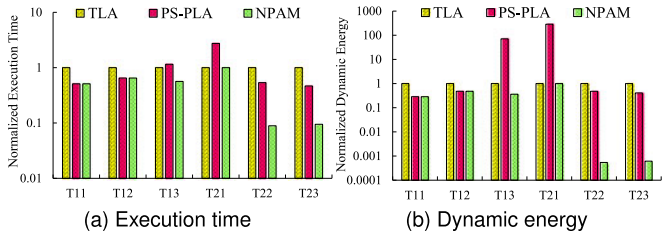


Fig. 4. The execution time and dynamic energy of each task when different allocation techniques are applied. The results are normalized to TLA [8].

to exploit the advantages of different memories while avoiding their disadvantages, and results in significant reduction of the energy of task sets when NPAM is applied.

It is noteworthy that when NPAM is applied, some tasks may consume more energy or spend more execution time compared to TLA [8] or PS-PLA [7], however the total energy consumption of the task set is reduced.

### 4.3 NPAM

Since the content of SPM needs to be managed explicitly, employing a proper data allocation technique is the key to achieve desirable efficiency in terms of energy and performance. An efficient allocation technique tries to bring the most accessed data to SPM in order to minimize the number of accesses to the off-chip memory which incurs more energy and latency.

It is assumed that the allocation of tasks to each core as well as the scheduling of tasks on each core are predetermined, and the content of the hybrid SPM can change dynamically upon the start of each task. The allocation technique should decide about the assignment of tasks' variables to the virtual pages, and then the mapping of virtual pages to each memory in order to minimize the energy consumption of the task set while respecting the deadline constraints. It should be noted that the optimal energy consumption of each task is calculated considering its average number of memory accesses. However, since it should be ensured that the timing constraints of the hard real-time tasks are not violated, the scheduling of the tasks is determined based on the worst case execution time (WCET) of each task and the WCET of each task is calculated based on the worst case memory accesses of the task. Considering that the WCETs of the tasks are used for the scheduling and each task starts exactly at its release time (determined by the scheduling algorithm), respecting the timing constraint is guaranteed and no unpredictable situations such as memory exhaustion or unpredictable tasks overlapping may occur.

The important note is that the NVM part of a hybrid SPM has asymmetric read and write accesses; i.e., the write accesses to NVM incur more cost in terms of energy and latency. Therefore, to improve the performance of the allocation technique, not only should the most accessed variables be brought to the on-chip memory, but also it is desirable to avoid the write-intensive variables to be mapped to NVM.

The outline of our proposed allocation technique, NPAM, is illustrated in Fig. 5. At first, each task is profiled in order to obtain the necessary information about its variables. Then in Variable Clustering phase, each task's variables are allocated to virtual pages in various ways that result in several different configurations for the task.

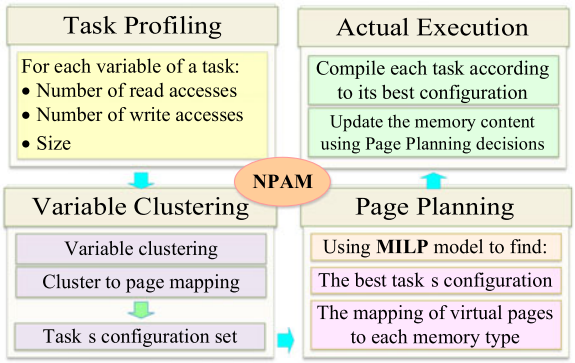


Fig. 5. Different phases of NPAM.

Afterward in *Page Planning*, the most beneficial configuration of each task is selected and the mapping of its virtual pages to each type of memory is decided. Finally, based on the results of the last two parts of the allocation technique, each task is compiled according to its best configuration, while the decisions about the mapping of the virtual pages to each memory are used at runtime in order to update the content of SPM upon the start of each task, dynamically. In the following sections, the major phases of the allocation technique are described in more details.

#### 4.3.1 *Variable Clustering*

Since embedded systems often serve a special purpose, their software applications are configured and installed at design time and rarely change thereafter [1], [33]. Hence, one way to take full advantage of SPM is through analyzing the embedded applications and configuring them according to the system requirements. Profiling the application programs gives the necessary information about the applications behavior according to which SPM can be managed effectively.

Therefore, the prerequisite step for our proposed allocation technique is the profiling of each task. Profiling offers useful information about the characteristics of each task's data variables. First, profiling determines the set of all variables of a task. Moreover, for each single variable of a task, the size of the variable, and the total number of read and write accesses are obtained by means of task profiling. These information about the variables of a task are fed to our allocation technique to find the most suitable memory for each variable.

After obtaining the required information about the variables of a task, the allocation of tasks' variables to virtual pages should be determined; this phase of NPAM is called Variable Clustering. The objective of Variable Clustering is to generate several different variables-in-page configurations from which the most beneficial one is selected by the next phase of NPAM. Hence, Variable Clustering determines different configurations specifying the different methods of allocating a task's variables to virtual pages.

Considering different attributes of memories and variables, several schemes may exist for the allocation of variables to virtual pages. Different kind of memories usually have diverse characteristics. Moreover, the set of all variables of a task may contain read-intensive or write-intensive variables. Regarding these dissimilar characteristics, Variable Clustering tries to group the variables of each task based on their

number of write accesses. The aim of the grouping is to put the most similar variables (in terms of the number of write accesses) in the same group while maximizing the dissimilarities between the members of different groups. For grouping the variables, we use *hierarchical clustering* algorithm [34]. This algorithm groups variables by creating a hierarchical cluster tree. The algorithm gives a multilevel hierarchy where a cluster at a higher level is generated by joining the clusters at lower levels. In this paper, we use agglomerative hierarchical clustering [34] that consists of the following steps:

- The similarity between every pairs of variables is calculated. In other words, the variation of the number of write accesses for every pair of variables of a task is computed.
- The variables (or pairs of variables) with close proximity are linked and grouped into a hierarchical cluster tree. In the process of making a cluster by pairing variables with similarity, hierarchical tree is generated by grouping the newly formed clusters into larger clusters.
- The hierarchical tree is cut into clusters at a specific height. In this step, the branches off the bottom of the hierarchical tree is pruned, and all of the variables below the cut are assigned to a single cluster. Cutting the hierarchical tree at different points leads to different partitioning schemes of the variables. Furthermore, each partitioning scheme is equivalent to a specific configuration of each task's variables.

After generating the hierarchical cluster tree for the set of all variables of a task, the tree is cut at every different heights. At each distinct height where the hierarchical cluster tree is cut, a specific clustering of the variables is generated. There is a tradeoff between the number of clusters and the similarity between the variables placed in each cluster. The more the number of clusters, the more the similarity between the variables in each cluster. The ultimate case is when each single variable is placed in a single cluster. The minimum number of clusters is one which happens when all variables with their diverse characteristics are placed in one single group.

It should be noted that in the process of grouping the variables based on the similarity of their write accesses, the size of the variables is not considered. So, it is not ensured that the variables in each cluster can be placed in a specific number of pages. Thus, the second step of Variable Clustering is to map the variables in each cluster to the minimum number of virtual pages.

Therefore, for each specific height where the tree is cut, Variable Clustering confronts a *knapsack problem* for each cluster to map its variables to virtual pages; it is desired to allocate variables to minimum number of pages in order to minimize the memory usage and reduce the page transfers between different memory types. Thus, for each single cluster generated by cutting the hierarchical tree at any specific height, there exists a set of variables with different sizes that need to be mapped to virtual pages with specific size $P_s$. So, the encountered knapsack problem is as follows:

"Each cluster has several variables with different sizes that must be placed in virtual pages with size $P_s$. The objective of the knapsack problem is to map the variables of a cluster to virtual pages in such a way that minimizes the total number of pages."

For solving this knapsack problem, we use *Best-Fit-Decreasing (BFD)* heuristic algorithm [35]. BFD is employed for each cluster at each specific cut of the hierarchical cluster tree to generate the mapping of the tasks' variables to virtual pages with specific size.

At the end of Variable Clustering, each task has a number of different configurations. Each configuration is equivalent to a different scheme of partitioning the task's variables based on their number of write accesses. All of the different configurations for a task are handed to the next phase of NPAM. From the set of all available configurations for a task, the most appropriate configuration is selected and the mapping of its virtual pages to the memories is determined by the second phase of NPAM.

The pseudo code of Variable Clustering is presented in Algorithm 1. In line 1, Variable Clustering starts. In line 2 through 15, the algorithm clusters the variables and generates different configurations for the variables of each task. Specifically, in lines 5 to 6, the algorithm clusters the variables based on their average number of write accesses. The upper bound on the maximum number of clusters for dividing the variables is equal to the total number of variables. Then, in line 7 to 12, the algorithm allocates the variables of each cluster to virtual pages considering the size of the variables in each cluster by the use of BFD algorithm. Each of the generated configuration is stored in the set of the task's configurations.

**Algorithm 1.** Variable Clustering

**Input:**
- a set of $N$ tasks $T$=$\{t_1, t_2, \ldots, t_N\}$.
- for each task $t_i$ from task set $T$, a set of $M_i$ variables $Vars_i = \{V_{i(1)}, V_{i(2)}, \ldots, V_{i(M_i)}\}$.
- the characterization of each variable $V_{i(j)}$ from $Vars_i$ that is denoted as a triple $V_{i(j)}$=$(R_{i(j)}, W_{i(j)}, S_{i(j)})$.
- $P_s$: memory page size.

**Output:**
- for each task $t_i$ from task set $T$, a set of $c_i$ configurations $Confs(t)$=$\{conf_{i(1)}, conf_{i(2)}, conf_{i(c_i)}\}$

```
1:  start Variable Clustering
2:  // making tasks' configurations
3:  for each t_i from T do
4:     // clustering based on the number of write accesses
5:     for k_W = 1 to M_i do
6:        GW_{k_W} ← cluster Vars_i into k_W groups
7:        for m = 1 to k_W do
8:           if GW_{k_W} is not empty then
9:              // Considering the size of variables in GW_{k_W},
                // applying the BFD algorithm to assign theses
                // variables to virtual pages with size P_s
10:             conf_{i(k_W)} ← BFD(P_s, GW_{k_W})
11:          end if
12:       end for
13:    end for
14: end for
15: end Variable Clustering
```

#### 4.3.2 Page Planning

Page Planning is in charge of selecting the best configuration of each task and determining the mapping of its virtual pages to each memory in order to optimize the allocation objective given the predefined mapping and scheduling of

tasks on processing cores. Indeed, Page Planning phase of NPAM plans about the management of the memory contents for the actual execution of the workload. Using the decisions made by Page Planning, the tasks are compiled according to their best configurations, and the content of the memory is updated during the execution of the tasks.

According to the predefined mapping and scheduling of the tasks on the processing cores, finding the best configurations and determining the mapping of their virtual pages to different memory types are performed by a *Mixed Integer Linear Programming (MILP)* model which is presented in the next section. From the set of all available configurations for a task, the one that optimizes the allocation objective is selected by the MILP model. The allocation objective could be the minimization of either the energy consumption or the execution time. Moreover, the MILP model determines the mapping of each page of the task's selected configuration to each memory. It is important to note that the MILP model is only used offline at design time and the results of the decisions about page mapping is used at run time to dynamically transfer memory pages between multiple memories upon the start of each tasks.

**Algorithm 2.** Page Planning

**Input:**
- the set of all configurations $Confs(t)$ for each task $t_i$
- *obj*: the optimization objective
- system configurations

**Output:**
- *bestConfigs*: The set of best configurations for the tasks
- *pagePlacement*: A set of page placements where each item $pagePlacement[t_i]$, represents the placement of each page of $bestConfs[t_i]$ for task $t_i$ in each type of memory

1: **start Page Planning**
// According to the *obj*, considering *the tasks to core*
// *allocation* and *task scheduling on each core*
2: **for** each task $t_i$ from task set $T$ **do**
3: use a *MILP* model to optimize *obj* as follows:
- $bestConfs[t_i] \leftarrow$ find the best configuration from the set of all available configurations $Confs(t)$ for $t_i$
- $pagePlacement[t_i] \leftarrow$ find the most appropriate memory for each virtual page of $bestConfs[t_i]$

4: **end for**
5: **end Page Planning**

The pseudo code of Page Planning phase of NPAM is shown in Algorithm 2. In lines 2 to 4, for each task of the task set, the algorithm uses the MILP model to select the best configuration, and also to specify the mapping of each virtual page of the selected configuration to each memory.

For transferring pages between the different physical memory locations, our proposed memory allocation technique employs conventional MMU mechanisms. It is important to note that the unpredictability associated with employing MMU has two sources [36]. First, the duration of address translation may be unpredictable, since each task may swap TLB entries or pages of other tasks. The second source of unpredictability stems from the difficulty of predicting whether a reference to a virtual page causes a page fault. However, In line with the predictable MMU proposed in [37], we exploit virtual memory in a way to preserve the predictability of the tasks. The details of the runtime tasks execution are as follows.

With the knowledge resulted from solving the MILP model, one page-table is constructed per task. The page-table includes information related to the best mapping of each task's virtual pages to different memories. NPAM allocates SPM pages to the tasks in such a way to ensure that the memory allocation cannot exceeds the maximum size of memories. Deciding about the mapping of tasks to each type of memory is performed offline at design time. During the execution of each task, its page-table is entirely stored in the hardware in order to minimize the penalty of a TLB miss. It should be noted that each task's page-table consists of an entry for each virtual page which is mapped to the SPM at run time. On the other hand, the pages specified by the MILP model to be mapped to the off-chip memory are pinned at fixed locations of the main memory. Hence, only the mapping of the pages that are denoted to be executed from SPM needs to be stored in the task's page-table. Therefore, it is guaranteed that each task's page-table can completely fit in the hardware page table.

At the start of each task, the process of task set-up takes place: First, the task's required resources (e.g., SPM pages) are allocated. Then, the physical regions of SPM are bounded to the task's virtual memory pages. The final step is the actual loading of the task and then, the task is ready for the execution.

Indeed, the two important actions at the start of each task include flushing the TLB entries and loading the task's page-table into the hardware page-table. These actions ensure that each task executes regardless of other tasks. Moreover, it is guaranteed that no page-faults can occur during the execution of the task. Although the task may experience some (cold) misses with low penalty (as the page table is stored in the hardware), the behavior of the task is independent from other tasks, so the pages' address translations are predictable. Moreover, no page replacement may occur during the execution of a task and no page faults can occur. Therefore, it is ensured that the predictability is preserved.

Page Planning phase of NPAM is implemented as an MILP model. The proposed MILP model is introduced in the next section.

#### 4.3.3 MILP Modeling

In this section the MILP formulation for finding the best possible tasks' configurations and the best page mapping is presented. Table 2 shows the notation used for variables throughout this section.

Among all different configurations of a task $t$, one and only one of them should be selected; it is ensured by

$$\forall t \in T : \sum_{c \in Confs(t)} T_{conf}(t,c) = 1. \tag{6}$$

Also, we must make sure that if a task configuration is not selected, none of its pages are assigned to memories

$$\forall t \in T, \forall c \in Confs(t), \forall p \in P(c) : \sum_{m \in M} Page(t,c,p,m) = T_{conf}(t,c). \tag{7}$$

TABLE 2
MILP Notation

| Notation | Description |
|---|---|
| $T_s(t)$ | Start time of task $t$ (in cycles) |
| $T_f^{ac}(t)$ | Average case finish time of task $t$ (in cycles) |
| $T_f^{wc}(t)$ | Worst case finish time of task $t$ (in cycles) |
| $T_{exe}^{ac}(t)$ | Average case execution time of task $t$ (in cycles) |
| $T_{exe}^{wc}(t)$ | Worst case execution time of task $t$ (in cycles) |
| $T_{exe}^{ac}(t,c)$ | Average case execution time of task $t$ using configuration $c$ (in cycles) |
| $T_{exe}^{wc}(t,c)$ | Worst case execution time of task $t$ using configuration $c$ (in cycles) |
| $T_{DE}(t)$ | Dynamic energy consumption of task $t$ (in Joules) |
| $T_{DE}(t,c)$ | Dynamic energy consumption of task $t$ using configuration $c$ (in Joules) |
| $Page(t,c,p,m)$ | Indicates whether page $p$ of configuration $c$ of task $t$ is mapped to memory $m$ |
| $T_{conf}(t,c)$ | Indicates whether configuration $c$ of task $t$ is selected |
| $T_{size}(t,m)$ | Memory usage of task $t$ on memory $m$ |
| $T_{ovl}(t,t')$ | Indicates whether task $t$ and task $t'$ overlap |
| $C_{exe}^{ac}(c)$ | Average case total execution time of core $c$ (in cycles) |
| $C_{exe}^{wc}(c)$ | Worst case total execution time of of core $c$ (in cycles) |
| $C_{DE}(c)$ | Total memory dynamic energy of core $c$ (in Joules) |
| $ExeCycles^{ac}$ | Average case total execution cycles of the system |
| $ExeCycles^{wc}$ | Worst case total execution cycles of the system |
| $TotalEnergy$ | Total memory energy consumption (in Joules) |
| $TotalDE$ | Total dynamic energy consumption (in Joules) |
| $TotalSE$ | Total static energy consumption (in Joules) |
| $P_{Leak}(m)$ | Leakage power of memory $m$ (in Watts) |
| $Page_{lat}^{ac}(p,c,m)$ | Average case latency of memory accesses of page $p$ of configuration $c$ on memory $m$ (in cycles) |
| $Page_{lat}^{wc}(p,c,m)$ | Worst case latency of memory accesses of page $p$ of configuration $c$ on memory $m$ (in cycles) |
| $Page_{dyn}(p,c,m)$ | Dynamic energy of memory accesses of page $p$ of configuration $c$ on memory $m$ (in Joules) |
| $MPage(p,m,t)$ | Whether page $p$ of memory $m$ is assigned to task $t$ |
| $MP(m)$ | Set of the pages of the memory $m$ |
| $CPU_{time}^{ac}(t)$ | Average case execution time of non-memory-related instructions of task $t$ (in cycle) |
| $CPU_{time}^{wc}(t)$ | Worst case execution time of non-memory-related instructions of task $t$ (in cycle) |
| $P(c)$ | Set of pages of configuration $c$ |
| $Confs(t)$ | Set of configurations of task $t$ |
| $Core(t)$ | The core where task $t$ is scheduled to execute |
| $T(c)$ | Set of tasks scheduled to execute on core $c$ |
| $CP$ | Clock cycle period in seconds |
| $M$ | Set of the memories of the systems |
| $C$ | Set of the cores of the system |
| $T$ | Set of the tasks of the system |
| $D$ | The deadline of the system (in cycles) |

The average execution time and the worst case execution time of each task under each configuration are calculated using (8) and (9), respectively. The execution time is divided into two parts: *memory-related* and *non-memory-related*. The former is caused by memory accesses and the later ($CPU_{time}^{ac}(t_i)$ and $CPU_{time}^{wc}(t_i)$) is caused by executing instructions other than load/store

$$\forall t \in T, \forall c \in Confs(t): \quad T_{exe}^{ac}(t,c) = CPU_{time}^{ac}(t_i) \times T_{conf}(t,c) + \sum_{p \in P(c)} \sum_{m \in M} (Page_{lat}^{ac}(p,c,m) \times Page(t,c,p,m)) \tag{8}$$

$$\forall t \in T, \forall c \in Confs(t): \quad T_{exe}^{wc}(t,c) = CPU_{time}^{wc}(t_i) \times T_{conf}(t,c) + \sum_{p \in P(c)} \sum_{m \in M} (Page_{lat}^{wc}(p,c,m) \times Page(t,c,p,m)). \tag{9}$$

In (8) and (9), $Page_{lat}^{ac}(p,c,m)$ and $Page_{lat}^{wc}(p,c,m)$ are calculated beforehand by (1) using the average number of reads and writes and the worst case number of reads and writes to page $p$ of configuration $c$ assuming that the page is mapped to memory $m$.

To calculate the average and worst case execution time and finish time of each task we present (10), (11), and (12)

$$\forall t \in T: T_{exe}^{ac}(t) = \sum_{c \in Confs(t)} T_{exe}^{ac}(t,c), T_{exe}^{wc}(t) = \sum_{c \in Confs(t)} T_{exe}^{wc}(t,c) \tag{10}$$

$$\forall t \in T: T_f^{ac}(t) = T_s(t) + T_{exe}^{ac}(t) \tag{11}$$

$$\forall t \in T: T_f^{wc}(t) = T_s(t) + T_{exe}^{wc}(t). \tag{12}$$

All tasks scheduled on a core must execute sequentially which means the start time of each task must be greater than or equal to the finish time of its predecessor

$$\forall c \in C, \forall t, t' \in T(c),\ t' \ executes\ right\ after\ t: \quad T_s(t') \geq T_f^{wc}(t). \tag{13}$$

The energy consumption of each task is divided into two parts: the dynamic portion and the static portion. The former is calculated by

$$\forall t \in T, \forall c \in Confs(t): T_{DE}(t,c) = \sum_{p \in P(c)} \sum_{m \in M} Page_{dyn}(p,c,m) \times Page(t,c,p,m), \tag{14}$$

where similar to $Page_{lat}^{ac}$, $Page_{dyn}$ is also calculated based on the average number of accesses to the page $p$ using (2).

The total memory dynamic energy consumption of task $t$ is calculated by accumulating the dynamic energy consumption of all configurations of $t$

$$\forall t \in T: T_{DE}(t) = \sum_{c \in Confs(t)} T_{DE}(t,c). \tag{15}$$

In order to detect whether two tasks overlap we use

$$\forall c, c' \in C | c \neq c', \forall t \in T(c), \forall t' \in T(c'): \quad T_{ovl}(t,t') = (T_s(t) \leq T_s(t')) \wedge (T_f^{wc}(t) > T_s(t')), \tag{16}$$

where $T_{ovl}(t,t')$ is set to one if tasks $t$ and $t'$ overlap during the execution.

To ensure that the memory allocation does not exceed the maximum capacity of the memories, we add (17) and (18)

$$\forall m \in M, \forall p \in MP(m), \forall t, t' \in T: (MPage(p,m,t) \wedge MPage(p,m,t')) \Rightarrow T_{ovl}(t,t') = 0 \tag{17}$$

$$\forall m \in M, \forall t \in T: \sum_{c \in Confs(t)} Page(t,c,p,m) = \sum_{p \in MP(m)} MPage(p,m,t), \tag{18}$$

where $MPage(p, m, t)$ represents whether page $p$ of memory $m$ is assigned to task $t$. Equation (17) dictates that if two tasks use the same memory page, they cannot run simultaneously.

The execution time of each core is set to the finish time of the last task scheduled to execute on that core

$$\forall c \in C,\ t\ is\ the\ last\ task\ on\ c : C^{ac}_{exe}(c) = T^{ac}_{f}(t) \tag{19}$$

$$\forall c \in C,\ t\ is\ the\ last\ task\ on\ c : C^{wc}_{exe}(c) = T^{wc}_{f}(t). \tag{20}$$

The total execution time of the system is the maximum value of the execution times of the cores, and the worst case execution time must not exceed the deadline of the system

$$\forall c \in C : ExeCycles^{ac} \geq C^{ac}_{exe}(c),\ ExeCycles^{wc} \geq C^{wc}_{exe}(c) \tag{21}$$

$$ExeCycles^{wc} \leq D. \tag{22}$$

As for the energy consumption of the system, the dynamic portion is calculated by accumulating the dynamic energy consumption of the tasks scheduled on each core

$$\forall c \in C : C_{DE}(c) = \sum_{t \in T(c)} T_{DE}(c, t) \tag{23}$$

$$TotalDE = \sum_{c \in C} C_{DE}(c). \tag{24}$$

The total static energy consumption of the system is sum of the products of the average system execution time and the static power consumption of each memory

$$TotalSE = \sum_{m \in M} (CP \times P_{Leak}(m) \times ExeCycles^{ac}). \tag{25}$$

Since $ExeCycles^{ac}$ represents the average execution time of the system in cycles, it is multiplied by the clock period to calculate the execution time in seconds.

Finally, the total energy consumption is calculated by accumulating the total dynamic energy and the total static energy of the system

$$TotalEnergy = TotalDE + TotalSE. \tag{26}$$

The objective is to minimize $TotalEnergy$.

## 5 Experiments

In this section the results of evaluations are demonstrated. First, the system configurations are introduced in Section 5.1. Then in Section 5.2, the effects of NPAM on the energy consumption of the task sets are investigated. Finally, the impacts of Variable Clustering are evaluated in Section 5.3.

### 5.1 Experimental Setup

To evaluate the performance of NPAM, we considered embedded systems equipped with MMU whose memory hierarchies include a hybrid SPM consisting of SRAM and STT-RAM, and a pure DRAM main memory. Among all different kinds of technologies available for NVMs, we chose STT-RAM, since it is an appropriate candidate to be used as part of the on-chip memory due to its desirable characteristics in terms of lifetime, access latency, and power consumption [29]. Nevertheless, other NVMs can be used as well.

To estimate the read/write access latency and energy consumption of a given size of each type of the memories, we used NVSim [38]. An in-house simulator that implements the MILP solution described in Section 4.3.3 was used to simulate the state of the system at run time, and to evaluate the efficiency of NPAM. We considered the requirements of adopting predictable MMU in our simulations according to [37]. The overheads of the extra assembly instructions executed during the context switches are added to the execution times and energy consumptions of the tasks. Furthermore, the TDMA policy of MMU is taken into account in analyzing the memory accesses during the simulations. The page size of the memory is considered to be 1 KB (in line with the supported page size of ARM platform [26], and similar researches [7], [33]). All of the processing cores of the system are homogeneous and work at the same frequency equal to 1 GHz. Although NPAM can be applied in order to minimize either the energy consumption or the execution time of the task set, in these experiments the objective was to reduce the energy consumption. The task allocation and scheduling are predetermined.

For investigating the performance of NPAM, two sets of experiments have been evaluated. In the first set, the efficiency of NPAM is compared to two of the most relevant memory allocation techniques. In the second set of experiments, the effects of clustering the variables of each task on the energy consumption and lifetime of the system are evaluated. In our simulations, the MILP solution time varied between 30 seconds and 72 hours. On average, for a task set consisting of 27 tasks, the MILP takes 23 hours on a system with *Intel Xeon E5-2690v2* processor to find the solution using *Gurobi v7* [39] as the MILP solver.

### 5.2 The Effects of NPAM on Energy Consumption

In this section, the effects of applying NPAM are compared to two distinct memory allocation techniques: TLA [8] and PS-PLA [7]. TLA [8] is proposed for determining the allocation of the tasks' data to each part of a hybrid main memory consisting of DRAM and NVM. The objective of TLA [8] is to minimize the energy consumption of the task set while satisfying the timing constraints associated with the tasks. We modified the heuristic memory allocation of TLA [8] to get the efficient allocation for a memory hierarchy consisting of DRAM main memory and hybrid SPM. On the other hand, PS-PLA [7] manages the memory at page-level for the data of each function of a program. The memory hierarchy considered in [7] includes DRAM main memory and pure SRAM SPM, and the goal is to minimize the energy consumption of a single-core system equipped with MMU. It is important to note that PS-PLA [7] does not propose a scheme for determining the mapping of the variables to virtual pages. We extended PS-PLA [7] in order to get the allocation scheme for a multi-core system where the tasks have real-time constraints.

In this section, we considered a quad-core embedded system equipped with MMU whose memory configuration is illustrated in Table 3. The workload consists of 27 embedded and real-time applications adopted from different categories of Mibench [40] and Mälardalen [41] benchmark

TABLE 3
The Memory Configuration for Task Set Consisting of Benchmark Tasks

| Characteristic | Memory Technology | | |
|---|---|---|---|
| | DRAM | SRAM | STT-RAM |
| Size | 4 MB | 4 KB | 16 KB |
| Read Access Latency (ns) | 62 | 2.32 | 1.79 |
| Write Access Latency (ns) | 62 | 2.32 | 11.21 |
| Read Access Energy (pJ) | 28,600 | 18.20 | 31.60 |
| Write Access Energy (pJ) | 28,600 | 7.50 | 77.75 |
| Leakage Power (mW) | 78.75 | 2.69 | 0.35 |

suits which are representatives of typical embedded applications. We ran each benchmark program in SimpleScalar [42] tool chain to collect the programs' traces. Each benchmark task has at least twenty different sets of input data. We profiled each task with a subset of all possible input sets that was randomly selected, considering that it is not usually possible to profile a given task for all feasible inputs. However, all sets of input data were used during the evaluation phase to gain more realistic results. The trace of each program consists of the information about different variables used in each program; e.g., the size of each variable as well as the average number of read and write accesses to each variable. NPAM uses these traces and the system configuration to decide on the memory allocation. Then, the findings are used by our MILP-based simulator when investigating the state of the system at runtime.

The normalized energy consumption of the task set for different memory allocation techniques is shown in Fig. 6, and the normalized execution time and dynamic energy of each task of the task set are represented in Fig. 7. Fig. 6 demonstrates that the total energy consumption of the task set when NPAM is applied is significantly reduced compared to other memory allocation techniques. PS-PLA [7] does not consider the different characteristics of the different parts of the hybrid SPM, and does not propose a scheme for

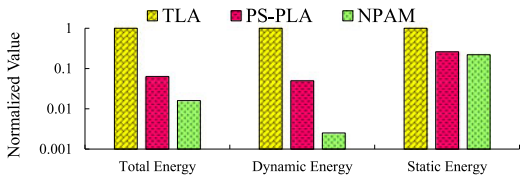


Fig. 6. The energy consumption of different memory allocation techniques for the task set consisting of benchmark tasks. The results are normalized to the results of TLA [8].

determining the mapping of the variables to pages. Therefore, pages with high write accesses may be mapped to NVM part of SPM, which negatively affect the energy consumption and lifetime of NVM. Moreover, the SPM space is partitioned among the cores in PS-PLA [7], and each core can only exploit its own share of SPM. So, if a core does not use its share of SPM completely, other cores cannot exploit that core's unused share. In this case, although there may be empty space on SPM, some tasks are forced to map their pages to off-chip memory. On the other hand, NPAM differentiates between SRAM and NVM parts of SPM and maps tasks' pages to the more suitable part. Moreover, since NPAM is designed for multi-core systems, it determines the memory allocation in order that each core can exploit the whole SPM space. Therefore, it is observed that NPAM is more energy efficient than PS-PLA [7].

It can be also observed that NPAM is more energy efficient than TLA [8] due to its finer allocation granularity. TLA [8] allocates the whole data of each task to one type of the memory. So, there may be some tasks that cannot be placed in their desired memory due to the capacity constraints. Moreover, the access pattern of the whole data of a task may not exactly suit one memory type. On the other side, NPAM maps the pages of each task to each memory which suites the access pattern of that page; this leads to the allocation of memory in a more efficient way.

As shown in Fig. 7, for majority of the tasks the dynamic energy and execution time of the cases where NPAM is

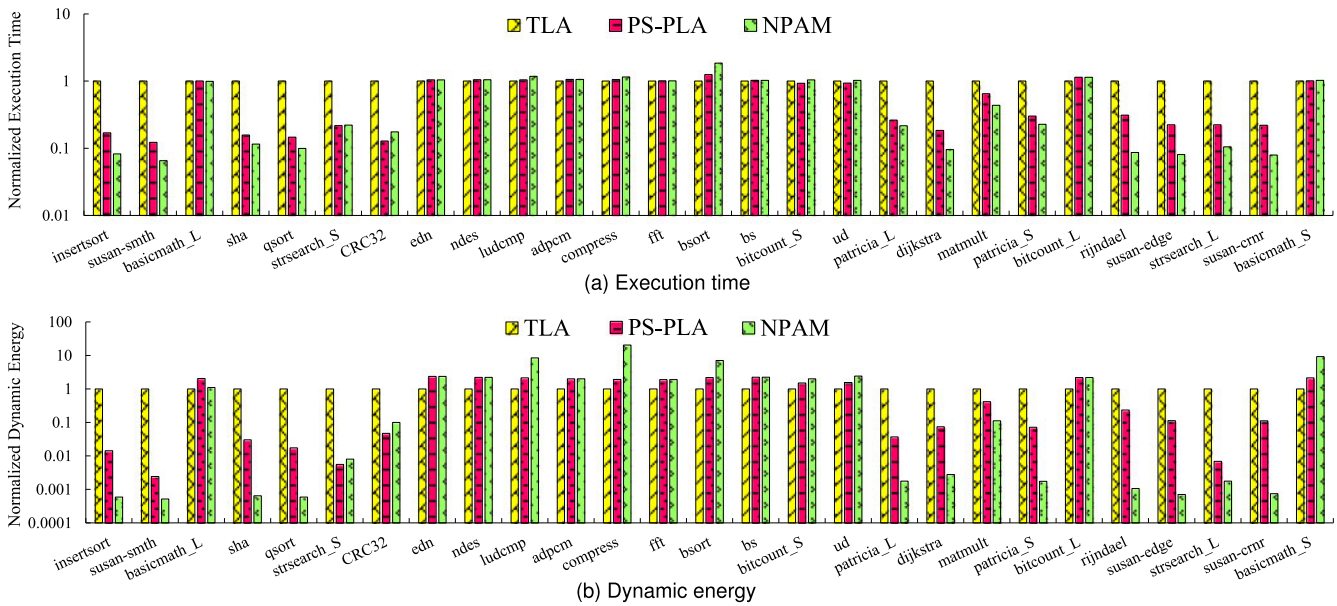


Fig. 7. The normalized execution time and dynamic energy of the benchmark tasks when different memory allocation techniques are applied. The results are normalized to TLA [8].

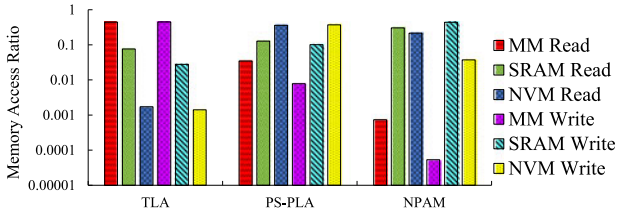


Fig. 8. The memory access pattern of different memory allocation techniques for the task set consisting of benchmark tasks.

applied is less than those of other allocation techniques. There are just few exceptional tasks that consumes more energy or execution time (e.g., *bsort*). Those exceptional cases may happen when some tasks run in parallel and compete for the shared resources; in this situation, by trying to optimize the allocation objective, NPAM prioritizes the tasks with dominant effect on the energy or execution time. Consequently, the tasks with dominant impact on the energy or time are given the more appropriate memories, while less effective tasks may consume more energy or time due to the forced condition in which they should use the less suitable memories. Considering that the objective of NPAM is to minimize the energy consumption of the whole task set, the extra energy or execution time for some individual tasks can be compensated.

When exploiting NVM as part of the memory hierarchy, the lifetime of NVM should be considered. Fig. 8 depicts the access pattern of the task set to different parts of the memory hierarchy. From Fig. 8, it can be observed that NPAM is able to utilize the on-chip memory more efficiently, since the ratio of main memory accesses have reduced. In addition, the write access ratio of NVM for NPAM (compared to PS-PLA [7]) is reduced, which results in improvement of NVM lifetime. The success of NPAM in reduction of main memory as well as NVM write accesses lead to more energy savings of NPAM.

The NVM lifetime improvement ratio of one memory allocation technique compared to another memory allocation technique, $LT_{imp}$, can be calculated according to (27) [18], [43]; where $M$ denotes the maximum number of erase count of NVM (about $10^{13}$ for STT-RAM [5]), and $W_1$ and $W_2$ represent the total number of write accesses to NVM using either memory allocation techniques

$$LT_{imp} = \frac{\frac{M}{W_2} - \frac{M}{W_1}}{\frac{M}{W_1}}. \quad (27)$$

By using (27), it is found that NPAM extends the lifetime of NVM by a factor of 8.92 compared to the case where PS-PLA [7] is applied. Compared to TLA [8], by applying NPAM the lifetime of NVM reduced by a factor of 0.77. This is due to the fact that in TLA [8], there may be some tasks whose size is larger than the size of the on-chip memory and cannot be placed in their appropriate memory due to the capacity constraints. By mapping less number of tasks to NVM, TLA [8] leads to less number of accesses to NVM and longer lifetime, but this longer lifetime is because of the incompetence of TLA [8] in efficient utilization of NVM due to the capacity constraints.

## 5.3 Variable Clustering Effects

The purpose of this set of experiments is to further investigate the impacts of variable clustering on the energy consumption of the system. We assessed various system architectures including dual-core and quad-core systems, each of which was evaluated with two different memory configurations: $MemConfig_1$ and $MemConfig_2$ which are illustrated in Table 4. These memory configurations are in line with the configurations adopted in [22], [29].

TABLE 4
The Memory Configurations for Task Sets Consisting of Synthetic Tasks

| | Characteristic | Memory Technology | | |
|---|---|---|---|---|
| | | DRAM | SRAM | STT-RAM |
| $MemConfig_1$ | Size | 4 MB | 1 KB | 4 KB |
| | Read Access Latency (ns) | 62 | 1.27 | 1.28 |
| | Write Access Latency (ns) | 62 | 1.27 | 10.72 |
| | Read Access Energy (pJ) | 28,600 | 8.10 | 5.92 |
| | Write Access Energy (pJ) | 28,600 | 3.94 | 48.91 |
| | Leakage Power (mW) | 78.75 | 0.70 | 0.13 |
| $MemConfig_2$ | Size | 4 MB | 2 KB | 8 KB |
| | Read Access Latency (ns) | 62 | 2.32 | 1.76 |
| | Write Access Latency (ns) | 62 | 2.32 | 11.18 |
| | Read Access Energy (pJ) | 28,600 | 18.20 | 7.44 |
| | Write Access Energy (pJ) | 28,600 | 7.50 | 53.58 |
| | Leakage Power (mW) | 78.75 | 2.69 | 0.173 |

In this section, as a *baseline* scheme of assigning the variables of a task to memory pages, we considered the case where the variables are not grouped and no modification on the assignment of the task's variables to pages has been applied, similar to the allocation scheme of the GCC compiler [44]. It should be noted that the mapping of virtual pages of the baseline configurations to each memory was also determined by MILP model of the Page Planning phase. To the best of our knowledge, this is the first research on exploring the effects of variable-to-page mapping in hard real-time multi-core embedded systems with hybrid memory configuration. Therefore, we chose the variable-to-page mapping scheme of the GCC compiler as the baseline with which we compared our proposed technique. The results of the cases where NPAM is applied are compared to the baseline.

### 5.3.1 Workload Generation

We performed extensive simulations to evaluate the impacts of Variable Clustering phase of NPAM on the energy consumption. In these experiments, the workload consists of synthetic task sets in order to eliminate the influence of specific applications on the results and to be able to investigate the performance of NPAM more generally. For task set generation, we employed the approach in [45] which ensures to generate tasks that are parameter independent and unbiased. In each set of experiments, *three* or *six* tasks can be assigned to each core. The total utilization of each core is equal to 0.7 and the individual utilization of each task is generated using *randfixedsum* algorithm [45]. The size of the frames is generated randomly between $8 \times 10^6$ and $10^7$ cycles using uniform distribution. For each case of the simulation, we considered 10 distinct randomly generated task sets for profiling phase. Then for the evaluation phase, another 10 sets of tasks were generated with the similar characteristics. The reported results include the average results of the corresponding task sets used for the evaluations.

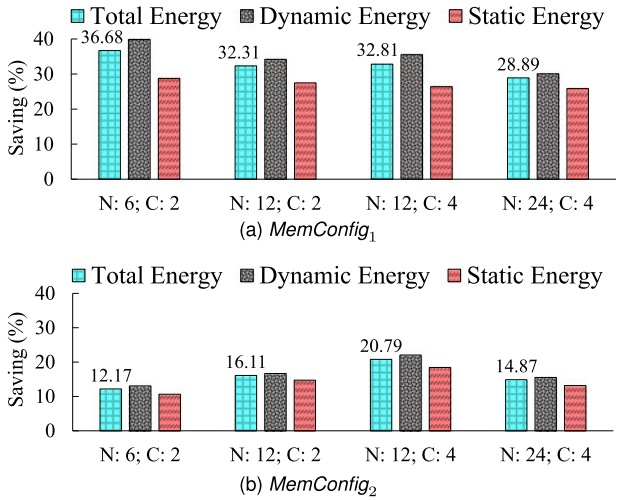


Fig. 9. Results of applying NPAM compared to the baseline when $AC/WC = 1$. The total number of tasks in the task sets and the number of processing cores are denoted by *N* and *C* respectively.

### 5.3.2 Evaluation Results

In the first part of this set of experiments, we evaluated the proposed memory allocation technique on a dual-core and a quad-core system where the memory can be configured according to $MemConfig_1$ or $MemConfig_2$ defined in Table 4. The workload includes task sets which assigns *three* or *six* tasks per core and it is assumed that the ratio of average case to worst case execution characteristics of the tasks, $AC/WC$, is equal to 1. The results of these experiments are demonstrated in Fig. 9.

As can be inferred from Fig. 9, applying NPAM results in significant energy savings. Specifically, for all different cases with memory configuration according to $MemConfig_1$, Fig. 9a, the total energy can be saved by 32.67 percent on average. This shows that NPAM is able to exploit the available memory more efficiently. Considering the different characteristics of the memories and diverse access patterns of the variables of the tasks, NPAM is able to allocate memory to the tasks' data in a way that results in the minimization of the total energy consumption. This is due to the fact that in systems where NVM is part of the memory hierarchy, regarding the different characteristics of the NVM and the conventional memories is of great importance. For the other evaluations in which the memory configurations are according to the $MemConfig_2$, the results of employing NPAM compared to the baseline is depicted in Fig. 9b. For all different evaluation cases represented in Fig. 9b, the average of the total energy saving is equal to 15.99 percent. It is noteworthy that the time of MILP analysis which is performed offline increases with the number of tasks. However, considering that the design time of high quality systems for hard real-time applications can take even months, it is reasonable.

The important note is that the less energy savings of cases with $MemConfig_2$ configurations in comparison to the cases with $MemConfig_1$ configurations, are due to the fact that when SPM has more capacity, most of the tasks' data are mapped to SPM in both the baseline case and when NPAM is applied. The point is that even when the on-chip memory capacity is large enough that the baseline scheme may consume little amount of energy, NPAM still excels over the baseline case; however, the energy savings may be less than the other experimental cases where there are more competitions over the limited resources.

Generally, NPAM tries to cluster the similar variables in terms of number of write accesses to the same group, and assign the pages of each group to the memory that results in the minimum costs; i.e., due to the high cost of write accesses to NVM, pages with high number of write activities are preferred to be mapped to SRAM, while pages with high read access counts are better to be mapped to NVM, since NVM is more efficient for read accesses.

It should be noted that although the aforementioned behavior is preferred, in some cases the baseline configuration may lead to less number of write accesses to NVM in comparison to NPAM. There are some reasons for these situations. First, the tasks that are running simultaneously compete for the shared resources; in order to gain the best savings, the tasks that contribute most to the total energy consumption of the system win and the loser tasks may be obliged to have more number of write accesses to NVM that incurs more cost. Moreover, observing the access pattern of the real-life tasks, in most cases a variable with high number of write accesses causes high number of read accesses as well. If the benefits of placing a variable with high number of read accesses in NVM outperform its disadvantages of having high number of write accesses to NVM, the allocation technique places this variable in NVM to achieve better energy or time efficiency; in this situation, although energy or execution time is improved, the lifetime of NVM is negatively affected. Hence, when exploiting NVM as part of the memory hierarchy, the lifetime of NVM should be considered. Using (27), we have examined the influence of NPAM on the lifetime of NVM in comparison to the baseline. The lifetime of NVM when NPAM is applied compared to the baseline degrades by 4.3 percent in the worst case, while in the best case the lifetime is improved by 12.8 percent. It is noteworthy that the negative effects of NPAM on the

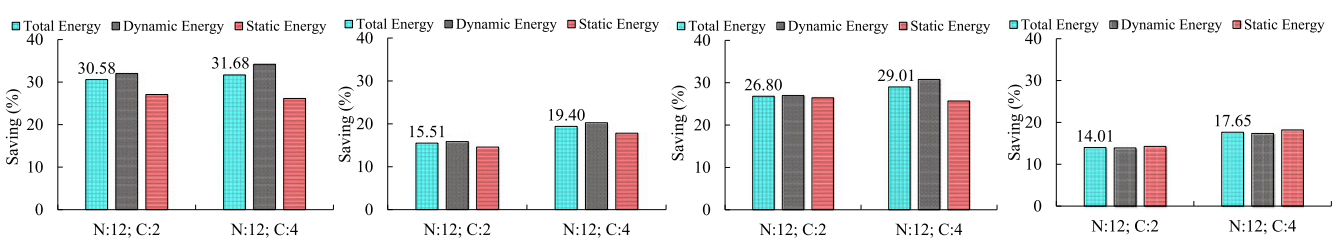


Fig. 10. Results of applying NPAM compared to the baseline. The total number of tasks in the task sets and the number of processing cores are denoted by *N* and *C* respectively.

lifetime are negligible in comparison to its positive effects on energy savings. Moreover, any negative impacts on the lifetime can be identified at design time and corresponding measurements can be taken (e.g., for improving the lifetime of NVM, any wear-leveling technique at the page-level can be applied in addition to NPAM).

To consider the effects of the variation of the estimated worst case and average case execution of each task, we considered two different sets of evaluations where the ratio of average case to worst case execution is either equal to 0.8 or 0.5. The results are represented in Fig. 10. In these experiments, the different system architectures were evaluated using workloads which contain *twelve* tasks. Investigation of Fig. 10 implies that when the difference between the estimated worst case and the average case grows, there would be less opportunity to save energy. It is justifiable as NPAM determines the memory allocation in order to reduce the energy consumption with the consideration of satisfying the worst case requirements of the task set. Therefore, the more difference between the worst case and average case requirements, the more waste of the system resources in situations when the worst case does not happen. However, all sets of experiments underline the ability of NPAM for energy saving.

We have also analyzed the impact of NPAM on the lifetime of NVM when the average case execution characteristics are not similar to the worst case. In these cases, the lifetime of NVM degrades by 3.9 percent in the worst case, while in the best case the lifetime is improved by 9.9 percent as a result of applying NPAM compared to the baseline.

## 6 Conclusion

In this paper we proposed a data allocation technique for multi-core embedded systems with hybrid SPMs that are equipped with MMU. The proposed technique tries to allocate the tasks' variables to virtual pages, and then specifies the mapping of virtual pages to each memory to take full advantage of a hybrid memory. By exploiting the high density and low leakage power of NVM and the efficient write access of conventional memory, our proposed technique presents an efficient memory allocation scheme. The experimental results show that the proposed allocation technique can reduce the energy consumption significantly without considerable adverse effects on NVM lifetime.

## References


[1] P. Marwedel, *Embedded Systems Design-Embedded Systems Foundations of Cyber-Physical Systems*. New York, NY, USA: Springer, 2011.

[2] G. Rodríguez, J. Tourino, and M. T. Kandemir, "Volatile STT-RAM scratchpad design and data allocation for low energy," *ACM Trans. Archit. Code Optimization*, vol. 11, no. 4, pp. 1–26, 2014.

[3] D.-W. Chang, C. Lin, Y.-S. Chien, C.-L. Lin, A. W.-Y. Su, and C.-P. Young, "CASA: Contention-aware scratchpad memory allocation for online hybrid on-chip memory management," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 33, no. 12, pp. 1806–1817, Dec. 2014.

[4] L. A. Bathen and N. Dutt, "HaVOC: A hybrid memory-aware virtualization layer for on-chip distributed scratchpad and non-volatile memories," in *Proc. Annu. Des. Autom. Conf.*, 2012, pp. 447–452.

[5] S. Yazdanshenas, M. R. Pirbasti, M. Fazeli, and A. Patooghy, "Coding last level STT-RAM cache for high endurance and low power," *IEEE Comput. Archit. Lett.*, vol. 13, no. 2, pp. 73–76, Jul.-Dec. 2014.

[6] J. Hu, Q. Zhuge, C. J. Xue, W.-C. Tseng, and E. H.-M. Sha, "Management and optimization for nonvolatile memory-based hybrid scratchpad memory on multicore embedded processors," *ACM Trans. Embedded Comput. Syst.*, vol. 13, no. 4, 2014, Art. no. 79.

[7] H. Cho, B. Egger, J. Lee, and H. Shin, "Dynamic data scratchpad memory management for a memory subsystem with an MMU," *ACM SIGPLAN Notices*, vol. 42, no. 7, pp. 195–206, 2007.

[8] W. Tian, et al., "Task allocation on nonvolatile-memory-based hybrid main memory," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 21, no. 7, pp. 1271–1284, Jul. 2013.

[9] Y. Liu and W. Zhang, "Scratchpad memory architectures and allocation algorithms for hard real-time multicore processors," *J. Comput. Sci. Eng.*, vol. 9, no. 2, pp. 51–72, 2015.

[10] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen, "WCET centric data allocation to scratchpad memory," in *Proc. 26th IEEE Int. Real-Time Syst. Symp.*, 2005, pp. 10–23.

[11] N. Nguyen, A. Dominguez, and R. Barua, "Memory allocation for embedded systems with a compile-time-unknown scratch-pad size," in *Proc. Int. Conf. Compilers Archit. Synthesis Embedded Syst.*, 2005, pp. 115–125.

[12] S. Udayakumaran, A. Dominguez, and R. Barua, "Dynamic allocation for scratch-pad memory using compile-time decisions," *ACM Trans. Embedded Comput. Syst.*, vol. 5, no. 2, pp. 472–511, 2006.

[13] A. Dominguez, S. Udayakumaran, and R. Barua, "Heap data allocation to scratch-pad memory in embedded systems," *J. Embedded Comput.*, vol. 1, no. 4, 2005, Art. no. 521540.

[14] B. Egger, J. Lee, and H. Shin, "Scratchpad memory management for portable systems with a memory management unit," in *Proc. 6th ACM IEEE Int. Conf. Embedded Softw.*, 2006, pp. 321–330.

[15] B. Egger, S. Kim, C. Jang, J. Lee, S. L. Min, and H. Shin, "Scratchpad memory management techniques for code in embedded systems without an MMU," *IEEE Trans. Comput.*, vol. 59, no. 8, pp. 1047–1062, Aug. 2010.

[16] W. Tian, J. Li, Y. Zhao, C. J. Xue, M. Li, and E. Chen, "Optimal task allocation on non-volatile memory based hybrid main memory," in *Proc. ACM Symp. Res. Appl. Comput.*, 2011, pp. 1–6.

[17] Z. Wang, Z. Gu, and Z. Shao, "Optimized allocation of data variables to PCM/DRAM-based hybrid main memory for real-time embedded systems," *IEEE Embedded Syst. Lett.*, vol. 6, no. 3, pp. 61–64, Sep. 2014.

[18] J. Hu, C. J. Xue, Q. Zhuge, W.-C. Tseng, and E. H.-M. Sha, "Write activity reduction on non-volatile main memories for embedded chip multiprocessors," *ACM Trans. Embedded Comput. Syst.*, vol. 12, no. 3, 2013, Art. no. 77.

[19] S. Gu, Q. Zhuge, J. Yi, J. Hu, and E. H.-M. Sha, "Data allocation with minimum cost under guaranteed probability for multiple types of memories," *J. Signal Process. Syst.*, vol. 84, no. 1, pp. 151–162, 2016.

[20] J. Hu, C. J. Xue, Q. Zhuge, W. C. Tseng, and E. H. M. Sha, "Towards energy efficient hybrid on-chip scratch pad memory with non-volatile memory," in *Proc. Des. Autom. Test Eur. Conf. Exhib.*, Mar. 2011, pp. 1–6.

[21] J. Hu, C. J. Xue, Q. Zhuge, W.-C. Tseng, and E. H.-M. Sha, "Data allocation optimization for hybrid scratch pad memory with SRAM and nonvolatile memory," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 21, no. 6, pp. 1094–1102, Jun. 2013.

[22] M. Qiu, Z. Chen, and M. Liu, "Low-power low-latency data allocation for hybrid scratch-pad memory," *IEEE Embedded Syst. Lett.*, vol. 6, no. 4, pp. 69–72, Dec. 2014.

[23] M. Qiu, et al. "Data allocation for hybrid memory with genetic algorithm," *IEEE Trans. Emerging Topics Comput.*, vol. 3, no. 4, pp. 544–555, Oct.-Dec. 2015.

[24] J. Whitham and N. Audsley, "Implementing time-predictable load and store operations," in *Proc. 7th ACM Int. Conf. Embedded Softw.*, 2009, pp. 265–274.

[25] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: Design alternative for cache on-chip memory in embedded systems," in *Proc. 10th Int. Symp. Hardware/Softw. Codes.*, 2002, pp. 73–78.

[26] A. N. Sloss, D. Symes, C. Wright, and J. Rayfield, *ARM System Developer's Guide: Designing and Optimizing System Software*. San Mateo, CA, USA: Morgan Kaufmann, 2004.

[27] G. J. Holzmann,"The power of 10: Rules for developing safety-critical code," *IEEE Comput.*, vol. 39, no. 6, pp. 95–99, Jun. 2006.

[28] C. Wang, C. Dong, H. Zeng, and Z. Gu, "Minimizing stack memory for hard real-time applications on multicore platforms with partitioned fixed-priority or EDF scheduling," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 21, no. 3, 2016, Art. no. 46.

[29] Z. Wang, Z. Gu, M. Yao, and Z. Shao, "Endurance-aware allocation of data variables on NVM-based scratchpad memory in real-time embedded systems," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 34, no. 10, pp. 1600–1612, Oct. 2015.
[30] C. Rusu, R. Melhem, and D. Mossé, "Maximizing rewards for real-time applications with energy constraints," *ACM Trans. Embedded Comput. Syst.*, vol. 2, no. 4, pp. 537–559, 2003.
[31] M. M. Kafshdooz and A. Ejlali, "Dynamic shared SPM reuse for real-time multicore embedded systems," *ACM Trans. Archit. Code Optimization*, vol. 12, no. 2, pp. 12:1–12:25, May 2015.
[32] Y. Wang and M. Saksena, "Scheduling fixed-priority tasks with preemption threshold," in *Proc. 6th Int. Conf. Real-Time Comput. Syst. Appl.*, 1999, pp. 328–335.
[33] B. Egger, J. Lee, and H. Shin, "Dynamic scratchpad memory management for code in portable systems with an MMU," *ACM Trans. Embedded Comput. Syst.*, vol. 7, no. 2, 2008, Art. no. 38.
[34] S. C. Johnson, "Hierarchical clustering schemes," *Psychometrika*, vol. 3, no. 32, pp. 241–254, 1967.
[35] D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham, "Worst-case performance bounds for simple one-dimensional packing algorithms," *SIAM J. Comput.*, vol. 3, no. 4, pp. 299–325, 1974.
[36] D. Hardy and I. Puaut, "Predictable code and data paging for real time systems," in *Proc. Euromicro Conf. Real-Time Syst.*, 2008, pp. 266–275.
[37] C. Meenderinck, A. Molnos, and K. Goossens, "Composable virtual memory for an embedded SoC," in *Proc. Euromicro Conf. Digit. Syst. Des.*, 2012, pp. 766–773.
[38] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "NVSim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 31, no. 7, pp. 994–1007, Jul. 2012.
[39] G. Optimization et al., "Gurobi optimizer reference manual," vol. 2, pp. 1–3, 2012, [Online]. Available: http://www.gurobi.com
[40] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. IEEE Int. Workshop Workload Characterization*, 2001, pp. 3–14.
[41] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The mälardalen WCET benchmarks: Past, present and future," *OASIcs-OpenAccess Series Informat.*, vol. 15, pp. 136–146, 2010.
[42] D. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," *ACM SIGARCH Comput. Archit. News*, vol. 25, no. 3, pp. 13–25, 1997.
[43] J. Hu, C. J. Xue, W. C. Tseng, Y. He, M. Qiu, and E. H. M. Sha, "Reducing write activities on non-volatile memories in embedded CMPs via data migration and recomputation," in *Proc. Des. Autom. Conf.*, 2010, pp. 350–355.
[44] W. Von Hagen, *The Definitive Guide to GCC*. New York, NY, USA: Apress, 2011.
[45] P. Emberson, R. Stafford, and R. I. Davis, "Techniques for the synthesis of multiprocessor tasksets," in *Proc. Int. Workshop Anal. Tools Methodologies Embedded Real-Time Syst.*, 2010, pp. 6–11.

**Farimah R. Poursafaei** received the BSc degree in computer engineering from Amirkabir University of Technology and the MSc degree from Sharif University of Technology, in 2014 and 2016, respectively. Her research interests include low-power real-time embedded systems and management of non-volatile memories.

**Mostafa Bazzaz** received the BSc degree in computer engineering from Amirkabir University of Technology and the MSc degree from Sharif University of Technology, in 2009 and 2011, respectively. He is working toward the PhD degree in the Computer Engineering Department, Sharif University of Technology. His research interests include low-power multi-core embedded systems, non-volatile memories, and real-time embedded systems.

**Alireza Ejlali** received the PhD degree in computer engineering from Sharif University of Technology, Tehran, Iran, in 2006. He is an associate professor of computer engineering with Sharif University of Technology. He is currently the director of the Embedded Systems Research Laboratory, Department of Computer Engineering, Sharif University of Technology. His current research interests include low power design, real-time embedded systems, and fault-tolerant embedded systems.