IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 1 November 2022, accepted 19 November 2022, date of publication 28 November 2022,
date of current version 5 December 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3225193

RESEARCH ARTICLE

Improving NVM Lifetime Using Task Stack Migration on Low-End MCU-Based Devices

JEONGMIN LEE[1], MOONSEOK JANG[1], KEXIN WANG[1], (Student Member, IEEE), INYEONG SONG[1], HYEONGGYU JEONG[1], JINWOO JEONG[1], (Student, IEEE), YONG HO SONG[2], AND JUNGWOOK CHOI[1], (Member, IEEE)

[1]Department of Electronics Engineering, Hanyang University, Seoul 04763, South Korea
[2]Samsung Electronics, Hwaseong, South Korea

Corresponding author: Jungwook Choi (choij@hanyang.com)

This work was supported in part by Samsung Electronics Company Ltd., and in part by the Institute of Information and Communications Technology Planning and Evaluation (IITP) Grant funded by the Korea Government (MSIT) (Logic Synthesis for NVM-based PIM Computing Architecture) under Grant 2022-0-00971.

ABSTRACT Tiny embedded devices are cost and energy-sensitive, and high-density emerging non-volatile memory (NVM) can help reduce energy consumption at a fraction of the cost. However, high-density NVM has low write endurance compared to volatile memory, so it is vulnerable to write concentration. Most NVM lifetime improvement studies in the existing embedded environment have distributed writes by modifying the mapping relationship between physical and logical addresses. However, applying the existing wear leveling techniques in low-end MCUs such as ARM Cortex M3/M4 that use only physical addresses is hard. Therefore, we wear-level the write-heavy stack area to improve the NVM lifetime in low-end MCUs. However, since the stack of bare metal applications is difficult to move during runtime, we implement the migration function targeting the task stack of FreeRTOS. The task stack moves based on time, and to avoid the pointer validation problem caused by the movement of the task stack, we migrate the stack under safe conditions. In addition, FreeRTOS uses a single heap to preferentially allocate to low free space, which reduces the degree of freedom where the stack moves, reducing the effect of distributing the writes. To alleviate this problem, we add another heap for the stack migration and introduce circular dynamic allocation in the heap. Through our experiments, the proposed method was about 19.6% larger than the ideal case of maximum write, and the computational overhead was about 0.2%.

INDEX TERMS Embedded software, memory management, nonvolatile memory, real-time systems.

I. INTRODUCTION

Emerging non-volatile memory (NVM) is capable of random access, consumes low standby power, and has a performance close to that of volatile memory, so it can simultaneously serve as the main memory and storage of a tiny embedded device. When ultra-compact embedded devices reduce energy consumption by using NVM [1], applications such as TinyML [2], [3], [4] that perform data acquisition and machine learning simultaneously can run for a long time by placing the device close to the data. NVM can be divided into low-density NVM and high-density NVM. High-density NVMs such as phase-change memory (PCM) [5] or multi-level cell (MLC) spin transfer torque magnetoresistive random-access memory (STT-MRAM) [6], [7] have lower performance and shorter lifetime than low-density NVMs such as a single-level cell(SLC) STT-MRAM [8], [9], but high-density NVMs are cheaper based on the same memory capacity. Although computing performance and energy consumption characteristics are essential for tiny embedded devices, the device's manufacturing cost is also important. In other words, using a high-density NVM in a tiny embedded device means that the tiny embedded device can be manufactured at a relatively low cost while having an NVM's performance and energy characteristics.

However, the high-density NVM has shorter write endurance than the conventional volatile memory, which may be a problem when using the high-density NVM as the

The associate editor coordinating the review of this manuscript and approving it for publication was Liang-Bi Chen.

main memory. A computation-intensive application such as TinyML generates many writes concentrated in the stack area. In this case, the lifetime problem of high-density NVM becomes more prominent. Tiny embedded devices sometimes use low-end MCUs without a memory management unit and cache, and in this case, applying the existing write distribution techniques is not easy.

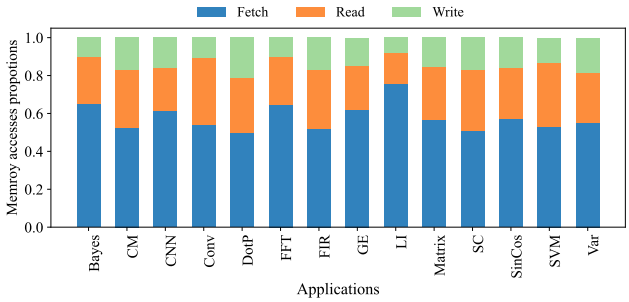
Previous studies mainly attempted wear leveling using the mapping relationship between logical and physical addresses. Write-related information was collected in units of memory blocks, and wear leveling was attempted based on this information [10], [11], [12]. Some studies have circularly moved the data for lighter wear leveling [13], [14]. However, the above studies modify the mapping through MMU, but it is challenging to apply to low-end MCU without MMU.

In this study, writes on the stack are distributed when NVM is used as the main memory using the existing hardware resources and RTOS task characteristics in low-end MCUs. RTOS tasks have their stack, so it is easier to distribute writes on the stack than in a bare metal environment. Since the amount of writes in the stack is different for each task and the location of the task stack is fixed, the task stack causes non-uniform wear out in the NVM. To increase the lifetime of the NVM, we measure the operation time of a task, and if a task operates for more than a critical time, the stack location of the task is changed during runtime to distribute writes. However, if the stack is migrated during runtime, the validation of pointers in the stack cannot be guaranteed, and task operation may fail. We made the stack migration condition not to damage pointer validation during stack migration. Also, the pointer is updated based on the new stack location to ensure the validation of the pointer after the stack is migrated.

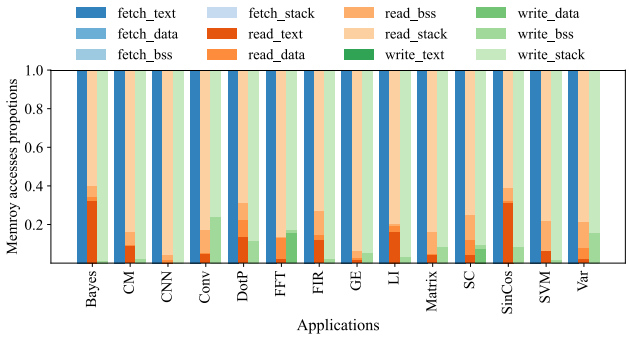
In addition, since the dynamic allocation of RTOS is designed based on the volatile memory system, distributing writes decreases if the existing dynamic allocation method is used for stack migration. Since the existing dynamic allocation prioritizes the low free space of the heap, the locations where the stack moves also be concentrated in the low space of the heap. In addition, other memory objects other than the stack can also be allocated on the heap. Since these other memory objects do not move during runtime, they affect the moveable position of the stack. To prevent write distribution degradation due to stack migration due to the existing dynamic allocation method, we create a separate heap to which only the task stack can be allocated. Moreover, the dynamic allocation is circularly performed on the heap. In addition, to increase the randomness of the location where the stack moves in the heap, we create random strides to allocate the stack to various locations in a heap.

The main contributions of this study are summarized as follows.

- Since obtaining write information in the low-end MCU environment is difficult, we perform time-based lightweight wear leveling.
- To avoid the risk of moving the stack during runtime, we limit the conditions that can be migrated, and update the pointer after migration.
- Since our proposed wear leveling performs software-based wear leveling using existing hardware resources and RTOS, portability to other environments is easy.



(a) Ratio of each memory access to total memory accesses



(b) Ratio of each memory access by memory section

FIGURE 1. Memory accesses of computation-intensive applications.

The following text is structured as follows. Section 2 introduces the motivation for our study. Section 3 deals with stack migration methods using time-based information and avoiding pointer validation issues. Section 4 describes methods to increase the write distribution effect by stack migration. Section 5 shows the measurement and analysis of overhead and write distribution effects caused by the proposed write distribution method. Section 6 discusses the significance and limitations of our study and future research directions. Section 7 examines NVM wear leveling and related studies and compares them with ours. Finally, we summarize this study.

II. MOTIVATIONS

In order to analyze the memory access that occurs in the low-end MCU in a computation-intensive application such as TinyML and DSP, ARM CMSIS-NN and CMSIS-DSP were performed on the ARM Cortex M4F and the memory access was measured [3], [15]. When 13 applications were executed once, memory access was shown in Fig. 1. Fig. 1(a) shows the proportion of *fetch*, *read*, and *write* to the total memory access of each application, and Fig. 1(b) shows the ratio of memory accesses occurring in each memory section. In all

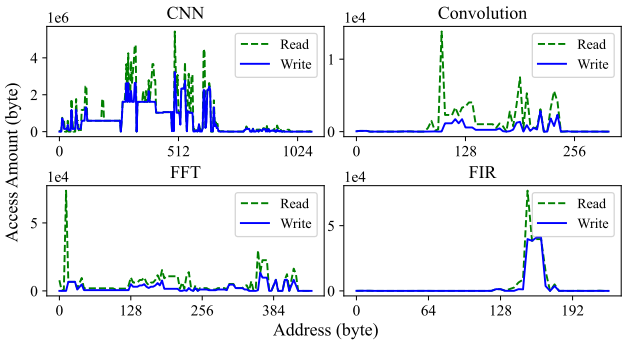


FIGURE 2. Distribution of memory accesses occurring on the stack.

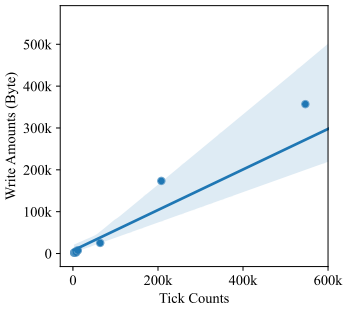


FIGURE 3. Correlation between task running time and write amounts in the task stack.

applications, *fetch* was the most, followed by *read* and *write* the least. However, most of the writes occurred in the stack area in all applications. This is because computation-intensive applications use more than a number of variables that can be handled simultaneously in the CPU's registers. Variables that cannot be accommodated in registers are temporarily stored on the stack. Also, in computationally intensive applications, function calls are frequent. When a function is called, registers, return addresses and passed arguments are stored on the stack. Due to the preceding operations, computationally intensive applications cause more writes to the stack. Fig. 2 shows more about the memory access that occurs inside the stack. When looking at the distribution of the amount of memory access that occurred in the stack of 4 applications, writes were concentrated on specific addresses, and this concentration of writes also occurred in other applications of ARM CMSIS. In conclusion, to use NVM, which has a limited lifespan, in low-end MCUs, a method for distributing writes occurring in the stack is required.

III. EXTENSION OF RTOS KERNEL FOR STACK MIGRATION

In low-end MCUs, obtaining write information during operation is challenging, and since there is no MMU, applying wear-leveling techniques that distribute writes using a mapping between physical addresses and logical addresses is challenging. When an application is executed, the greater the amount of computation, the greater the memory access amount, and the longer the application's operating time. This study used the operation time to infer the amount of writing instead of directly acquiring write information using these characteristics.

A single application runs in a bare metal environment, and most operate using a single stack. On the other hand, in a real-time operating system (RTOS), a plurality of tasks (or processes) are executed, and each task has an individual stack allocated on the heap. Therefore, when applications performing the same function are run in each environment, writes occurring in the stacks are relatively distributed due to a plurality of task stacks in the RTOS environment. Also, since the task's stack is not affected by the actions of other tasks, the task stack can be moved to a different location in memory to distribute writes. However, the movement of the task stack may make the pointers that point directly to a specific location in the stack invalid, which may cause an obstacle to the operation of the task. In addition, the stack can be moved only in a partial range due to the existing dynamic memory allocation of the RTOS preferentially allocating a low address in a heap, so write distribution can occur only in a partial range of the heap. In this study, to alleviate these problems, the task stack is moved to another location under a condition that is safe from pointer validity, and the task stack is moved to the whole heap through circular allocation.

A. TIME-BASED TASK STACK MIGRATION

Because low-end MCU-based devices struggle to obtain memory access information during runtime, we use the task running time to determine whether stack migration should be performed. In RTOS, the task scheduler periodically works through a tick interrupt generated by the tick timer built into the low-end MCUs. The proposed task stack migration uses a tick timer to measure time; therefore, additional hardware is not required to measure the task running time.

In tinyML and DSP applications, the longer the CPU time, the more write instructions are generated. Fig. 3 shows the running time measured by the tick timer and the number of write instructions during that time when 13 DSP applications and CNNs were run on a Cortex M4. In all cases, the running time of the task and the number of write instructions have a high positive correlation. The number of writes induced by the task can be inferred from the task running time. Therefore, task running time is used as a criterion for stack migration in this study.

However, a high positive correlation does not exist for running times and write amounts for all task types. Tasks with few mathematical operations generate a small number of writes on the stack, even if the running time is long. These types of tasks should be excluded from stack migration because the write distribution effect of stack migration is small. Therefore, we use flags to identify tasks that allow stack migration. Since application designers can quickly identify tasks requiring significant computation, they must set the flags in the task creation process.

The proposed time-based stack migration consists of task running-time measurement and stack migration, and most

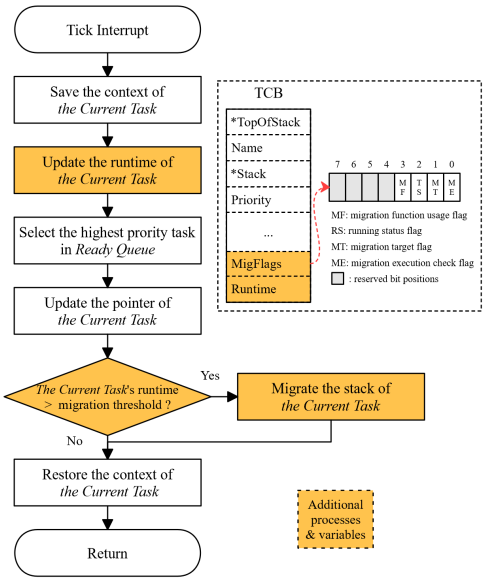


FIGURE 4. Context switching and TCB structure with the proposed time-based stack migration function.

operations are performed in context switching. Context switching operates in the kernel mode, and because the task operation stops at this time, a relatively accurate time measurement is possible. The task running time measurement used a tick timer built into the low-end MCU, and a space for storing the measured time is added to the TCB.

The context-switching operation, including the proposed stack migration function, is shown in Fig. 4. When the task is switched in, the time measurement starts, and the time is measured until the corresponding task is switched out. In the TCB, in addition to information for storing the running time of a task, a variable for storing flag information for managing the migration information of the corresponding task is required. Fig. 4 shows the configuration of the migration information flags, and the migration information is recorded in the lower 4-bit of one byte. The *MF* flag indicates whether stack migration is allowed for the task, and this flag is set when the task is created. The task with the *MF* flag of *SET* has much computation. The *RS* flag indicates whether the task is running, paused, or suspended. The *RS* flag is *SET* if the task is running or paused, and *RESET* if it is suspended. A high possibility exists that pointers that refer to the stack pointer or point to a specific location on the stack are used during code execution in the task function. Therefore, migrating a stack while the task function is running or paused is prone to pointer validation problems. If the stack migrates when the task is complete, the risk of pointer validation problem is low; thus, the safety of stack migration through the *RS* flag is checked. The *MT* flag is used to determine whether a task is a target for stack migration during the following restart process, and when the operating time of the task exceeds the migration threshold time, it becomes *SET*. The *ME* flag records whether a task stack migration has occurred and causes the task to update the local pointer immediately after stack migration. In particular, the *RS* and *ME* flags are used to avoid pointer validation problems caused by stack migration, as described in the next section.

B. AVOIDING POINTER VALIDATION PROBLEM CAUSED BY STACK MIGRATION

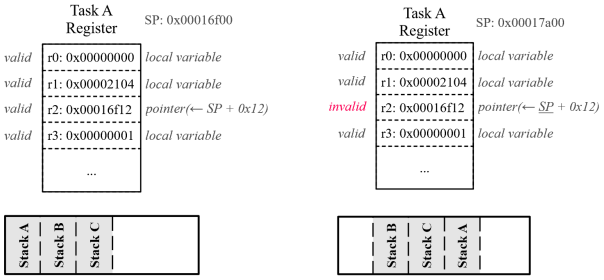
Because stack migration moves the task stack location, it can cause pointer validation problems. When the local pointer points to a specific space in the stack, it still points to the previous location, even if the task stack migrates. Furthermore, when a function call occurs, the current registers and stack pointer values are stored in the stack. The values stored in the stack are restored when the called function ends. Then, the codes of the location where the call occurred are executed continuously. If task stack migration arises before returning the called function, and if function return occurs, the restored stack pointer references an address on the previous stack. In other words, task stack migration can cause pointer validation problems.

Pointer validation problems can be avoided by limiting the conditions under which the task stack migration is performed. DSP or tinyML tasks can be implemented to operate at regular intervals or when a defined condition is satisfied. For these types of tasks, it is easy to distinguish between a start point and an endpoint. As mentioned, migrating the stack while a task runs or pauses can cause a pointer validation problem. Therefore, stack migration should only be performed when the task is complete to avoid the pointer validation problem. When a function that suspends a task is called, the *TS* flag becomes *RESET*. When the task is restarted by context switching, and *TS* flag becomes *SET*. When a task is paused, the *TS* flag remains in the *SET* state. If stack migration is allowed only when the *TS* flag is *RESET*, the pointer validation problem can be partially avoided.

However, if a local pointer exists on a register, a pointer validation problem may occur even though stack migration is performed under limited conditions. Fig. 5 shows an example of a pointer validation problem during stack migration, which occurs because a local pointer exists in a register. Even after stack migration, the pointers in the registers are not updated automatically. Therefore, the local pointer on *register 2* points to a location in the stack prior to migration. Therefore, the pointer in *register 2* is invalid. Fig. 6 shows an example of avoiding the validity problem of a pointer located in a register. In this study, when stack migration occurred, the *ME* flag is *SET* before task restart. When a task restarts, it reads the *ME* flag at the restart point of the task function to determine whether migration has occurred. If the *ME* flag is set to *SET*, the local pointer is updated. As the local pointer updates the referenced address by calculating the relative position based on the current stack pointer, the validity of the local pointer is guaranteed.

C. CIRCULAR ALLOCATION BY DEALLOCATED STATE

The dynamic allocation of the FreeRTOS first allocates the lower free space of the heap, and the deallocated space is



(a) Before stack migration (b) After stack migratoin

FIGURE 5. Pointer validation problem in registers owing to stack migration.

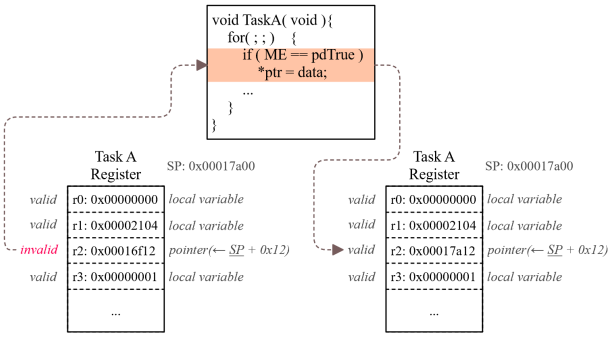
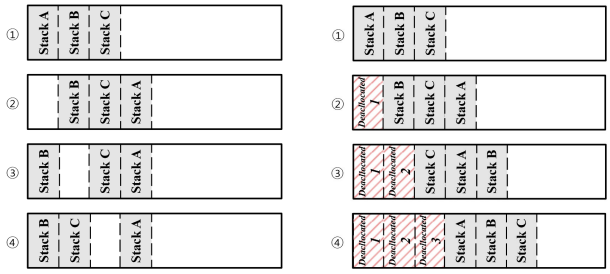


FIGURE 6. Avoiding pointer validation problems by updating pointers after stack migration.

immediately converted into free space. When the stack is moved to a higher space on the heap, the space previously allocated to the stack becomes a free space. When the stack is migrated again, it is moved to the lower space of the heap where the previous stack is located, even though sufficient free space is available in the higher space of the heap. In other words, in the existing dynamic allocation method, even if the stack is migrated, most of it is moved only to the low spaces of the heap. Therefore, the distribution of writes is made only in part of the heap. Fig. 7 compares the migration processes using existing FreeRTOS and the proposed dynamic allocation methods. Fig. 7(a) shows the dynamic allocation of the FreeRTOS. The existing dynamic allocation method allows the stack to migrate only within a limited heap area. Fig. 7(b) shows the dynamic allocation method proposed to solve the existing dynamic allocation problem. When the free space on the heap is less than the requested allocation size, the deallocated spaces are converted into free spaces. The proposed dynamic allocation delays the free-space transition by introducing a deallocated state space, which induces dynamic allocation to be performed circularly from low to high. The circular allocation also moves the space where writes occur circularly, resulting in the spread of writes across the heap.

Newly defined deallocated state spaces must be managed for circular allocation. Since the data in the deallocated state space are invalid, we place the linked list in the deallocated state space to manage the deallocated state space. Fig. 8 shows an example of managing deallocated state spaces as a



(a) Dynamic allocation of FreeRTOS (b) Proposed circular allocation

FIGURE 7. Comparison of stack migration using the existing and proposed memory allocations.

linked list in circular allocation. To reduce management load, they are linked in the order in which they are deallocated. The previously deallocated state space is at the front of the linked list, and the subsequently deallocated state space is at the back. The deallocated state space at the front of the linked list is preferentially converted into free space. Depending on the situation, performing a free space conversion operation several times may be necessary for creating the required free space. However, searching or sorting is unnecessary for managing the deallocated state; therefore, the overhead required for management is negligible.

IV. TECHNIQUES FOR IMPROVING THE WRITE DISTRIBUTION

When performing time-based stack migration, the allocation status in a heap and the task stack size limit the stack's moveable position. If the movable position of the stack is restricted, the effect of distributing writes is also reduced. In particular, the heap of FreeRTOS is a single structure, and various memory objects, such as the task stack, TCB, queue, and user space, are allocated. Other memory objects can limit the migration of the stack and consequently reduce stack migration efficiency. A dual heap structure with a heap added only for stack migration is applied to alleviate this problem. Additionally, the location where the stack is migrated may be more concentrated at specific addresses. This concentration is because the dynamic allocation location may be fixed depending on the direction in which the circular dynamic memory allocation proceeds and the size of the task stack. Before a stack is migrated, a random-size stride is allocated to a heap to increase the randomness of the stack's moveable position.

A. DUAL HEAP STRUCTURE TO PREVENT HEAP FRAGMENTATION

The existing dynamic memory allocation of RTOS uses a single heap space to allocate the task stack, TCB, queue, semaphore, user space, and others. Memory objects generally do not move from where they are initially allocated until deallocated. However, in terms of stack migration, fixed-memory objects fragment the heap. If the heap is fragmented, the location where stack migration is possible is restricted, and the write distribution by stack migration is also reduced.

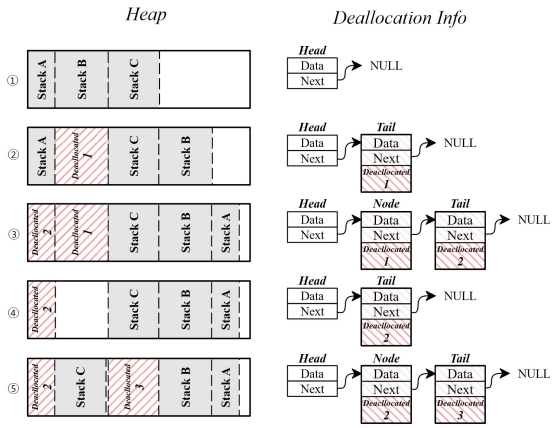
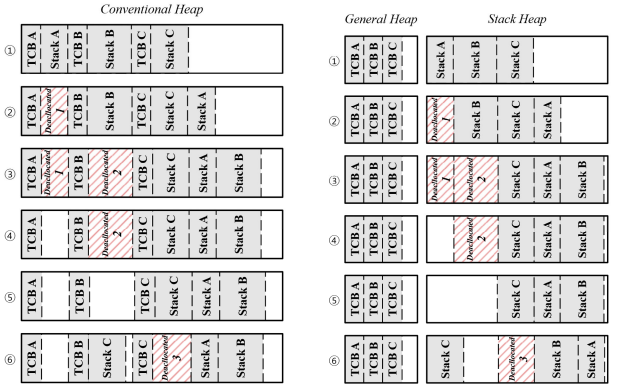


FIGURE 8. Example of circular allocation.



(a) Typical single heap structure (b) Proposed dual heap structure

FIGURE 9. Comparison of the existing single heap structure and the dual heap structure.

To prevent heap fragmentation caused by fixed memory objects, we propose a dual heap structure with a heap for stack migration. The existing single-heap structure allocates all the memory objects to one heap. In contrast, the proposed dual heap structure has a *stack heap* where only task stacks for which stack migration is allowed are allocated and a *general heap* where all other memory objects are allocated. Fig. 9 shows an example of stack migration operation in the existing single-heap structure and the proposed dual heap structure. Fig. 9(a) shows the case of a single heap when only three tasks, A, B, and C, are created, and stack migration is performed. Because TCBs do not move, the space between TCBs in terms of stack migration can be considered fragmented. Only stack A can enter the space between TCBs A and B in Fig. 9(a), whereas stacks B and C cannot migrate. Therefore, only stack A can be located in this space, and only the write patterns that occur in task A are accumulated. Fig. 9(b) shows the dual heap structure. This Fig. is an example of stack migration when TCBs are aligned to a *general heap*, and only stacks are allocated to the *stack heap*. The total size of the *general heap* and *stack heap* is the same as that of a single heap, as shown in Fig. 9(a). As shown in Fig. 9(b),

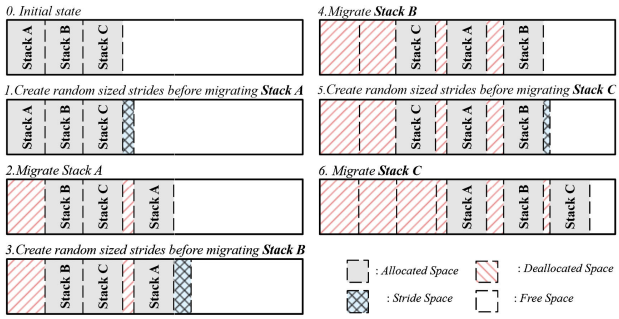


FIGURE 10. Example of random size stride generation.

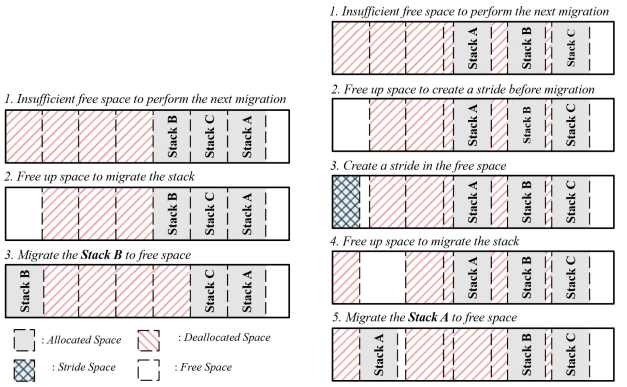
unlike in the case of a single heap, the stack movement is unrestricted owing to the TCB when migrating the stack. In a real RTOS, more memory objects are allocated to the heap; therefore, the heap becomes more fragmented. This further limits the stack movement. The proposed dual heap structure is more effective in distributing writes because it is free from the heap fragmentation caused by other memory objects during stack migration.

B. RANDOM SIZE STRIDE FOR IMPROVING WRITE DISTRIBUTION

Other stacks, deallocated state space, and free space limit the stack moves. If the location the stack moves is constrained, the stack moves more to a specific location within the heap. The more frequently the stack is migrated to specific locations, the more writes are accumulated in those locations. So we use random size strides to increase the randomness of where the stack moves. 10 shows an example of creating a random size stride, created before stack migration and immediately converted to a deallocated space. Fig. 11 is an example of the stack migration difference depending on whether or not stride is used when a stack of the same size is migrated in a situation where there is not enough free space. It is a condition that all stacks have the same size. In this case, if a random size stride is not used, as shown in Fig. 11(a), the stack is circularly allocated at regular intervals. Because stacks move at regular intervals, writes also move at regular intervals and accumulate, limiting the distribution of writes. Fig. 11(b) is an example of stack migration when using a random size stride. Because random size strides are created before the stack is migrated, the stack can be moved to various locations. In other words, random size strides accumulate writes on the stack to various locations on the heap, distributing the writes. However, additional computational overhead occurs if a random size stride is used.

C. TIME BASED STACK MIGRATION WITH LIMITED COMPUTATION OVERHEAD

As mentioned earlier, our proposed time-based stack migration method migrates the task stack based on time. If a task runs for more than a threshold time, the stack is migrated to another location in the heap before the task is restarted.



(a) Without random size stride (b) With random size stride

FIGURE 11. Comparison of stack migration with and without random size strides.

TABLE 1. Variables of stack migration process.

Variable	Definition
M_{TH}	Stack migration threshold time
S	Size of current task's stack
T	Runtime of current task
R	Random number
D_r	Max free space conversion depth of *stride generation* (maximum stride depth)
D_m	Max free space conversion depth of *stack migration* (maximum migratoin depth)
d_r	Current free space conversion depth of *stride generation*
d_m	Current free space conversion depth of *stack migration*
F_i	i-th free space in linked list
RS	Task running status check flag

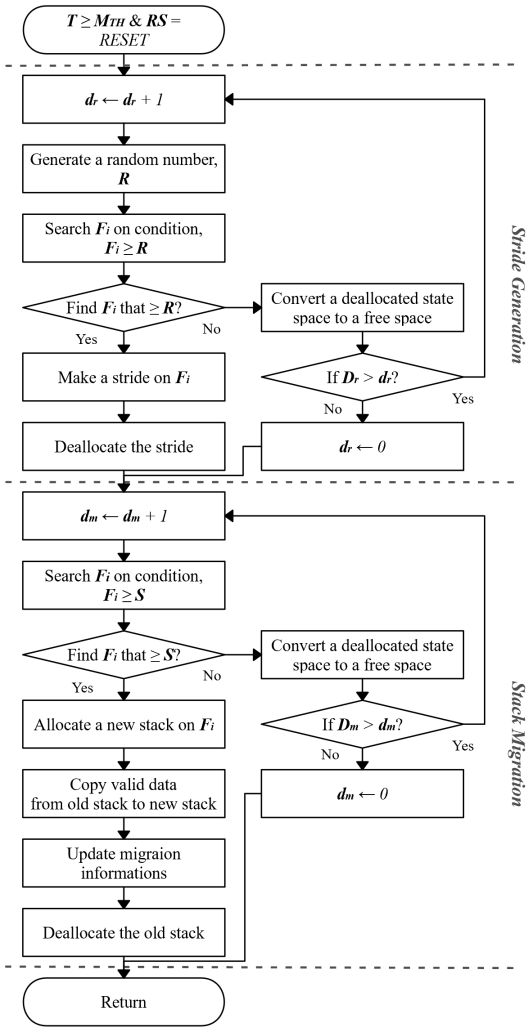


FIGURE 12. Proposed stack migration process.

However, in creating a random size stride and migrating the stack, it is necessary to secure space for allocation. Due to circular allocation using deallocated state, the heap often does not have enough free space for allocation. In this case, the deallocated state space is converted to free space. However, the newly created free space may still not be sufficient for allocation, and in this case, the free space conversion operation is performed again. If this free space conversion operation occurs continuously, the task deadline can be significantly violated. Therefore, we set the maximum depth to perform the free space conversion operation during the stride generation and stack migration processes.

Fig. 12 shows the proposed time-based stack migration method, and Table 1 shows the symbols in Fig. 12. The proposed time-based stack migration is divided into the *stride generation* process and the *stack migration* process. In the *stride generation* process, the maximum stride depth is used to limit the maximum number of free space searches and conversions. If there is the insufficient size of free space for stride generation until the maximum stride depth is reached, the stride generation fails and goes to *stack migration*. In the *stack migration* process, the maximum migration depth is used to limit the maximum number of free space searches and conversions. Also, if a sufficient size of free space for stack migration is not secured until the maximum migration depth is reached, the stack migration is regarded as failed, and the operation is completed. Therefore, our proposed time-based stack migration may fail even if the task stack meets the migration conditions. Even if the migration fails at a specific point in time, the writes are distributed because the stack moves to a different location based on a long term.

The time complexity of the proposed algorithm is $O(n^2)$. Free space search searches for free space in the stack heap through a loop operation, and the time complexity is $O(n)$. Also, the stride creation process and stack migration process, which include searching for this free space, have loop operations depending on the situation, so the time complexity is $O(n^2)$. However, since the number of iterations is limited by the maximum stride depth and the maximum migration depth, the computational overhead is suppressed so that it does not become larger than a certain amount.

TABLE 2. Primary experiement envirionments.

Enviroment	
Simulator	OVPsim21118.0
MCU	ARM Cortex M4F
RTOS	Amazon FreeRTOS 10.3.1
Application	ARM CMSIS
Stack heap size	12 KB
Tick length	20000 inst.

V. EXPERIMENT

The time-based stack migration method proposed in this study aims to improve the lifetime of NVM in a low-end MCU environment. The proposed method moves the write-intensive task stack so that the writes are distributed in the *stack heap*. In this study, we used software simulation to verify the proposed method and measure performance. The experiment simulator is Imperas OVPsim [16], and ARM Cortex M4, among various processors supported by OVPsim, was used. Amazon FreeRTOS [17], [18] was used as an RTOS to implement the proposed method, and the task management and memory management functions of the FreeRTOS kernel were expanded. The proposed time-based stack migration method was verified while performing the examples mentioned above of ARM CMSIS. However, in OVPsim, it is not easy to measure the detailed information related to the movement of the stack, and the size of memory access information collected in a short period is enormous, so OVPsim is not suitable for long-term experiments. Therefore, we measured the write distribution over a long period by modeling the memory access and stack migration behaviors that occur in *stack heap* when simulated with OVPsim. Table 2 shows the primary environments in which experiments were performed.

We measured the overhead caused by the proposed write distribution method and analyzed the worst-case execution time (WCET) at this time. Moreover, we implemented the applications of ARM CMSIS as tasks and measured the write distribution effect while migrating the stack of these tasks. Since our proposed method is affected by the number of tasks, the running time of the task stack, the migration threshold time, the maximum depth, Etc., we performed performance measurements while changing the conditions. Table 3 shows the operation time and stack size of the tasks used in these experiments, and the amount of write access generated in the task stack when each task operates once is shown in Fig. 13.

A. COMPUTATIONAL OVERHEAD

Time-based stack migration causes more operations in the RTOS kernel mode than previously, and time-based stack migration may require additional operations, such as free space conversion, depending on the situation at the time of migration. It may be challenging to meet the real-time performance required by the application due to the computational overhead of executing time-based stack migration once. As mentioned earlier, in this study, the maximum stride depth and the maximum migration depth were introduced to

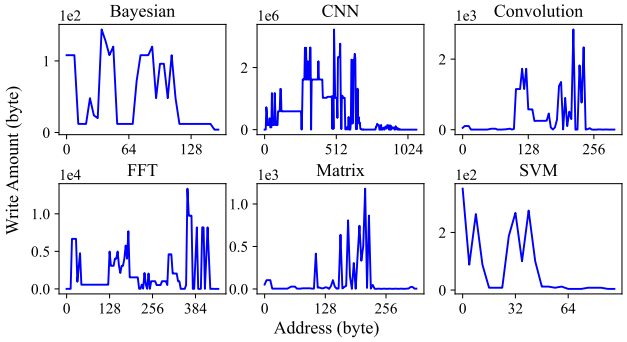


FIGURE 13. Write patterns.

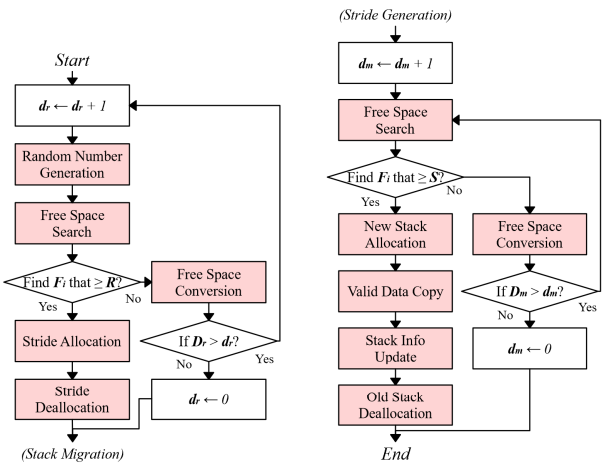
TABLE 3. Running time and stack size of tasks.

Application	Stack size (B)	Running time (tick)
Bayesian	160	0.22
CNN	1088	290.0
Convolution	300	3.2
FFT	456	33.7
Matrix	324	0.57
SVM	156	0.14

limit the maximum computational overhead in the time-based stack migration process. By limiting the maximum amount of computation through the maximum stride depth and maximum migration depth, it is possible to determine WCET by time-based stack migration, which makes it easy to guarantee the real-time of the application.

Since the status of *stack heap*, the size of stacks, and the number of stacks affect the operation of *stride generation* and *stack migration*, and it is not easy to accurately measure the computation overhead of time-based stack migration. Therefore, we divided the time-based stack migration's primary operations and performed these operations several times to obtain the average computation overhead to the second decimal place. The computation overhead is expressed to the second decimal place because the average overhead changes frequently in the first decimal place depending on the *stack heap* status and stack type.

In Fig. 14, the central computation overheads in the *stride generation* and *stack migration* processes are shown in red. Table 4 shows the cycles of these computation overhead generating operations and the cycle required to measure the task running time. *time management* is the process of measuring and updating the running time of a task, and it occurs every tick interrupt and other overheads occur when performing time-based stack migration. We measured the process of searching free space for dynamic allocation by dividing it into *initial free space search* and *free space search*. When the stack is moved more than a certain number of times, a significant amount of space inside *stack heap* is deallocated, which makes it difficult to initially find a sufficient amount of free space during the *stride generation* and *stack migration* process. If the free space search fails, free space conversion occurs, so it is easy to find a relatively sufficient free space



(a) Stride generation (b) Stack migration

FIGURE 14. Major computational overhead factors.

TABLE 4. Average computational overheads in time based stack migration.

Overhead Case	Cycle
Time management (TM)	40
Random number generation (RG)	60
Initial free space search (SS_{Init})	200
Free space search (SS_{Free})	30
Free space conversion (SC_{Free})	110
Stride allocation (A_{Stride})	110
Stride deallocation (D_{Stride})	90
New stack allocation (A_{Stack})	110
Old stack deallocation (D_{Stack})	90
Valid data copy (C_{Stack})	200
Stack info update (U_{Stack})	50

in a subsequent free space search attempt. Also, *initial free space search* contains the action to be performed in dynamic allocation before the free space search proceeds. As a result, in the *stride generation* process and *stack migration* process, *initial free space search*, which searches for free space for the first time, required more cycles than *free space search*, which is an additional search operation.

The overhead that occurs when creating a stride depends on whether or not the stride was successfully created and the number of free space conversions and searches to create the stride. $Success_{Stride}$ represents whether or not the stride generation succeeded as 1 if it succeeded in creating a stride and 0 if it failed. Based on this, when the number of free space switching and search attempts is d_r, the overhead O_{Stride} generated in the *stride generation* process is as follows.

$$\begin{aligned} O_{Stride} = {} & RG \times d_r + SS_{Init} + SS_{Free} \times (d_r - 1) + SC_{Free} \\ & \times (d_r - Success_{Stride}) + (A_{Stride} + D_{Stride}) \\ & \times Success_{Stride} \end{aligned}$$

The overhead incurred during *stack migration* also varies depending on whether the migration was successful and the

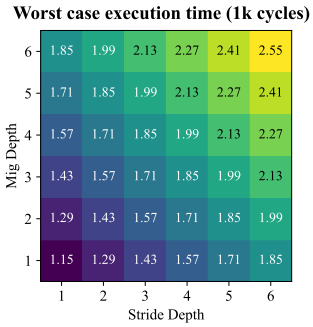


FIGURE 15. WCET according to maximum stride depth and maximum migration depth.

number of free space conversions and searches for migration. $Success_{Stack}$ represents 1 if the migration succeeds and 0 if it fails, indicating whether the migration was successful. Based on this, when the number of free space switching and search attempts is d_m, the overhead $O_{Migration}$ generated in the *stack migration* process is as follows.

$$\begin{aligned} O_{Migration} = {} & SS_{Init} + SS_{Free} \times (d_m - 1) + SC_{Free} \\ & \times (d_m - Success_{Stack}) + (A_{Stack} \\ & + D_{Stack} + C_{Stack} + U_{Stack}) \times Success_{Stack} \end{aligned}$$

Furthermore, when time-based stack migration operates at one tick time, the total computational overhead, including the time management overhead, is as follows.

$$O_{Total} = TM + O_{Stride} + O_{Migration}$$

Referring to the overhead factors above, when the maximum stride depth is 1-6, and the maximum stack migration depth is 1-6, the maximum execution times under each condition are shown in Fig. 15. The maximum execution time occurred when stride generation and stack migration succeeded at the maximum depth of each stride depth and migration depth. Therefore, as the stride depth and migration depth increased, the WCET increased. When the stride depth and migration depth limited in this study were the largest, the WCET corresponded to about 2550 cycles. This cycle occupied about 12.8% of the tick length of this study. The lower the WCET, the easier it is to respond to the real-time requirement, so finding the appropriate stride depth and migration depth is necessary.

B. MEMORY OVERHEAD

The proposed time-based stack migration causes additional writes to the *stack heap* by moving the stack, circular allocation, Etc. First, as the stack moves to a new location, valid data from the old stack must be moved onto the new stack. The stack is migrated from the bottom of the task function in the proposed algorithm. Therefore, the size of valid data in the task stack was 66 bytes, which was constant regardless of the task type. Moreover, circular allocation causes additional writes in creating allocated space, deallocated state space, and free space. Each state space stores information for search

and management, and each state requires 8 bytes. In addition, additional writes occur in the *general heap* by task operation time management, migration flag management, and stack address update, but at the level of 1-4 bytes. However, the above additional writes are negligible, with a size much smaller than the size of the writes that occur on the task stack.

C. ANALYSIS OF STACK MIGRATION

The write variance of the time-based stack migration method is influenced by various factors such as *stack heap* size, task stack size, task stack count, write pattern of task stack, maximum stride depth, maximum migration depth, maximum stride size, task operation time, and migration threshold time. We set up two task configurations to check the effect of write distribution of time-based stack migration according to different task configurations: {*CNN*}, {*Bayesian, Convolution, FFT, Matrix, SVM*}. The two task configurations were tested while changing the migration parameters, and the maximum write, relative standard deviation, average overhead, and migration efficiency were measured. The maximum write indicates the write amount of the address with the most significant accumulated write amount in a *stack heap*. The smaller the maximum write, the slower the NVM lifetime may be reached. Relative standard deviation indicates how far the data falls from the mean. Since a less relative standard deviation means that the data are closer to the mean, a less relative standard deviation means that the writes are better distributed across a *stack heap*. In the same task configuration, since the total and average of write in a *stack heap* are the same regardless of the migration parameters, the lower the relative standard deviation, the higher the write distribution. The average overhead represents the average computation overhead per tick time, and the smaller the size, the smaller the average computation overhead caused by the time-based stack migration. Distribution efficiency refers to the degree of distribution of write per cycle, and the higher the distribution efficiency, the better the distribution of write per the overhead generated during the migration process. We performed experiments to analyze the effect of changing the maximum stride depth, maximum migration depth, maximum stride size, and migration threshold time on the write distribution in two task combinations.

1) ONE CNN TASK

We tried to verify the effect of write distribution by migration parameters when one task was stack migrated through *CNN* single-task configuration experiment. In the *CNN* single task configuration experiment, there was no difference in the migration threshold time. The reason is that the *CNN* task takes 290.0 ticks for one operation, which is longer than the 100 tick time, which is the longest migration threshold time covered in this study, so the result does not change depending on the difference in the migration threshold time. Fig. 16 shows the maximum write, relative standard deviation, average overhead, and distribution efficiency when the CNN task stack was migrated 10^6 times. In the *CNN* single task experiment, the maximum write and relative standard deviation tended to decrease as the maximum migration depth decreased and the maximum stride size increased. However, the maximum stride depth had little effect on the write distribution when it exceeded a specific size.

In this experiment, when the maximum migration depth was one, the reason for the highest write distribution was that there was only one stack to be migrated. If only one task is being migrated in a *stack heap*, the space size transitioned to a deallocated state during the *stack migration* process is always the same. If the maximum migration depth is more significant than a specific size, there is a high possibility that the stack will be reallocated in the space where the stack previously stayed during the *stack migration* process. In addition, near the start position of the *stack heap*, the randomness of the position where the stack moves due to the stride is not sufficiently random, so it was often allocated at a position slightly away from the existing allocation position. As a result, the area in which large write occurred overlaps, and the area in which the write was concentrated is not well distributed. However, if the maximum migration depth was one, if sufficient free space was not found during the initial free space search attempt, the free space conversion operation was performed, and the *stack migration* process was terminated. Therefore, the case of stack reallocation to the same location was reduced; most of the deallocated space was created during the *stride generation* process; the randomness of allocation was increased when the stack was migrated. It can be seen from the stack migration success rate according to the change in the maximum migration depth in Fig. 17. When the maximum migration depth was one, the stack migration success rate was very low, but the maximum write and the relative standard deviation were lower than those with larger maximum migration depths.

As the maximum stride size increased, the maximum write and relative standard deviation decreased. This tendency is because as the maximum stride size increases, the distribution of the sizes of the generated strides becomes more diverse, thus increasing the randomness of the position of movement of the stack through the strides. The average overhead decreased as the maximum migration depth, maximum stride depth, and maximum stride size decreased (Fig. 16(c)). This result is because the number of occurrences of random number generation, free space search, and free space conversion increases as the migration parameters increase. However, since the running time of a *CNN* task is 290 ticks, the time-based stack migration is performed every 290 ticks. There was no significant difference in the average overhead per tick in the change of migration parameters, and it converges to 40 cycles, which is the time management overhead that occurs every tick interrupt. As mentioned above, migration efficiency indicates the degree of distribution of writes per overhead. Since there was no significant difference in the average overhead overall, the case where the maximum migration depth was one was the best. In this case, the distribution efficiency was highest when the maximum

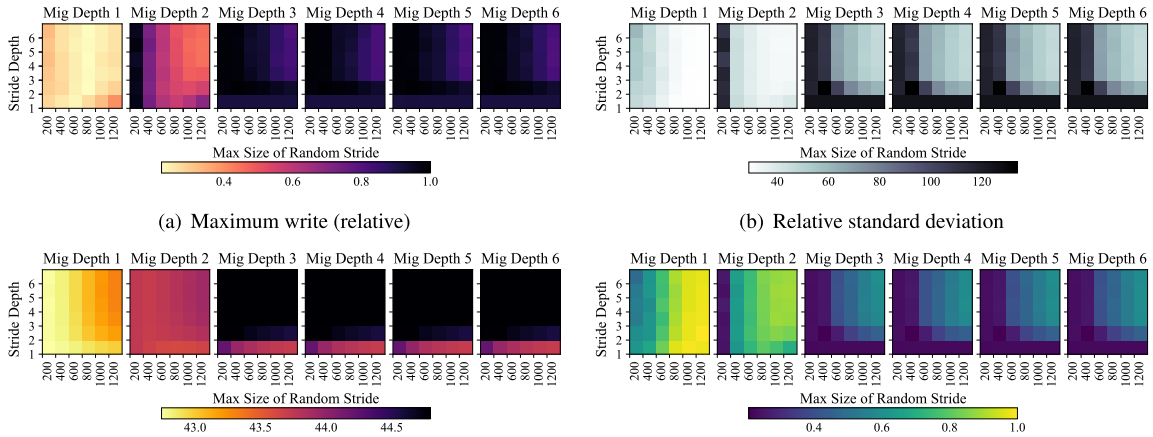


FIGURE 16. When running CNN tasks, the largest write in the *stack heap* according to the migration depth, stride depth, and maximum stride size difference of time-based stack migration, relative standard deviation, average overhead, and distribution efficiency.

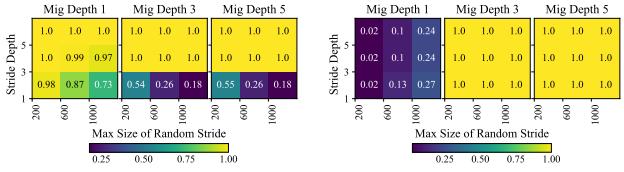


FIGURE 17. When excuting CNN task, stride generation success rate and migration success rate.

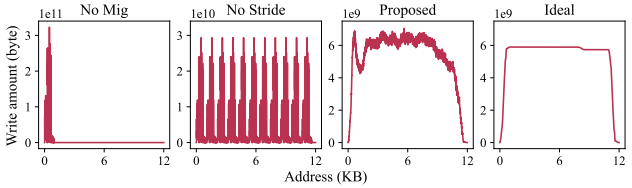


FIGURE 18. Write in the *stack heap* with migration policies.

stride depth was two or more and the maximum stride size was 1000 or more. Since the WCET also increases as the maximum stride depth and maximum stride size increase, the smaller the maximum stride depth and the maximum stride size possible, the better when the distribution efficiency is at a similar level. Therefore, when migrating the stack of a *CNN* single task, the writes were effectively distributed when the maximum migration depth was one, the maximum stride depth was two, and the maximum stride size was 1000.

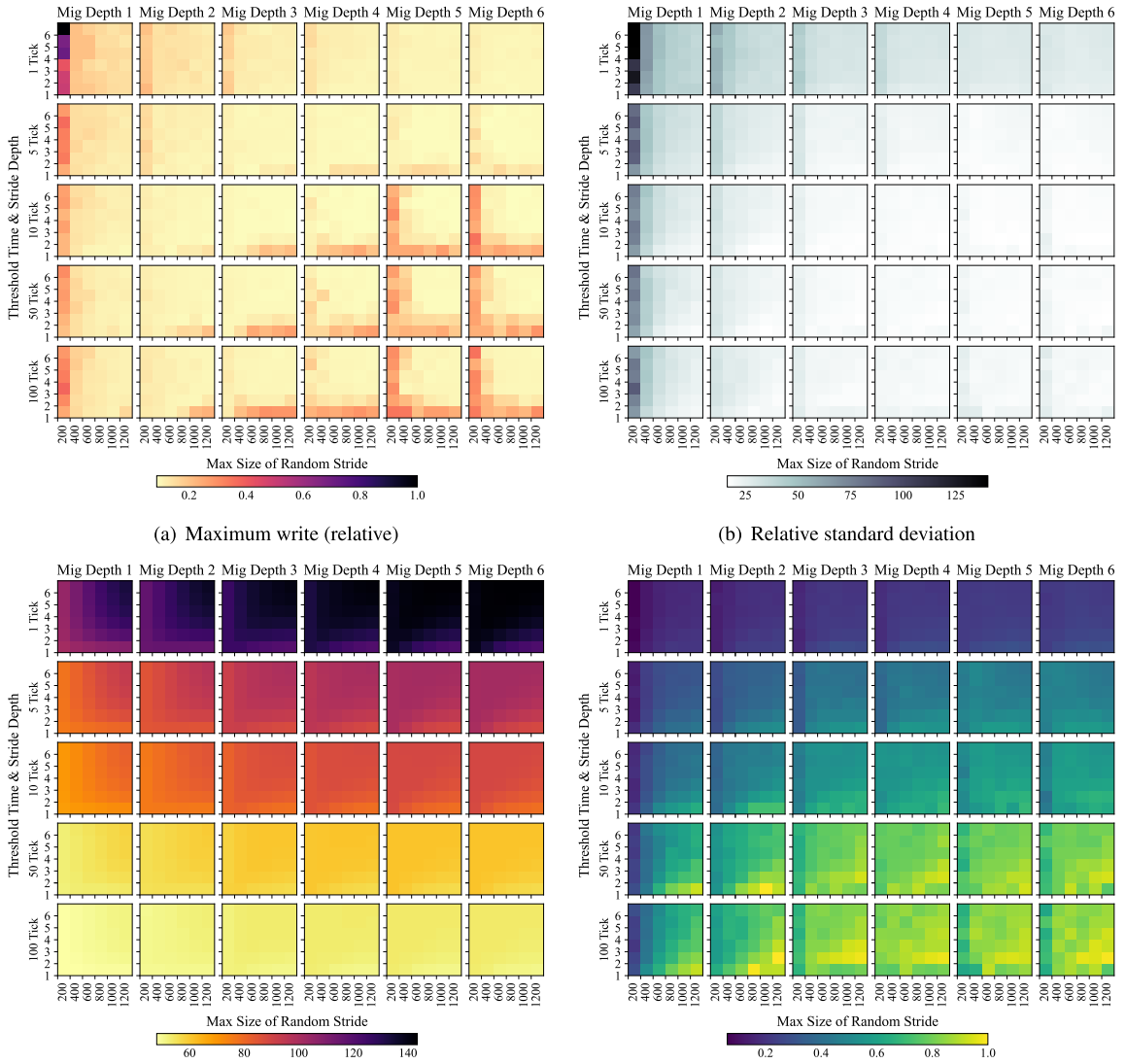
Fig. 18 shows the distribution of writes in a *stack heap* according to the migration policy in a CNN single task case. **No Mig** had no stack migration, so writes were concentrated where the stack was first allocated. **No Stride** was a case of stack migration without generating a stride, and the stack was moved in the *stack heap* according to the size of the stack. **Proposed** was the result of applying the proposed stack migration method, and it was the write distribution when the relative standard deviation was the smallest in the previous Fig. 19. Compared to **No Strdie**, writes were more evenly distributed by random size stride. **Ideal** was an ideal write distribution result that could be expected from the stack migration proposed by the stack and was the write distribution when the stack was moved at 4-byte intervals.

No Stride had a size of 9.1% compared to the maximum write of **No Mig** as the stack is moved to the free space of the *stack heap* by circular allocation. Furthermore, the maximum write of **Proposed** was about 1/4 the size of the maximum write of **No Stride**. Also, **Propose** had a 19.60% higher maximum write compared to **Ideal**, which showed that the proposed time-based stack migration could distribute the writes close to the ideal case.

2) FIVE DSP TASKS

Five tasks were sequentially operated to verify write distribution when migrating multiple stacks for a total of 10^6 tick time. Moreover, while changing the migration parameters, the maximum write, average standard deviation, average overhead, and migration efficiency were measured. As can be seen from Fig. 13 and Table 3, the stack size, write pattern, and operation time of each task are different. That is, there may be a difference in the times each task stack migrated for a certain period.

Fig. 19 shows the maximum write, relative standard deviation, average overhead, and distribution efficiency when five DSP tasks were operated, and time-based stack migration was performed while changing the migration parameters. The greater the deviation of the operation time, the greater the difference in the frequency of stack movement, so it is easy to compare the effect of the difference in the migration threshold time. Therefore, we used five tasks of *Bayesian*, *Convolution*, *FFT*, *Matix*, and *SVM* in this experiment because the difference in operation time of each task was significant. Unlike the previous single task experiment, the operation time of all five tasks used in this experiment is shorter than the largest migration threshold time, so each task's time-based stack



(a) Maximum write (relative)

(b) Relative standard deviation

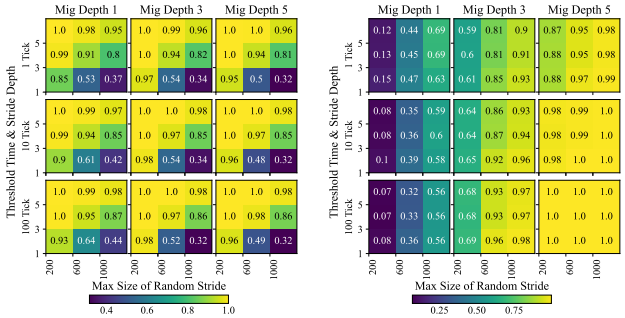
(c) Average overhead (cycle per tick)

(d) Distribution efficiency

FIGURE 19. When executing five DSP tasks, the largest write in the *stack heap* according to the migration depth, stride depth, and maximum stride size difference of time-based stack migration, relative standard deviation, average overhead, and distribution efficiency.

migration execution frequency varies according to the change in the migration threshold time. Therefore, the experiment was conducted while changing the four migration parameters: the maximum migration depth, the maximum stride depth, the maximum stride size, and the migration threshold time. Moreover, Fig. 20 shows the stride generation success rate, stack migration success rate, free space conversion per stride generation attempt, and free space conversion per stack migration attempt.

The maximum writes tended to decrease as the maximum migration depth increased when the migration threshold time was one tick. However, when the migration threshold time was five ticks or more, the maximum write may be relatively large depending on the maximum stride depth and maximum stride size conditions. This increase in maximum write was because, as the migration threshold time increased, the deviation in the number of time-based stack migrations between tasks became relatively large. Table 5 shows the ratio of time-based stack migration according to the change in the migration threshold time. When *SVM* with the shortest one-time operation time executed time-based stack migration once, it indicates the number of times each task executed time-based stack migration. As the migration threshold time increased, the deviation in the number of time-based stack migrations between tasks with short operation time and long operation time increased. If the number of successful stack migrations of one task is much higher than that of other tasks, the stack of the task is more likely to be reallocated to the same location. Another factor that increased the reallocation possibility was a situation in which it was difficult to obtain sufficient randomness due to the stride, and the maximum stride depth was one, or the maximum stride size was 200.



(a) Stride generation success rate (b) Stack migration success rate

FIGURE 20. When executing five DSP tasks, stride generation success rate and migration success rate.

TABLE 5. Time-based stack migration operation ratio by task according to migration threshold time.

Threshold	Bayes	Conv	FFT	Matrix	SVM
1 tick	1.6	8.0	8.0	4.0	1.0
5 ticks	1.6	18.0	36.0	4.0	1.0
10 ticks	1.6	18.0	72.0	4.0	1.0
50 ticks	1.6	22.4	179.0	4.1	1.0
100 ticks	1.6	22.3	238.3	4.1	1.0

In other words, the larger the migration threshold time, the more the stack of the *FFT* with a long operation time is migrated relatively more than the stacks of other tasks; In this situation, writes are relatively less distributed if the randomness caused by the stride generation is low.

The relative standard deviation showed a similar trend to the maximum writing. If the maximum stride depth was two or more, the stride generation was successful in most cases. Therefore, when the maximum stride depth was two or more, there was no significant difference in the write distribution according to the change in the size of the maximum stride depth. In addition, although the relative standard deviation tended to decrease as the maximum stride size increased when the maximum migration depth was three or more, decreasing the relative standard deviation according to the increase in the maximum stride size was insignificant. Moreover, when the migration threshold time was one tick, the relative standard deviation was more significant than when the other migration threshold times were, so the write distribution was low. This phenomenon was also affected by the time-based stack migration frequency of tasks. When the migration threshold time was one tick, more stacks of tasks with short running time were migrated than at longer migration threshold times, and a lot of deallocation state space with a smaller size than the stack of *FFT* task was created. Therefore, securing sufficient free space was difficult when *FFT* stack was migrated. This could be indirectly confirmed by looking at Fig. 20(a), that when the migration threshold time was one, the stack migration success rate was lower than the other migration threshold time conditions under the migration depth three and five conditions. However, when the maximum migration depth was one, the migration success rate was higher than when the migration threshold time was significant because tasks with short operation times and small stacks succeeded in stack migration more. It should be noted that when the migration threshold time was 100 ticks, the relative standard deviation was, on average, 5-10% greater than when the migration threshold time was 10 and 50 ticks; This was because when the migration threshold time was100 ticks, *FFT* task successfully migrated the stack relatively more than other tasks.

The average overhead increased as the maximum migration depth, maximum stride depth, and maximum stride size increased. On the other hand, the average overhead decreased as the migration threshold time increased. The distribution efficiency increased as the migration threshold time increased due to the low average overhead. However, as mentioned above, the write distribution decreased when the migration threshold time was 100 ticks. Therefore, the distribution efficiency was similar to when the migration threshold time was 50 ticks despite the reduced overhead.

VI. DISCUSSION

We proposed a software-based NVM lifetime enhancement technique for low-end MCUs without MMU. We migrated the task stack of RTOS to distribute writes on the stack. When running the CNN task through this, an average of about 0.2% computational overhead occurred, and the maximum write is about 19.6% larger than the ideal case, so the proposed method can effectively distribute the writes with less overhead. However, most studies on NVM lifetime improvement through write distribution did not consider the environment without MMU. On the other hand, we have shown that the lifetime of NVM main memory in low-end MCUs can be improved using only existing hardware.

Regardless, since our study requires an RTOS, there is a limitation in that additional computation overhead occurs compared to the bare metal environment. Second, stack migration may occur consecutively in multiple tasks, making it challenging to ensure hard real-time. Finally, when a plurality of tasks performs migration using the stack heap, the task currently being migrated may have restrictions on movement by other tasks in the stack heap, thereby increasing migration overhead. Therefore, we will find ways to reduce the computational overhead that can occur when stack migration of multiple tasks in a future study.

VII. RELATED WORK

Various wear-leveling studies have improved the lifetime of NVMs over the past few decades. First, there were [10], [11], [14], [19], [20], [21], [22], [23], [24], [25], and [26] in wear-leveling research that changes the location of data based on write-related information. On the other hand, there was [10], [13], and [11] in a study to perform wear leveling by randomly changing the location of data. *Ouroboros* [12] proposed a method to perform wear leveling using both write information and randomness. Some of these studies [19], [24], [25], [27] improved the NVM lifetime by modifying the memory allocation method. However, all of the above

studies remapped a specific unit of memory space or adjusted the location of data in memory through separate hardware. These studies require dedicated hardware to coordinate the data location. In addition, a significant number of studies have remapped data using MMU. However, low-end MCUs do not have an MMU and are small in size, making it difficult to apply such wear-leveling techniques.

Considering an environment without an MMU, such as a low-end MCU, some studies have attempted wear leveling of the stack through a software technique. *Dynamic stack* [28] distributes writes to the stack by adjusting the position of the stack frame to account for wear. However, writes occurring inside the stack frame are not considered. *Loop2Recursion* [29] performs wear-leveling in an environment without MMU. The loop operation induces the concentration of writes by repeated operations in a specific stack area. *Loop2Recursion* changes the loop operation to the form of a recursive function and raises the level of the stack, so that writes occur in a wide area within the stack. However, depending on the execution depth of the recursive function, writes may be more concentrated at a specific address, and the write deviation inside the recursive function is not considered.

Unlike previous studies, our study performed software-based write distribution in an environment without MMU. At this time, the randomness of the position of the stack is increased, so that writes occurring on the stack are evenly distributed within the stack heap. Unlike previous studies, our study performed software-based write distribution in an environment without MMU. At this time, the randomness of the position of the stack is increased, so that writes occurring on the stack are evenly distributed within the stack heap.

VIII. CONCLUSION

We proposed a method of distributing writes by migrating the task stack of FreeRTOS to improve the lifespan of NVM in a low-end MCU environment without an MMU. We used a task operation time for selecting the migration target task. Since moving the stack can impair pointer validation, we limited the migration condition of the stack and updated the pointer after the stack moved. We introduced a circular allocation and a dual heap structure to avoid pointer validation problems due to stack movement and improve write distribution limited by the existing dynamic allocation. Also, we used random strides to allocate the stack in various places within the heap. In addition, we set migration parameters to limit the amount of computational overhead. Through this, the maximum write size was 19.6% larger than the ideal stack migration under the condition of executing a CNN single task. At this time, the instruction increased by 0.2% due to computational overhead. The proposed method can operate only with existing hardware resources and has a small overhead, so it is expected to be easily applied to other low-end MCUs.

REFERENCES

[1] S. H. Kang, "Embedded STT-MRAM for energy-efficient and cost-effective mobile systems," in *Symp. VLSI Technol. (VLSI-Technology), Dig. Tech. Papers*, Jun. 2014, pp. 1–2.
[2] P. P. Ray, "A review on TinyML: State-of-the-art and prospects," *J. King Saud Univ. Comput. Inf. Sci.*, vol. 34, no. 4, pp. 1595–1623, Apr. 2022.
[3] N. Suda and D. Loh, "Machine learning on ARM cortex-M microcontroller," Tech. Rep., 2019.
[4] X. Wang, M. Magno, L. Cavigelli, and L. Benini, "FANN-on-MCU: An open-source toolkit for energy-efficient neural network inference at the edge of the Internet of Things," *IEEE Internet Things J.*, vol. 7, no. 5, pp. 4403–4417, May 2020.
[5] F. Arnaud, P. Ferreira, F. Piazza, A. Gandolfo, P. Zuliani, P. Mattavelli, E. Gomiero, G. Samanni, J. Jasse, C. Jahan, and J. P. Reynard, "High density embedded PCM cell in 28 nm FDSOI technology for automotive micro-controller applications," in *IEDM Tech. Dig.*, Dec. 2020, p. 24.
[6] Y. Chen, X. Wang, W. Zhu, H. Li, Z. Sun, G. Sun, and Y. Xie, "Access scheme of multi-level cell spin-transfer torque random access memory and its optimization," in *Proc. 53rd IEEE Int. Midwest Symp. Circuits Syst.*, Aug. 2010, pp. 1109–1112.
[7] M. Aoki, H. Noshiro, K. Tsunoda, Y. Iba, A. Hatada, M. Nakabayashi, A. Takahashi, C. Yoshida, Y. Yamazaki, T. Takenaga, and T. Sugii, "Novel highly scalable multi-level cell for STT-MRAM with stacked perpendicular MTJs," in *Proc. Symp. VLSI Technol.*, Jun. 2013, pp. T134–T135.
[8] C. Yoshida, T. Ochiai, Y. Iba, Y. Yamazaki, K. Tsunoda, A. Takahashi, and T. Sugii, "Demonstration of non-volatile working memory through interface engineering in STT-MRAM," in *Proc. Symp. VLSI Technol. (VLSIT)*, Jun. 2012, pp. 59–60.
[9] J. J. Kan, C. Park, C. Ching, J. Ahn, Y. Xie, M. Pakala, and S. H. Kang, "A study on practically unlimited endurance of STT-MRAM," *IEEE Trans. Electron Devices*, vol. 64, no. 9, pp. 3639–3646, Sep. 2017.
[10] A. P. Ferreira, M. Zhou, S. Bock, B. Childers, R. Melhem, and D. Mosse, "Increasing PCM main memory lifetime," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2010, pp. 914–919.
[11] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *Proc. Int. Symp. Comput. Archit. (ISCA)*, vol. 37, no. 3, pp. 14–23, Jun. 2009.
[12] Q. Liu and P. Varman, "Ouroboros wear leveling for NVRAM using hierarchical block migration," *ACM Trans. Storage*, vol. 13, no. 4, pp. 1–31, Dec. 2017.
[13] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Dec. 2009, pp. 14–23.
[14] D. Liu, T. Wang, Y. Wang, Z. Shao, Q. Zhuge, and E. Sha, "Curling-PCM: Application-specific wear leveling for phase change memory based embedded systems," in *Proc. 18th Asia South Pacific Design Automation Conf. (ASP-DAC)*, Jan. 2013, pp. 279–284.
[15] *ARM Cortex-M4 Processor Technical Reference Manual*, ARM, 2009.
[16] *Simulation Control of Platforms and Modules User Guide Imperas Software Limited Simulation Control of Platforms and Modules User Guide*, I. S. Limited, 2020.
[17] *The FreertosT Reference Manual*, A. W. Services, 2017.
[18] R. Barry, "Mastering the freertos T real time kernel," Tech. Rep., 2016.
[19] H. Aghaei Khouzani, Y. Xue, C. Yang, and A. Pandurangi, "Prolonging PCM lifetime through energy-efficient, segment-aware, and wear-resistant page allocation," in *Proc. Int. Symp. Low Power Electron. Design*, Aug. 2014, pp. 327–330.
[20] C.-H. Chen, P.-C. Hsiu, T.-W. Kuo, C.-L. Yang, and C.-Y.-M. Wang, "Age-based PCM wear leveling with nearly zero search cost," in *Proc. 49th Annu. Design Autom. Conf. (DAC)*, New York, NY, USA, 2012, pp. 453–458.
[21] J. Dong, L. Zhang, Y. Han, Y. Wang, and X. Li, "Wear rate leveling: Lifetime enhancement of PRAM with endurance variation," in *Proc. 48th Design Autom. Conf. (DAC)*, New York, NY, USA, 2011, pp. 972–977.
[22] V. Gogte, W. Wang, S. Diestelhorst, A. Kolli, P. M. Chen, S. Narayanasamy, and F. T. Wenisch, "Software wear management for persistent memories," in *Proc. 17th USENIX Conf. File Storage Technol.*, Boston, MA, USA, Feb. 2019, pp. 45–63.
[23] Y. Han, J. Dong, K. Weng, Y. Wang, and X. Li, "Enhanced wear-rate leveling for PRAM lifetime improvement considering process variation," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 24, no. 1, pp. 92–102, Jan. 2016.
[24] W. Li, Z. Shuai, C. J. Xue, M. Yuan, and Q. Li, "A wear leveling aware memory allocator for both stack and heap management in PCM-based main memory systems," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2019, pp. 228–233.

[25] S. Yu, N. Xiao, M. Deng, Y. Xing, F. Liu, Z. Cai, and W. Chen, "WAlloc: An efficient wear-aware allocator for non-volatile main memory," in *Proc. IEEE 34th Int. Perform. Comput. Commun. Conf. (IPCCC)*, Dec. 2015, pp. 1–8.
[26] C. Hakert, K.-H. Chen, H. Schirmeier, L. Bauer, P. R. Genssler, G. von der Brüggen, H. Amrouch, J. Henkel, and J.-J. Chen, "Software-managed read and write wear-leveling for non-volatile main memory," *ACM Trans. Embedded Comput. Syst.*, vol. 21, no. 1, pp. 1–24, Jan. 2022.
[27] J. Zhu, S. Li, and L. Huang, "Wamalloc: An efficient wear-aware allocator for non-volatile memory," in *Proc. IEEE 22nd Int. Conf. Parallel Distrib. Syst. (ICPADS)*, Dec. 2016, pp. 625–634.
[28] Q. Li, Y. He, Y. Chen, C. J. Xue, N. Jiang, and C. Xu, "A wear-leveling-aware dynamic stack for PCM memory in embedded systems," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2014, pp. 1–4.
[29] W. Li, L. Wu, M. Yuan, C. J. Xue, J. Xue, and Q. Li, "Loop2Recursion: Compiler-assisted wear leveling for non-volatile memory," in *Proc. IEEE 38th Int. Conf. Comput. Design (ICCD)*, Oct. 2020, pp. 581–588.

JEONGMIN LEE received the B.S. degree from the Department of Electronic Engineering, Hanyang University, South Korea, in 2014, where he is currently pursuing the Ph.D. degree with the Department of Electronics and Computer Engineering.

His research interests include embedded computing and the IoT devices.

MOONSEOK JANG received the B.S. degree in electronic engineering from Hanyang University, Seoul, South Korea, in 2014, where he is currently pursuing the Ph.D. degree with the Department of Electronics and Computer Engineering. His research interests include computer architecture, embedded systems, and flash memory storage.

KEXIN WANG (Student Member, IEEE) received the B.S. degree from the School of Optoelectronic Engineering, Changchun University of Science and Technology, in 2017. She is currently pursuing the Ph.D. degree with the Department of Electronic and Computer Engineering, Hanyang University, Seoul, South Korea. Her research interest includes high-performance solid state drive architecture.

INYEONG SONG received the B.S. degree in electronic engineering from Myongji University, Yongin, South Korea, in 2018. He is currently pursuing the Ph.D. degree with the Department of Electronics and Computer Engineering, Hanyang University, Seoul, South Korea. His research interests include embedded systems and NAND flash memory-based storage systems.

HYEONGGYU JEONG received the B.S. degree in electronic engineering from Hanyang University, Seoul, South Korea, in 2019, where he is currently pursuing the Ph.D. degree with the Department of Electronic Engineering. His research interests include computer architecture, next generation memory, and storage systems.

JINWOO JEONG (Student, IEEE) received the B.S. degree from the Department of Electronic Engineering, Hanyang University, South Korea, in 2015, where he is currently pursuing the Ph.D. degree with the Department of Electronics and Computer Engineering. His research interests include nand flash-based storage systems and error-correction codes.

YONG HO SONG received the B.S. and M.S. degrees in computer engineering from Seoul National University, Seoul, South Korea, in 1989 and 1991, respectively, and the Ph.D. degree in computer engineering from the University of Southern California, Los Angeles, CA, USA, in 2002.

He worked as a Professor at the Department of Electronic Engineering, Hanyang University, Seoul. He is currently the Executive Vice President with Samsung Electronics Company Ltd. His current research interests include the system architecture and software systems of mobile embedded systems, which further include SoC, NoC, multimedia on multicore parallel architecture, and NAND flash-based storage systems.

Dr. Song has served as a Program Committee Member of several prestigious conferences, including the IEEE International Parallel and Distributed Processing Symposium, IEEE International Conference on Parallel and Distributed Systems, and IEEE International Conference on Computing, Communication, and Networks.

JUNGWOOK CHOI (Member, IEEE) received the B.S. and M.S. degrees in electrical and computer engineering from Seoul National University, South Korea, in 2008 and 2010, respectively, and the Ph.D. degree in electrical and computer engineering from the University of Illinois at Urbana—Champaign, USA, in 2015. He worked at the IBM T. J. Watson Research Center as a Research Staff Member, from 2015 to 2019. He is currently an Assistant Professor with Hanyang University, South Korea. His research interest includes the efficient implementation of deep learning algorithms. He has received several research awards, such as the DAC 2018 Best Paper Award. He has actively contributed to academic activities, such as the Technical Program Committee of DATE 2018–2020 (the Co-Chair) and DAC 2018–2020, and the Technical Committee (DiSPS) in IEEE Signal Processing Society.