Thread Evolution Kit for Optimizing Thread Operations on CE/IoT Devices

Geunsik Lim, *Graduate Student Member, IEEE*, Donghyun Kang, and Young Ik Eom

***Abstract*—Most modern operating systems have adopted the one-to-one thread model to support fast execution of threads in both multi-core and single-core systems. This thread model, which maps the kernel-space and user-space threads in a one-to-one manner, supports quick thread creation and termination in high-performance server environments. However, the performance of time-critical threads is degraded when multiple threads are being run in low-end CE devices with limited system resources. When a CE device runs many threads to support diverse application functionalities, low-level hardware specifications often lead to significant resource contention among the threads trying to obtain system resources. As a result, the operating system encounters challenges, such as excessive thread context switching overhead, execution delay of time-critical threads, and a lack of virtual memory for thread stacks. This article proposes a state-of-the-art Thread Evolution Kit (TEK) that consists of three primary components: a CPU Mediator, Stack Tuner, and Enhanced Thread Identifier. From the experiment, we can see that the proposed scheme significantly improves user responsiveness (7x faster) under high CPU contention compared to the traditional thread model. Also, TEK solves the segmentation fault problem that frequently occurs when a CE application increases the number of threads during its execution.**

***Index Terms*—Thread model, thread optimization, thread stack, thread scheduling, thread manager.**

I. Introduction

As DIGITAL consumer electronics (CE) devices such as a smart refrigerator [1] and smart television become common, it is important for traditional software layers to be optimized to mitigate the limitations of CE devices [2]–[4]. Nowadays, such devices are generally called IoT devices because they interoperate each other with Internet facilities and sensor modules. Meanwhile, traditional operating systems of computing systems have adopted a model of one-to-one mapping between kernel-space and user-space threads because it allows opportunities for improving the scalability and performance of the system [5]–[15]. Unfortunately, this model does not fit for CE/IoT devices that have lower hardware specifications, because the model incurs some problems in that the threads running on CE/IoT devices often unintentionally spend a significant amount of time in taking the CPU resource and the frequency of context switch rapidly increases due to the limited system resources, degrading the performance of the system significantly. In addition, since CE/IoT devices usually have limited memory space, they may suffer from the segmentation fault [16] problem incurred by memory shortages as the number of threads increases and they remain running for a long time.

Some engineers have attempted to address the challenges of IoT environments such as smart homes by using better hardware specifications for CE/IoT devices [3], [17]–[21]. Unfortunately, this approach is inefficient and expensive because high-performance hardware requirement increases the manufacturing costs of CE/IoT devices. Other researchers and engineers have implemented *dual-version applications*: a generic version for normal computing systems and a light-weight version for CE/IoT systems [20]. However, this approach also increases the cost of maintaining the applications because both versions of the software code must be modified when a software update is made. Meanwhile, in traditional systems, there is no concept of thread priority, and thus, it is difficult to identify time-critical threads that require quick responsiveness. As a result, when many active threads are running at the same time, all the threads, including those that are time-critical, unprejudicedly compete for system resources. This leads to performance collapse along with dropping user responsiveness.

This article proposes a *Thread Evolution Kit (TEK)* that adopts the modern one-to-one thread model for CE/IoT devices while leveraging its benefits in the user space. TEK is composed of three primary components: a CPU Mediator, Stack Tuner, and Enhanced Thread Identifier. First, we designed the CPU Mediator to help developers set the priority of time-critical threads. TEK is also implemented with a priority-based scheduler that isolates high-priority threads (i.e., time-critical threads) from normal ones. The goal of the Stack Tuner is to determine how much memory space should be allocated for thread stacks. Also, it is used to avoid the problem of *coarse-grained stack management* in which existing operating systems give each thread more stack memory than is actually used by the thread. To implement the Stack Tuner, this article revisited the existing stack management mechanism and

Manuscript received July 24, 2020; revised October 12, 2020; accepted October 20, 2020. Date of publication October 23, 2020; date of current version November 25, 2020. This work was partly supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (IITP-2015-0-00284, (SW Starlab) Development of UX Platform Software for Supporting Concurrent Multi-users on Large Displays) and Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (NRF-2017R1A2B3004660). *(Corresponding author: Young Ik Eom.)*

Geunsik Lim and Young Ik Eom are with the Department of Electrical and Computer Engineering, Sungkyunkwan University, Suwon 16419, South Korea (e-mail: leemgs@skku.edu; yieom@skku.edu).

Donghyun Kang is with the Department of Computer Engineering, Changwon National University, Changwon 51140, South Korea (e-mail: donghyun@changwon.ac.kr).

Digital Object Identifier 10.1109/TCE.2020.3033328

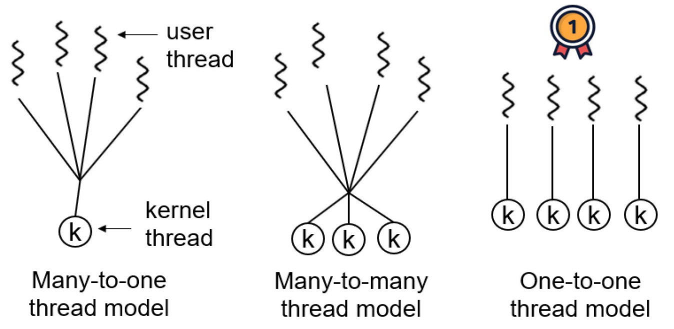

Fig. 1. Three types of thread models. Popular operating systems [5], [22]–[24] adopt the one-to-one thread model as their major thread model.

designed a scheme that could automatically assign an appropriate stack size by obtaining the thread's actual stack usage. With the proposed scheme, software developers for CE/IoT devices can easily use TEK to prevent virtual memory shortages. Meanwhile, to employ TEK, software developers must specify information on the threads using the special APIs provided by TEK. In order to correctly handle each thread, the Enhanced Thread Identifier inspects this information whenever the program codes are compiled. For evaluation, TEK was implemented on an CE/IoT development board and compared with the conventional system. Surprisingly, the results show that the response time of time-critical threads was accelerated by up to 7x, and the amount of memory space was saved by up to 3.4x, compared with the conventional software platform.

The remainder of this article is organized as follows. Section II describes the strong and weak points of each thread model. Section III discusses the observation results of conventional thread operations on CE/IoT devices. Section IV presents the design and implementation of the proposed schemes, and Section V shows the evaluation results. Related work is described in Section VI, and finally, Section VII concludes this article.

II. Background

This section compares the strong and weak points of the three major thread models: many-to-one, many-to-many, and one-to-one, as shown in Fig. 1. Then, it addresses how modern operating systems such as Linux have evolved the thread mapping model.

A. Comparison of Thread Models

The model for thread mapping between the user space and kernel space has a great influence on the behavior of the threads from their creation to termination. Fig. 1 shows the different operation flows of the common thread models and how each works. This section addresses the features, strengths, and weaknesses of each thread model.

Many-to-one thread model: Many operating systems have commonly used this model as it allows for simple implementation and portability. In addition, this model provides for much quicker context switching compared with other models because all threads are managed in the user space. Unfortunately, when this model is applied, a thread can block the flows of other threads for a long time. For example, if one thread triggers a system call that leads to resource contention, other threads of the same application must also wait until the thread gets the resource. Moreover, this model cannot fully take advantage of the benefits of a multi-processor architecture because it handles only one application with a single processor, even though the application is composed of multiple threads [12], [13].

Many-to-many thread model: This model allows for parallel mapping between the user-space and kernel-space threads. In this model, most context switching operations occur in the user space, thus, this model shares kernel objects among the user-space threads. However, this model requires more effort to implement than the many-to-one model due to its complicated structure. Also, now that priority is assigned to the threads through the thread manager in the user space, the threads cannot directly access the kernel objects to obtain CPU resources [25], [26].

One-to-one thread model: Most modern operating systems employ the one-to-one thread model. This model manages threads directly in the kernel space so that kernel-level scheduling is possible for each thread. In multi-processor systems, multiple threads can run on different CPUs simultaneously, and there may be no delay, even during blocks of system services. When one thread calls a blocking system call, the other threads do not need to be blocked. However, since this model performs context switching in the kernel space, the context switching operation is relatively slow compared to the many-to-one model. Also, with each thread taking up kernel resources, if an application creates many threads, the performance of the threads depends heavily on the hardware specifications [5], [27], [28].

It is now common for an application to be developed by dozens of developers due to the increased size and complexity of applications, and so, the number of threads in modern CE/IoT devices has increased from tens to hundreds. Meanwhile, when the number of user-space threads in one application increases to hundreds, it is difficult for the application developer to know the roles of threads created by other developers because the existing thread models do not provide developers with a mechanism to monitor thread functionalities [27]–[29]. As a result, it becomes increasingly difficult to control and optimize their behavior.

B. Evolution of Linux Thread Model

In recent decades, developers implemented a multi-threaded application with the many-to-one thread model. The many-to-one thread model made thread implementation easy by providing application developers with intuitive user-space thread operations as discussed in Section II-A. However, user-level threads based on the many-to-one thread model could not utilize multi-processors directly without kernel-level support from the operating system. As a result, with the advent of multi-processor systems, operating systems added kernel-level support for threads [5], [13].

The Linux kernel introduced Fast User-Level muTEX (FUTEX) [25] to support light and fast synchronization. This can be used when the threads access shared resources in the multiprocessor environment. The system calls of the FUTEX facility provide application developers with a fast user-level synchronization mechanism [30]–[32]. The one-to-one thread model of Linux supports real-time FUTEX behavior for user-space threads. Then, the Linux kernel adopts LinuxThread, which uses a *clone* system call to enable user-space real-time thread operations [5], [25]. Since the *clone* system call of the one-to-one thread model provides a mechanism that tracks threads to speed up thread creation, Linux dramatically improves the scalability and performance of thread execution. Also, now that the one-to-one thread model eliminates the user-level thread manager for fast thread operation, it both simplifies the operation flow of the threads and significantly accelerates the execution speed of thread termination. As a result, applications no longer depend on a thread manager that may cause context switching and performance degradation [5], [10], [33].

The one-to-one threaded programs can run even faster since Linux introduced the O(1) scheduler, which consists of run queues and bitmap priority arrays. This improves scalability without adding a performance penalty in multi-processor environments [6], [34]. As a result, when the O(1) scheduler allowed threads to run quickly in high-performance multi-core server environments, modern operating systems adopted the one-to-one thread model as its standard thread model.

Finally, the one-to-one thread model safely solves existing POSIX compliance problems because it performs signal handling of the threads in kernel space. Moreover, because Linux maps the user-space thread to the Light-Weight Process (LWP) in kernel space, it completely links the system resource usage of the thread to that of the LWP of the Linux kernel. On Linux, the LWP refers to processes sharing the same memory address space and other resources in the kernel space. Therefore, the thread library can correctly monitor the thread behaviors of an application using the pseudo filesystems (e.g., *proc* and *sysfs*) [35] in Linux.

Even though the Next-Generation POSIX Threads (NGPT) [25] proposed a many-to-many model in which many user threads are mapped onto many kernel threads, unlike the existing one-to-one thread model of the Linux kernel, it cannot solve all the problems of the user-space library threads. The main reason that the traditional LinuxThread has been used as the dominant thread library for so long time is that the kernel-level threads of the operating system solve fault handling and thread performance problems.

III. Observation

This section discusses the observation that modern CE/IoT devices bring new challenges to the existing one-to-one thread model.

A. CPU Contention Among the Threads on Low-End CE/IoT Devices

The latest Linux kernel supports two types of CPU schedulers: the O(1) and the Completely Fair Scheduler (CFS) [21]. The O(1) scheduler provides CPU scheduling of real-time threads with a First-In-First-Out (FIFO) or Round-Robin (RR) scheduling policy, and controls each thread according to its fixed priority. On the other hand, the CFS scheduler supports fair scheduling of non-real-time threads with a NORMAL scheduling policy, controlling each thread according to its dynamic priority [36]. The fixed priority does not change while scheduling threads, while strictly maintaining the scheduling sequence. On the contrary, the dynamic priority changes from time to time while scheduling threads because CFS dynamically recalculates the weight values of the threads in the system. Typically, user-space applications create new threads that are controlled by the NORMAL scheduling policy, and they are scheduled in a time-sharing manner. In detail, the NORMAL scheduling policy manages CPU resources using the virtual run-time [37] with the red-black tree which is usually used for efficient self-balancing binary search [38] to ensure that all threads use CPU resources fairly [36]. The CFS scheduler adopts the notion of virtual run-time, which should be equal for all tasks, where the virtual run-time normalizes the value of the real run-time of a given thread with its *nice* value (user-level thread priority). At this time, the CFS scheduler employs a red-black tree to support efficient self-balancing binary search algorithm. The red-black tree, in combination with the virtual run-time, ensures that higher priority tasks gain access to CPU resources more frequently without starving the lower priority tasks.

However, the CFS scheduler does not consider low-end CE/IoT devices, especially when the CE/IoT device has to execute time-critical threads with more favor under high CPU contention. The *nice* value of −20 to 19 entered by the application developer is based on the 40 weight values defined by CFS as an array. For example, if a developer creates four threads with *nice* values of 1, 2, 3, and 4, the CPU usage becomes 26.5%, 25.5%, 24.5%, and 23.5%, respectively, because the operating system applies equitable weight values to the threads. In other words, the CFS scheduler focuses on fair resource distribution among the threads in the system. Therefore, if the threads frequently compete to obtain available CPU resources, the existing scheduler decreases the response time of the time-critical threads in which user responsiveness is important.

Fig. 2 shows the delay in processing time-critical threads that measure the current temperature during high CPU

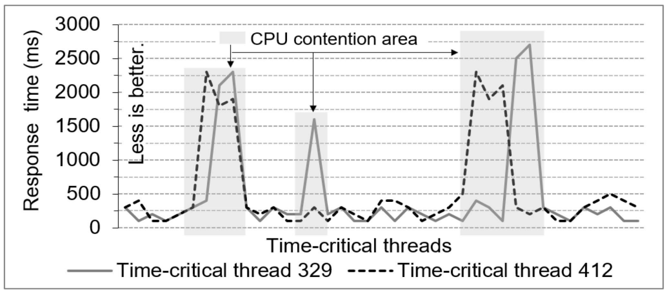

Fig. 2. The response time of two time-critical threads in an urgent group during CPU contention. The gray rectangle represents a region where threads compete for available CPU resources. Time-critical thread 412 probes the current temperature from a thermal peripheral device, while time-critical thread 329 displays the temperature.

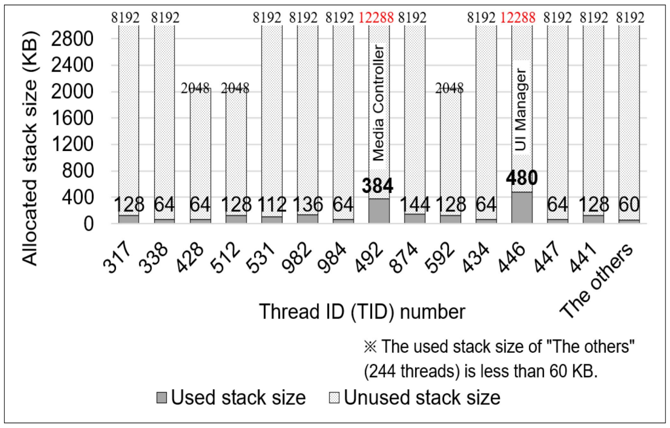

Fig. 3. The actual stack usage of 258 user-space threads on a CE/IoT device.

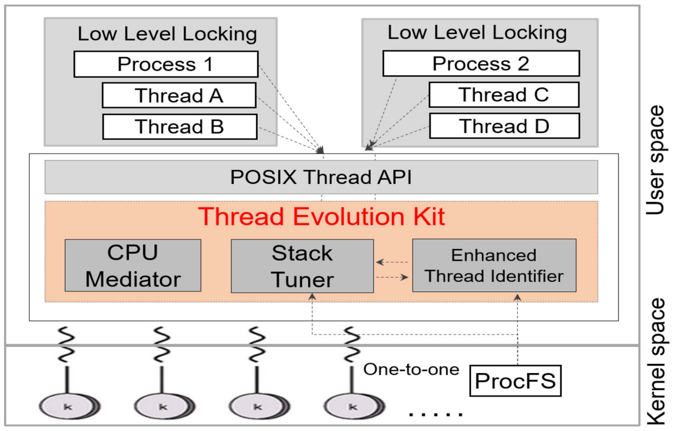

Fig. 4. Overall architecture and operation flow of TEK.

(Embedded CPU quad-core 1.2GHz) contention in a real CE/IoT system environment such as a refrigerator. In the experiment, the time-critical threads and the background service threads in the same thread group were intensely competing with one another for CPU resources. As a result, the system frequently reproduced unpredictable processing delays for the time-critical threads. Even though high-end hardware (e.g., X86 CPU) can minimize the frequency of CPU contention compared to low-end hardware (e.g., Embedded CPU), most CE/IoT devices require an energy-efficient CPU for low power consumption and smaller die size. Therefore, it is crucial to optimize the processing speed of time-critical threads under high CPU contention in low-end devices. In Section IV, this article describes the design and implementation of the technique proposed to solve this problem in detail.

B. Segmentation Fault of New Threads on CE/IoT Devices

In general, a segmentation fault occurs when a running thread accesses an illegal memory location or tries to write onto a read-only memory location [16]. Surprisingly, it is observed that there is another case when segmentation faults are generated while running an application. When an application requests to create a new thread, the operating system builds a new stack on the virtual memory space and then clones the contexts of the parent process, such as code and data, to that of the new thread. However, whenever a thread is created, the existing operating system gives each thread more space in stack memory than the amount of space the thread actually uses (e.g., in Linux, a stack space of 8 MB or 12 MB is automatically assigned for each thread). Therefore, this coarse-grained stack management problem, which induces a lack of system stack space, may be accelerated over time. Unfortunately, in 32-bit architecture, it is not easy to solve this problem because an application can use a total of 3 GB for the user-space area. For example, if an application simultaneously runs 300 threads with 8 MB stack for each thread, the existing operating system may incur a segmentation fault when creating a new thread. Of course, the frequency of segmentation faults depends somewhat on how the software platform handles the virtual memory area.

Fig. 3 shows the actual stack usage of 258 user-space threads that are run on a CE/IoT device (1 GB RAM LPDDR2 900MHz, 4 GB virtual memory with the memory management unit) with coarse-grained stack management. In Linux, a new thread requires a minimum stack size of 16 KB to establish a data structure of the user-level thread. As shown in Fig. 3, most of the threads allocate much higher stack size than they use in reality. From the analysis, although all threads run tasks with the default stack size (8 MB) of the system, more stack space is allocated for the UI Manager (TID 446) and Media Controller (TID 492). On the contrary, less stack space is allocated for the threads of TID 428, 512, and 592. The reason for this is that the developers employ the *pthread_attr_setstacksize* API [11] in order to directly manipulate the stack space of the threads.

These observations give further motivation to propose TEK because it is believed that it is possible to resolve the resource management problems incurred due to excessive resource contention while running the one-to-one mapping model between kernel-space and user-space threads.

IV. TEK: Design and Implementation

This section introduces the *Thread Evolution Kit (TEK)*, designed in the same spirit as the traditional one-to-one thread model to handle application threads in the user space. However, to develop applications in CE/IoT environments without re-design of the existing one-to-one thread model, TEK optimizes the previous thread model with three key components:

- *CPU Mediator (Section IV-A):* This component supports fine-grained thread management in which each thread is handled based on the priority given by the application developers.
- *Stack Tuner (Section IV-B):* The goal of this component is to optimally allocate stack memory in the virtual address space whenever an application creates a new thread.
- *Enhanced Thread Identifier and New APIs (Section IV-C):* This component is responsible for handling the hundreds of threads running on a CE/IoT device and provides new APIs to designate a thread as time-critical or non-time-critical.

Fig. 4 shows how applications are managed with TEK. TEK provides application developers with POSIX-compatible

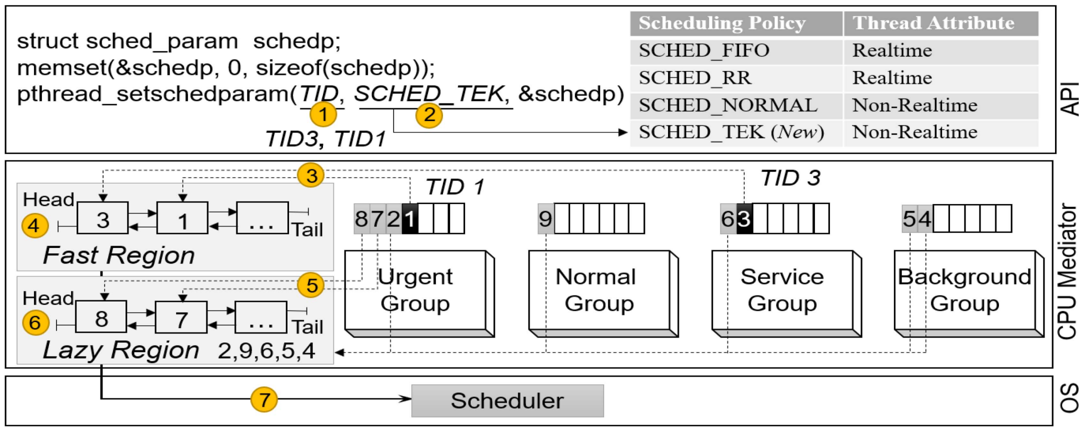
struct sched_param schedp;
memset(&schedp, 0, sizeof(schedp));
pthread_setschedparam(TID, SCHED_TEK, &schedp)
TID3, TID1
Scheduling Policy | Thread Attribute
SCHED_FIFO Realtime
SCHED_RR Realtime
SCHED_NORMAL Non-Realtime
SCHED_TEK (New) Non-Realtime
API
Head
Tail
Fast Region
Lazy Region 2,9,6,5,4
TID 1
TID 3
Urgent Group
Normal Group
Service Group
Background Group
CPU Mediator
Scheduler
OS

Fig. 5. Thread programming model with SCHED_TEK of CPU Mediator for accelerating the response time of time-critical threads.

TABLE I
THE POSIX APIs FOR CPU SCHEDULING OF PROCESSES AND THREADS

API name	Arguments	Who	Type
nice	❶inc	Process	Syscall
setpriority	❷which, ❸who, ❹prio	Process	Syscall
pthread_set schedparam	❺thread, ❻policy, ❼priority	Thread	Libcall

❶ inc: a nice value for the calling process.
❷ which: PRIO_PROCESS, PRIO_PGRP, or PRIO_USER.
❸ who: a process group or real user ID of the calling process.
❹ prio: a value in the range -20 to 19. The default priority is 0.
❺ thread: a thread ID.
❻ policy: a scheduling policy (e.g., SCHED_TEK for CPU Mediator).
❼ priority: a scheduling priority (e.g., a nice value for SCHED_TEK).

thread APIs (e.g., *pthread_setschedparam*) that support optimization of resource management in both low-end CE/IoT devices and existing high-end server systems (i.e., in TEK, the improved *pthread_setschedparam* API is used to run a unified application that is compatible with both low-end and high-end devices). Now, this article discusses the key components of TEK in detail.

A. CPU Mediator

The existing software layer for CE/IoT devices was designed to handle threads with a group scheduling policy (i.e., coarse-grained thread management). Because modern applications create more and more threads, this technique can effectively control CPU resources by grouping the threads of each application. However, this coarse-grained thread management technique may be harmful in modern CE/IoT environments in which threads may require fast responsiveness because it is not easy to predict which thread will be run next.

The CPU Mediator is designed to support fast and predictable thread execution. In particular, the CPU Mediator classifies all threads running on a CE/IoT device into two categories according to their priority: time-critical and non-time-critical. The scheduling priority (❼ in Table I) of each thread is set by calling the APIs supported by TEK as described in Table I and would not be adjusted until the thread terminates. For time-critical threads, this article further implements a new scheduler policy, called SCHED_TEK (❻ in Table I), that offers more chances to obtain CPU resources by delaying non-time-critical threads.

This article considers an example scenario in which time-critical threads are processed. When an application runs time-critical threads to guarantee fast response time, the CPU Mediator changes the policy of the scheduler to SCHED_TEK by calling the *pthread_setschedparam* API in user space. Fig. 5 shows a logical thread migration flow of the CPU Mediator along with the SCHED_TEK policy. The SCHED_TEK policy starts to control the CPU resources according to the following two steps. First, the CPU Mediator looks up the time-critical threads in the group where the user-space thread lays based on its kernel-space thread ID,[1] and then logically migrates them to the *Fast Region*, where the probability of obtaining CPU resources is relatively high, as shown in Fig. 5. Second, the CPU Mediator dynamically drops the priority of each non-time-critical thread running on the CE/IoT device to yield CPU resources to the time-critical threads. Then, it logically migrates all non-time-critical threads to the *Lazy Region*, where the execution of the threads will be delayed until the *Fast Region* becomes empty. The purpose of the *Fast Region* is to accelerate the processing speed of time-critical threads, while that of the *Lazy Region* is to delay the other threads. These regions link or unlink the threads of the existing groups with a doubly-linked list. After the time-critical threads in the *Fast Region* are terminated, the CPU Mediator unlinks the threads belonging to the *Lazy Region* and puts them into their original groups, instantly restoring their scheduling policy.

B. Stack Tuner

As the number of threads increases, segmentation faults may occur frequently due to the lack of system stack space in the virtual memory. Traditional operating systems always allocate stack space as requested by the thread,[2] regardless of how much stack space is actually used by the thread at runtime; this situation is very similar to the *internal fragmentation issue*

[1]The kernel-space thread ID is acquired by using the modified *gettid()* system call where this article replaces FUTEX with the Read-Copy-Update (RCU) mechanism because it is more suitable for read-intensive operations.

[2]*The pthread_attr_setstacksize()* API is used to explicitly allocate the stack memory space.

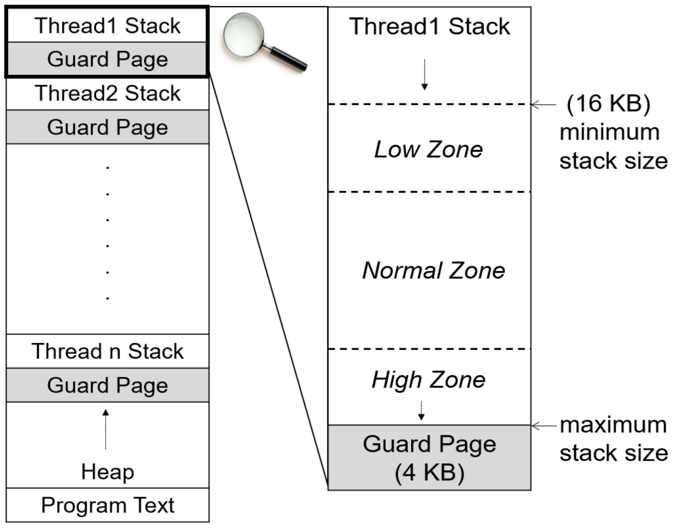

Fig. 6. The stack management structure of the Stack Tuner used to avoid a shortage of virtual memory.

in physical memory. For example, if a thread running for an application only uses 1 MB of stack space after being allocated 100 MB of stack space, a significant amount of virtual memory (i.e., 99%) is wasted. Therefore, the coarse-grained stack management technique mentioned in Section III-B may accelerate the lack of the system stack space over time.

To address the lack of system stack space, this article designed the Stack Tuner to monitor the stack space during the lifetime of each thread. In order to measure the stack usage of each thread, the Stack Tuner periodically obtains information on the *procFS* [35] filesystem and records the peak stack usage of each thread in the *Thread Information Table*, which will be discussed in the next section. Based on the recorded stack usage, the Stack Tuner automatically gives each thread suitable stack space to optimize the memory usage of the applications. For exact guidelines, this article additionally configured three types of zones in the stack space: *Low Zone*, *Normal Zone*, and *High Zone*.[3] Fig. 6 shows how to classify the stack space of each thread. If the peak stack usage of a thread belongs to the *Low Zone* or *High Zone*, the Stack Tuner informs the developers, allowing them to fix the stack space requirement at the next compilation. To deliver this information, this article modifies the Glibc [39] library, which is well-established as the standard library for handling system calls in Linux. In the *Low Zone*, the Stack Tuner points out that the thread is wasting the virtual memory space of the application. On the other hand, if the peak stack usage of the thread reaches the *High Zone* range, the Stack Tuner generates the information that the thread may end up with a stack overflow in the near future. Finally, the Stack Tuner puts the *Guard Page* at the end of the thread's stack to detect a stack overflow.

C. Enhanced Thread Identifier and New APIs

The main purpose of the Enhanced Thread Identifier is to easily identify the characteristics and attributes of each thread

[3]The default size of each zone is configured by the configuration file at the boot time of the CE/IoT device.

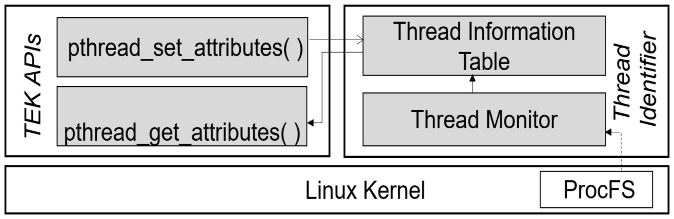

Fig. 7. The operation flow of the Enhanced Thread Identifier.

among hundreds of threads running on a CE/IoT device. In reality, an application calls the *pthread_create* API to create a new thread and the new thread just executes the thread function specified by the third argument of the *pthread_create* API [11], [13], [15], [22]. As a result, it is not easy to determine the role of the thread with the thread ID only. To enhance the identification of a thread, the Enhanced Thread Identifier records the information on a new thread created by an application into an auxiliary table, called *Thread Information Table*.

Fig. 7 shows the structure and relationship of the components of the Enhanced Thread Identifier. The Enhanced Thread Identifier extracts the thread attributes from the parameters of the *pthread_create* or *pthread_set_attributes* API, and then records them in the *Thread Information Table* on-the-fly. The thread attributes include the information on the role of the thread.

Application threads often call a function that connects to a sensor device to receive data from it (e.g., humidity sensor, temperature controller, air pressure sensor, or gas detection sensor). For example, a developer can set "gas detection" as the role of a thread with the *pthread_set_attributes* API in order to easily lookup the gas detection thread running on a device. The *Thread Monitor* in Fig. 7 periodically collects thread information on the running threads (e.g., scheduling policy, scheduling priority, thread creation time, stack size, and virtual memory size) from the *ProcFS* [35] filesystem. The peak usage of the stack space of each thread is measured in this way.

Meanwhile, this article designed novel APIs to set or get the attributes of a thread in the *Thread Information Table*. Application developers can mark a thread as time-critical or non-time-critical by triggering the *pthread_set_attributes* API. When this function is called in the user space with a thread ID and its attributes, the Enhanced Thread Identifier searches for the thread ID in the *Thread Information Table* and saves the thread attributes, including its priority and scheduling policy, into the table. On the other hand, the *pthread_get_attributes* API is used to return the attributes of the thread.

The *Thread Information Table* requires additional memory space to store thread attributes. Considering that modern CE/IoT applications usually run more than 300 threads and the Enhanced Thread Identifier allocates 40 bytes to store the thread attributes for each thread, the Enhanced Thread Identifier requires just 12 KB (300 threads multiplied by 40 bytes) of additional memory space for 300 threads, and so, the additional memory cost is not significant. Also, considering that the read and write operations for managing the thread information are completed within 46 ns and

TABLE II
SYSTEM CONFIGURATION FOR EXPERIMENTS

Content	Item	Specifications
H/W	CPU	Embedded CPU quad-core 1.2GHz
	RAM	1GB LPDDR2 (900MHz)
	Storage	32GB MicroSD
	Sensor Interface	GPIO-40 pin header
S/W	OS	Linux 4.4.15 32bit (LTS)
	Virtual memory	1 GB kernel space and 3 GB user space
	Compiler	GCC 9.1
	C library	Glibc 2.29
	Thread Model	NPTL (Native POSIX Thread Library) [5]

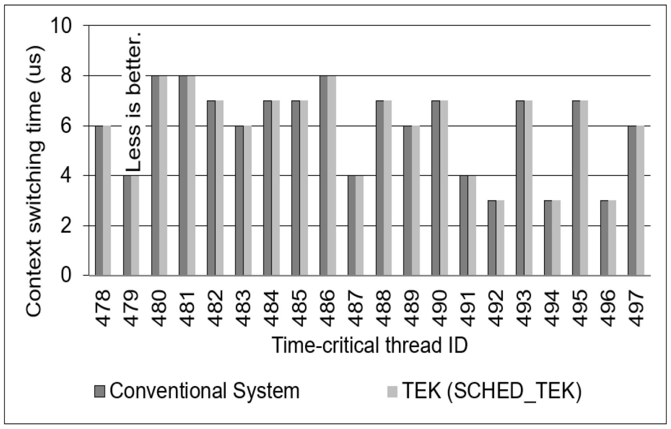

Fig. 8. The context switching time of the threads.

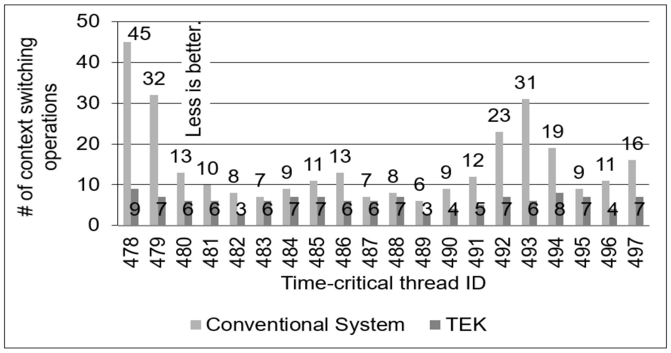

Fig. 9. The frequency of context switching operations on time-critical threads.

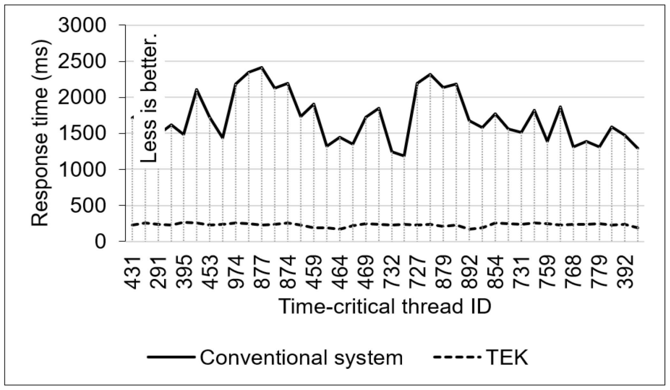

Fig. 10. The evaluation results of the user-space SCHED_TEK policy for improving user responsiveness of time-critical threads under CPU contention.

67 ns, respectively, when the Enhanced Thread Identifier saves the thread information into a typical memory device (1 GB LPDDR2 900MHz), it has little effect on the thread performance of the devices.

V. EVALUATION

This section introduces an experimental environment and then explores how the proposed scheme, TEK, improves not only the response time of the time-critical threads but also the utilization of memory space. In particular, the evaluations in this section answer the following questions: (1) what is the difference between TEK and the conventional system in terms of context switching? (Section V-B), (2) where does the improvement of the response time come from when TEK is enabled on CE/IoT devices? (Section V-C), and (3) how does TEK contribute to stack management at the kernel level? (Section V-D).

A. Experimental Setup

A prototype of TEK was implemented based on a commercial CE/IoT device using an Embedded CPU with 1 GB memory running Linux kernel 4.4. Table II summarizes the evaluation setup in detail. The benefits of TEK were compared with the conventional kernel wherein the CPU scheduler provides coarse-grained control of threads and provides memory management based on a fixed-sized stack space. In this article, all evaluations were conducted by categorizing threads as time-critical or non-time-critical so as to understand the performance difference of the events triggered by users. If a thread frequently handles user-level events during a short time period, it is considered to be time-critical because it requires a short response time. Otherwise, it is considered to be non-time-critical. The evaluation results were measured during the creation of 2000 threads after finishing the boot procedure.

B. Context Switching of Threads

This article first focuses on the performance of TEK in terms of context switching since a performance drop may be caused by the use of fine-grained thread management along with the thread's priority. Fig. 8 shows the experimental results for the context switching time. In the figure, the x-axis represents the thread ID of the time-critical threads, and the y-axis represents the context switching time of the time-critical threads. The proposed scheme has results similar to the conventional scheme even though it includes more behaviors for categorizing threads into time-critical and non-time-critical ones. This is possible because the proposed scheme offloads the operations for thread classification to the CFS scheduler by logically migrating the threads between the regions implemented with the doubly-linked list as depicted in Section IV-A.

Meanwhile, Fig. 9 shows the number of context switching operations centered around the time-critical threads. This figure confirms that TEK significantly reduces the number of context switching operations compared with the conventional method. In the best case, TEK reduces the number of context switching operations by up to 41%. The reason behind this is that TEK assigns more CPU time to the time-critical threads by

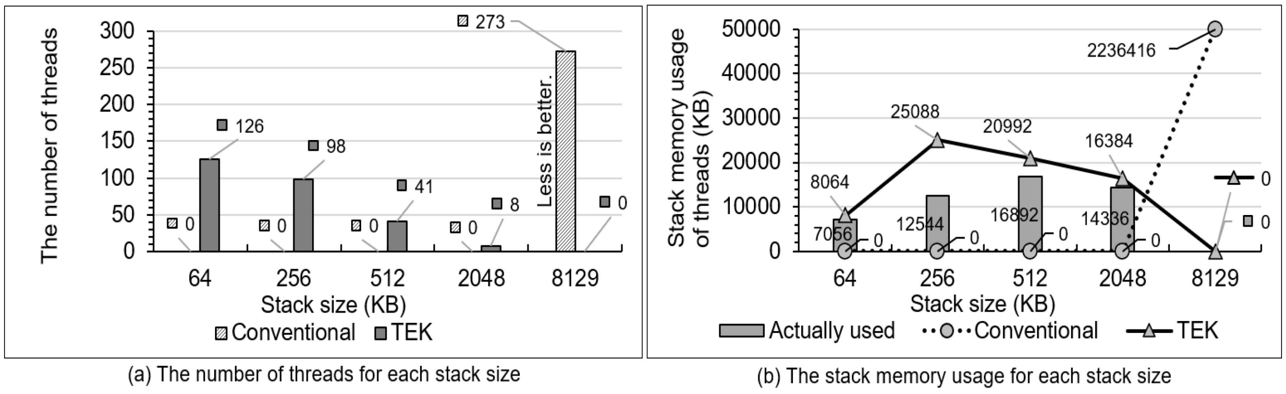

Fig. 11. Virtual memory consumption for each stack size of the threads.

isolating them from the group scheduling policy. As a result, the time-critical threads can speed up their response time by up to 42% compared with those of the conventional system. In summary, TEK provides more opportunities to time-critical threads in terms of CPU scheduling with little overhead. In addition, TEK does not require any modifications to the conventional one-to-one thread model because it uses the API of the conventional thread model.

C. Response Time of User-Level Event

This section discusses the improvement obtained to the response time of the time-critical threads using the TEK user space thread manager. To measure the response time of the time-critical threads, the experiment was conducted under a real-life scenario that included two steps. As described in Section IV-A, modern operating systems can limit the CPU usage of the running threads by classifying them into four different scheduling groups: urgent, normal, service, and background [40]. Now, the first step in the scenario creates threads that are evenly deployed to each thread group so as to create a CPU intensive situation. Then, the second step measures the response time of each thread when it handles a user-level event, such as a touch screen input on the CE/IoT device. In other words, the response time of each time-critical thread was measured during 100% CPU utilization.

Fig. 10 shows the evaluation results of TEK compared with the conventional system. As shown in the figure, the conventional system leads to latency fluctuations; the average response time was 1719 ms when all of the time-critical threads competed for the CPU. This is because the non-time-critical threads can frequently stagnate and even hang on the time-critical threads. Alternatively, TEK shows steady performance results, dramatically reducing the average response time of the time-critical threads by up to 235 ms. These results are meaningful because there was a lot of competition for CPU resources in the thread group. The reason behind such significant improvements is that the CPU Mediator efficiently isolates and handles time-critical threads in terms of the CPU resources. Unfortunately, the CPU Mediator causes a negative impact on the performance of non-time-critical threads because they can be preempted to yield resources to the time-critical threads. In the worst case, the response time of the non-time-critical threads was stalled by up to 1487 ms. However, it is important to mention that the non-time-critical threads, such as software update threads, reserved task threads, and system management threads, do not react to user activities on-the-fly.

D. Stack Management of Threads

Generally, whenever one thread is created, the memory manager in the kernel allocates a fixed-size chunk of stack memory for the created thread. Unfortunately, if the thread uses a smaller region of memory than the allocated chunk, the unused memory space is wasted. This unused memory space may indirectly cause a segmentation fault because it leads to a shortage of free memory. On the other hand, as mentioned, TEK efficiently allocates a stack memory chunk that best fits the thread using the Stack Tuner. Fig. 11 shows the accumulated usage of stack memory allocated to the created thread. For accurate evaluation, the data in Fig. 11 were monitored during creation of a total of 273 threads over 15 days. As expected, Fig. 11–(a) clearly confirms that TEK uses a much smaller amount of memory space than the conventional system. Also, TEK allocated only 70 MB of memory, even though a total of 273 threads were running simultaneously, as shown in Fig. 11–(b).

Fig. 12 plots the number of segmentation faults that actually occurred while running the threads on the experimental CE/IoT device. As mentioned before, the conventional systems employ a fixed-sized chunk of memory to support the creation of a new thread, and therefore, the evaluation was performed by varying the size of the fixed-sized stack space from 2 MB to 8 MB. In the conventional system, the evaluation results clearly show that the number of segmentation faults increased significantly with an increase in the number of threads. In particular, after the number of accumulated threads reached 200, the conventional system became unstable and could not guarantee a stable response for the creation of a new thread because of the exception handling of the segmentation fault. On the other hand, TEK maintained good conditions over most of the experiment because the proposed system supports a fine-grained stack allocation that exactly allocates stack memory space for the amount actually used in the thread. Of course, TEK also wastes a small portion of memory because of page alignments. To understand how much memory space is wasted,

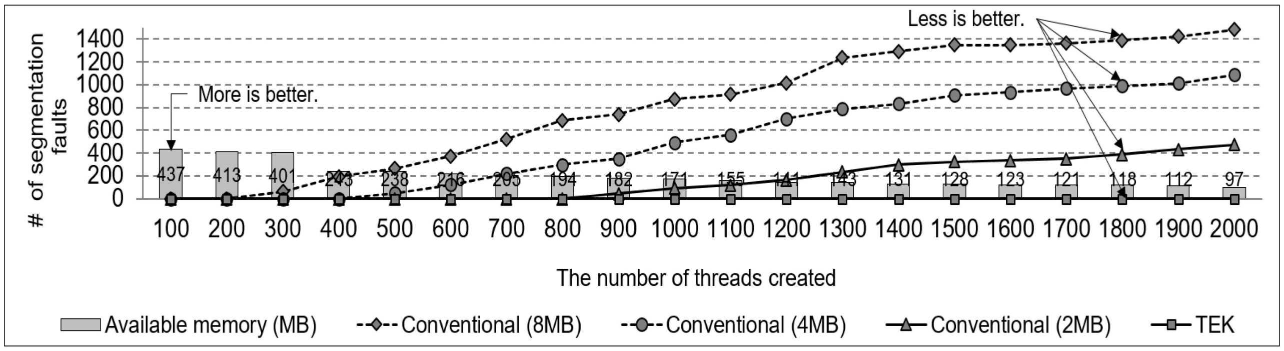

Fig. 12. The experimental results for segmentation fault frequency while creating threads.

the amount of allocated memory usage was monitored on both the TEK and the conventional system. As shown in Fig. 11, TEK only used 39% (70528 KB) of the stack memory space thanks to the Stack Tuner, while the conventional system used 4300% (2236416 KB), compared with the stack memory space actually used by the threads (50828 KB).

VI. Related Work

This section summarizes prior work to clearly understand the difference between the proposed scheme and the conventional system in terms of the thread model, thread performance, and thread management.

Thread Model and Performance: Many studies have been performed to enhance the thread model. NPTL [5] pointed out the issue of the scalability of the Linux scheduler, then proposed the O(1) Linux scheduler to address the issue both on multi-core and single-core architecture. In particular, NPTL designed a FUTEX synchronization mechanism to support the one-to-one thread model without additional overhead. Wong *et al.* [6] presented the CFS scheduler to ensure a fair allocation of CPU resources to tasks without sacrificing interactive performance. As a result, this scheduler could replace the O(1) scheduler [6] in the Linux kernel. Meanwhile, PK [12] focused on the performance of threads and proposed a concurrency model based on POSIX threads (Pthreads) to improve thread performance, including real-time threads. Engelschall [13] described a portable multi-threading mechanism that supports the expeditious creation and execution of threads during the simultaneous execution of multiple threads. In addition, to achieve backward compatibility, Engelschall developed a mechanism based on ANSI-C on the Unix system. In summary, all of the above studies focused on improving the performance of threads in high-performance computing environments equipped with large-scale hardware resources. Therefore, they are different from TEK in that TEK considers small-scale hardware environments, like CE/IoT devices.

Thread Management: Adya *et al.* [33] focuses on cooperative task management to guide the concurrency conditions of the system for program architects. A prototype of the cooperative task management method was implemented based on the event-driven approach so as to meet the requirements of thread concurrency. On the other hand, Arachne [10] addresses low-latency and high-throughput applications by designing short-lived threads. In this scheme, threads running in the user level are handled with core-aware scheduling, which assigns the cores to each thread according to the application requirements. In other words, the desired scheduling can be achieved with core-aware thread management. However, since the APIs are not POSIX compatible, it is difficult to immediately port them to modern CE/IoT devices.

VII. Conclusion

Contemporary CE devices, which have sensor and network modules, have unique characteristics compared with general desktop or server systems in that the threads of an application must be handled by limited resources, such as low clock speed CPU and small capacity memory. This allows for low power requirements, miniaturization, and cost competitiveness. This article targeted enhancing the existing one-to-one thread model to resolve the resource management problems of the user-space threads on low-end CE/IoT devices. To handle threads more efficiently in these systems, this article proposed state-of-the-art resource management facilities for CE devices: a CPU Mediator, Stack Tuner, and Enhanced Thread Identifier. This article shows that the proposed system dramatically improves the response time of time-critical threads by up to 7x and saves available virtual memory space by up to 3.4x. In addition, the proposed system supports a POSIX-compatible thread scheduling API that allows developers to run unified applications on both small-scale and large-scale hardware platforms. Also, the proposed system supports a light-weight system resource manager to improve naive stack management on the low-end CE devices.

References

[1] W. Z. Khan, M. Y. Aalsalem, and M. K. Khan, "Communal acts of IoT consumers: A potential threat to security and privacy," *IEEE Trans. Consum. Electron.*, vol. 65, no. 1, pp. 64–72, Feb. 2019, doi: 10.1109/TCE.2018.2880338.

[2] S. K. Roy, S. Misra, and N. S. Raghuwanshi, "SensPnP: Seamless integration of heterogeneous sensors with IoT devices," *IEEE Trans. Consum. Electron.*, vol. 65, no. 2, pp. 205–214, May. 2019, doi: 10.1109/TCE.2019.2903351.

[3] D. Jo and G. J. Kim, "ARIoT: Scalable augmented reality framework for interacting with Internet of Things appliances everywhere," *IEEE Trans. Consum. Electron.*, vol. 62, no. 3, pp. 334–340, Aug. 2016, doi: 10.1109/TCE.2016.7613201.

[4] S. Raj, "An efficient IoT-based platform for remote real-time cardiac activity monitoring," *IEEE Trans. Consum. Electron.*, vol. 66, no. 2, pp. 106–114, May 2020, doi: 10.1109/TCE.2020.2981511.

[5] S. J. Hill, "Native POSIX threads library (NPTL) support for uClibc," in *Proc. OLS*, Ottawa, ON, Canada, 2006, pp. 409–420.
[6] C. S. Wong, I. K. T. Tan, R. D. Kumari, J. W. Lam, and W. Fun, "Fairness and interactive performance of O(1) and CFS Linux Kernel schedulers," in *Proc. ITCC*, Kuala Lumpur, Malaysia, 2008, pp. 1–8.
[7] F. Mueller, "A library implementation of POSIX threads under UNIX," in *Proc. USENIX Conf.*, San Diego, CA, USA, 1993, pp. 29–41.
[8] H.-J. Boehm, "Threads cannot be implemented as a library," *SIGPLAN Notices*, vol. 40, no. 6, pp. 261–268, Jun. 2005, doi: 10.1145/1065010.1065042.
[9] J. Nakashima and K. Taura, "MassiveThreads: A thread library for high productivity languages," in *Proc. LNCS*, Berlin, Germany, 2014, pp. 222–238.
[10] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. Ousterhout, "Arachne: Core-aware thread management," in *Proc. USENIX OSDI*, Carlsbad, CA, USA, 2018, pp. 145–160.
[11] B. Barney. (2009). *POSIX Threads Programming*. Accessed: Mar. 30, 2020. [Online]. Available: https://computing.llnl.gov/tutorials/pthreads
[12] F. W. Miller, "PK: A POSIX threads Kernel," in *Proc. USENIX ATC*, Monterey, CA, USA, 1999, pp. 179–181.
[13] R. S. Engelschall, "Portable multithreading: The signal stack trick for user-space thread creation," in *Proc. USENIX ATC*, San Diego, CA, USA, 2000, pp. 239–250.
[14] J. Howell, B. Parno, and J. R. Douceur, "How to run POSIX apps in a minimal picoprocess," in *Proc. USENIX ATC*, San Jose, CA, USA, 2013, pp. 321–332.
[15] M. Rieker, J. Ansel, and G. Cooperman, "Transparent user-level check-pointing for the native POSIX thread library for Linux," in *Proc. PDPTA*, Las Vegas, NV, USA, 2006, pp. 492–498.
[16] K.-A. Tran *et al.*, "Clairvoyance: Look-ahead compile-time scheduling," in *Proc. CGO*, Austin, TX, USA, 2017, pp. 171–184.
[17] J. Yun, I.-Y. Ahn, N.-M. Sung, and J. Kim, "A device software plat-form for consumer electronics based on the Internet of Things," *IEEE Trans. Consum. Electron.*, vol. 61, no. 4, pp. 564–571, Nov. 2015, doi: 10.1109/TCE.2015.7389813.
[18] P. Sundaravadivel, K. Kesavan, L. Kesavan, S. P. Mohanty, and E. Kougianos, "Smart-Log: A deep-learning based automated nutrition monitoring system in the IoT," *IEEE Trans. Consum. Electron.*, vol. 64, no. 3, pp. 390–398, Aug. 2018, doi: 10.1109/TCE.2018.2867802.
[19] G. Lee and M. Rho, "IoT connectivity interface in Tizen: Smart TV scenarios," in *Proc. DUXU*, Toronto, ON, Canada, 2016, pp. 357–364.
[20] M. Ham and G. Lim, "Making configurable and unified platform, ready for broader future devices," in *Proc. ICSE-SEIP*, Montreal, QC, Canada, 2019, pp. 141–150.
[21] S. Dhotre, P. Patil, S. H. Patil, and R. Jamale, "Analysis of scheduler settings on the performance of multi-core processors," in *Proc. IEEE ICEI*, Tirunelveli, India, 2017, pp. 687–691.
[22] L. Gong, Z. Li, T. Dong, and Y. Sun, "Rethink scalable M:N threading on modern operating systems," *J. Comput.*, vol. 11, no. 3, pp. 176–189, May 2016, doi: 10.17706/jcp.11.3.176-188.
[23] B. D. Veerasamy and G. M. Nasira, "JNT—Java native thread for Win32 platform," *Int. J. Comput. Appl.*, vol. 70, no. 24, pp. 1–9, May 2013, doi: 10.5120/12212-8249.
[24] G. Blake, R. G. Dreslinski, T. Mudge, and K. Flautner, "Evolution of thread-level parallelism in desktop applications," in *Proc. ISCA*, San Jose, CA, USA, 2010, pp. 302–313.
[25] H. Franke, R. Russell, and M. Kirkwood, "Fuss, futexes and furwocks: Fast user-level locking in Linux," in *Proc. OLS*, Ottawa, ON, Canada, 2002, pp. 479–495.
[26] N. Brown, "C++CSP2: A many-to-many threading model for multicore architectures," in *Proc. CPA*, Surrey, U.K., 2007, pp. 183–205.
[27] J. J. Harrow, "Runtime checking of multithreaded applications with visual threads," in *Proc. SPIN*, Beijing, China, 2000, pp. 331–342.
[28] K. Pouget, M. Pérache, P. Carribault, and H. Jourdren, "User level DB: A debugging API for user-level thread libraries," in *Proc. IEEE IPDPSW*, Atlanta, GA, USA, 2010, pp. 1–7.
[29] M. Leske, A. Chis, and O. Nierstrasz, "Improving live debugging of concurrent threads through thread histories," *Sci. Comput. Program.*, vol. 161, pp. 122–148, Sep. 2018, doi: 10.1016/j.scico.2017.10.005.
[30] A. Gidenstam and M. Papatriantafilou, "LFTHREADS: A lock-free thread library," *SIGARCH Comput. Archit. News*, vol. 36, no. 5, pp. 88–92, Jun. 2009, doi: 10.1145/1556444.1556456.
[31] U. Drepper. (2005). *Futexes Are Tricky*. Accessed: Mar. 30, 2020. [Online]. Available: https://www.akkadia.org/drepper/futex.pdf
[32] P. Holman and J. H. Anderson, "Locking under Pfair scheduling," *ACM Trans. Comput. Syst.*, vol. 24, no. 2, pp. 140–174, May 2006, doi: 10.1145/1132026.1132028.
[33] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur, "Cooperative task management without manual stack management," in *Proc. USENIX ATC*, Monterey, CA, USA, 2002, pp. 289–302.
[34] L. Soares and M. Stumm, "FlexSC: Flexible system call scheduling with exception-less system calls," in *Proc. USENIX OSDI*, Vancouver, BC, Canada, 2010, pp. 33–46.
[35] B. Wang, B. Wang, and Q. Xiong, "The comparison of communication methods between user and Kernel space in embedded Linux," in *Proc. IEEE ICCP*, LiJiang, China, 2010, pp. 234–237.
[36] C. S. Wong, I. Tan, R. D. Kumari, and F. Wey, "Towards achieving fairness in the Linux scheduler," *ACM SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 34–43, Jul. 2008, doi: 10.1145/1400097.1400102.
[37] M. Kim, S. Noh, J. Hyeon, and S. Hong, "Fair-share scheduling in single-ISA asymmetric multicore architecture via scaled virtual run-time and load redistribution," *J. Parallel Distrib. Comput.*, vol. 111, pp. 174–186, Jan. 2018, doi: 10.1016/j.jpdc.2017.08.012.
[38] C. Davis, J. Jackson, J. Oldfield, T. Johnson, and M. Hale, "A time comparison between AVL trees and red black trees," in *Proc. FCS*, Las Vegas, NV, USA, 2019, pp. 49–54.
[39] R. McGrath. (2019). *The GNU C Library (glibc)*. Accessed: Mar. 30, 2020. [Online]. Available: https://www.gnu.org/software/libc/
[40] P. Bellasi, G. Massari, and W. Fornaciari, "Effective runtime resource management using Linux control groups with the BarbequeRTRM framework," *ACM Trans. Embed. Comput. Syst.*, vol. 14, no. 2, pp. 1–17, Mar. 2015, doi: 10.1145/2658990.

Geunsik Lim (Graduate Student Member, IEEE) received the B.S. degree in computer science and engineering from Ajou University, South Korea, in 2003, the M.S. degree from the College of Information and Communication Engineering, Sungkyunkwan University, South Korea, in 2014, where he is currently pursuing the Ph.D. degree with the Department of Electrical and Computer Engineering. He is also a Principal Software Engineer with Samsung Electronics, South Korea. His current research interests include system optimization, operating systems, software platforms, and on-device artificial intelligence.

Donghyun Kang received the Ph.D. degree from the College of Information and Communication Engineering, Sungkyunkwan University, South Korea, in 2018. He is an Assistant Professor with the Department of Computer Engineering, Changwon National University, South Korea. He was an Assistant Professor with Dongguk University, Gyeongju, from 2019 to 2020 and a Software Engineer with Samsung Electronics, South Korea, from 2018 to 2019. His research interests include file and storage systems, operating systems, and emerging storage technologies.

Young Ik Eom received the B.S., M.S., and Ph.D. degrees in computer science and statistics from Seoul National University, South Korea, in 1983, 1985, and 1991, respectively. Since 1993, he has been a Professor with Sungkyunkwan University, South Korea. From 2000 to 2001, he was a Visiting Scholar with the Department of Information and Computer Science, University of California at Irvine, USA. He was also the President of the Korean Institute of Information Scientists and Engineers in 2018. His research interests include virtualization, operating systems, file and storage systems, cloud systems, and UI/UX system.