IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 14, 2020, accepted January 25, 2020, date of publication February 3, 2020, date of current version February 18, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2970998

Catnap: A Backoff Scheme for Kernel Spinlocks in Many-Core Systems

YOUNGJOO WOO[1,2], SUNGHUN KIM[3], CHEOLGI KIM[4], AND EUISEONG SEO[1]

[1]Department of Computer Science and Engineering, Sungkyunkwan University, Suwon 16419, South Korea
[2]Artificial Intelligence Research Laboratory, Electrical and Telecommunications Research Institute, Daejeon 34129, South Korea
[3]Samsung Research, Samsung Electronics Company, Ltd., Suwon 16677, South Korea
[4]Department of Software and Computer Engineering, Korea Aerospace University, Goyang 412791, South Korea

Corresponding author: Euiseong Seo (euiseong@skku.edu)

This work was supported in part by the National Research Foundation of Korea through the PF Class Heterogeneous High Performance Computer Development under Grant NRF-2016M3C4A7952587, and in part by the Institute of Information and Communications Technology Planning and Evaluation funded by the Ministry of Science and ICT (MSIT), Korean Government, (Research on High Performance and Scalable Manycore Operating System) under Grant 2014-3-00035.

ABSTRACT As the number of cores equipped in a system grows, the impact of the spinlock waiting inside the operating system (OS) kernel on the performance and energy efficiency of a system worsens. In particular, it deteriorates the effectiveness of simultaneous multithreading (SMT). Because spinlocks are indispensable in OS kernels, it is necessary to suppress the spin wait overhead in the many-core systems. To address this issue, we propose the catnap spinlock that exploits the ACPI-C state, which is named as the catnap state and is induced by the `MONITOR`/`MWAIT` instruction pair. The catnap state releases the processor resources while deceiving the kernel that the thread is iterating a busy-waiting loop. Because entering and exiting from the C-state require considerable time, we applied the catnap loop only to the non-head waiters not to delay the lock handover operation. Furthermore, we selectively applied the catnap spinlock to the lock instances for sufficiently long critical sections based on the observation made in profiling runs. The proposed scheme was implemented in the Linux kernel and evaluated in a many-core processor system with a few workloads from the *PARSEC* and *Re-aim* benchmark suites. Our evaluation showed that the proposed scheme improved the performance by up to 33.59% and reduced energy consumption by 39.11%.

INDEX TERMS Spinlocks, symmetric multithreading, energy efficiency, synchronization, many-core.

I. INTRODUCTION

Spinlocks have been widely used for decades because of their fast handover and lightweight design, although the lock waiters consume significant processor resources while they are waiting. In particular, in operating system (OS) kernels, the sleep locks, such as semaphores or mutexes, cannot be used in the scheduling or interrupt handling routines because the design of such locks depends on the scheduler and interrupt handlers. Consequently, the use of spinlocks is frequent and unavoidable in the kernel code.

As the number of concurrent threads in the system increases, the synchronization cost for accessing shared resources in the OS kernel escalates, and this leads to performance degradation and limited scalability [1]. As a result, there have been many research efforts to address the lock contention overhead inside the kernel since the inception of the many-core era [2]–[4].

The inflated lock contention caused by increased parallelism is also applied to spinlocks. Because a kernel spinlock waiter disables preemption while it is waiting, the increased contention significantly worsens the system-wide average lock waiting time. In our experiments, we verified that up to 25% of the overall processor cycles could be wasted for spinlock waiting while executing an I/O-intensive workload with 252 parallel threads. The wasted processor cycles severely compromised both performance and energy efficiency.

The number of transistors that are integrated into a chip or in a core is continuously increasing owing to advancements in processor manufacturing. As a means to utilize these surplus transistors, most modern processors for servers or high-end PCs are equipped with a simultaneous multithreading (SMT) feature, which allows multiple logical threads to share the resources of a single physical core.

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

The rationale behind SMT is that the core resources yielded by the idle cores can be temporarily utilized by the active cores. Therefore, if a logical thread continuously consumes such resources for busy waiting, the amount of shared resources that are available to the other logical threads must be reduced, and this results in the diminished effectiveness of SMT. As a result, the prolonged spinlock waiting time adversely affects not only the performance of the waiting thread but also that of the sibling threads.

To address the increased spinning overhead in the many-core systems, this paper proposes *catnap* spinlock for the OS kernel. The waiters of the catnap spinlocks wait in the catnap state, which is a temporary core sleep state, induced by the `MONITOR/MWAIT` instruction pair. The catnap state is considered as an actively working state of the kernel while it is, in fact, an idle state at the hardware-level. Therefore, the catnap spinlock can be applied in any context where a conventional spinlock is being used to improve performance and energy efficiency.

Spinlocks are targeted for the short critical sections and are thus expected to have quick handover delays. The `MONITOR/MWAIT` instruction pair enables spinlock waiting without resource waste, but the idling core requires notable time to wake up from the low power state. The waiter in the catnap state may not immediately acquire the lock when its predecessor releases it because of the exit latency from the idle state. Therefore, applying the catnap state may degrade the execution time.

However, the delay can be hidden when the wake-up notification to the waiter is delivered at a proper time point in advance, and the critical section is sufficiently long to allow for this. To minimize the handover delay caused by the catnap state, we apply the catnap waiting loop only to the non-head waiters, which are expected not to acquire the lock right next to the current holder. In addition, we analyzed the critical section lengths of kernel spinlock instances while executing benchmark suites. Based on the profile results, we selectively applied the catnap spinlock to the kernel instances that have critical sections longer than the harmless catnap threshold, which is the minimum length of the critical section that can hide the wake-up delay in the proposed design.

The catnap spinlock is implemented in the Linux kernel based on *qspinlock*, which is the current Linux spinlock implementation. We evaluated the proposed scheme in terms of performance and energy efficiency using various workloads from the *PARSEC* and *Re-aim* benchmark suites running on an Intel Xeon Phi many-core server.

The remainder of this paper is organized as follows. In Section II, we introduce the history and mechanisms of the in-kernel spinlock and its performance impact in highly paralleled multicore systems. In Section III, we propose the design and implementation of the catnap spinlock, and in Section IV, we present its evaluation with multiple benchmarks. Section V introduces the related work on reducing overhead and securing scalability for locks in many-core systems. Finally, Section VI concludes this paper.

II. BACKGROUND AND MOTIVATION

A. ANATOMY OF MODERN SPINLOCKS

A spinlock is one of the representative synchronization primitives, which is a non-sleep lock that does not require scheduling intervention. Therefore, it has been popularly used on several occasions because of its simplicity and straightforward semantics. However, threads in the spinlock discipline waiting for lock acquisition have to iterate a busy-waiting loop continually. This spinning loop consumes a significant amount of processor cycles, and therefore, energy as well. However, because both of its lock acquisition and release operations are simple and fast, it provides efficient synchronization for short critical sections under low contention [5], [6].

For these reasons, the Linux kernel takes the spinlock as a primary synchronization mechanism. The Linux kernel uses spinlocks for serializing non-preemptible contexts, including interrupt handlers and task schedulers, and for protecting short critical sections. Furthermore, several high-level synchronization mechanisms of the Linux kernel, including *read-copy-update (RCU)*, *mutex*, *semaphore*, and *reader-writer lock*, are implemented using the spinlock as a basic building block.

Spinlocks in the early versions of the Linux kernel used the simple *test-and-set (TAS)* spinlock [5]. The waiters of the TAS spinlock compete with each other while spinning on a globally shared lock variable. When the lock is released, the ownership of the lock goes to the waiter that executes the TAS instruction first. Consequently, some of the waiters may starve regardless of when they have begun waiting. To address this unfairness, the Linux kernel adopted the ticket spinlock [7], which supports the first-in-first-out (FIFO) ordering, which has been used as a replacement of the TAS spinlock since the release of version 2.6.25.

In conventional spinlocks, including the TAS and ticket spinlocks, every waiter repeatedly accesses one global lock variable. This repetitive lock variable access may generate a considerable amount of remote cache accesses in a multi-core system that has separate caches for each processing unit to provide cache coherence. In turn, the remote cache accesses increase the degree of contention in the interconnect networks and memory buses. Consequently, the frequent use of spinlock may damage the performance not only of the lock-related cores but also of the overall system [5]. Because the number of cores integrated into a system is rapidly growing, the limited scalability of the spinlock is becoming a critical issue [2], [3].

A few queue-based spinlock structures were proposed to divide the problematic global spinning into per-waiter local spinning. By doing so, the waiters do not produce any remote cache accesses while waiting. The two representative examples of this queue-based spinlock design are the *MCS* [8] and *CLH* locks [9], [10].

The MCS lock uses a linked list for the wait queue management. Each lock waiter has a dedicated node in the wait queue. When the lock is released, the holder hands over the

lock ownership by writing a non-zero value into the head node of the queue to pass the lock ownership, and then, the waiter watching the head node enters its critical section. The CLH lock is similar to the MCS lock, but the waiter of the CLH lock spins on its predecessor node. Therefore, to release the lock, the lock holder simply changes its node to false. Because of this, the release of the CLH lock is faster than that of the MCS lock. However, the CLH waiter must prepare a waiting node for the next waiter before entering the wait loop. Therefore, queueing the waiter and transferring the lock ownership with the CLH lock is more complicated than with the MCS lock. Because the waiters spin on their private lock variables, both MCS and CLH lock structures generate a limited amount of remote cache accesses regardless of the number of waiters or length of the waiting time. The scalability earned from suppressing the remote cache accesses makes them appropriate for the many-core systems.

In response to the increasing number of cores, besides replacing the classic ticket spinlock, the Linux kernel also adopted the qspinlock [11], which is an enhanced version of the MCS lock, since its 3.15 release. There are three main differences between the design of the qspinlock and the original MCS lock. First, the qspinlock has a pending state to expedite the lock handover. Second, a qspinlock holder can quickly release the lock by changing the global lock variable value that reflects the lock state. Lastly, the qspinlock manages a pre-allocated queue node pool for each logical CPU, so it does not need memory allocation for a queue node when a new waiter arrives. The details of the qspinlock are as follows.

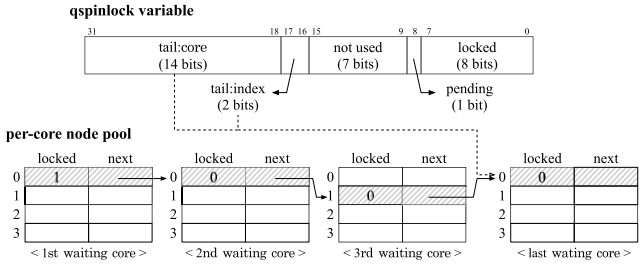


FIGURE 1. Data structures for a qspinlock; global lock variable and per-core wait queue.

The qspinlock state is represented by a 32-bit variable, as depicted in Fig. 1. This lock variable consists of three fields: the tail of the wait queue, the pending thread existence bit, and the status of the lock instance. A thread first performs the `CMPXCHG` atomic instruction over this variable to acquire the lock instance. If the lock variable value is zero, it means the lock is in a free state, and the thread immediately acquires it. This procedure is the fast path of the qspinlock acquisition. If the thread cannot acquire the lock through the fast path, it will invoke the slow path to wait for the lock to be released.

If there are currently no waiters in the queue, the waiter that arrives first starts waiting in the pending state. Because the MCS lock has to manipulate the wait queue node, it is slower than the ticket spinlock when the lock is not contended. In contrast, the performance of the qspinlock is comparable to the ticket spinlock due to the pending state even while the degree of lock contention is low. The pending waiter of qspinlock does not manipulate a wait queue node but waits with spinning on the global lock variable instead after setting the pending bit. Therefore, the pending waiter can acquire the lock immediately when the lock is released because there are no queue clean-up operations required.

If more than one thread starts waiting for the lock, a newly coming waiter will create a wait queue or adds a node to the existing queue. The waiter, by using the `CMPXCHG` instruction, atomically updates the tail field with the address of the added node and saves the previous node address formerly stored in the tail field. The tail field consists of a core id and an index of the node in the per-core wait node pool. As the Linux kernel does not allow context switches during spinlock waiting, the number of cores determines the maximum number of waiters. Therefore, the qspinlock can allocate memory space statically for per-core node pools. The node location can be retrieved by the core index and the index in the per-core pool, and thus, the qspinlock uses only 16 bits to keep track of the last node position. The per-core node pool has four nodes, considering the possibility of nesting up to four levels of spinlocks: task, `softirq`, `hardirq`, and non-maskable interrupt (NMI).

All waiters in the queue, except the head waiter, spin on their own queue nodes. The head waiter has a queue node but spins on the global lock variable like the pending waiter. When a holder releases a lock, the head waiter can grasp the change of in the lock value and immediately acquire it. After that, the head waiter notifies the second waiter to become the next head waiter. In summary, the qspinlock has three kinds of spin-loops for the pending, head, and other waiting threads, respectively.

In the qspinlock scheme, only up to two waiters can spin on the global lock variable at a particular time point, the pending thread, if it exists, and the head waiter. The pending waiter temporarily exists only when there are no waiters in the queue. Other than the spin-loop, the lock variable is accessed when the lock holder releases it, or a new waiter arrives. Therefore, the lock variable in this design will rarely experience the remote cache accesses, and thus, the qspinlock design significantly improves scalability.

B. PERFORMANCE EFFECT DUE TO SPIN WAITING

The performance degradation due to spinlock contention is caused by both handover delay, which is the time interval from the end of a critical section to the beginning of the next critical section, and resource utilization by spinning waiters. As previously stated, the qspinlock resolved the increased handover delay in many-core systems by reducing the remote cache accesses. However, the resource waste from spin waiting, which is an inherent problem of spinlocks [12], remains the same in the qspinlock design.

As the number of cores is continuously growing in the multi-core era, the resource waste from spinlock contention is

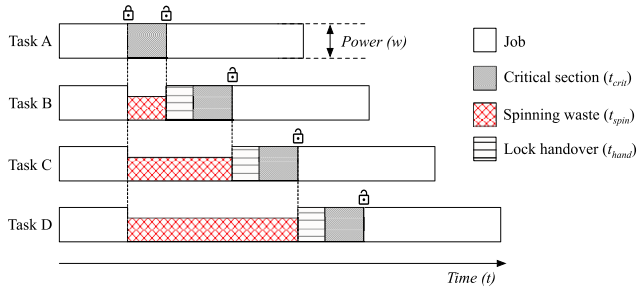


FIGURE 2. Quadratic increase in the busy waiting due to spinlock contention in a multicore system.

rapidly increasing. For example, as shown in Fig. 2, suppose that four threads request a lock instance at the same time and each thread holds the lock for time t_{crit} after it acquires the lock. When thread A acquires the lock, thread B needs to wait in the spin-loop for $t_{spin}(B) = t_{crit}$ where t_{crit} is the length of the critical section. However, threads C and D have to wait for up to $t_{spin}(C) = 2\ t_{crit} + t_{hand}$ and $t_{spin}(D) = 3\ t_{crit} + 2\ t_{hand}$, respectively, where t_{hand} is the hand over delay of the spinlock. Therefore, the total wait time with spinning will be $t_{spin}(B) + t_{spin}(C) + t_{spin}(D) = 6\ t_{crit} + 3\ t_{hand}$. If N threads are attempting to acquire the lock instance at the same time, the expected total waiting time will be $\sum t_{spin} = \frac{N(N-1)}{2} t_{crit} + \frac{(N-1)(N-2)}{2} t_{hand}$ in the worst case.

As mentioned above, OS kernels frequently use spinlocks. Therefore, in many-core systems, spinlock waiting in the kernel context, such as memory management, file system, and I/O operations, may significantly contribute to the processor utilization. To quantitatively analyze processor utilization caused by spinlocks in the kernel, we performed an experiment using the *Re-aim* benchmark suites and *vips* from the *PARSEC* benchmark executed in a many-core system providing 256 hardware threads of which configurations are described in Table 4.

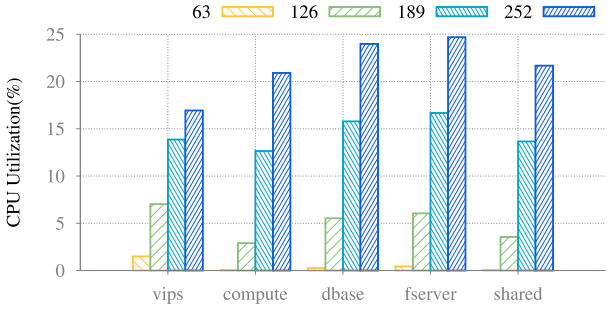


FIGURE 3. CPU time share of the wait function of the *qspinlock*.

Fig. 3 shows the ratio of processor cycles consumed by spinlock waiting to the all the execution cycles, including the idle and stall cycles, while each benchmark runs with a varying number of threads. The wasted processing time for the *shared* workload worsened by 56 times when using 256 threads in comparison to the case using 64 threads. This tendency could be observed from the other workloads used in the experiment, as well. This shows that the processor resource waste by spinlock waiting is critically determined by the degree of parallelism.

As stated earlier, the processor cycles consumed in spin waiting may negatively affect the efficiency of the SMT feature, which is pervasive in modern server processors [13]. SMT allows multiple logical cores to share core resources in one physical core and run multiple threads simultaneously to improve the utilization of core resources. By doing so, SMT can improve not only performance but also energy efficiency by up to 30 % [14], [15].

In the SMT architecture, each logical core has a complete set of independent architectural states, such as general-purpose registers, control registers, advanced programmable interrupt controller (APIC) registers, and machine status registers. However, they share the remaining resources, such as cache memory, execution units, branch predictors, control logic, and buses [14]. For instance, Intel Xeon Phi Knights Landing (KNL) processors simultaneously execute four logical threads with one physical core through their SMT technology named *Hyper-threading*. A physical core can decode up to two instructions at a single cycle. Moreover, the allocation unit, which manages the pipeline for executing μ-operations in an out-of-order manner, and the memory execution unit can execute up to two instructions in a single cycle. Therefore, if two logical cores are inactive, the instructions of sibling logical cores can be executed every cycle [16].

Therefore, the efficiency of SMT is determined by the behaviors of logical cores sharing a physical core [17]. If one of the logical cores continually dominates resources in a physical core to acquire a spinlock, the performance of sibling logical cores will significantly deteriorate [18], [19].

The energy efficiency of data centers and server systems is becoming increasingly critical these days [20]. It is evident that the wasted processor cycles by spinlocks will lead to reduced energy efficiency.

To summarize, the growing number of cores has resulted in quadratic increase in the waiting time of kernel spinlocks. This, in turn, degrades the effectiveness of the SMT feature and worsens the energy efficiency of the overall system.

Therefore, the spinlock waiting loop should minimize its resource usage so that the yielded resources can be used for other productive jobs. For this, the spin-loops usually use a backoff mechanism, which slows down the spinning speed after unsuccessful probes of the lock variable. Several backoff mechanisms have been proposed to relax the adversary effects of the spin-loops to the shared resources and power consumption [5], [8], [21].

Backoffs are normally implemented by adding delay instructions, which insert some stall cycles in the spin-loop. For example, the x86 architecture provides the `PAUSE` instruction for hinting execution of a spin-loop to the processor [22], [23]. Furthermore, `PAUSE` is strongly recommended to avoid the branch misprediction that occurs when the waiter is exiting from a tight spin-loop [24]. `PAUSE` has an identical opcode (F390) to `REP;NOP`, repetition of `NOP`, in the old

microarchitectures before Intel Pentium 4. The behavior of the PAUSE instruction is the same as the repetition of the NOP instructions. However, the difference is that the processor automatically adds a decent number of stall cycles to maintain the memory order.

Currently, the qspinlock in the Linux kernel uses the PAUSE instruction for backing off its spin-loop. However, the spinlock waiters still read the lock variable at a rapid rate, and this causes the side effects mentioned above. Therefore, a new backoff mechanism is necessary to mitigate the performance and energy loss from spinlock contention in many-core systems.

III. OUR APPROACH

A. CATNAP SPINLOCK

Relaxing spinning is effective only when the lock acquisition is highly congested. However, if the backoff mechanism delays the lock handover operation, the overall performance will degrade regardless of the degree of lock contention because lock handover would occur in the critical paths.

Spinning in a core consumes only a small fraction of the dynamic power. However, all active cores in a system continuously consume static power, therefore contributing a significant to overall power consumption unless they are not in a deep idle state. Therefore, although a backoff mechanism may reduce the power consumption of a waiting core, the overall energy consumption will increase if the total execution time is extended owing to the prolonged handover operation.

Consequently, it is more important not to degrade the performance than to mitigate the spinning overhead. In other words, a desirable backoff mechanism should slow down or relax only spinning without delaying the lock handover operation.

The most straightforward way for backing off a spin-loop is executing a few NOP instructions in every iteration. Slowing down the spin wait with multiple NOP instructions alleviates the spinning overhead. It reduces the frequency of memory references and branch instructions, and thus, it can lessen the shared resource contention when the spinlock contention is severe. Moreover, the repetition of NOP instructions can be applied to most of the processors without specialized hardware support.

However, the delayed inspection of the lock state due to the prolonged spin-loop exit can increase the lock handover latency in the queue-based spinlock design. Moreover, it is challenging to find an appropriate number of repetitions of the NOP, as it varies depending on the processor architecture and the degree of the lock contention. Determining the number of NOP instructions per a spin-loop iteration on the fly is also difficult because it requires access to global variables for status monitoring, and additional branch instructions in the spinlock code, and thus, it would significantly deteriorate the performance and scalability of the spinlock.

Other instructions that suspend the processor operation, such as HLT or MWAIT of the x86-64 instruction set, can be used for backing off the spin-loops. The HLT instruction suspends the execution and puts the core in an idle state. These instructions are being used to implement the energy-efficient idle loop by a few OSs, and other ISAs also provide similar instructions.

Both instructions, as mentioned earlier, enforce the core to spend time in the low-power core idle state. However, the HLT instruction is not appropriate for use as a backoff instruction because a core suspended by the HLT instruction can be reactivated only through receiving an interrupt. If the lock waiter suspends its core in the spin wait loop using the HLT instruction, and then the lock holder sends the inter-processor interrupt (IPI) to the first waiter to wake the core up, the number of iterations in the spin-loop can be remarkably reduced. However, the performance of the spinlock will be significantly degraded because sending IPI causes a lock handover delay for a few thousands of clock cycles for the initiation, delivery, and reception of the interrupt.

MWAIT instruction also suspends the core activity. However, the wake-up mechanism of MWAIT works differently. A logical core should execute a MONITOR instruction before it uses MWAIT. MONITOR specifies a range of address space to be monitored by the processor for the forthcoming store instructions over that range. When a logical core executes a MWAIT instruction, the logical core stops operation and enters into an idle state. After that, when another logical core issues a store instruction onto the monitored address range, the suspended logical core is woken up and resumes its execution [25]. By doing so, the MONITOR/MWAIT pair allows resuming the execution without an additional memory read or branch operation. Therefore, we concluded that the MONITOR/MWAIT pair have the traits required for the backoff instruction of the kernel spinlock.

When the spinlock suspends the core with a backoff instruction, the waiter must not sleep from the perspective of the task scheduler. However, the activity and power consumption of the core should be suppressed to the level of the idle state. Therefore, a spinlock with a backoff mechanism is different from the sleep locks or the blocking locks that the waiter releases the core to the scheduler to enter into the sleep state. Therefore, we define the backoff-induced temporary sleep state as the *catnap* state to distinguish it from the sleeping wait state of the sleep locks. In addition, we propose the catnap spinlock, of which waiters wait in the catnap state.

The catnap spinlock is based on the qspinlock design. However, in certain conditions, waiters will enter into the catnap state using the MONITOR and MWAIT pair. The logical core that is executing a waiter in the catnap state does not execute any instructions and remains idle in an advanced configuration and power interface (ACPI) C-state, if possible, while the OS kernel regards it as actively running. Therefore, the catnap spinlock can simply replace the existing spinlocks, and it can mitigate the resource waste problem of spin waiting.

The pseudo-code of the catnap waiting loop is shown in Fig. 4.

Procedure 1 spinlock_contended_wait

```
1:  while True do
2:    monitor_target ← address_of (locked)
3:    MONITOR (monitor_target)
4:    if Locked then
5:      break
6:    else
7:      hint ← designated_C–state
8:      wakeup_mode ← interrupt_wakeup_disable
9:      MWAIT(hint, wake_up_mode)
10:   end if
11:   if Locked then
12:     break
13:   end if
14: end while
```

FIGURE 4. Spinlock_contended_wait.

When the waiting loop begins, at lines 2 to 3, the logical core sets the per-core lock variable of qspinlock, which is illustrated in Fig. 1, as the monitoring target with the `MONITOR` instruction.

After that, the waiter checks the status of the per-core variable at line 4. If the lock field in the variable is set, then the waiter will exit from the wait loop because it means that the waiter just became the head waiter, which is the first waiter to acquire the lock. Because the head waiter is expected to acquire the lock soon and entering into and exiting from the sleep state take considerable time, the head waiter of the catnap spinlock iterates the conventional spinning wait loop for the performance. The details of this issue will be covered in Section III-B.

`MWAIT` requires two operands; the desired C-state level and the wake-up events. The catnap loop sets both operands before executing the `MWAIT` instruction.

The deeper the C-states a core stays in, the less power it will consume. However, the exit latency from the idle state will increase as the level of the C-state deepens. The prolonged exit latency can delay lock acquisition. Moreover, because of the power consumption during the state transition, as the level of the C-state deepens, the target residency, which is the shortest time interval needed to achieve energy savings from staying in a C-state, will increase [26]. Therefore, the desired C-state level hint, which will be given to the `MWAIT` instruction, should be carefully chosen so that it does not affect the lock handover delay.

Table 1 presents the exit latency and the target residency of each C-state of the Intel KNL and Skylake processors, respectively. These are defined in the `intel_idle` driver of the Linux kernel. We could not find any publicly available numbers for the state transition latency, or entry latency, to the deeper C-states.

TABLE 1. Exit latency and target residency of Core C-states.

Processor	Core C-state	Exit latency (μs.)	Target residency (μs.)
Knights Landing	C1	1	2
	C6	120	500
Skylake	C1	2	2
	C1E	10	20
	C6	133	600

The SMT feature prevents a physical core from entering into the desired C-state of its logical core because other logical cores that share the physical core may actively execute instructions. The designated depth can be reached only when the degree of the lock contention is high or when the system load is low. Therefore, the C-state hint can be considered to be the maximum allowable depth that does not prolong the handover delay.

The target residency of the C6 state of the KNL processor is 500 μs. However, according to our observations, which are presented in detail in Section III-B, only two spinlock instances showed an average waiting time longer than 500 μs during the execution of benchmark applications. In addition, the exit latency from the C6 state, which affects the handover delay, is 120 μs while the average holding times of lock instances are around 1 μs. Considering these circumstances, the catnap waiting loop uses the C1 state as the designated idle level, and delivers the hint to the `MWAIT` instruction at line 7.

Because the `MONITOR` instruction monitors write access to a target address in the cache line granularity, writing to the area around the lock variable can wake up the catnap state. In addition, NMIs may disturb the catnap as well. Therefore, even if the monitor was set for the per-node lock variable, after waking up, the waiter must check whether the cause of awakening was a change to the lock variable or not, as illustrated at line 11.

As described in Section II-A, the current qspinlock has three different waiting states: pending, head, and queue. The catnap is applied only to the queue waiters, who arrived after the pending and head waiters, among the three waiting states. The pending and head waiters, which spin over the global lock variable, should quickly react, or acquire the lock when the global lock variable changes to the released state because the response time to the lock variable change directly affects the handover delay. In addition, if the pending and head waiters wait in the catnap state, they will frequently and repetitively enter into and exit from the catnap state under highly contended conditions because the waiting queue tail field of the global variable changes every time a new waiter is inserted into the waiting queue. These frequent state changes drive the core to stay in the idle state for a period shorter than its target residency. Therefore, for the sake of performance and energy efficiency, the catnap spinlock lets the leading two spinners wait in the conventional `PAUSE` spin-loop.

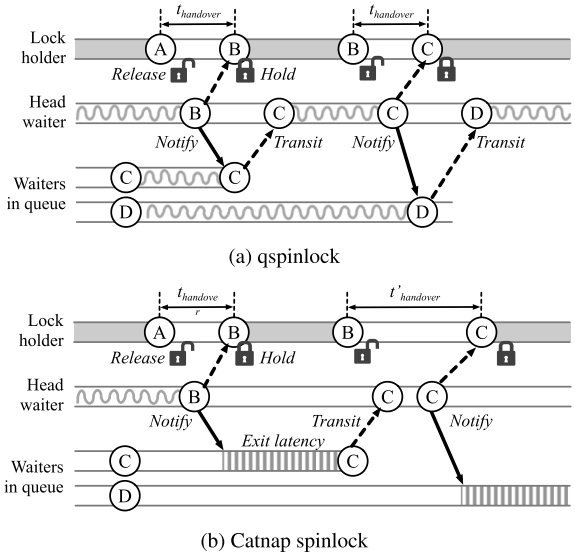


FIGURE 5. Lock hand-over timing depending on spinning scheme when the critical section is short.

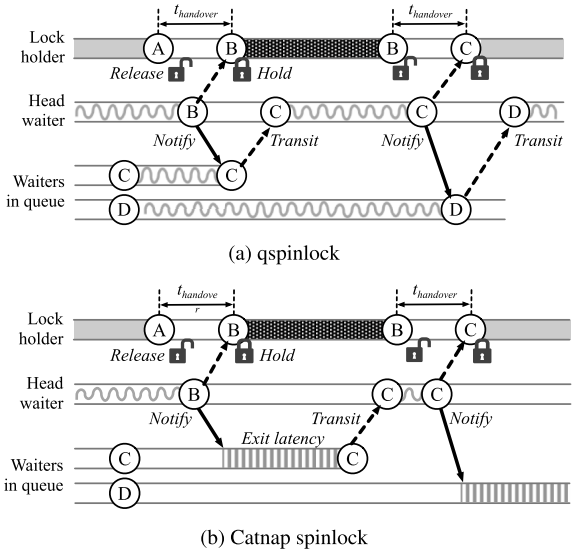


FIGURE 6. Lock hand-over timing depending on spinning scheme when the critical section is long.

The proposed *catnap* spinlock enables a logical core running a spinning thread to yield physical core resources while waiting. Therefore, it is expected to improve energy efficiency and the effectiveness of the SMT. However, it may adversely affect the handover delay when the critical section lengths are not sufficiently long to offset the transition overhead.

B. EXIT LATENCY OPTIMIZATION

As previously mentioned, the catnap spinlock imposes the catnap state only upon the waiters that have the head node in the waiting queue to prevent possible handover delay. However, when the critical section of a spinlock instance is shorter than the state transition latency, the handover delay can still occur in the proposed design, and the use of the catnap, in such cases, will negatively affect the performance.

The exit from the catnap state of the second waiter begins when the head waiter modifies the per-core queue node of the second waiter, which is being monitored. The wake-up delay allowable for the second waiter is determined by the critical section length of the head waiter, as shown in Fig. 5 and Fig. 6.

Fig. 5 illustrates the case where the catnap spinlock prolongs the handover operation because the critical section length is not sufficiently long to offset the transition delay. Let us assume that four threads, *A*, *B*, *C*, and *D*, are sharing a spinlock instance, and *A* initially holds the lock. When *A* releases the lock, *B*, which is iterating the `PAUSE` spin-loop, notifies *C* of its promotion to the head node and acquires the lock. The time interval between *A*'s release and *B*'s acquisition is $t_{handover}$ for both the qspinlock and catnap spinlock schemes. However, when *B* releases the lock shortly after its acquisition, in the catnap spinlock scheme, the time interval between *B*'s release and *C*'s acquisition is extended to $t'_{handover}$ because *C* is still in the middle of the state transition when *B* releases the lock.

On the contrary, if the critical section is sufficiently long, as shown in Fig. 6, *C* will become ready before *B* finishes its critical section. In such cases, the handover delay of a catnap spinlock between *B* and *C* will be the same as that of the qspinlock. To summarize, to completely offset the exit latency of the second waiter, which waits in the catnap state, the critical section of the current lock holder has to be longer than the state transition time.

To determine the critical section length that does not prolong the handover delay, we measured the lock performance using the *locktorture* [27], [28] test kernel module by varying the length of the critical section and the number of concurrent threads. The configuration of the many-core server used for this experiment was described in Table 4. The number of cycles for each thread needed to obtain the lock instance was measured to compare the performance of the qspinlock and catnap spinlock.

Each pixel in Fig. 7 shows the relative lock acquisitions per unit time of the catnap spinlock normalized to the conventional qspinlock for each combination of the number of threads and the critical section length. The exit latency from the C1 state of the processor used in the experiment is approximately 1 μs.

The catnap spinlock being contended by four threads performed slower than the qspinlock even when the critical section was longer than the exit latency. This is because, as shown in Fig. 8, the waiter, *C*, received a notification right after it started the state transition to the catnap state. A state transition operation cannot be aborted. Therefore, thread *C* had to exit after it reached the catnap state immediately. When only two threads are waiting in the queue, this situation occurs every time as long as the critical section length does not exceed the sum of the entry latency and the exit latency.

The results in Fig. 7 show that the catnap spinlock retards the lock handover time when the critical section length was

TABLE 2. Top 20 spinlock instances having the longest average waiting time.

	Lock instance	No. of acquisitions / total no. of acquisitions (%)	Acquisitions longer than exit latency (%)	Average holding time (μs)	No. of contentions / no. of acquisitions (%)	Average waiting time (μs)
1	pgdat→lru_lock	4.0	99.1	14.3	93.8	2339.9
2	ptlock_ptr(page)	2.4	67.8	28.8	27.0	1633.9
3	(mapping→i_pages)→xa_lock	16.0	92.5	1.8	4.5	280.5
4	random_read_wait.lock	0.0	100.0	30.2	0.6	103.5
5	sighand→siglock	1.4	78.8	3.1	0.0	91.6
6	(__futex_data.queues)[i].lock	0.1	81.4	2.7	31.0	87.6
7	p→pi_lock	2.3	66.2	8.8	0.1	73.4
8	mount_lock	0.0	73.7	7.0	5.3	57.2
9	vmap_area_lock	0.1	68.1	3.9	1.9	51.6
10	iclog→ic_force_wait	0.0	94.9	10.8	7.1	39.0
11	dd→lock	0.1	96.2	3.6	30.2	33.7
12	journal→j_wait_done_commit	0.0	86.0	6.0	28.0	30.3
13	fq→mq_flush_lock	0.0	99.9	48.0	1.3	30.1
14	semaphore→lock	0.1	34.0	1.1	0.0	22.2
15	rq→lock	18.5	91.0	10.9	1.6	15.9
16	slock-AF_INET	0.0	32.9	2.6	0.0	15.0
17	host→lock	0.0	100.0	11.1	12.7	14.4
18	s→s_inode_list_lock	0.1	76.3	4.6	1.0	11.2
19	pool→lock	0.0	66.5	1.3	0.0	11.1
20	page_wait_table[i]	0.7	86.0	12.4	14.6	11.1

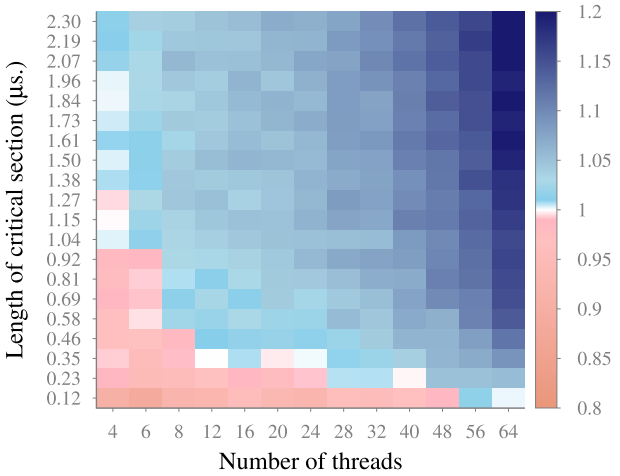


FIGURE 7. Number of lock acquisitions per unit time of catnap spinlock normalized to that of qspinlock under various combinations of thread numbers and critical section lengths.

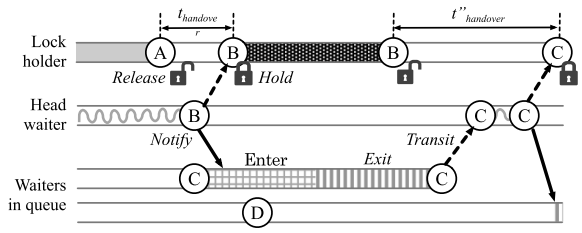


FIGURE 8. Worst case scenario in which the state transition delays the handover operation.

shorter than a certain threshold, and the threshold changes depending on the number of competing threads. We name this threshold the *harmless catnap threshold* and use it as a judgmental criterion to selectively apply the catnap spinlock only to the spinlock instances, as the performance will not be degraded by applying it.

As stated, the harmless catnap threshold depends on two factors: the critical section length and number of competing threads. The critical section length is a static parameter, while the number of contending threads is determined dynamically. Because the spinlock code should be fast and short, it cannot allow for the dynamic determination of the harmless catnap threshold. Therefore, as a conservative approach, assuming few concurrent threads, we settle on a value, 1.4 μs, for the threshold, based on the observation made in Fig. 7. However, the value of the harmless catnap threshold may differ depending on the microarchitecture of the target processor.

As stated earlier, the execution time may be slowed down by applying the catnap spinlock to the lock instances whose holding times are shorter than the threshold. Consequently, the selective application of the catnap spinlock only to the lock instances that have sufficiently long holding times is expected to confer benefits mentioned above while minimizing negative effects. However, the dynamic or runtime selection of the waiting loop is technically challenging because it additionally requires access to global lock usage statistics and conditional branches in the lock acquisition path. The former threatens the scalability, and the latter causes the pipeline stalls.

We used the *lockstat* profiler to observe the dynamic characteristics of the kernel lock instances while executing the workloads listed in Table 5. The holding time of a lock instance significantly varies because a single lock instance may protect multiple critical sections, and the execution time of a critical section depends on the input and other circumstances. Therefore, we used the average holding time of a lock instance as the judging criterion to determine whether we should apply the catnap spinlock to the instance.

The 84 lock instances were observed to be contended at least once during the execution of the workloads. Among them, 61 lock instances had an average holding time longer than the harmless catnap threshold. Table 2 lists the top

TABLE 3. Top 20 spinlock instances having the shortest average holding time.

	Lock instance	No. of acquisitions / total no. of acquisitions (%)	Acquisitions longer than exit latency (%)	Average holding time (μs)	No. of contentions / no. of acquisitions (%)	Average waiting time (μs)
1	pgdat→ split_queue_lock	0.0	4.3	0.5	5.3	0.9
2	tty→ write_wait	0.0	12.6	0.7	0.7	1.5
3	group→ notification_lock	0.0	18.7	0.7	0.1	1.1
4	mapping→ private_lock	0.0	14.2	0.8	0.0	1.3
5	hctx→ lock	0.0	22.3	0.8	0.2	1.2
6	fs→ lock	0.0	22.4	0.9	2.0	1.5
7	rlock-AF_UNIX	0.1	37.7	0.9	0.0	1.3
8	lru→ node[i].lock	0.0	40.9	1.0	0.0	1.4
9	dentry→ d_lockref.lock	8.6	37.7	1.0	1.1	1.6
10	cfs_rq→ removed.lock	0.2	46.2	1.0	0.0	1.0
11	ip→ i_flags_lock	0.4	46.0	1.0	0.2	1.1
12	sb_lock	1.4	49.8	1.1	2.3	1.1
13	semaphore→ lock	0.1	34.0	1.1	0.0	22.2
14	sb→ s_type→ i_lock_key	0.3	46.6	1.1	0.0	1.4
15	sbinfo→ stat_lock	0.1	63.7	1.1	0.2	1.4
16	sbinfo→ shrinklist_lock	0.4	64.1	1.1	0.8	1.3
17	cil→ xc_push_lock	0.0	56.9	1.1	0.4	1.1
18	init_fs.lock	0.0	60.0	1.2	30.0	1.3
19	tbl→ locks	0.0	62.5	1.3	0.0	2.8
20	unix_table_lock	0.0	69.0	1.3	0.2	1.4

20 spinlock instances that have long average waiting times. They are expected to benefit from the catnap spinlock while the spinlock instances, shown in Table 3, which are the top 20 instances with short average holding times, are likely to be penalized. The second column is the ratio of the acquisitions of the instance to the total lock acquisitions that occurred in the system during the execution. The fifth column is the ratio of the occurrences of the waiting threads to all acquisition threads to the lock instance. The average waiting time shows the sum of waiting time divided by the number of waiting thread occurrences. Therefore, even when the number of contentions is extremely low, the average waiting time can be substantial.

A lock instance with a short average holding time, listed in Table 3, generally exhibits short average waiting times because the short holding time lowered the degree of the contention, and thus the length of its waiting queue could not be extended further. Therefore, the expected benefit of applying the catnap spinlock to such instances is minimal, while its handover operation could be significantly delayed.

As the average waiting time of an instance gets longer, the expected benefit from using the catnap spinlock increases. Table 2 shows that most heavily contended instances fall in the range of the catnap spinlock application. The 14th and 19th instances were two of the few exceptions. However, their busy waiting loop will barely affect the system performance and energy efficiency because the occurrences of waiting threads for such instances are infrequent.

The shortest average holding time was 1.44 μs among the instances of which 80 percentile waiting times were longer than the target residency. It is longer than the harmless catnap threshold. Therefore, we can conclude that if we apply the catnap spinlock only to the lock instances that have an average holding time longer than the threshold, the performance damage from the state transition will be negligible, while the benefit earned from using the catnap spinlock will be maximized.

In order to selectively apply the catnap spinlock, we added the catnap counterparts of the spinlock functions to the Linux kernel source code. If spinlock functions are invoked with the `_catnap` postfix, the catnap spinlock will be used instead of the conventional qspinlock.

IV. EVALUATION

A. EVALUATION ENVIRONMENT

To assess its performance and energy efficiency benefit, we evaluated the proposed spinlock scheme using a manycore server system described in Table 4. We disabled the *lockstat* feature of the kernel to remove the probing effect. The processor used in our evaluation has 64 physical cores, and each core can simultaneously execute up to four logical threads through its SMT feature, called Hyper-threading. Therefore, a total of 256 threads can concurrently run in the processor. This provides sufficient parallelism to reproduce the contended situation of the kernel spinlocks.

In the multi-socket non-uniform memory access (NUMA) multiprocessor systems, the lock handover delay may be significantly affected by the topology or distance between the involved cores. However, such handover jitters do not exist in our evaluation system, which has only one processor. In addition, its SMT configuration shows a high logical-thread-to-physical-core ratio in comparison to the other processors being sold in the market today. Therefore, we believe that it can clearly show the effectiveness of the catnap spinlock, i.e., whether it is an adversary or favorable.

The energy consumption was measured using the 32-bit energy status counter provided with the Intel processor running average power limit (RAPL) feature. The energy status counter is designed to overflow every 1 s when the degree of power consumption is high. Therefore, when using RAPL to measure energy consumption, the monitor program must read the counter at least once per second, which can affect other threads sharing the core. Therefore, the energy measurement monitor uses one dedicated core so that it does not

TABLE 4. System configurations used for evaluation.

		Specification
Processor	Model	Intel Xeon Phi 7230F
	Number of Cores	64
	SMT per Core	4
	Clock Frequency	1.30 GHz
	On-chip Memory	16GB 7.2 GT/s (8 MCDRAM Modules)
	TDP	230 W
Memory		DDR4 2400 MHz 32 GB × 6
Storage		Samsung NVMe Z-SSD (MZPJB800HMCP-00003)
Software	OS	CentOS Linux release 7.5.1804
	Kernel	Linux kernel 4.19.44

interfere with other threads running simultaneously. Through the *taskset* feature, therefore, the benchmark is assigned to the remaining cores where the measurement monitor is not running. The measured energy consumption includes both static and dynamic energy that was consumed while each target workload was being executed.

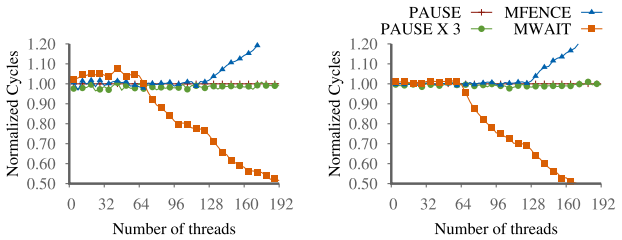
The remainder of this chapter introduces the methods and results of evaluation experiments. Two methods were used to evaluate the catnap spinlock and a system applying it. First, we investigate the impacts of the backoff of the spin-loop for queue waiters on the performance and power consumption of the SMT through a test benchmark, *locktorture*. The other method measures the overall performance and energy consumption of well-known benchmarks and is used to compare the catnap spinlock and selective catnap approach with the qspinlock of vanilla Linux kernel. Each experiment was performed ten times.

B. EVALUATION OF BACKOFF MECHANISMS

The catnap spinlock changed the backoff instruction of the spin wait loop from the `PAUSE` instruction to the `MONITOR/MWAIT` instruction pair. We analyzed the performance and energy consumption characteristics of the `MONITOR/MWAIT` pair in comparison to the `PAUSE` and `MFENCE` instructions, respectively.

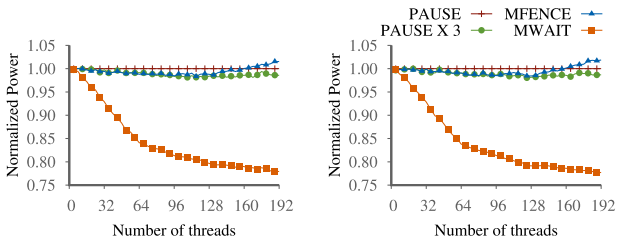
The `MFENCE` instruction provides a memory barrier and is one of the instructions that prevent speculative execution through branch prediction. Therefore, it is known to improve energy efficiency when it is used as a spinning backoff instruction instead of `PAUSE` [20]. In addition, we also conducted experiments in triplicate for a spinning loop of the `PAUSE` instruction to further prolong the backoff duration.

locktorture was used for these experiments. *locktorture* is a performance test tool for kernel lock primitives in which a varying number of threads repeats the acquisition-and-release loop and counts the number of acquisitions as its performance metric. The lock holding time of the original *locktorture* was randomly determined. Therefore, we modified it to adjust the holding time by changing the number of loop iteration in the critical section. Each test thread was pinned to each logical core.



(a) Short critical section (b) Long critical section

FIGURE 9. Lock performance change depending on the *backoff* instruction.



(a) Short critical section (b) Long critical section

FIGURE 10. Power consumption change depending on the *backoff* instruction.

Fig. 9(a) and Fig. 10(a) show the performance and average power consumption while varying the backoff instruction when the lock holding time was set to 1200 cycles (0.92 μs), which is shorter than the harmless catnap threshold. The results were normalized to that of the qspinlock kernel. When the number of concurrent threads was smaller than 64, `MONITOR/MWAIT` performed poorer by up to 10% than the other backoff instructions in terms of execution time. However, even in such cases, the average power consumption was decreased owing to catnap waiting. When the number of threads was larger than 64, its performance and energy efficiency improved because the waiters yielded processor resources to the sibling threads, including the lock holder, in the same core.

Fig. 9(b) and Fig. 10(b) show the results when the lock holding time was set to 2400 cycles (1.85 μs), which is longer than the threshold. When the holding time was longer than the threshold, the `MONITOR/MWAIT` pair did not worsen the performance in any cases, as expected. Furthermore, the degree of the performance and power consumption improvement were significantly improved compared with the case with short holding time as the number of concurrent threads increased. Regardless of the duration of lock holding time, the performance and power consumption dramatically decreased when the number of threads exceeded the number of available physical cores because the `MWAIT`-induced sleep state significantly lessened the resource competition among logical threads sharing a physical core.

Both MFENCE and PAUSE×3 reduced the power consumption by approximately 2–3%. However, their benefits did not increase proportionally with the number of threads.

On the contrary, the performance of MFENCE, along with its power consumption, was linearly worsened because the memory barrier operation adversely impacted the operation of the other logical threads when the number of concurrent threads exceeded 128. The effectiveness of both PAUSE and PAUSE×3 did not show a meaningful relationship with the degree of SMT.

C. APPLICATION BENCHMARKS

Applications running in user mode do not directly use the kernel spinlock. The kernel spinlocks are used when applications perform memory management operations, I/O or file system operations, and other kernel-related activities through system calls or other kernel interfaces. Therefore, to assess the performance and energy efficiency effectiveness of the catnap spinlock, we picked five application workloads that cause severe kernel spinlock contention from the *Re-aim* [29] and *PARSEC* [30] benchmark suites. *Re-aim* is a benchmark that simulates various real workloads by combining test units, and *PARSEC* is a benchmark suite consisting of diverse multithreaded programs that represent shared-memory programs for chip multiprocessor systems. The descriptions of the select applications are listed in Table 5.

TABLE 5. Descriptions of workloads used in evaluation.

Benchmark	Description
compute (Re-aim)	Simulation of a compute-intensive server.
dbase (Re-aim)	Simulation of a database load.
fserver (Re-aim)	Simulation of a file server.
shared (Re-aim)	Simulation of a multi-user shared server, assumed to be supporting telnet clients.
vips (PARSEC)	Fundamental image operations, such as an affine transformation and a convolution.
raytrace (PARSEC)	Rendering a three-dimensional scene onto a two-dimensional image plane. This generates little spinlock contention.

The threads in the all *Re-aim* workloads are independent and thus do not explicitly synchronize with each other while the threads of *vips* and *raytrace* explicitly synchronize with each other for concurrent processing. We set up an in-memory file system, *tmpfs*, as storage space for the applications to remove the disk I/O waiting time from the measured execution time. We measured their execution time and energy consumption while varying their configurations.

All the workloads of *PARSEC* other than *vips* and *raytrace* were excluded in our evaluation because we could not find any noticeable changes in their performance and energy consumption depending on the spinlock mechanism due to their low spinlock contention. *raytrace* was one of the workloads that do not produce any significant spinlock waiting. However, it was included to evaluate the proposed scheme under such a low contention environment.

Each workload was executed with the all-catnap spinlock kernel, selective-catnap spinlock kernel, and unmodified vanilla kernel. The results, which are shown in Fig. 11, are normalized to that from the vanilla kernel. The graph also shows the confidence interval for each result.

As shown in Fig. 11(f), *raytrace* showed negligible changes in execution time and energy consumption. The difference between the results under the proposed scheme and the vanilla kernel was mostly due to the performance variation caused by the uncontrollable factors, such as interrupts, I/O jitters, and so on. The other workloads of *PARSEC* showed the similar behavior to *raytrace*. From this point on, we will analyze the results obtained from only the five workloads that produced heavy spinlock contention.

When running with 64 threads or less, each thread was mapped to a physical core. Because the threads did not share the physical cores, no noteworthy performance gain with the improvement of the SMT effectiveness was observed. In addition, the energy consumption was barely improved in such cases because the spinlock waiting contributed to the processor utilization by merely up to 1.5% when the number of running threads was 63, as depicted in Fig. 3. However, the energy consumption for *vips*, as shown in Fig. 11(e), was reduced by 11.51% even when running with 64 threads, although the performance gain was only 5.09%. This is because *vips* showed exceptionally higher processor utilization for spin waiting than the other workloads under low numbers of running threads.

As the number of running threads grew to 128, 192, and 255, the processor utilization caused by spinlock waiting also increased. Because the number of logical threads that shared the same physical core increased to 2, 3, and 4, there would have been more chances to yield core resources to the productive threads when using the catnap spinlock. Consequently, the execution time was reduced by 7.59%, 15.70% and 19.86% on average when running with 128, 192, and 255 threads, respectively. The execution time reduction was super-linearly increased as the number of running threads grew. The average energy consumption reductions were 10.87%, 20.71% and 25.39%, respectively. The proportion of energy consumption reduction was larger than that of execution time reduction because it was contributed by both reduced power consumption of the waiting loop and shortened execution time from increased SMT efficiency.

Although most workloads used in our evaluation produced severe kernel spinlock contention, the set of highly contended spinlock instances differed depending on the workload. In addition, we observed that, for every workload, only a few spinlock instances dominated the entire lock waiting time.

Each workload showed different spinlock usage patterns, and thus the spinlock instances that impacted performance and energy consumption differed from workload to workload. The catnap spinlock was most effective for *fserver* when

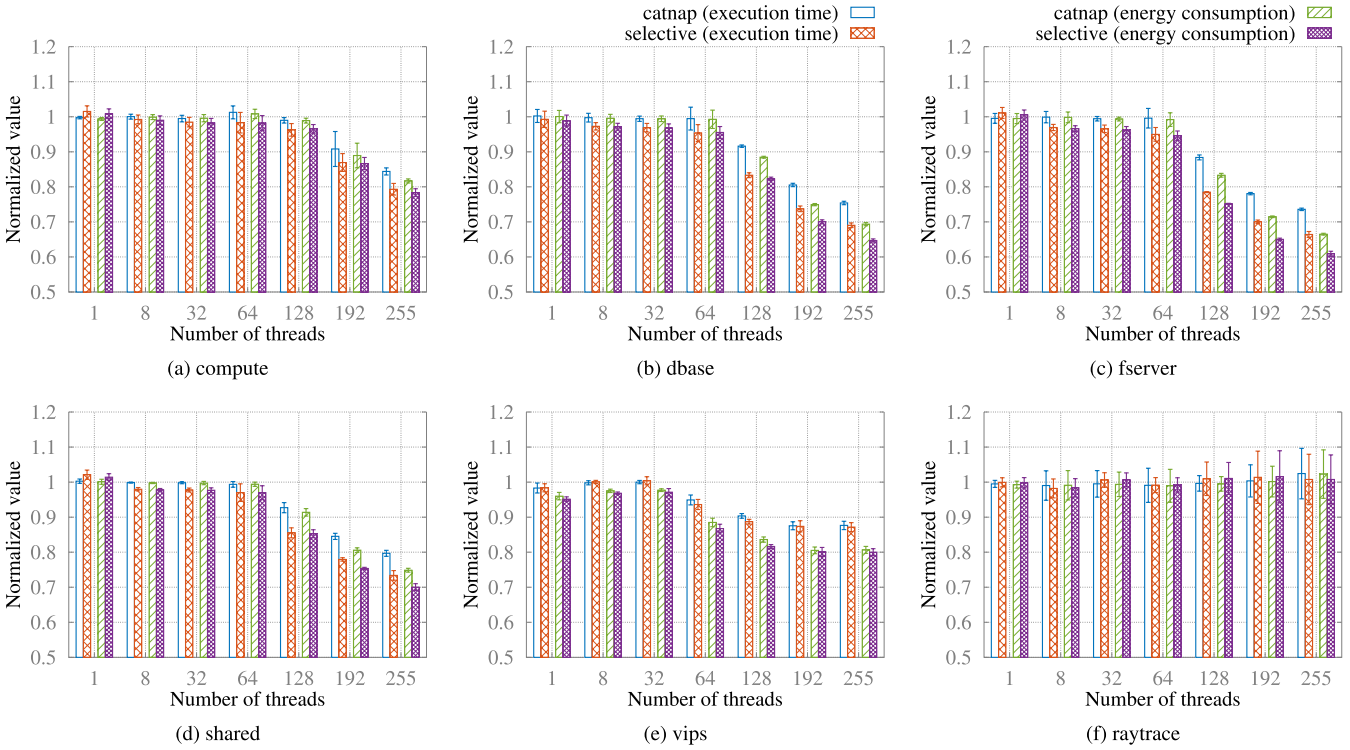


FIGURE 11. Normalized execution time and energy consumption of each workload.

running with 255 threads. The catnap spinlock reduced its execution time by 26.41%. Most of its contended waiting time, which took 24.69% of its total execution time, was caused by the `lru_lock` in the `struct pglist_data` used in virtual memory management. This lock had an average holding time of 14.3 μs, and its contention was so severe that the average waiting time was 2339.9 μs. A few spinlock instances impacted the execution time of `vips`, while a single spinlock instance dominated the total busy waiting time of `fserver`. The largest spinning waste of `vips` was caused by the `i_page` lock in the `struct address_space`, which is used for writing back the page cache. Its average holding time was 1.8 μs, and average waiting time was 280.5 μs.

The selective catnap approach showed further performance and energy efficiency improvement for a part of the workload set. For example, the selective kernel improved the execution time of `fserver` further by 9.90% when running with 128 threads. The third and forth highly contended lock instances while executing `fserver` were `d_lockref` of the directory entry cache and `sb_lock` used for the file system super block management, respectively. The average holding times of these two instances were 1.0 μs and 1.1 μs, respectively, and these are shorter than the harmless catnap threshold. Therefore, their average waiting times were 1.6 μs and 1.1 μs, respectively, even though their contention ratios were quite high at 1.1% and 2.3%, respectively. These average waiting times are shorter than the target residency. Their short average waiting times, combined with their high contention ratios, made the catnap spinlock inappropriate for these instances. Consequently, the selective catnap kernel could improve the performance and energy efficiency gain by using the qspinlock for these two.

Across all of our evaluation configurations, the performance enhancement by the catnap spinlock was 6.44% on average, and a maximum of 26.41% was achieved. Combined with the relaxed spin-loop, this performance gain reduced the energy consumption of the target workloads by 8.87% on average and by up to 33.50%. The selective catnap approach improved performance by 9.52% on average and up to 33.59% and reduced energy consumption by 11.59% on average and up to 39.11% by removing the side effects of the catnap spinlock.

V. RELATED WORK

As previously mentioned in Section II-A, the conventional spinlocks such as TAS spinlock may produce frequent remote cache accesses in modern multi-socket or multi-core systems. To resolve such issues, several spinlock mechanisms have been proposed, and they can be categorized into two approaches; scalable lock algorithms and reducing references on lock variables.

A few spinlock schemes, such as MCS spinlock [8], CLH spinlock [9], [10], k42 lock [31], and C-MCS-TKT lock [32], are built considering the underlying hardware characteristics to obtain low-interference among cores and thus scalability. In addition, a few backoff mechanisms [2], [21], [33] have been proposed to reduce the number of lock variable

references or the intensity of shared hardware resource contention due to accessing global lock variables.

To deal with the problems in congested situations, several approaches, such as the Malthusian lock [34] and requester-based lock [4], have hybridized the busy-waiting lock and the blocking lock. The requester-based lock, which is based on the Linux ticket spinlock, puts the waiters to the ACPI C-state through the `MONITOR/MWAIT` pair when the number of waiters becomes larger than the predefined threshold to resolve the scalability collapse mentioned above. However, if the number of waiters in the queue is the only criterion to determine the backoff mechanism, the handover delay may get extended when the holding time or critical section is extremely short. On the contrary, the waiters will iterate the `PAUSE` spin-loop even when the critical section is excessively long; thus, using `MWAIT` is beneficial. In addition, the authors did not consider the positive impact of their backoff mechanism on SMT effectiveness. We, on the other hand, quantitatively analyzed it in this paper.

There have been a few studies that focused on energy consumption due to spinning waste. One spinner lock [35] intended to resolve scalability issues and energy wastes in OpenMP has been proposed. The `MONITOR/MAIT` combination was applied to the user-level spinlock mechanisms used in the OpenMP framework to improve the energy efficiency of the OpenMP applications. Falsfi et al. proposed a user-level blocking lock that spins with the `MFENCE` instruction for a certain amount of time and blocks [20] to improve energy efficiency while obtaining fair performance.

In non-uniform memory access (NUMA) architecture, the latency of internal communication in a node, and the latency of inter-node communication are significantly different from each other. Therefore, many modern NUMA-aware spinlock algorithms tend to reduce inter-node communication to improve locking performance.

The first NUMA-aware spinlock is the HBO lock [36]. The TAS spinlock uses a node ID as the value of the lock, and thus it can be aware of the placement of a lock holder. If the lock holder is placed on the remote node, the HBO lock increases the backoff delay to increase the possibility of lock acquisition by the local thread. However, the HBO lock is based on the TAS spinlock; thus, it cannot guarantee scalability and can experience starvation. Resolving this drawback, a few hierarchical locks, such as the HCLH lock [37], HMCS lock [38], AHMCS lock [39], FC-MCS lock [40], and FSL lock [41] have been proposed to secure scalability. The hierarchical lock structure narrows down from the NUMA domain to the socket and to the core to select the next holder, and each waiter is continually checking on its local lock variable while waiting.

The hierarchical NUMA-aware locks introduced in this section commonly have weak fairness because they prioritize the node that has the lock ownership. In this regard, lock cohorting [42], a general approach to design NUMA-aware lock with weak fairness by combining both NUMA-oblivious and NUMA-aware approaches, has been proposed.

A compact NUMA-aware (CNA) lock is one of the recently introduced NUMA-aware spinlocks. The CNA lock is designed based on the qspinlock to use two queues for waiters in the local node and waiters in the remote node, respectively. A CNA lock requires only one word of memory, regardless of the number of sockets in the underlying machine. The CNA lock has the single-thread performance of the MCS but significantly outperforms the latter under high contention, achieving a similar level of performance when compared to the other state-of-the-art NUMA-aware locks that require substantially more space [28]. The catnap approach proposed in this paper can be easily ported to the CNA lock.

VI. CONCLUSION

The spinlock is an essential component in the OS kernel that guarantees the mutual exclusion property for critical sections in the scheduling or interrupt handling routines, which cannot be preempted, or for the extremely short critical sections. However, under high contention, the increased number of spinlock waiters in a many-core system results in poor energy efficiency and performance degradation not only of the waiting core but also of the sibling cores sharing the same physical core in the SMT architecture.

In this study, we quantitatively analyzed the impact of the kernel-internal spinlocks on the performance and energy efficiency of an SMT-featured many-core system and proposed the catnap spinlock to resolve it. The catnap spinlock used the `MONITOR/MWAIT` instruction pair in the busy waiting loop to put the waiting core into the idle state, which is recognized as an active execution state by the kernel scheduler. To avoid the extra handover delay caused by adopting the catnap state, in the proposed scheme, only the contended waiters use the catnap spin-loop, while the waiters that are expected to acquire the lock instance soon use the conventional busy waiting loop. In addition, we further reduced the possible handover delay by selectively applying the catnap spinlock only to the lock instances that have a sufficiently long average holding time based on the spinlock instance usage patterns, which were collected while executing the target workloads.

The proposed approach was implemented in the Linux kernel and evaluated with a many-core server that has 64 physical cores supporting 256 logical threads. Our evaluation showed that the selective catnap spinlock kernel reduced the execution time of the benchmark workloads by up to 33.59% and improved the energy consumption by up to 39.11%

Our approach can be easily ported to various kinds of locks, including the blocking locks with the spin-and-park policy and the NUMA-aware hierarchical locks. In addition, our analysis and approach introduced in this paper can be applied to wherever busy waiting loops are being used, such as the I/O polling loop for high-performance I/O operation. Busy waiting loops are quintessential for lots of reasons, and being used in various parts. We believe that our approach can significantly suppress the resource and energy waste caused

by the busy waiting loops, which is exponentially growing as the number of cores equipped in a system increases.

REFERENCES

[1] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich, "An analysis of linux scalability to many cores," in *Proc. USENIX OSDI*, 2010, pp. 1–16.

[2] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich, "Non-scalable locks are dangerous," in *Proc. OLS*, 2012, pp. 119–130.

[3] D. Bueso, "Scalability techniques for practical synchronization primitives," *Commun. ACM*, vol. 58, no. 1, pp. 66–74, Dec. 2014.

[4] Y. Cui, Y. Wang, Y. Chen, and Y. Shi, "Requester-based spin lock: A scalable and energy efficient locking scheme on multicore systems," *IEEE Trans. Comput.*, vol. 64, no. 1, pp. 166–179, Jan. 2015.

[5] T. E. Anderson, "The performance of spin lock alternatives for shared-money multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 1, no. 1, pp. 6–16, Jan. 1990.

[6] T. David, R. Guerraoui, and V. Trigonakis, "Everything you always wanted to know about synchronization but were afraid to ask," in *Proc. ACM SOSP*, 2013, pp. 33–48.

[7] J. Corbet. (2008). *Ticket Spinlocks*. [Online]. Available: https://lwn.net/Articles/267968/

[8] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, no. 1, pp. 21–65, Feb. 1991.

[9] T. Craig, "Building FIFO and priority queuing spin locks from atomic swap," Univ. Washington, Seattle, WA, USA, Tech. Rep. TR 93-02-02, Feb. 1993.

[10] P. Magnusson, A. Landin, and E. Hagersten, "Queue locks on cache coherent multiprocessors," in *Proc. 8th Int. Parallel Process. Symp.*, Dec. 2002, pp. 165–171.

[11] W. Long. (2013). *qspinlock: Introducing a 4-Byte Queue Spinlock Implementation*. [Online]. Available: https://lwn.net/Articles/561775/

[12] J. K. Ousterhout, "Scheduling techniques for concurrent systems," in *Proc. ICDCS*, vol. 82, 1982, pp. 22–30.

[13] V. Cakarevic, P. Radojkovic, J. V. Mulá, F. J. C. Almeida, R. Gioiosa, M. A. P. González, M. Nemirovsky, and M. V. Cortes, "Understanding the overhead of the spin-lock loop in CMT architectures," in *Proc. WIOSCA*, 2008.

[14] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton, "Hyper-threading technology architecture and microarchitecture," *Intel Technol. J.*, vol. 6, no. 1, pp. 1–12, 2002, pp 1–10.

[15] H. Esmaeilzadeh, T. Cao, Y. Xi, S. M. Blackburn, and K. S. McKinley, "Looking back on the language and hardware revolutions: Measured power, performance, and scaling," in *Proc. ASPLOS*, 2011, pp. 319–332.

[16] J. Jeffers, J. Reinders, and A. Sodani, "Knights landing architecture," in *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*, T. Green, Ed. San Mateo, CA, USA: Morgan Kaufmann, 2016, ch. 4, pp. 63–71.

[17] "Multicore and hyper-threading technology," in *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Santa Clara, CA, USA: Intel Corporation, Apr. 2019, ch. 10, pp. 8–12.

[18] H. Ou. (May 2010). *Long Duration Spin-Wait Loops on Hyper-Threading Technology Enabled Intel Processors*. Intel Corporation. [Online]. Available: https://software.intel.com/en-us/articles/long-duration-spin-wait-loops-on-hyper-threading-technology-enabled-intel-processors

[19] N. Anastopoulos and N. Koziris, "Facilitating efficient synchronization of asymmetric threads on hyper-threaded processors," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, Apr. 2008, pp. 1–8.

[20] B. Falsafi, R. Guerraoui, J. Picorel, and V. Trigonakis, "Unlocking energy," in *Proc. USENIX ATC*, 2016, pp. 393–406.

[21] J. Corbet. (2013). *Improving Ticket Spinlocks*. [Online]. Available: https://lwn.net/Articles/531254/

[22] "PAUSE—Spin loop hint," in *Intel 64 and IA-32 Architectures Software Developer Manuals: Instruction Set Reference, A-Z*, vol. 2. Santa Clara, CA, USA: Intel Corporation Jan. 2019, ch. 4, p. 4:233.

[23] "PAUSE," in *AMD64 Architecture Programmer's Manual: General-Purpose and System Instructions*, vol. 3. Santa Clara, CA, USA: Advanced Micro Devices, May 2018, ch. 3, p. 260.

[24] *Using Spin-Loops on Intel Pentium 4 Processor and Intel Xeon Processor*, Intel Corp., Santa Clara, CA, USA, May 2001.

[25] "MWAIT—Monitor wait," in *Intel 64 and IA-32 Architectures Software Developer Manuals: Instruction Set Reference, A-Z*, vol. 2. Santa Clara, CA, USA: Intel Corporation, Jan. 2019, ch. 4, pp. 4:162–4:164.

[26] S. Kanev, K. Hazelwood, G.-Y. Wei, and D. Brooks, "Tradeoffs between power management and tail latency in warehouse-scale applications," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Oct. 2014, pp. 31–40.

[27] P. E. McKenney. *Kernel Lock Torture Test Operation*. Accessed: Feb. 10, 2020. [Online]. Available: https://www.kernel.org/doc/Documentation/locking/locktorture.txt

[28] D. Dice and A. Kogan, "Compact NUMA-aware Locks," in *Proc. 14th EuroSys Conf. (EuroSys)*, 2019, p. 12.

[29] C. White, "Performance testing the Linux Kernel," in *Proc. OLS*, 2003, pp. 457–469.

[30] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proc. PACT*, Oct. 2008, pp. 72–81.

[31] M. Auslander, D. Edelsohn, O. Krieger, B. Rosenburg, and R. Wisniewski, "Enhancement to the MCS lock for increased functionality and improved programmability," U.S. Patent 20 030 200 457 A1, Oct. 23, 2003.

[32] S. Awamoto, H. Chishiro, and S. Kato, "Scalable and memory-efficient spin locks for embedded tile-based many-core architectures," in *Proc. IEEE 21st Int. Symp. Real-Time Distrib. Comput. (ISORC)*, May 2018, pp. 25–32.

[33] P. Hoai Ha, M. Papatriantafilou, and P. Tsigas, "Reactive spin-locks: A self-tuning approach," in *Proc. 8th Int. Symp. Parallel Archit., Algorithms Netw. (ISPAN)*, Jan. 2006, p. 6.

[34] D. Dice, "Malthusian Locks," in *Proc. 12th Eur. Conf. Comput. Syst. (EuroSys)*, 2017, pp. 314–327.

[35] H. Akkan, M. Lang, and L. Ionkov, "HPC runtime support for fast and power efficient locking and synchronization," in *Proc. IEEE Int. Conf. Cluster Comput. (CLUSTER)*, Sep. 2013, pp. 1–7.

[36] Z. Radovic and E. Hagersten, "Hierarchical backoff locks for nonuniform communication architectures," in *Proc. 9th Int. Symp. High-Perform. Comput. Archit. (HPCA)*, Aug. 2003, pp. 241–252.

[37] V. Luchangco, D. Nussbaum, and N. Shavit, "A hierarchical CLH queue lock," in *Proc. 12th Int. Euro-Par Conf.*, Dresden, Germany, Aug./Sep. 2006, pp. 801–810.

[38] M. Chabbi, M. Fagan, and J. Mellor-Crummey, "High performance locks for multi-level NUMA systems," *SIGPLAN Not.*, vol. 50, no. 8, pp. 215–226, Jan. 2015.

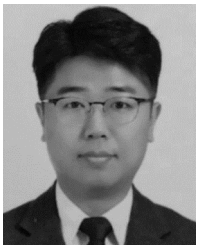
[39] M. Chabbi and J. Mellor-Crummey, "Contention-conscious, locality-preserving locks," *SIGPLAN Not.*, vol. 51, no. 8, pp. 1–14, Feb. 2016.

[40] D. Dice, V. J. Marathe, and N. Shavit, "Flat-combining NUMA locks," in *Proc. 23rd ACM Symp. Parallelism Algorithms Archit. (SPAA)*, 2011, pp. 65–74.

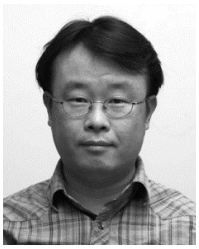
[41] B. Zhang, J. Kang, T. Wo, Y. Wang, and R. Yang, "A flexible and scalable affinity lock for the Kernel," in *Proc. IEEE Int. Conf High Perform. Comput. Commun., IEEE 6th Int. Symp. Cyberspace Saf. Secur., IEEE 11th Int. Conf. Embedded Softw. Syst (HPCC,CSS,ICESS)*, Aug. 2014, pp. 34–37.

[42] D. Dice, V. J. Marathe, and N. Shavit, "Lock cohorting: A general technique for designing NUMA locks," *ACM Trans. Parallel Comput.*, vol. 1, no. 2, p. 13, 2015.

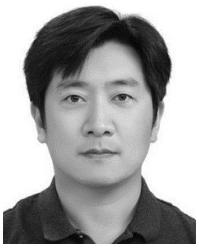
YOUNGJOO WOO received the B.S. degree in electronic engineering from Inha University, South Korea, in 2009, the M.S. degree in computer science from the Ulsan National Institute of Science and Technology (UNIST), in 2012, and the Ph.D. degree from Sungkyunkwan University, in 2019. She is currently a Postdoctoral Researcher with the Electrical and Telecommunications Research Institute (ETRI). Her research interests include operating systems, many-core systems, energy-efficient computing, and cloud computing.



SUNGHUN KIM received the B.S., M.S., and Ph.D. degrees in computer engineering from Sungkyunkwan University, in 2012, 2014, and 2019, respectively. He is currently working with the Samsung Research, Samsung Electronics Company, Ltd., South Korea, where he is currently developing and optimizing the Tizen operating system. His research interests include operating systems, memory systems, and embedded system optimization.



CHEOLGI KIM received the B.S. degree in computer science and the Ph.D. degree from the Korea Advance Institute of Science and Technology (KAIST), in 1996 and 2004, respectively. He is currently an Associate Professor with the Software and Computer Engineering Department, Korea Aerospace University, South Korea. In 2017, he founded Ratio LLC, South Korea, an IoT device company. His research areas are safety critical system software architecture, low-energy embedded systems, and real-time systems.



EUISEONG SEO received the B.S., M.S., and Ph.D. degrees in computer science from KAIST, in 2000, 2002, and 2007, respectively. He is currently an Associate Professor with the Department of Computer Science and Engineering, Sungkyunkwan University, South Korea. Before joining Sungkyunkwan University, in 2012, he was an Assistant Professor with the Ulsan National Institute of Science and Technology (UNIST), South Korea, from 2009 to 2012, and a Research Associate with the Pennsylvania State University, from 2007 to 2009. His research interests are system software, embedded systems, and cloud computing.