차 례

표 차례

그림 차례

제 1 장 머릿말

본문을 한글로 작성할 때 머릿말로 시작을 하시는 게 좋습니다. [1] 인용은 다음과 같이 합니다 [2]-[8]. 인용은 뒤에 인용을 쓰는 칸이 있습니다. 참고하여서 인용하시길 바랍니다 [9, 10]. 한글 논문에는 영어를 쓰지 마시기 바랍니다.

제 2 장 본문

2.1 서론

최근 코어수가 증가하고 있다. 따라서 멀티코어에서 매니코어 시스템으로 바뀌고 있다. 매니코어 시스템에 대한 운영체제 커널의 parallelism은 시스템 전체의 parallelism에서 가장 중요하다. 만약 커널이 scale하지 않으면, 그 위에 동작하는 응용프로그램들도 역시 scale하지 않는다[]. 이처럼 중요한 운영체제 커널 중 멀티코어 또는 매니코어 환경에서 많이 사용되는 운영체제가 리눅스 커널이다[]. 하지만 리눅스 커널은 아직 확장성 문제가 있다 [?] [?]. 확장성 문제 중 하나는 락 경합 때문에 발생하는 업데이트 직렬화 문제이다 [?] [?]. 그 이유는 업데이트 오퍼레이션은 여러 쓰레드가 동시에 수행되지 못하기 때문이다 [?].

이처럼 업데이트 직렬화 문제를 해결하기 위해 여러 동시적 업데이트 방법 들이 연구되고 있다 [?] [?]. 이러한 동시적 업데이트 방법들은 워크로드 특성인 업데이트 비율에 따라 많은 성능 차이를 보인다 [?]. 이 중 높은 업데이트 비율을 가진 자료 구조 때문에 발생하는 학장성 문제를 해결하기 위한 여러 방법이 연구 되고 있다. 그 중 하나는 cache communication bottleneck을 줄인 log-based 알고리즘 [?] [?] [?]을 사용하는 것이다. Log-based 알고리즘은 업데이트가 발생하면, data structure의 업데이트 operation을 per-core 또는 atomic 하게 log로 저장하고 read operation을 수행하기 전에 저장된 로그를 수행하는 것이다 . 이것은 마치 CoW(Copy On Write)와 유사하다 [?].

S. Boyd-Wickizer et al.는 동기화된 타임스탬프 카운터(synchronized timestamp counters) 기반의 per-core log를 활용하여 update-heavy한 자료구조를 대상으로 동시적 업데이트 문제를 해결함과 동시에 cache communication bottleneck을 줄였다 [?]. 동기화된 타임스탬프 카운터 기반의 per-core log를 활용한

동시적 업데이트방법은 업데이트 부분만 고려했을 때, per-core에 데이터를 저장함으로 굉장히 높은 scalability를 가진다[]. 하지만 per-core 기반의 동기화된 타임스탬프 카운터를 사용한 방법은 결국 timestamp merging and ordering 작업을 야기한다. 만약 코어 수가 늘어 날 경우, 로그를 자료 구조에 적용하는 과정에서 timestamp 때문에 발생하는 추가적인 sequntial 프로세싱이 요구된다. 이것은 결국 확장성과 성능을 저해한다.

본 논문은 동기화된 타임스탬프 카운터를 이용함에 따라 생기는 추가적인 sequencial processing 문제를 해결하기 위해 LDU(Lightweitgh log-based Deferred Update)를 개발하였다. LDU는 타임스탬프 카운터가 필요한 operation log를 업데이트 순간마다 지우고, 매번 로그를 생성하지 않고 재활용하는 방법이다. 이로인해 synchronized timestamp counter 문제와 cache communication bottleneck 문제를 동시에 해결하였다. 해결 방법은 분산 시스템에서 사용하는 log기반의 concurrent updates 방식과 shared-memory system의 hardware-based synchronization 기법(compare and swap, test and set, atomic swap)을 조합하여 동시적 업데이트 문제를 해결하였다.

이처럼 동기화된 타임스탬프 카운터를 제거함과 동시에, cache communication bottleneck 줄인 LDU는 log-based 알고리즘들의 장점들을 모두 포함할 뿐만아니라 추가적인 장점을 가진다. 첫째로, update가 수행하는 시점 즉 로그를 저장하는 순간에는 lock이 필요가 없다. 따라서 lock에 대한 오버헤드 없이 concurrent updates를 수행할 수 있다 둘째로, 저장된 update operation log를 coarse-grained lock과 함께 하나의 코어에서 수행하기 때문에, cache 효율성이 높아진다 [**?**]. 셋째로, 기존 여러 자료구조에 쉽게 적용할 수 있는 장점이 있다. 게다가 마지막으로, log를 저장하기 전에 로그를 삭제하므로 보다 빠르게 log의 수를 줄일 수 있다.

우리는 위와 같은 장점을 가지는 LDU를 리눅스 커널에서 high update rate 때문에 scalability 문제를 야기시키는 anonymous reverse mapping과 file reverse mapping에 적용하였다. 또한 우리는 LDU를 Linux 4.5.rc4에 구현하였고, fork-intensive 워크로드인 AIM7 [**?**], Exim [**?**] from MOSBENCH [**?**], lmbench [**?**]를 대상으로 성능 개선을 보였다. 개선은 stock 리눅스 커널에 비해 120코어에서 각각 x,x,x 배이다.

Contributions. This paper makes the following contributions:

- 우리는 리눅스 커널을 위한 새로운 log-based concurrnet updates 방법인 LDU를 개발하였다. LDU는 동기화된 타임스탬프 카운터를 이용함에 따라 생기는 시간 정렬과 머징에 의한 추가적인 sequencial processing 문제를 해결하였다. 이를 위해 LDU는 로그를 업데이트 순간 지우고 로그를 재활용하는 방법을 개발하였다.

- 우리는 LDU을 practical한 manycore system인 intel xeon 120코어 위에 동작하는 리눅스 커널의 2가지 reverse mapping(anonymous, file)에 적용하여, fork scalability 문제를 해결하였다. Fork 관련 벤치마크 성능은 워크로드 특성에 따라 1.6x부터 2.2x까지 개선되었다.

The rest of this paper is organized as follows. Section 2 describes the background and Linux scalability problem. Section 3 describes the design of the LDU algorithm and Section 4 explains how to apply to Linux kernel. section 5 explains our implementations in Linux and Section 6 shows the results of the experimental evaluation. Finally, section 8 concludes the paper.

‘

2.2 Background and Problem

2.3 LDU Design

LDU는 리눅스 커널의 hight update rates를 가진 data strcuture의 Scalability 위한 새로운 로그 기반의 위한 Concurrent Updates 방법이다. LDU는 timestamp를 이용하여 발생하는 log를 관리에 대한 어려움을 해결하였다. timestamp를 사용하지 않기 위해 LDU는 global queue를 이용하는데 이 때 발생

하는 head pointer에 대한 cache invalidate 줄이기(mitigating) 위해 3가지 방법 (light weight queue, Update-side Abosrbing, reusing garbage object)을 병행 하였다. 이러한 LDU의 기본적인 철학은 distributed system에서 주로 사용하는 time-stamp log기반 방식의 concurrent updates 방법과 shared memory system 에서만 사용할 수 있는 CAS와 같은 atomic operation을 절묘하게 결합하여 설계하였다. 즉 저장된 log를 순서대로 수행 하기 위해 최소한의 atomic operation 을 사용하도록 설계하였다. This section explains the algorithmic design aspects of LDU.

2.3.1 Log-based Concurrent updates

2.3.2 Approach

2.3.3 The LDU Algorithm

2.3.4 LDU logical update

2.3.5 LDU Physical update

2.3.6 LDU Correctness

2.4 Concurrent updates for Linux kernel

2.4.1 Case study:reverse mapping

2.4.2 anon vma

* LDU.
* PLDU.

2.4.3 file mapping

* LDU.

* PLDU.

2.5 Implementation

2.6 Evaluation

This section answers the following questions experimentally:

- Does LDU's design matter for applications?
- Why does LDU's scheme scale well?

2.6.1 Experimental setup

To evaluate the performance of LDU, we use well-known three benchmarks:AIM7 Linux scalability benchmark, Exim email server in MOSBENCH and lmbench. We selected these three benchmarks because they are fork-intensive workloads and exhibit high reverse mapping lock contentions. Moreover, AIM7 benchmark has widely been used in practical area not only for testing the Linux but also for improving the scalability. To evaluate LDU for real world applications, we use Exim which is the most popular email server. A micro benchmark, Lmbench, has been selected to focus on Linux fork operation-intensive fine grained evaluations. Finally, we wanted to focus on Linux fork performance and scalability;therefore, we selected lmbench, a micro benchmark.

In order to evaluate Linux scalability, we used four different experiment settings. First, we used the stock Linux as the baseline reference. Second, we used ordered Harris lock-free list while we apply unordered Harris lock-free list for the third setting (see section 2.5). Finally, we used combination of unordered Harris lock-free list for anonymous mapping and our LDU for file mapping.

Since we cannot obtain detailed implementation of Oplog, we could not include comparison between LDU and Oplog in this paper.

We compare our LDU implementation to a concurrent non-blocking Harris linked list [**?**];therefore, we implement the Harris The code refers from synchrobench [**?**] and ASCYLIB [**?**], and we convert their linked list to Linux kernel style. Because both synchrobench and ASCYLIB leak memory, we implement additional garbage collector for the Linux kernel using Linux's work queues and lock-less list.

In order to further improve performance, we move their ordered list to unordered list. A feature of the Harris linked list is all the nodes are ordered by their key. Zhang [**?**] implements a lock-free unordered list algorithm, whose list is each insert and remove operation appends an intermediate node at the head of the list;these approach is practically hard to implement. Indeed, Linux does not require contains operation because the Linux data structures such as list, tree and hash table not depended on search key;they depend on their unique object. This feature can eliminates the ordered list in Harris linked list. Therefore, we perform each insert operation appends an intermediate node at the first node of the list;on the other hand, each remove operation searches from head to their node.

We ran the three benchmarks on Linux 3.19.rc4 with stock Linux with the automatic NUMA balancing feature disabled because the Harris linked list has the iteration issue [**?**]. All experiments were performed on a 120 core machine with 8-socket, 15-core Intel E7-8870 chips equipped with 792 GB DDR3 DRAM.

2.6.2 AIM7

AIM7 forks many processes, each of which concurrently runs. We used AIM7-multiuser, which is one of workload in AIM7. The multiuser workload is composed of various workloads such as disk-file operations, process creation, virtual memory operations, pipe I/O, and arithmetic operation. To minimize IO bottlenecks, the workload was executed with tmpfs filesystems, each of which is 10 GB. To increase the number of users during our experiment and show the results at the peak user numbers, we used the crossover.

The results for AIM7-multiuser are shown in Figure 2.3, and the results show the throughput of AIM7-multiuser with four different settings. Up to 60 core, the stock Linux scales linearly while serialized updates in Linux kernel become bottlenecks. However, up to 120core, unordered harris list and our LDU scale well because these workloads can run concurrently updates and can reduce the locking overheads due to reader-writer semaphores(`anon_vma`, `file`). The combination of LDU with unordered harris list has best performance and scalability outperforming stock Linux by 1.7x and unordered harris list by 1.1x. While the unordered harris list has 19% idle time(see Table 2.1), stock Linux has 51% idle time waiting to acquire both `anon_vma's rwsem` and `file's i_mmap_rwsem`. We can notice that although LDU has 23% idle time, the throughput is higher than unordered harris list. In this benchmark, the ordered harris list has the lowest performance and scalability because their `CAS` fails frequently.

2.6.3 Exim

To measure the performance of Exim, shown in Figure 2.4, we used default value of MOSBENCH to use tmpfs for spool files, log files, and user mail files. Clients run on the same machine and each client sends to a different user to prevent contention on user mail file. The Exim was bottlenecked by per-directory locks protecting file creation in the spool directories and by forks performed on different cores [?]. Therefore, although we eliminate the fork problem, the Exim may suffer from contention on spool directories.

Results shown in Figure 2.4 show that Exim scales well for all methods up

AIM7	user	sys	idle
Stock(anon, file)	2487 s	1993 s	4647 s(51%)
H(anon, file)	1123 s	3631 s	2186 s(31%)
H-unorder(anon, flie)	3630 s	2511 s	1466 s(19%)
H-unorder(anon), L(file)	3630 s	1903 s	1662 s(23%)

EXIM	user	sys	idle
Stock(anon, file)	41 s	499 s	1260 s(70%)
H(anon, file)	47 s	628 s	1124 s(62%)
H-unorder(anon, file)	112 s	1128 s	559 s(31%)
H-unorder(anon), L(file)	87 s	1055 s	657 s(37%)

lmbench	user	sys	idle
Stock(anon, file)	11 s	208 s	2158 s(91%)
H(anon, file)	11 s	312 s	367 s(53%)
H-unorder(anon, file)	11 s	292 s	315 s(51%)
H-unorder(anon), L(file)	12 s	347 s	349 s(49%)

표 2.1: Comparison of user, system and idle time at 120 cores.

to 60 core but not for higher core counts. The stock Linux shows performance degradation for more than 60 core. Both unordered harris list and our LDU do not suffer from performance loss because they do not acquire the `anon_vma` semaphore and `i_mmap` semaphore in fork. LDU performs better due to the fact that it uses both update-side absorbing and lock-less list, outperforming stock Linux by 1.6x and unordered harris list by 1.1x. Even though we applied scalable solution, Exim shows limitation on scalability improvement since the main bottleneck is per-directory lock contention on spool directories. The unordered harris list has 31% idle time, whereas LDU has 37% idle time due to the their efficient concurrent updates.

2.6.4 lmbench

lmbench has various workloads including process creation workload(fork, exec, sh -c, exit). This workload is used to measure the basic process primitives such as creating a new process, running a different program, and context switching. We configured process create workload to enable the parallelism option which specifies the number of benchmark processes to run in parallel [**?**]; we used 100 processes.

The results for lmbench are shown in Figure 2.5, and the results show the execution times of the fork microbenchmark in lmbench with four different methods. Three methods outperform stock Linux by 2.2x at 120 cores;however, before 30 core, two harris list have lower performance due to their execution overheads. While stock Linux has 90% idle time, other methods have approximately 50% idle time since stock Linux waits to acquire reverse mapping locks such as `anon_vma's rwsem` and `mapping's i_mmap_rwsem`.

2.7 Discussion and future work

2.8 Related work

In order to improve Linux scalability, researchers have been optimized memory management in Linux by finding and fixing scalability bottlenecks.

Shared address spaces in multithreaded applications easily become scalability bottlenecks since kernel operations including `mmap` and `munmap` system calls and `page faults` handling require per-process locks for synchronization. Multithreaded application, for example, can become bottleneck by kernel operations on their shared address space, whose operations are the `mmap` and `munmap` system calls and `page faults`. These operations are synchronized by a single per-process lock. BonsaiVM [**?**] solved this address space problem by using the RCU; RadixVM [**?**] created a new VM using refcache and radix tree, which enable `munmap`, `mmap`, and `page fault` on non-overlapping memory regions to scale perfectly. Alternatively, to avoid contention caused by shared address space locking, system programmers change their multithreaded applications to use processes [**?**].

Though sufficient level of performance scalability has been achieved for reader intensive operations through RCU and Hazard pointer, solutions to scalability for update-heavy operations has not been satisfiable. A recent paper by Arbel and Attiya [**?**] shows a new design of concurrent search tree called the Citrus tree. The Citrus tree combines RCU and fine-grained locks, and it supports concurrent write operations that traverse the search tree by using RCU concurrently. When increasing the update rate, Citrus tree still suffers from bottlenecks. RLU [**?**] presents a new synchronization mechanism that allows unsynchronized sequences of reads to execute concurrently with updates. In high update rate, Oplog can achieve substantially multi-core scaling for update-heavy data structures. Our work focus on update-heavy data structures and uses non-blocking method to store the operation log instead of per-core processing.

One method for the concurrent update is using the non-blocking algorithms [?] [?] [?], which are based on CAS. In non-blocking algorithms, each core tries to read the values of shared data structures from its local location, but has possibility of reading obsolete values. CAS is performed at the time of reading values that are not the current values and CAS fails and requires retrials sometimes when the values have been overwritten. These algorithms execute optimistically as though they read the value at location in their data structure;they may obtain stale data at the time. When they observed against the current value, they execute a CAS to compare the against value. The CAS fails when the value has been overridden, and they must be retried later on. Consequently, both repeated CAS operation and their iteration loop caused by CAS fails cause bottlenecks due to inter-core communication overheads [?]. Moreover, none of the non-blocking algorithms implements an iterator, whose data structure just consists of the insert, delete and contains operations [?]. The Linux, however, commonly uses the iteration to read, so when applying non-blocking algorithms to the Linux, they may meet this iteration problem. Petrank [?] solved this problem by using a consistent snapshot of the data structure; this method, however, may require a lot of effort to apply its sophisticated algorithms to Linux. For evaluation purposes, we implemented Harris linked list [?] to Linux, and we sometimes have failure where reading the pointer that had been deleted by updater concurrently result of the problem of the iteration.

MCS [?], a scalable exclusive lock, is used in the Linux kernel [?]. to avoid unfairness at high contention levels, so this scalable exclusive lock can be used for fine grained locking in Linux. However, in MCS, since only one thread may hold the lock at a time, it can cause low scalability in case of long critical regions. Reader-writer lock [?] allows either number of readers to execute concurrently or single writer to execute. Thus, readers-writer locks allow better scalability in case of read-mostly objects.

In read-mostly data structures, RCU [?] can be quite useful since it allows read operations to proceed without read locks, and delays freeing of data structures to avoid races. One drawback is that as the update rate increases, their performance and scalability decrease due to a single writer and their synchronization function. Consequently, scalable exclusive lock, reader-writer lock and RCU require serialization for updates and thus show significant limitation on scalability.

2.9 Conclusion

We propose a concurrent update algorithm, LDU, for update-heavy data structures scalable for many-core systems. To achieve the scalability during process spawning, we applied deferred log processing with global log queue and update-side absorbing to Linux reverse mapping. Evaluation results using the AIM7, Exim and lmbench reveal that LDU shows better performance up to 2.2 times compared to existing solutions. LDU is implemented on to Linux kernel 3.19 and available as open-source from `https://github.com/KMU-embedded/scalablelinux`.

2.10 Acknowledgments

This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (14-824-09- 011, “Research Project on High Performance and Scalable Many-core Operating System”)

2.11 작성

장과 절 그리고 부절로 본문을 작성하실 수 있습니다.

2.11.1 자동

이것들은 자동으로 차례에 들어가게 됩니다.

2.12 한글 논문

한글 논문에는 영어를 쓰지 마시기 바랍니다.

```
function logical_insert(obj, root):
  If xchg(obj.del_node.mark, 0) ≠ 1:
     BUG(obj.add_node.mark)
     obj.add_node.mark ← 1
     If test_and_set_bit(OP_INSERT, obj.exist) ≠ true:
        set_bit(OP_INSERT, obj.used):
        obj.add_node.op ← OP_INSERT
        obj.add_node.key ← obj
        obj.add_node.root ← root
        add_lock_less_list(obj.add_node)


function logical_remove(obj, root):
  If xchg(obj.add_node.mark, 0) ≠ 1:
     BUG(obj.del_node.mark)
     obj.del_node.mark ← 1
     If test_and_set_bit(OP_REMOVE, obj.exist) ≠ true:
        set_bit(OP_REMOVE, obj.used):
        obj.del_node.op ← OP_REMOVE
        obj.del_node.key ← obj
        obj.del_node.root ← root
        add_lock_less_list(obj.del_node)
```

그림 2.1: LDU logical update algorithm. `logical_insert` represents non-blocking insert function. It may be called by original insert position without locks. The fastpath is that when their object was removed by `logical_remove`, `logical_insert` just changes node's marking field.

```
function synchronize_ldu(obj, head):
  If (head.first = NULL):
    return;
  entry ← xchg(head.first, NULL);
  for each list node:
    obj ← node.key
    If !xchg(node.mark, 0):
      physical_update(node.op, obj, node.root)
    clear_bit(node.op, obj.used)
    If !xchg(node.mark, 0):
      physical_update(node.op, obj, node.root)

function physical_update(op, obj, root):
  If op = OP_INSERT :
    call real insert function(obj, root)
  Else If op = OP_REMOVE :
    call real remove function(obj, root)
```

그림 2.2: LDU physical update algorithm. `synchronize_ldu` may be called by reader and converts update log to original data structure traversing the lock-less list.

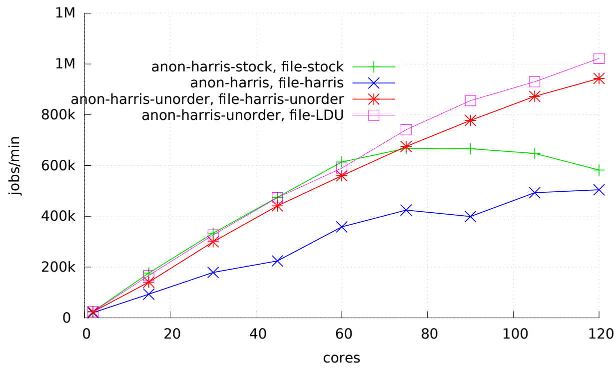

그림 2.3: Scalability of AIM7 multiuser for different method. The combination LDU with unordered harris list scale well;in contrast, up to 60 core, the stock Linux scale linearly, then it flattens out.

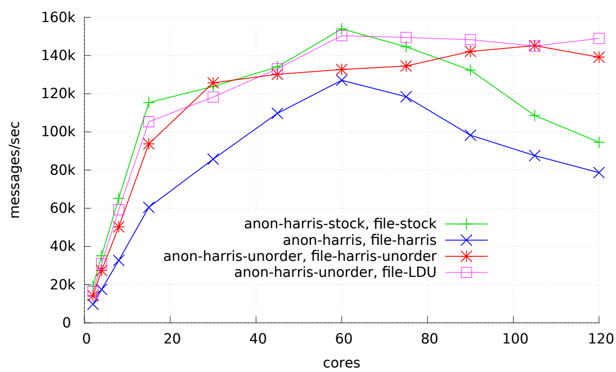

그림 2.4: Scalability of Exim. The stock Linux collapses after 60 core;in contrast, both unordered harris list and our LDU flatten out.

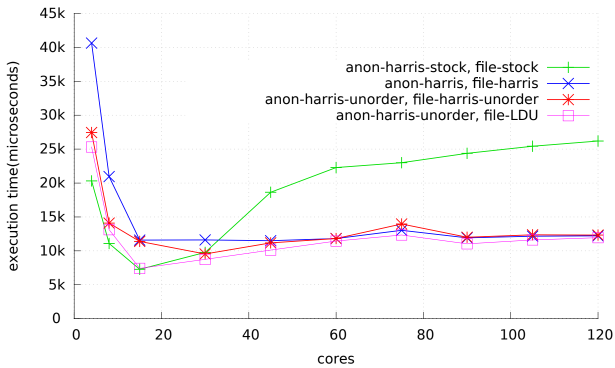

그림 2.5: Execution time of lmbench's fork micro benchmark. The fork micro benchmark drops down for all methods up to 15 core but either flattens out or goes up slightly after that. At 15 core, the stock Linux goes up;the others flattens out

제 3 장 그림, 표

3.1 그림과 표를 본문에서 이야기하기

본문에서 그림과 표에 관해 이야기를 할 때도 인용에서처럼 하시면 됩니다.

표 3.1: 표 제목을 넣으십시오.

	BF	SW-I		SW-II		SW-III	CAP
		Para	Ferro	Para	Ferro		
E (eV)	0	7.796	7.832	10.418	10.408	11.5	13.2
M (μ_B)	0	0	1.94	0	2.06	0	0

제 4 장　맺음말

마지막은 맺음말로 하는 것을 권합니다.

참 고 문 헌

[1] 박상우, 동시 송수신 안테나를 두 개 쓰는 협력 인지 무선통신망에 알맞은 전 이중 통신, 한국과학기술원 석사 학위 논문, 2016.

[2] 송익호, 박철훈, 김광순, 박소령, 확률변수와 확률과정, 자유아카데미, 2014.

[3] 송익호, 안태훈, 민황기, 인지 무선에서의 광대역 주파수 검출 방법 및 장치, 특허등록번호 10-1494966, 2015년 2월 12일.

[4] 호우위시, 이원주, 이승원, 안태훈, 이선영, 민황기, 송익호, "선형 판별 분석에서 부류안 분산 행렬의 영 공간 재공식화," 한국통신학회 2012년도 추계종합학술발표회, 대한민국 고려대학교, 242-243쪽, 2012년 11월.

[5] 민황기, 안태훈, 이승원, 이성로, 송익호, "비간섭 전력 부하 감시용 고차 적률 특징을 갖는 전력 신호 인식," 한국통신학회논문지, 제39C권, 제7호, 608-614쪽, 2014년 7월.

[6] S. Park, *Full-Duplex Communication for Cooperative Cognitive Radio Networks with Two Simultaneous Transmit and Receive Antennas*, Master Thesis, Korea Adv. Inst. Science, Techn., Daejeon, Republic of Korea, 2016.

[7] I. Song, J. Bae, and S. Y. Kim, *Advanced Theory of Signal Detection: Weak Signal Detection in Generalized Observations*, Springer-Verlag, 2002.

[8] I. Song, T. An, and J. Oh, *Near ML decoding method based on metric-first search and branch length threshold,* registration no. US 8018828 B2, Sep. 13, 2011, USA.

[9] H.-K. Min, T. An, S. Lee, and I. Song, "Non-intrusive appliance load monitoring with feature extraction from higher order moments," in *Proc. 6th IEEE Int. Conf. Service Oriented Computing, Appl.,* Kauai, HI, USA, pp. 348-350, Dec. 2013.

[10] I. Song and S. Lee, "Explicit formulae for product moments of multivariate Gaussian random variables," *Statistics, Probability Lett.,* vol. 100, pp. 27-34, May 2015.

사 사

언제나 저를 바른 길로 이끌어 주시는 송익호 교수님께 큰 고마움을 느낍니다. 끝으로 오늘의 제가 있을 수 있도록 사랑으로 키워 주신 가족들에게 감사드립니다. 저의 이 작은 결실이 그분들께 조금이나마 보답이 되기를 바랍니다.