KIPS Tr. Comp. and Comm. Sys.
Vol.6, No.12 pp.469~478 pISSN: 2287-5891

https://doi.org/10.3745/KTCCS.2017.6.12.469

A Performance Evaluation on Classic Mutual Exclusion Algorithms for Exploring Feasibility of Practical Application

Hyung-Bong Lee† · Ki-Hyeon Kwon††

ABSTRACT

The mutual exclusion is originally based on the theory of race condition prevention in symmetric multi-processor operating systems. But recently, due to the generalization of multi-core processors, its application range has been rapidly shifted to parallel processing application domain. POSIX thread, WIN32 thread, and Java thread, which are typical parallel processing application development environments, provide a unique mutual exclusion mechanism for each of them. Applications that are very sensitive to performance in these environments may want to reduce the burden of mutual exclusion, even at some cost, such as inconvenience of coding. In this study, we implement Dekker's and Peterson's algorithm in the form of busy-wait and processor-yield in various platforms, and compare the performance of them with the built-in mutual exclusion mechanisms to evaluate the usability of the classic algorithms. The analysis result shows that Dekker's algorithm of processor-yield type is superior to the built-in mechanisms in POSIX and WIN32 thread environments at least 2 times and up to 70 times, and confirms that the practicality of the algorithm is sufficient.

Keywords : Mutual Exclusion, Dekker's Algorithm, Peterson's Algorithm, POSIX Thread, WIN32 Thread, Java Monitor

실제 적용 타당성 탐색을 위한 고전적 상호배제 알고리즘 성능 평가

이 형 봉† · 권 기 현††

요 약

상호배제는 원래 다중처리기 시스템을 지원하는 운영체제의 경쟁상황 예방 이론에서 출발하였으나, 최근에는 다중코어처리기의 일반화로 그 적용범위가 급격하게 병렬처리 어플리케이션 영역으로 이동되었다. POSIX 스레드, WIN32 스레드, Java 스레드 등이 대표적인 병렬처리 어플리케이션 개발환경인데, 이들은 각자 고유한 상호배제 메커니즘을 제공하고 있다. 이들 환경에서 성능에 매우 민감한 어플리케이션들은 코딩의 불편함 등 약간의 희생을 감수하더라도 상호배제를 위한 부담 경감을 필요로 할 수 있다. 이 연구에서는 두 스레드 사이에서 Dekker와 Peterson 알고리즘을 플랫폼별로 바쁜 대기와 처리기 양보 형태로 구현하여 각각의 고유 상호배제 메커니즘들과의 성능을 비교하고, 그 알고리즘들의 유용성을 평가한다. 분석 결과 POSIX 및 WIN32 스레드 환경에서 처리기 양보 형 Dekker 알고리즘이 최소 2배에서 최대 70배까지 우수한 것으로 나타나 이 알고리즘의 실용성이 충분한 것으로 확인되었다.

키워드 : 상호배제, Dekker 알고리즘, Peterson 알고리즘, POSIX 스레드, WIN32 스레드, Java 모니터

1. 서 론

상호배제는 둘 이상의 프로세스가 공유 데이터에 접근하여 읽기·연산·쓰기를 비동기적으로 시도함으로써 나타나는 경쟁상황(race condition)이 존재하는 임계영역을 보호한다는 개념으로, 컴퓨터과학에서 이미 잘 알려진 고전적 주제이다. 애초의 상호배제 필요성은 다중처리기(병렬처리) 시스템의 운영체제 내에서 필연적으로 나타날 수밖에 없는 경쟁상황을 해결하려는 데에서 출발했기 때문에 오늘날에도 운영체제 분야에서 다루어지고 있다. 다중처리기 운영체제를 보호하기 위해서 처리기들을 주(master)·종(slave)으로 분리하여 주 처리기 하나에게만 운영체제 진입을 허용하기도 하지만[1], 확장성이나 신뢰성 측면에서 모든 처리기들이 동등한 자격으로 운영체제에 진입할 수 있는 대칭형

† 종신회원 : 강릉원주대학교 컴퓨터공학과 교수
†† 정 회 원 : 강원대학교 전자정보통신공학부 교수
Manuscript Received : May 29, 2017
Accepted : October 13, 2017
* Corresponding Author : Ki-Hyeon Kwon(kweon@kangwon.ac.kr)

다중처리(SMP: Symmetric Multiprocessing) 시스템 도입은 불가피하다. 특히, 오늘날에는 인텔의 다중코어처리기 보급의 일반화로 이를 탑재하고 있는 주변의 거의 모든 유닉스·리눅스·윈도우 운영체제가 SMP 모드로 운영 중이고, 어플리케이션들은 이러한 SMP의 특성에 힘입어 병렬처리에 의한 성능향상을 위해 다중스레드 기반 프로그래밍이 일상화되고 있다. 즉, 상호배제 개념은 더 이상 시스템 프로그래밍 수준에 머물지 않고 급격하게 일반 어플리케이션 프로그래밍 영역으로 확대된 것이다. 이런 현실에서 다중스레드 프로그래밍에 필수적인 상호배제 원리를 어려워하는 일선 교육기관의 대다수 소프트웨어 전공 학생들을 위한 특별한 교육 방법론[2]이 제안되기도 하고, 어플리케이션 수준에서 상호배제를 인식하지 않고 접근할 수 있는 원자적 연산 자료구조나 입·출력 동기화에 기반한 메시징 기법 등을 활용하는 이른바 잠금 없는(lockless or lock-free) 병렬 프로그래밍 기법[3, 4]이 소개되어 있기도 하다. 그러나 상호배제 기법을 완전히 도외시하는 병렬 프로그래밍은 상상할 수 없다. 두 프로세스 사이에서 특별한 하드웨어 기계명령어 지원을 받지 않고 공유 변수를 이용한 순수한 소프트웨어 상호배제 알고리즘이 Dekker에 의해 최초로 고안되었고[5], 이를 보다 간편하게 변형시킨 알고리즘이 Peterson에 의해 제안된 바 있는데[6], 이들 고전적 알고리즘은 교육 및 이론적 연구 차원에서 단편적으로 인용될 뿐이고 최근에는 주로 분산 모바일 환경 분야에서의 상호배제 방법론이나[7-9] 상호배제 알고리즘 자체의 증명방안[10, 11]에 관한 연구가 이루어지고 있다. 이 연구에서는 고전적 상호배제 알고리즘의 단순 구현을 다뤘던 연구[12]를 기반으로 성능 측면에서 이들 알고리즘의 실질적 활용 가치에 대한 탐색을 시도한다. 이를 위하여 POSIX 스레드, WIN32 스레드, 그리고 Java 스레드 환경에서 상호배제가 필요한 두 스레드를 동기화하는 이들 고전적 알고리즘의 상호배제 부담을 각 스레드 고유 상호배제 메커니즘들과 비교·분석하여 그 실용성을 검증한다.

이 논문은 총 5장으로 구성되어, 2장에서 두 알고리즘의 실제 구현을 위한 논점들을 살펴보고, 3장에서 실험 계획을 설계한다. 4장에서는 플랫폼별 성능 측정결과를 분석·평가한다. 그리고 5장의 결론으로 맺는다.

2. 고전적 상호배제 알고리즘 및 구현 논점

2.1 Dekker 알고리즘

Fig. 2는 Fig. 1의 Dekker 알고리즘[5]을 C언어 스타일의 의사코드 형태로 표현한 것인데 상호배제와 관련된 대부분의 문헌들이 이와 유사한 표현을 사용한다. 오늘날 크게 진전된 컴퓨팅 환경에서 이 알고리즘을 구현하기 위해서는 아래와 같이 몇 가지 논점에서 보완이 이루어져야 한다.

1) 컴파일러 상수전파(constant propagation) 최적화

공유변수에 대하여 다른 처리기가 임의 시점에 접근할 수 있다는 점을 고려하지 않는다면 Fig. 2의 ①, ② 부분은 굳이 메모리에서 turn 변수를 다시 읽을 필요가 없기 때문에 컴파일러는 성능향상을 위해 위에서 읽은 값 false를 그대로 사용하는 최적화를 시도할 것이므로 이 알고리즘은 실패한다. 이를 예방하기 위해서는 volatile 특성의 변수를 명시하거나 이러한 최적화를 거부해야 한다.

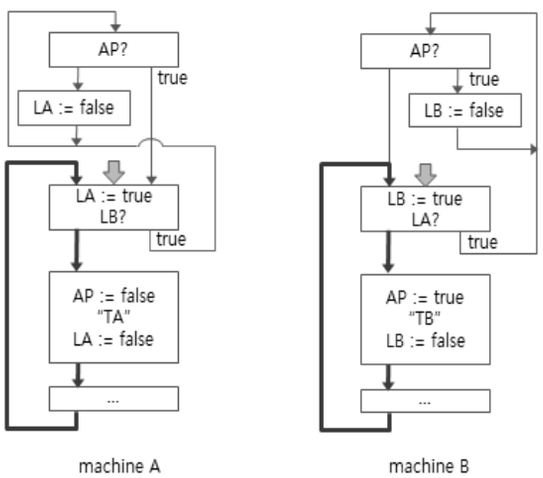


Fig. 1. Dekker's Algorithm

```
variables
  flag[2] : array of 2 booleans with initial value false
  turn    : integer with initial value 0 or 1

p0:                                  p1:
  do {                                 do {
③ flag[0]← true;                       flag[1] ← true;
④ while(flag[1]) {                     while(flag[0]) {
      if (turn ≠ 0) {                    if (turn ≠ 1) {
        flag[0] ← false;                   flag[1] ← false;
      ① while (turn ≠ 0)                 ② while (turn ≠ 1)
      ⑥     ;                                ;
⑤       flag[0] ← true;                    flag[1] ← true;
      }                                    }
    }                                    }
    turn ← 1;.                           turn ← 0;.
    ... // critical section              ... // critical section
    flag[0] = false;                     flag[0] = false;
    ... // other section                 ... // other section
  } while(p0_not_end);                 } while(p1_not_end);
```

Fig. 2. Pseudo Code for Dekker's Algorithm

2) 처리기 메모리 접근 재정렬(memory reordering)

고도로 진전된 현대 처리기들은 최종 결과가 동일하다는 전제 하에 기계명령어(인스트럭션)의 배치 순서와 관계없이 읽기(load)·쓰기(store)의 순서를 실행시간에 변경하여 메모리 뱅크 분리에 따른 버스 사용률 향상이나 캐시 정책의 효율화를 추구한다[13]. 이런 관점에서 Fig. 2의 ③(flag[0] 쓰기)과 ④(flag[1] 읽기)의 실행순서는 바뀔 수 있는데, 만약 다른 프로세스 p1이 없다면 그 결과는 동일하겠지만 두 처리기가 공유하는 경우에는 실패한다. 이를 예방하기 위해서는 실행시간 메모리접근 순서가 지켜져야 할 양쪽 기계 명령어 사이에 메모리 접근 재정렬을 금지하는 기계명령어 즉, 메모리 장벽(memory barrier 혹은 memory fence)를 삽입해야 한다. Fig. 2의 경우 ③과 ④, ④와 ⑤ 사이에 메모리 장벽이 각각 필요하다.

3) 단일처리기 시스템과 다중처리기 시스템

Fig. 1의 Dekker 알고리즘을 자세히 살펴보면 처리 주체를 machine A와 B로 명명하여 독립된 두 개의 처리기를 강하게 시사하고 있다. 오늘날의 시분할 운영체제는 처리기가 하나인 경우에도 여러 개의 프로세스에게 독립된 가상 처리기로 할당하기 때문에 상호배제 관점에서 Dekker 알고리즘은 여전히 유효하다. 그러나 처리기가 하나인 경우 임계영역에 진입한 p1이 선점되어 p0가 실행된다면 p0는 주어진 시간동안 ④와 ⑤ 사이에서의 불필요한 바쁜 대기가 불가피하여 현실적으로 적용이 불가능하다. 이런 상황은 반대의 경우에도 마찬가지다. 따라서 단일처리기 환경이라면 ①, ⑥의 바쁜 대기를 대신하는 처리기 양보가 필요하다.

2.2 Peterson 알고리즘

Fig. 3은 Dekker 알고리즘을 크게 간편화시킨 Peterson 알고리즘[6]이고, 이에 대한 C언어 스타일의 일반적인 의사코드는 그림 4와 같다. 이 알고리즘에서도 컴파일러 최적화, 처리기 메모리 접근 재정렬, 단일처리기 시스템과 다중처리기 시스템 관련 등의 논점은 Dekker 알고리즘과 동일하게 존재한다. 즉, Fig. 4에서 flag[]와 turn 변수에 대한 상수전파 최적화를 예방해야 하고, ①② 와 ③사이에는 메모리 접근 재정렬을 예방해야 한다. 또한 단일처리기 시스템에서는 ③의 바쁜 대기 대신 처리기를 양보해야 한다.

```
/* trying protocol for P1 */
Q1 := true;
TURN := 1;
wait until not Q2 or TURN=2;
Criticla Section;
/* exit protocol for  P1 */
Q1 := false;
```

```
/* trying protocol for P2 */
Q2 := true;
TURN := 2;
wait until not Q1 or TURN=1;
Criticla Section;
/* exit protocol for  P2 */
Q2 := false;
```

Fig. 3. Peterson's Algorithm

```
variables
 flag[2] : array of 2 booleans with initial value false
 turn    : integer with no initial value
```

```
p0:
do {
① flag[0]← true;
② turn ← 1;
③ while (flag[1] &&
                turn≠ 0) ;
  ... // critical section
  flag[0] = false;
  ... // other section
} while(p0_not_end);
```

```
p1:
do {
   flag[1]← true;
   turn ← 0;
   while (flag[0] &&
                turn≠ 1) ;
  ... // critical section
  flag[1] = false;
  ... // other section
} while(p1_not_end);
```

Fig. 4. Pseudo Code for Petersonr's Algorithm

2.3 고전적 상호배제 알고리즘의 구현

이 연구에서는 알고리즘 구현 언어로 C와 Java 두 가지를 사용하고, 그 각각에 대한 구현된 Dekker 알고리즘을 Fig. 5와 Fig. 6에 보였다. 이들 그림에서 CPU_YIELD()와 MEM_FENCE()는 필요치 않는 경우는 주석처리하고, 필요한 경우는 해당 플랫폼 별 고유 프로시저를 대입한다. Table 1에 이들에 대한 플랫폼 별 프로시저를 보였다. 그리고 Fig. 6의 자바 구현에서 Flag 변수를 배열 대신 개별 변수로 정의한 이유는 배열에 대한 volatile 한정이 배열 객체 자체에만 적용되고 배열 원소에는 적용되지 않기 때문이다. 이를 해결하는 방안으로 AtomicInteger나 AtomicIntegerArray 등 원자적 연산 타입을 활용할 수 있으나 여기서는 최대한 원래의 알고리즘 변수 형을 유지하기 위해 개별 변수를 활용하였다. 이 내용들은 Peterson 알고리즘에도 동일하게 적용된다.

Table 1. Procedures for Memory Fence and CPU Yield

System	MEM_FENCE()	CPU_YIELD()
Alpha, POSIX	not needed	sched_yield()
Intel, POSIX	asm("mfence")	
Sparc, POSIX	not needed	
Intel, WIN32	MemoryBarrier()	SwitchToThread()
JVM	not needed	Thread.yield()

```
volatile  int   Flag[2] = {0, 0,}, Turn = 0;
void Dekker_lock(int  me)
{
      Flag[me] = 1;
      MEM_FENCE();
      while (Flag[1-me]) {
            if (Turn != me) {
                  Flag[me] = 0;
                  while (Turn != me)
                        CPU_YIELD();
                  Flag[me] = 1;
                  MEM_FENCE();
            }
      }
}
void Dekker_unlock(int me)
{
      Turn = 1-me, Flag[me] = 0;
}
```

Fig. 5. Dekker's Algorithm in C Language

```
class Dekker {
      volatile boolean Flag0, Flag1;
      volatile int        Turn;
      Dekker() {Flag0 = Flag1 = false; Turn = 0; }
      void lock(int  me) {
            if (me == 0) this.lock0();
            else           this.lock1();
      }
      void unlock(int  me) {
            if (me == 0) { Turn = 1; Flag0 = false; }
            else           { Turn = 0; Flag1 = false; }
      }
      void lock0 () {
            Flag0 = true; MEM_FENCE();
            while(Flag1) {
                if (Turn != 0) {
                      Flag0 = false;
                      while(Turn != 0) CPU_YIELD;
                      Flag0 = true; MEM_FENCE();
                }
            }
      }
      void lock1() {
           Flag1 = true;
           while(Flag0) {
                if (Turn != 1) {
                    Flag1 = false;
                    while(Turn != 1) CPU_YIELD();
                     Flag1 = true;
                }
           }
      }
}
```

Fig. 6. Dekker's Algorithm in Java Language

3. 실험 계획 설계

고전적 상호배제 알고리즘들의 성능평가 결과에 대한 보편성을 확보하기 위해서는 가급적 다양한 플랫폼에서 성능 측정 및 분석이 이루어져야 한다.

3.1 상호배제 어플리케이션 및 측정 성능

상호배제가 요구되는 어플리케이션으로 Fig. 7과 같이 공유변수 Count에 주어진 수를 주어진 횟수만큼 덧셈을 반복하는 스레드 add와 동일한 방법으로 뺄셈을 반복하는 스레드 sub의 시작과 종료시간 차를 1/100초 정밀도로 측정한다. 종료 시점에서 Count의 최종 값은 초기 값 0과 같아야 함은 물론이다. 이 때 반복 횟수는 10,000,000단위로 10,000,000에서 90,000,000까지(식별부호 "10M" ~ "90M") 변화를 주며 9번 측정한다.

```
int          Count = 0;
int do_something(int  i)
{
      int  q = i / 2;
      if (q * 2 == i)  return 2;
      else               return 1;
}
```

```
void add(int  n)
{
  int    i;
  for (i = 0; i < n; i++) {
    v = do_something(i);
    MUTEX_LOCK(0);
    Count += v;
    MUTEX_UNLOCK(0);
  }
}
```

```
void sub(int  n)
{
  int  i;
  for (i = 0; i < n; i++) {
    v = do_something(i);
    MUTEX_LOCK(1);
    Count -= v;
    MUTEX_UNLOCK(1);
  }
}
```

Fig. 7. Sample Application for Performance Evaluation

3.2 상호배제 비교 메커니즘 그룹 설정

이 연구의 관심은 시스템 간의 성능비교가 아닌 어느 한 시스템 내에서의 여러 가지 상호배제 메커니즘 간의 성능비교에 있다. 따라서 비교 메커니즘 그룹을 Table 2와 같이 운영체제에 따라 세 가지 유형으로 분류한다. 이 표에서 Spin과 Yield는 각각 해당 알고리즘의 바쁜 대기와 처리기 양보를 의미한다. Table 3에는 실험에 사용된 운영체제별 하드웨어 플랫폼 목록을 보였다.

Table 2. Classification of Mutual Exclusion Mechanism Group based on Operating System

Operating System	Mechanism Group	Notation
Unix, Linux, MacOS (POSIX thread)	• POSIX Mutex, • Dekker Spin/Yield • Peterson Spin/Yield	PM DS/DY PS/PY
Window (WIN32 thread)	• Win32 Mutex, • Dekker Spin/Yield • Peterson Spin/Yield	WM DS/DY PS/PY
JVM (Java thread)	• Java Monitor • Dekker Spin/Yield • Peterson Spin/Yield	JM DS/DY PS/PY

Table 3. Hardware Platforms for Operating Systems

Operating System	Version	CPU	Core (bit)	Notation
Unix	DEC OSF1	Alpha 21264	1(64)	PD
	SUN Solaris2	Sparc V9	2(64)	PS
Linux	Ubunto 14.04LTS	I7-4770	4(64)	PL
MacOS	Darwin 14.5.0	ZeonW3860	6(64)	PM
Window	Window7	E7500(TG)	2(32)	W1
		Q9300(Offc)	4(32)	W2
		E7200(Trad)	2(64)	W3
		I7-260(Wrd)	4(64)	W4
JVM	PD, PL, PM, W1, W3(selected five platforms)			

4. 상호배제 메커니즘의 성능 측정 및 분석

성능측정 결과는 Table 2와 같이 POSIX 스레드, Window 스레드, Java 스레드 등 크게 세 가지 환경으로 분류하여 분석한다.

4.1 POSIX 스레드 환경

PD~PL(DEC OSF1, SUN Solaris, Ubunto, Darwin) 네 플랫폼에서 고전적 상호배제 알고리즘 DS~PY(Dekker와 Peterson의 바쁜 대기 및 CPU 양보 형 알고리즘)에 대한 성능 측정 결과를 Fig. 8에 보였는데, PD 플랫폼은 처리기가 하나이기 때문에 DS, PS 등 바쁜 대기 형 알고리즘을 적용할 수 없었다. 이 그림으로부터 플랫폼과 무관하게 DY 알고리즘이 가장 안정적인 성능을 유지하고 있음을 발견할 수 있다.

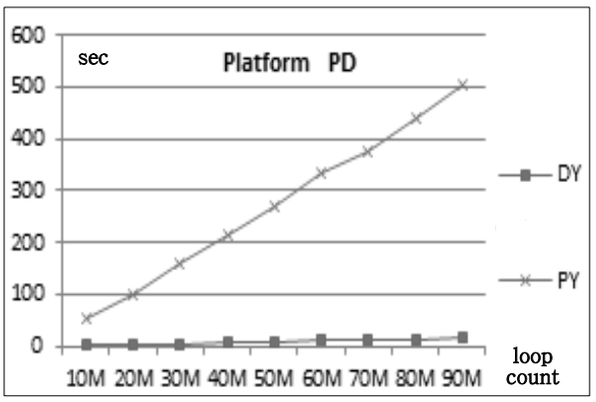
Fig. 9는 위와 동일한 플랫폼에서의 POSIX 뮤텍스와 DY 알고리즘 간의 성능 비교 결과인데 큰 차이가 보이지 않는 리눅스 플랫폼을 제외하고는 DY 알고리즘이 2~70배의 월등한 우수성을 보이고 있다. 리눅스 운영체제의 경우 대체로 DY 알고리즘이 우수하나, 처리기 스케줄링의 특이 상황이 일어나는 경우 DY 알고리즘이 불리할 수 있는 것으로 분석된다.

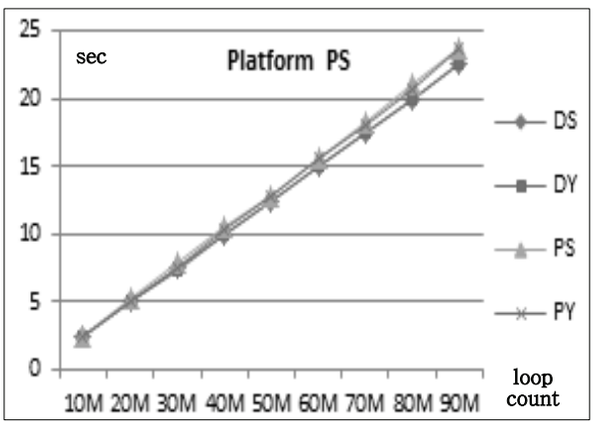
4.2 WIN32 스레드 환경

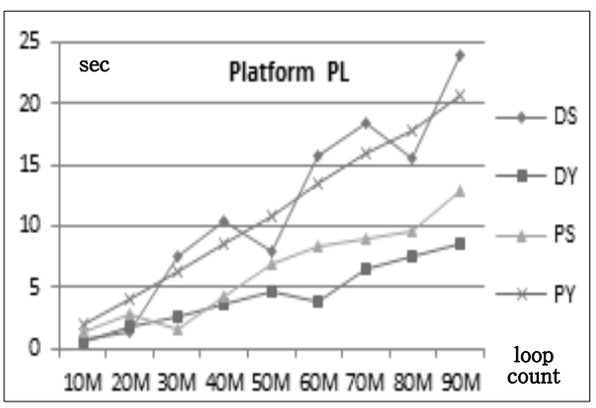
Fig. 10은 W1~W4(Window 7, 2/4 Core, 32/64bit)의 네 플랫폼에서 POSIX 스레드 환경에서 적용했던 고전적 상호배제 알고리즘 DS~PY의 성능을 WIN32 스레드 환경으로 측정 결과이다. 이 결과로부터 알고리즘 DY를 대표적 고전적 상호배제 알고리즘으로 채택하고, WIN32 뮤텍스 성능과 비교한 결과를 Fig. 11에 보였는데 여기서도 DY 알고리즘의 성능이 10~20배 정도 우수하다.

4.3 Java 스레드 환경

JVM 호스트 플랫폼으로 PD, PL, PM, W1, W2을 선정하고, 그 각각에서 DS~PY의 네 가지 알고리즘에 대한 성능 측정 결과를 Fig. 12에 보였다. 단, PD 플랫폼에서는 처리기가 하나이므로 바쁜 대기형인 DS와 PS 알고리즘을 제외시켰다. 이 그림으로부터 Java 스레드 환경에서도 DY를 대표 알고리즘으로 설정하는데 무리가 없음을 알 수 있다. 이에 따라 동일 플랫폼에서 Java 모니터와 DY 알고리즘의 성능을 비교한 결과를 Fig. 13에 보였는데, Java 스레드 환경에서는 POSIX 및 WIN32 스레드와는 달리 플랫폼과 무관한 DY 알고리즘의 일관된 우수성을 발견할 수 없는 것으로 나타났다.







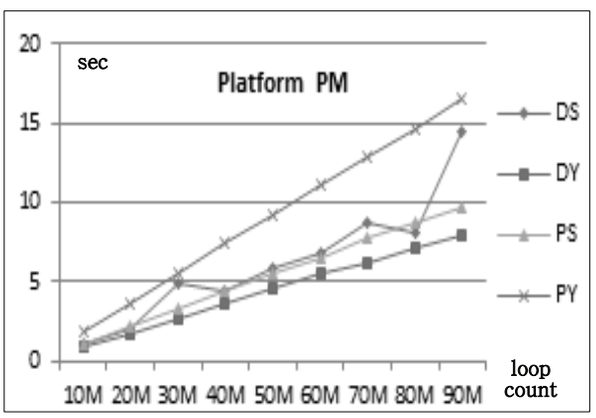
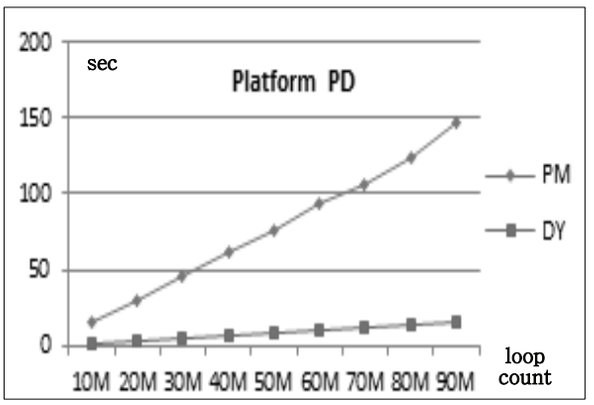
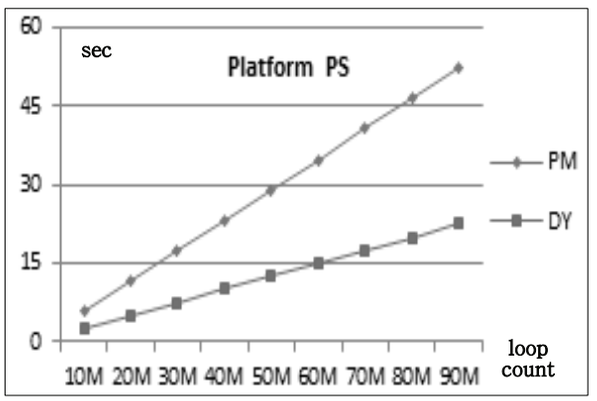
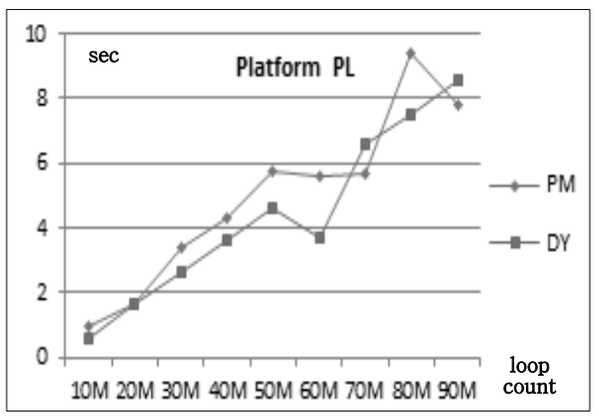


Fig. 8. Performance of Classic Mutual Exclusion Algorithms in POSIX Thread Environment







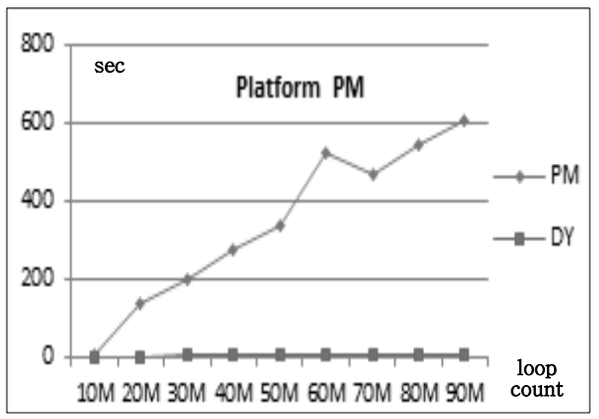
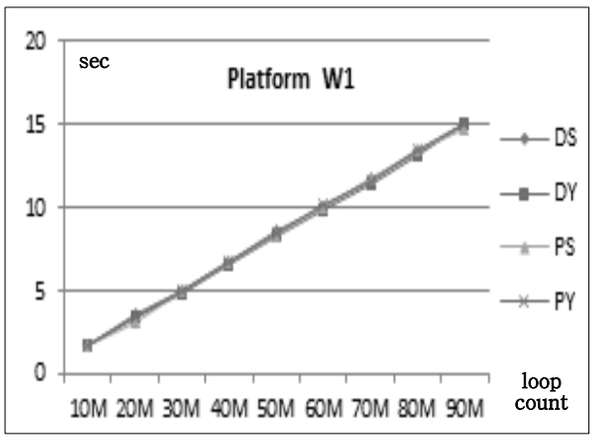
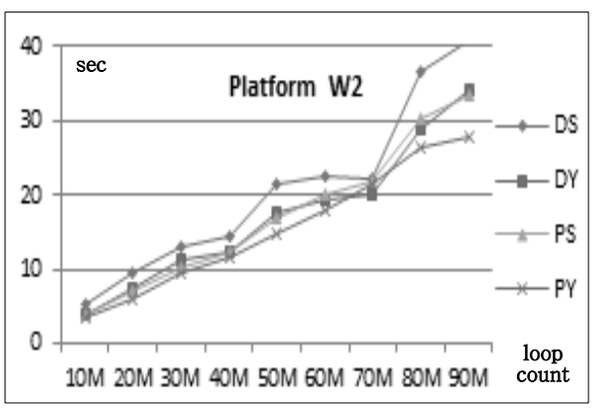
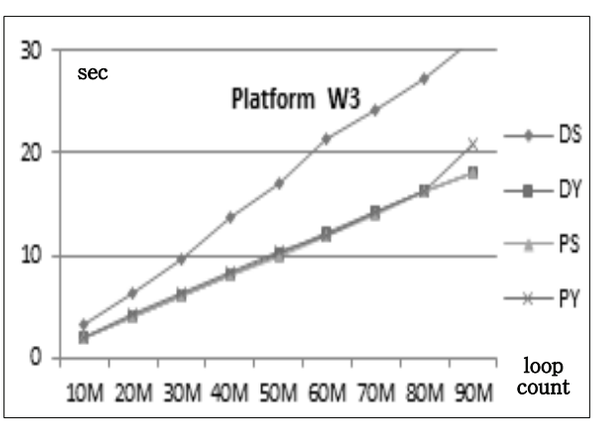


Fig. 9. Performance Comparison Between POSIX Mutex and Dekker's Algorithm







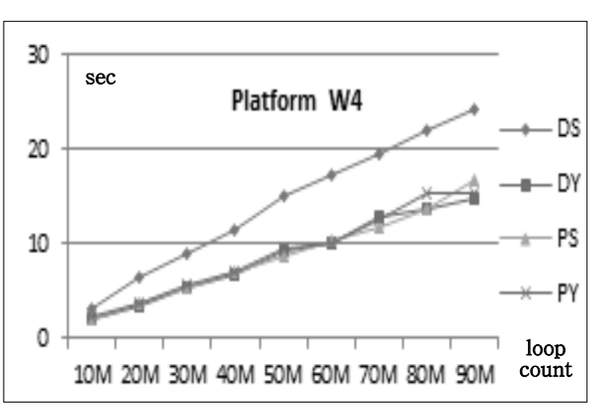
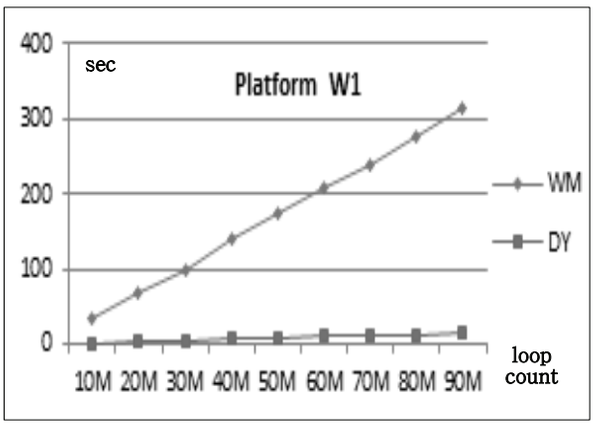
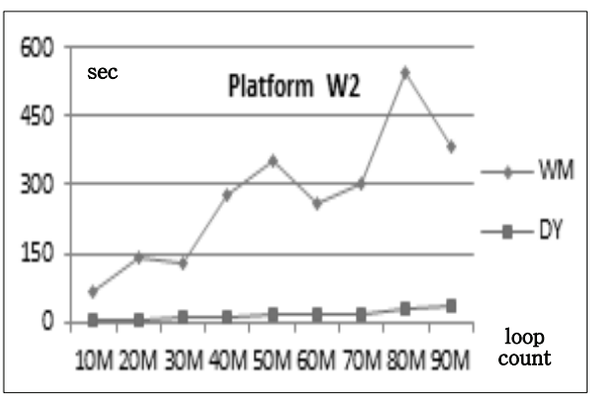
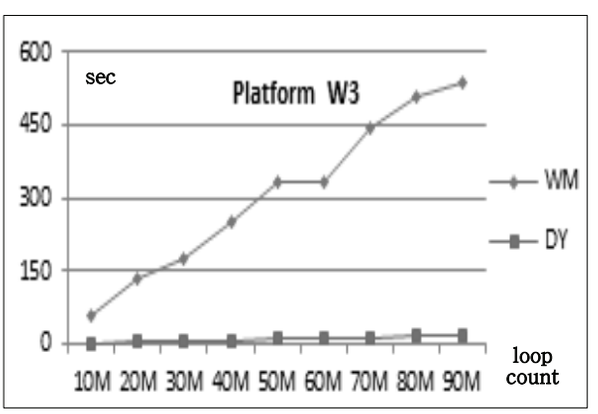


Fig. 10. Performance of Classic Mutual Exclusion Algorithms in WIN32 Thread Environment







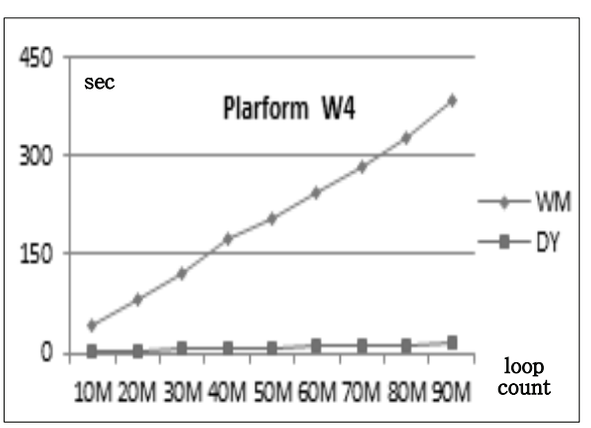


Fig. 11. Performance Comparison Between WIN32 Mutex and Dekker's Algorithm

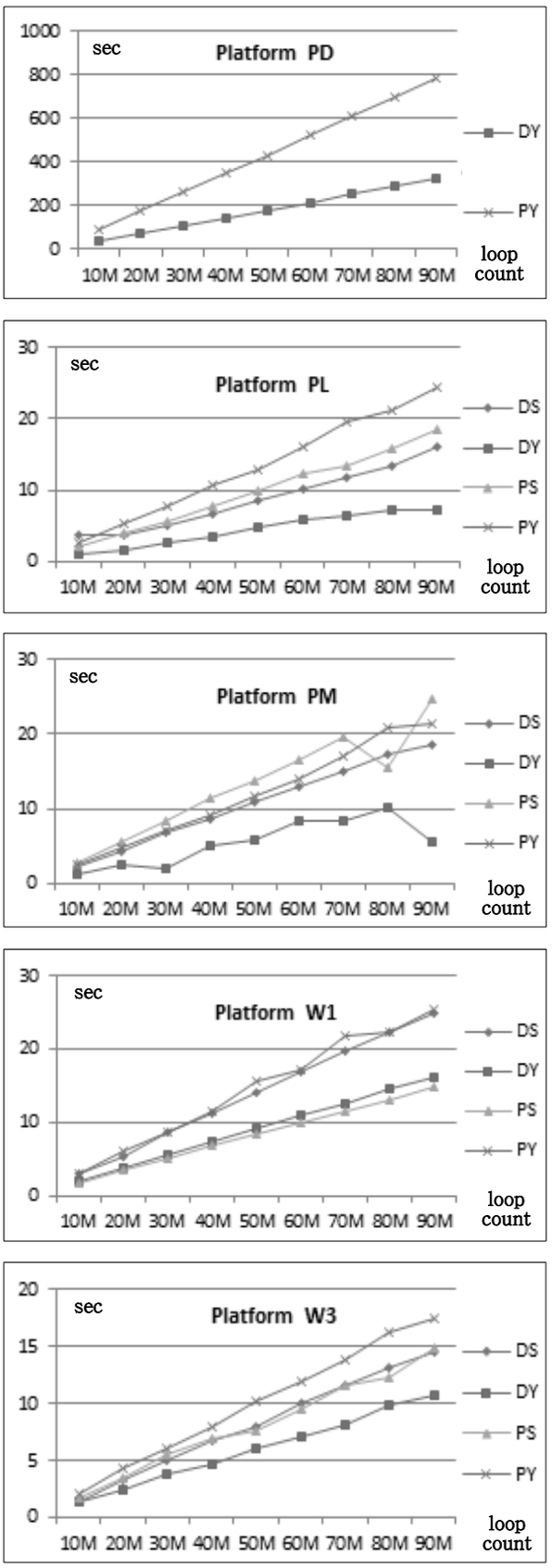


Fig. 12. Performance of Cassic Mutual Eclusion Algorithms in Java Thread Environment

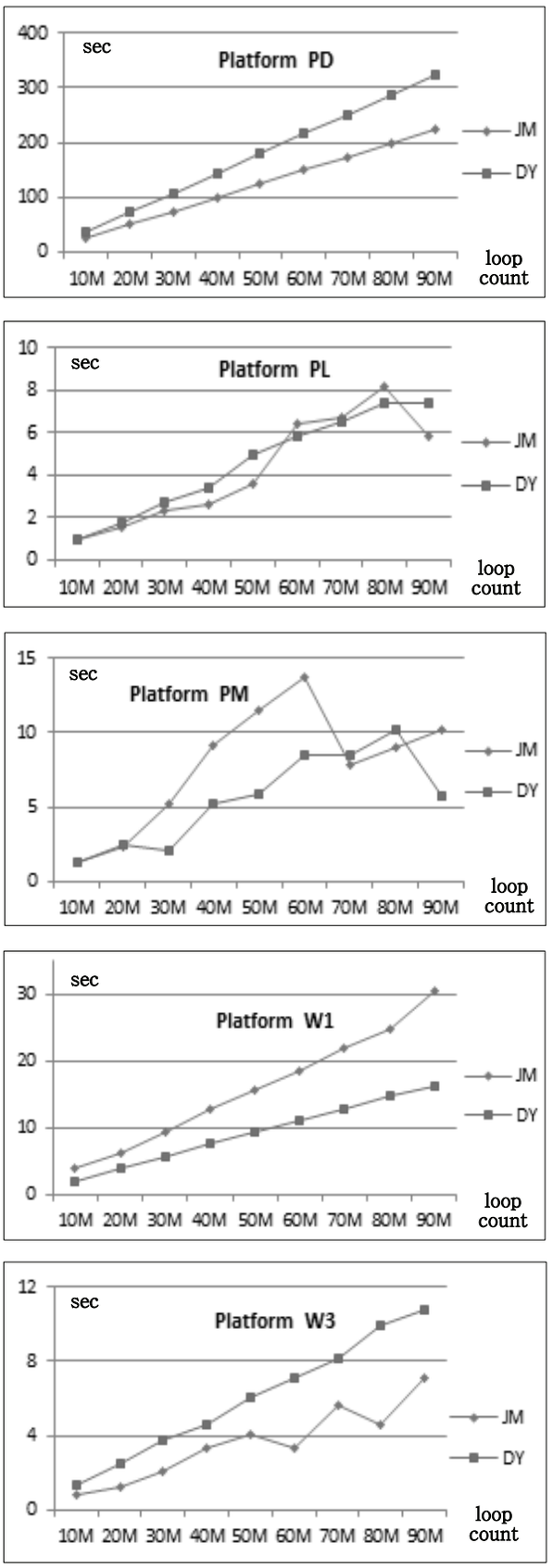


Fig. 13. Performance Comparison Between Java Monitor and Dekker's Algorithm

4.4 플랫폼 별 성능 분석

유닉스 계열의 성능평가 결과인 Fig. 9를 플랫폼 별 성능 관점에서 분석해보면 처리기가 네 개인 PL, 두 개인 PS, 하나인 PD 순으로 DY 알고리즘의 성능이 우수함을 확인할 수 있지만, 처리기 클록 속도, 메모리 크기, 다중 프로세스 정도 등이 모두 다르기 때문에 이 결과가 플랫폼 별 절대적 비교 결과라고 단정할 수는 없다. 실제로, 윈도우 계열의 성능 평가 결과인 Fig. 11에서는 처리기가 두 개인 W2에서의 성능이 하나인 W1에서의 성능보다 낮게 측정되었다.

5. 결 론

지금까지는 Dekker 알고리즘으로 대표되는 순수 소프트웨어에 기반한 고전적 상호배제 알고리즘들이 실제 적용을 위한 실용적 관점보다는 교육 및 이론 연구차원에서 다루어져 왔다. 그러나 이 연구의 실험결과 비록 두 스레드 간이기는 하지만 이 알고리즘들이 POSIX와 WIN32 스레드 환경에서 각각의 고유 상호배제 메커니즘보다 최소 2배에서 최대 70배 이상의 월등한 성능을 보이고, Java 스레드에서는 플랫폼에 따라 2배 정도의 우수성을 보이는 것으로 측정되었다. 이 실험에서 어플리케이션 처리 시간이 거의 없도록 설계했기 때문에 이 결과는 순수한 상호배제 부담에 대한 측정으로 볼 수 있고, 그 원인은 시스템 제공 상호배제 메커니즘이 운영체제 호출에 따른 부담을 동반하는 반면 소프트웨어 상호배제는 사용자 영역에서 이루어져 운영체제 호출 부담이 없기 때문인 것으로 분석된다. 즉, 성능에 민감한 어떠한 어플리케이션이라도 적용을 고려해볼만한 가치가 충분히 있는 것으로 나타난 것이다. 다만, 성능 극대화를 위해 실행시간 메모리 접근 재정렬 기능이 내장된 인텔 등 현대 처리기 를 탑재한 플랫폼에서는 스레드 간 순서처리 일관성(sequential consistency) 유지를 위한 보완장치를 어플리케이션 개발자가 직접 고안해야 하는 부담은 감수해야 한다. 이 연구가 고전적 상호배제 알고리즘이 가지고 있는 실용적 가치를 재평가하는 기회와 이들 알고리즘에 대한 교육이 단순한 이론적 접근보다는 실질적 구현 및 활용 중심으로 변화하는 계기가 될 것으로 기대한다.

References

[1] A. Silberschatz, P. B. Galvin, and G. Gagne, “Operating System Concepts,” 9th Ed., pp.278-280, Wiley, 2013.

[2] Young-Suk Lee, Young-Ho Nam, “A Study on Instruction Method for Mutual Exclusion Using Simulation Based on Graphic,” *The Journal of Korean Association of Computer Education*, Vol.13, No.6, pp.9-21, 2010.

[3] Microsoft, “Lockless Programming Considerations for Xbox 360 and Microsoft Windows,” [Internet], https://msdn.microsoft.com/en-us/library/windows/desktop/ee418650(v=vs.85).aspx (accessed at May 15th, 2017).

[4] Geoff Langdale, “Lock-Free Programming,” [Internet], https://www.cs.cmu.edu/~410-s05/lectures/L31_LockFree.pdf (accessed at May 15th, 2017).

[5] E. W. Dijkstra, “About the sequentiality of process descriptions (EWD-35),” E.W. Dijkstra Archive. Center for American History, University of Texas at Austin, 1962~1963 (http://www.cs.utexas.edu/users/EWD/translations/EWD35-English.html, accessed at May 15th, 2017).

[6] G. L. Peterson, “Myths About the Mutual Exclusion Problem,” *Information Processing Letters*, Vol.12, No.3, pp.115-116, 1981. (http://cs.nyu.edu/~lerner/spring12/Read03- Mutual Exclusion.pdf, accessed at May 15th, 2017).

[7] S. H. Park and S. H. Lee, “Token-Based Mutual Exclusion Algorithm in Mobile Cellular Networks,” *Proc. of 26th International Conference on Advanced Information Networking and Applications Workshops*, pp.460-465, 2012.

[8] S. H. Park, Y. M. Kim, and S. C. Yoo, “A Design of Mutual Exclusion Protocol for Mobile Game in Cellular Wireless Networks,” *Journal of the Korea Entertainment Industry Association*, Vol.10, No.6, pp.431-438, 2016.

[9] A. Luo, W. Wu, J. Cao, and M. Raynal, “A Generalized Mutual Exclusion Problem and Its Algorithm,” *Proc. of 42nd International Conference on Parallel Processing(ICPP)*, IEEE, pp.300-309, 2013.

[10] F. Cicirelli and L. Nigro, “Modelling and Verification of Mutual Exclusion Algorithms,” *Proc. of IEEE/ACM 20th International Symposium on Distributed Simulation and Real Time Applications(DS-RT)*, pp. 136-144, 2016.

[11] X. Ji and L. Song, “Mutual Exclusion Verification of Peterson’s Solution in Isabelle/HOL,” *Proc. of third International Conference on Trustworthy Systems and Their Applications(TSA)*, IEEE, pp.81-86, 2016.

[12] S. M. Choi and H. B. Lee, “A Study on Mutual Exclusion Algorithms,” *Proc. of the KIPS Fall Conference 2016*, Vol. 23, No.2, pp.38-39, 2016.

[13] H. W. Cain and M. H. Lipasti, “Memory Ordering: A Value-Based Approach,” *Proc. of 1st Annual International Symposium on Computer Architecture(ISCA)*, 2004.

이 형 봉

http://orcid.org/0000-0003-4944-5265
e-mail : hblee@gwnu.ac.kr
1984년 서울대학교 계산통계학과(학사)
1986년 서울대학교 계산통계학과(석사)
2002년 강원대학교 컴퓨터과학과(박사)
1986년~1993년 LG전자 컴퓨터연구소 선임
1994년~1998년 한국디지탈(DEC Korea) 책임
1999년~2003년 호남대학교 정보통신공학부 조교수
2004년~현 재 강릉원주대학교 컴퓨터공학과 교수
관심분야 : 임베디드 시스템, 센서 네트워크, 데이터마이닝 알고리즘

권 기 현

http://orcid.org/0000-0001-8711-7049
e-mail : kweon@kangwon.ac.kr
1993년 강원대학교 전자계산학과(학사)
1995년 강원대학교 전자계산학과(석사)
2000년 강원대학교 컴퓨터과학과(박사)
1998년~2002년 동원대학 인터넷정보과 교수
2002년~현 재 강원대학교 전자정보통신공학부 교수
관심분야 : 패턴인식, 미들웨어, 임베디드 소프트웨어