[분산시스템] HW3 보고서

2017-26932 컴퓨터공학부 김형모

1. Test-and-Set

BTS Instruction을 이용해서 다음과 같이 work thread를 구현하였다.
(Bit Test and Set)

```
void * work_TAS(void * vargp) {
  __asm__ __volatile__(
    "L_main%=:" ENDL

    "L_aquire%=:" ENDL
    LOCK_PREFIX "bts $0, %0" ENDL
    "jc L_aquire%=" ENDL

    "movl %1, %%eax" ENDL
    "cmpl %2, %%eax" ENDL
    "jge L_release%=" ENDL
    "incl %1" ENDL

    "L_release%=:" ENDL
    "movl $0, %0" ENDL

    "movl %1, %%eax" ENDL
    "cmpl %2, %%eax" ENDL
    "jl L_main%=" ENDL
    : "+m"(lock), "+m"(mem), "+m"(count)
  );

  return NULL;
}
```

BTS $0, lock을 실행하면, lock[0] bit의 이전 값을 CF에 할당한 후 lock[0] bit에 1을 할당하게 된다. 여기서 lock[0] == 0은 어느 thread도 lock을 가지지 않은 상태를 의미하고, lock[0] == 1은 어떤 thread가 lock을 가지고 있는 상태를 의미한다. **BTS**에 lock prefix를 설정하면, 맨 처음에 **BTS** $0, lock을 실행한 thread만이 CF에 0이 할당되므로 Critical section으로 진행하게 된다. CF에 1이 할당된 thread들은 spin lock을 돌면서 **BTS**를 수행한다.

2. Test-and-Test-and-Set

BT 및 **BTS** Instruction으로 다음과 같이 work thread를 구성하였다.
(Bit Test, Bit Test and Set)

```
void * work_TTAS(void * vargp) {
  __asm__ __volatile__(
    "L_main%=:" ENDL

    "L_aquire%=:" ENDL
    "L_block%=:" ENDL
    "bt $0, %0" ENDL
    "jc L_block%=" ENDL
    LOCK_PREFIX "bts $0, %0" ENDL
    "jnc L_enter%=" ENDL
    "jmp L_aquire%=" ENDL

    "L_enter%=:" ENDL
    "movl %1, %%eax" ENDL
    "cmpl %2, %%eax" ENDL
    "jge L_release%=" ENDL
    "incl %1" ENDL

    "L_release%=:" ENDL
    "movl $0, %0" ENDL

    "movl %1, %%eax" ENDL
    "cmpl %2, %%eax" ENDL
    "jl L_main%=" ENDL
    : "+m"(lock), "+m"(mem), "+m"(count)
  );

  return NULL;
}
```

TAS에서와 마찬가지로 lock[0] == 0은 어느 thread도 lock을 가지지 않은 상태를 의미하고, lock[0] == 1은 어떤 thread가 lock을 가지고 있는 상태를 의미한다. **TAS**와의 차이점은 spin lock을 돌 때 **BTS**를 수행하는 것이 아니라 **BT**를 수행한다. lock[0] == 0을 확인하고 spin lock을 빠져나온 thread들끼리만 **BTS**를 수행해서 경쟁하여 lock을 획득한다.

3. Compare-and-Exchange

CMPXCHG Instruction으로 다음과 같이 work thread를 구성하였다. (Compare and Exchange)

```
void * work_CAS(void * vargp) {
  __asm__ __volatile__(
    "L_loop%=:" ENDL
    "movl %0, %%eax" ENDL
    "cmpl %1, %%eax" ENDL
    "jge L_done%=" ENDL

    "movl %%eax, %%edx" ENDL
    "incl %%edx" ENDL
    LOCK_PREFIX "cmpxchgl %%edx, %0" ENDL
    "jmp L_loop%=" ENDL
    "L_done%=:" ENDL
    : "+m"(mem), "+m"(count)
  );

  return NULL;
}
```

경쟁에 참여하는 각 thread는 mem을 load한 후, mem < COUNT이면 **CMPXCHG**로 mem ← mem+1을 시도한다. 어떤 thread가 update에 성공하게 되면 반드시 다른 thread는 실패하게 되므로 data race가 일어나지 않는다. 반드시 한 thread는 **CMPXCHG**를 통해 update에 성공하므로 lock-free이지만, starvation이 발생할 수 있으므로 wait-free는 아니다.

4. Fetch-and-Add

XADD Instruction으로 다음과 같이 work thread를 구성하였다.
(Exchange and Add)

```
void * work_FAA(void * vargp) {
  __asm__ __volatile__(
    "L_main%=:" ENDL
    "movl $1, %%eax" ENDL
    LOCK_PREFIX "xaddl %%eax, %0" ENDL

    "L_aquire%=:" ENDL
    "cmpl %%eax, %1" ENDL
    "jne L_aquire%=" ENDL

    "movl %2, %%eax" ENDL
    "cmpl %3, %%eax" ENDL
    "jge L_release%=" ENDL
    "incl %2" ENDL

    "L_release%=:" ENDL
    "addl $1, %1" ENDL

    "movl %2, %%eax" ENDL
    "cmpl %3, %%eax" ENDL
    "jl L_main%=" ENDL
    : "+m"(stamp), "+m"(lock), "+m"(mem), "+m"(count)
  );

  return NULL;
}
```

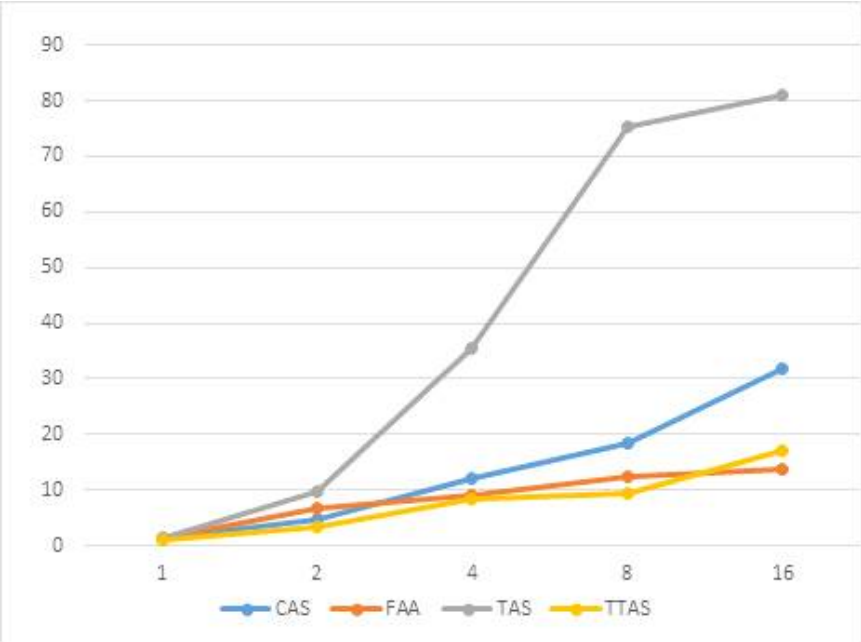
mem을 update하고자하는 thread는 lock prefix가 있는 **XADD**(Get-and-Add와 동일함)를 사용하여 stamp를 얻는다. 따라서 각 stamp는 고유하다는 것이 보장된다. 경쟁하는 thread들은 spin lock을 돌면서 **CMP**를 수행하는데, stamp == lock인 하나의 thread만이 Critical section으로 진입할 수 있다. 이 thread가 Critical section을 빠져나가면서 lock을 1 증가시키고, 다음 stamp를 가지고 있는 하나의 thread가 Critical section에 진입하게 된다.

5. Evaluation

다음은 thread의 개수를 1, 2, 4, 8, 16개로 증가시켜가며 각 work thread들이 shared memory에 1억 번 연산하는데 걸리는 시간(초)을 표로 나타낸 것이다.

thread #	TAS	TTAS	CAS	FAA
1	1.2	1	1.2	1.4
2	9.8	3.4	4.6	6.6
4	35.4	8.4	12	9
8	75.4	9.4	18.4	12.4
16	81	17.2	31.8	13.8

다음은 표의 내용을 꺾은선 그래프로 나타낸 것이다.



그래프로부터 **TAS**의 성능은 thread가 증가함에 따라 급격히 떨어진다는 것을 확인할 수 있다. 이는 spin lock 내부의 **BTS**가 memory에 read와 write를 동시에 해서 cache coherence를 보장하기 위해 cache line invalidation이 빈번하게 일어나기 때문이다.

TTAS는 **TAS**와 알고리즘은 동일하지만 spin lock 내부에서 **BTS** 대신 **BT**를 이용하기 때문에 memory를 read만 하므로 cache line invalidation이 줄어들어 성능이 훨씬 좋다.

CAS는 lock-free이지만 **CMPXCHG**가 memory/register에 read, write를 하는 등 Instruction 자체의 overhead가 크다. 실험 결과 **TAS**보다는 성능이 좋은 것으로 나타났지만, thread가 4개를 넘어가면 **TTAS**나 **FAA**보다 성능이 나빴다.

thread가 16개일 때의 성능이 가장 좋았던 **FAA**는 stamp를 얻는 XADD에서 blocking이 없고, spin lock에서는 RMW나 Atomic Instruction이 아니고 memory를 read만 하는 **CMP**를 수행하기 때문에, thread가 많은 환경에서의 성능 저하가 적은 편이었다.