Distributed System Homework1 Report

2017-26932 김형모

2018년 3월 14일

1 Server and CPU specs

컴퓨터공학부 숙제 서버인 `martini.snu.ac.kr`의 사양을 조사하였다.

OS: Ubuntu 16.04.2 LTS (GNU/Linux 4.8.0-39-generic `x86_64`)

CPU: Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz (8 cores)

Cache: L1d: 32KB, L1i: 32KB, L2: 256KB, L3: 20MiB

Details: `martini.snu.ac.kr`에 장착된 CPU는 `ia64` ISA를 지원하는 Intel Xeon E5-2630 v3로, 하스웰 마이크로아키텍쳐의 서버용 CPU이다. 물리 코어의 수는 8개이고 하이퍼 스레딩에 의한 논리 스레드 수는 16개이다. 8개 코어가 모두 한 소켓에 모여 있어 NUMA 노드는 1개이다. L1 캐시는 `dCache`와 `iCache`가 각각 32KB, L2 캐시는 256KB, L3 캐시는 20MiB이다. 벡터 명령어 확장은 `AVX2.0`까지 지원한다.

CPU flags: `fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm epb tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid cqm xsaveopt cqm_llc cqm_occup_llc dtherm ida arat pln pts`

2 Synchronization Operations

대표적인 Synchronization Operation에는 `Test-and-Set`, `Compare-and-Swap`, `Fetch-and-Add`, `Memory barrier` 등이 있다. 본 과제에서 이용한 멀티코어 숙제 서버인 `martini.snu.ac.kr`에서는 `ia64` ISA를 지원하는데, `Test-and-Set`에 해당하는 것으로는 **BTS** Instruction이, `Compare-and-Swap`에 해당하는 것으로는 **CMPXCHG** Instruction이, `Fetch-and-Add`에 해당하는 것으로는 **LOCK** Prefix를 붙인 **ADD**, **XADD** Instruction이 있으며, `Memory barrier`에 해당하는 것으로는 **LFENCE**, **SFENCE**, **MFENCE** Instruction이 있다.

3 Sequential code

본 과제는 각 thread가 공유하는 32-bit 메모리 변수 `mem`에 1을 10억번 더하는 연산을 수행하는 것이다. 이를 1개의 코어에서 실행하는 Serial code는 다음과 같다. 이때, 수행 시간은 1초였다.

```
1 while (mem < COUNT) {
2   mem += 1;
3 }
```

4 Test-and-Set

`Test-and-Set`에 해당하는 `ia64` Instruction은 **BTS**이다. 이를 이용하여 다음과 같이 work thread를 구현하였다.

```
1  L_loop5:
2    lock; bts $0, lock(%rip)
3    jc L_loop5
4    movl mem(%rip), %eax
5    cmpl count(%rip), %eax
6    jge L_done5
7    incl mem(%rip)
8    movl $0, lock(%rip)
9    jmp L_loop5
10 L_done5:
11   movl $0, lock(%rip)
```

각 thread는 `lock`이 0이면 1로 바꾼 다음 Critical section으로 진행하고, 이미 1이면 처음으로 돌아가 재시도한다. **BTS** Instruction에 **LOCK** prefix를 붙여 `lock`을 atomic하게 접근하도록 했다. 이때, 수행 시간은 573초였다.

5 Compare-and-Swap

`Compare-and-Swap`에 해당하는 `ia64` Instruction은 **CMPXCHG**이다. 이를 이용하여 다음과 같이 work thread를 구현하였다.

```
1 L_loop15:
2   movl mem(%rip), %eax
3   cmpl count(%rip), %eax
4   jge L_done15
5   movl %eax, %edx
6   incl %edx
7   lock; cmpxchgl %edx, mem(%rip)
8   jz L_loop15
9 L_done15:
```

각 thread는 `mem`을 load한 후, `mem`의 기존 값에 변화가 없으면 그보다 1만큼 큰 값으로 교체한다. 만약 다른 thread에 의해 `mem`의 값이 이미 변했을 경우, 처음부터 재시도하게 된다. **CMPXCHG** Instruction에 **LOCK** prefix를 붙여 `mem` 값을 비교하고 교체하는 과정을 atomic하게 했다. 이때, 수행 시간은 11초였다.

6 Fetch-and-Add

`Fetch-and-Add`에 해당하는 `ia64` Instruction은 **LOCK** prefix를 붙인 **ADD**이다. 이를 이용하여 다음과 같이 work thread를 구현하였다.

```
1   sarl    $31, %edi
2   notl    %edi
3   addl    $125000001, %edi
4   movl %edi, %edx
5 L_loop29:
6   cmpl $0, %edx
7   jle L_done29
8   decl %edx
9   lock; addl $1, mem(%rip)
```

```
10    jmp L_loop29
11 L_done29:
```

각 thread는 자신의 thread-id로부터 자신이 더해야 할 횟수를 static하게 계산하고, 그만큼 `mem`에 1을 더하는 과정을 반복한다. **LOCK** prefix에 의해 `mem`에 1을 더한 값을 store하는 과정이 atomic하게 일어나게 된다. 이때, 수행 시간은 24초였다.

7 Memory Barrier and Store Barrier

`Memory Barrier`에 해당하는 `ia64` Instruction은 **MFENCE**이고, `Store Barrier`에 해당하는 `ia64` Instruction은 **SFENCE**이다. 이를 이용하여 다음과 같이 **MFENCE**를 이용한 work thread를 구현하였다.

```
1  L_loop40:
2    mfence
3    movl mem(%rip), %ecx
4    cmpl count(%rip), %ecx
5    jge L_done40
6    movl %ecx, %eax
7    cltd
8    idivl tnum(%rip)
9    cmpl %edi, %edx
10   jne L_loop40
11   incl mem(%rip)
12   jmp L_loop40
13 L_done40:
```

마찬가지로 다음과 같이 **SFENCE**를 이용한 work thread를 구현하였다.

```
1  L_loop51:
2    sfence
3    movl mem(%rip), %ecx
4    cmpl count(%rip), %ecx
5    jge L_done51
6    movl %ecx, %eax
7    cltd
8    idivl tnum(%rip)
9    cmpl %edi, %edx
10   jne L_loop51
```

```
11    incl mem(%rip)
12    jmp L_loop51
13 L_done51:
```

mem에 1을 더하고 store한 값이 global하게 visible하면 되기 때문에, load에서의 memory fence는 필요 없다. 따라서 **MFENCE**를 사용한 구현과 **SFENCE**를 사용한 구현은 거의 동일하다. 단순히 작업이 완료되었는지 확인하기 전에 store fence를 하여 update된 값을 정확히 확인하고, 확인된 값과 자신의 thread-id를 이용하여 한 번에 한 thread만 덧셈을 수행한다. 이때, 수행 시간은 각각 132초, 135초였다.

8 Conclusion

실험 결과, software logic의 비중이 크거나 다른 thread의 작업에 영향을 많이 받을수록 성능이 떨어졌다. 동기화를 위해 필요한 부가 연산이 많아지고, 여러 thread가 충돌했을 때 적어도 하나는 진행할 수 있는 지의 여부가 달라지기 때문이다.

BTS는 명시적으로 Critical section으로 진입하고 나갈 때, lock을 이용하기 때문에 공유 메모리에 대한 atomic 접근 부가 연산이 많아져 크게 느려진다. **CMPXCHG**는 적어도 하나의 thread는 반드시 성공하며, 코드가 간단하여 이 명령어 이외의 부가 연산이 크지 않다. 이는 **LOCK ADD**도 마찬가지인 것으로 보인다. **MFENCE**나 **SFENCE**는 여러 thread가 반드시 각 iteration에서 대기하면서 순서를 맞추어 진행해야 하며, mem 접근에서 반드시 분기하므로 느려지는 경향이 있다.

그 어떤 방법도 Sequential 구현에 비해 느릴 수밖에 없는데, 여러 thread가 결국 하나의 메모리 공간에 접근하기 때문에 근본적으로 Parallel하지 않기 때문이다. 이 문제에서는 Data parallelism도, Task parallelism도 이용할 수 없으며, 멀티스레드 구현을 하게 되면 Instruction level parallelism도 완전히 활용하기 어려워진다. 따라서 이런 종류의 문제에서는 멀티스레드를 이용하지 말아야 한다.