Brian Ouellette
gth677b
April 16, 2010

# Array Based Queue Lock Implementation in Alpha ASM
# ECE 6101 Term Project

## *Design of the Lock:*

The lock implemented is following the algorithm for an array based queue lock that was provided by Tom Anderson in “The performance of spin lock alternatives for shared-memory multiprocessors” published in IEEE Transactions on Parallel and Distributed Systems in January 1990. The pseudocode published in the article is shown below.

```
type lock = record
     slots : array [0..numprocs -1] of (has_lock, must_wait)
         := (has_lock, must_wait, must_wait, ..., must_wait)
         // each element of slots should lie in a different memory module
         // or cache line
     next_slot : integer := 0

 // parameter my_place, below, points to a private variable
 // in an enclosing scope

 procedure acquire_lock (L : ^lock, my_place : ^integer)
     my_place^ := fetch_and_increment (&L->next_slot)
         // returns old value
     if my_place^ mod numprocs = 0
         atomic_add (&L->next_slot, -numprocs)
         // avoid problems with overflow; return value ignored
     my_place^ := my_place^ mod numprocs
     repeat while L->slots[my_place^] = must_wait      // spin
     L->slots[my_place^] := must_wait                  // init for next time

 procedure release_lock (L : ^lock, my_place : ^integer)
     L->slots[(my_place^ + 1) mod numprocs] := has_lock
```

The idea is to give each process its own lock variable to spin on so that there is not bus contention over any specific variable. When a process attempts to acquire a lock it gets assigned a spot in an array of lock variables and waits till its variable is set, indicating it has the lock. When each process releases its lock it sets the next processes' lock, bringing the next process out of its busy-waiting loop.

Fetch-and-increment and atomic_add must both be atomic operations, which with the Alpha ISA is implemented through the use of the load-link (or load-locked) and store-conditional instructions, ldl_l and stl_c respectively. A store-conditional will fail without writing if the memory address it is storing too has been modified since the load-link was executed. By repeating the load-link and store-conditional until the store-conditional reports success and writes the value out will ensure atomicity for the memory operations.

The core of my implementation is provided below. The variable “mine” corresponds to “my_place” and “base” is just the index of the start of the “L->slots[]” array. The implementation of the fetch-and-add is at label “1:” and atomic_add can be seen at label “2:”. Otherwise it aligns very closely to the pseudocode above, optimized for a max of 16 processes. Hand-written pseudocode is available in the comments in “lock.h” for potentially easier to interpret version.

Acquire:

```
1:  ldl_l  %[mine],%[next]
    addl   %[mine],1,%[tmp]
    stl_c  %[tmp],%[next]
    beq    %[tmp],1b
    cmpult %[mine],16,%[tmp]
    bne    %[tmp],3f
2:  ldl_l  %[tmp],%[next]
    subl   %[tmp],16,%[tmp]
    stl_c  %[tmp],%[next]
    beq    %[tmp],2b
3:  and    %[mine],15,%[mine]
4:  lda    %[tmp],%[base]
    sll    %[mine],4,%[tmp2]
    s4addq %[tmp2],%[tmp],%[tmp]
5:  ldl    %[tmp2],0(%[tmp])
    beq    %[tmp2],5b
    stl    $31,0(%[tmp])
```

Release:

```
addl   %[mine],1,%[mine]
and    %[mine],15,%[mine]
lda    %[tmp],%[base]
sll    %[mine],4,%[tmp2]
s4addq %[tmp2],%[tmp],%[tmp]
addl   $31,1,%[tmp2]
stl    %[tmp2],0(%[tmp])
```

### *Lock Interaction with Cache Coherency Protocol:*

The Alpha computer the benchmark is running on is using a bus-based MESI cache coherence protocol. Sixteen processors share the bus and each has its own private local memory; there is no directory. To approach the issue of how the implemented lock interacts with the cache coherence system used by the Alpha computer first the spinlock, then small improvements on it will be examined, followed by the lock implemented here.

The simplest spinlock would just busy-wait on a single lock variable attempting a load-link/store-conditional. The biggest problem with this is that since the lock variable appears in each processor's cache each attempt to set it is going to invalidate all other local copies, forcing the other processors spinning on the lock to fetch a new copy from memory. So the original write causes an elevation to the modified state in the local processor and sends out an invalidation for the others. Then another processor pushes the variable into the modified state, invalidating all copies again. While more than one processor is attempting to acquire the lock, the bus and the memory modules the processors are fetching the lock variable from are easily saturated with useless traffic.

The first basic improvement is to make the algorithm spin with a load instead of a load-linked, using the idea from a construct known as “test and test-and-set”. This will only elevate the lock variable to a shared state initially and does not require invalidations to be sent out unless the lock variable indicates the lock is available at which point the processor attempts to write to it. When the lock becomes available an invalidation will be received and the new lock variable will be fetched from memory. Until that point the processor can busy-wait on a local copy of the lock variable. This eliminates a lot of bus traffic but it results in bursty traffic since every processor spinning to acquire the lock will receive the invalidation more or less simultaneously and will all attempt to elevate to a modified state and read from memory at the same time.

An array based queue lock addresses this problem by having each processor spin on a separate lock variable. When the lock is released the processor must fetch the address of the lock variable for the next processor in line and update it. This will cause an invalidation to be sent out for the lock variable  but only the processor spinning on that variable will be affected so there will just be one

processor attempting to fetch the new value. Each processor can keep its lock variable cached locally and in a way the cache coherence scheme itself is used to send a message specifically to the next processor in line that it can acquire the lock. It is critical that each lock variable be in a different cache block or that the cache coherence protocol is update based since the entire cache block will be invalidated on writes. If the lock variables all reside sequentially in an array in memory they will almost assuredly be in the same cache block and invalidates meant for a single processor will be received by every processor with a locally cached lock variable dragging performance back to below that of a “test and test-and-set” based spinlock.

### *Simulation Results:*

Data has been gathered on the lock throughout the programming process. Each stage is described below. Each subsequent stage was built upon the last one with the exception of o3base.

```
base           -original lock.h
arrayBadCache  -Anderson's Array Based Queueing lock, in one cache line
withJumps      -on different cache lines, jumps to subsections and back as in atomic.h
withoutJumps   -direct jumps to labels
o3base         -original lock.h compiled with -O3
final          -unlock indexing in asm, w/o jumps, compiled with -O3, small optimizations in lock code
```

All data points gathered from the data cache represent the sum total from all 16 CPUs. Raw unsummarized data and output logs for each stage can be found in the attachment accompanying this report.

The set of data corresponding to o3base is not completely reliable as GCC's internal optimizations caused the benchmark to fail; however, I have included it in order to give a sense of how much improvement in final comes from the change in algorithm and how much comes from being programmed such that GCC optimzations are able to run safely. Keep in mind that since the benchmark produced incorrect results for o3base the optimizations performed were more aggressive than those performed on the final dataset.

Beginning with arrayBadCache, it is not surprising to see that this implementation of the lock performs significantly worse than the provided simple “test and test-and-set” spinlock. Since each threads variable is in the same cache block the interaction with the cache coherence protocol will be just the same as the spinlock and generate unnecessary invalidations. This implementation gains nothing over a spinlock and at the cost of a more computationally intense algorithm.

Spreading the slots array out such that each element is in a different cache line causes the number of cache misses to plummet and correspondingly the time the benchmark takes to run decreases as well, which can be seen in the withJumps results.

The next step taken to try and optimize performance was to see the difference between having jumps for looping be to a subsection past the main-line code and then back into original code and just using direct jumps. In “atomic.h”, the file which provides atomic instructions for use in programming C for the Alpha it explains the practice of jumping to a subsection and back as necessary in order to get proper branch prediction; however, in practice the two were almost identical in terms of every metric. This result was surprising as I had assumed that either the branch predictor would fail, degrading performance, or the instruction cache would have a lowered miss rate, increasing it. Apparently the instruction cache is large enough that this is not an issue or the branch predictor is better than the one “atomic.h” was initially written for. Direct jumps were used in the final code to reduce code size and increase code readability.

Up until this point array indexing for my_spin_unlock() had been done in C and passed directly into the asm statement. For final this was redone in assembly, netting a small increase in performance. There was some slight code rearranging and the removal of an assembly statement in my_spin_lock()

which resulted in negligible performance gain. The “memory” statement was also added to the clobbered list at this point preventing GCC from caching memory values across the assembly sections. The lack of this may have been partially responsible for o3base failing to pass the benchmark although that is doubtful since the assembly statements were in their own functions and the values in the registers would have to be held onto across multiple function calls in order for that to have an effect. At this point the optimized and compiled code has a significant advantage over the original spinlock. The complete comparison against all sizes of benchmarks can be seen in the Appendix A.

The next step to be taken is to increase the granularity of the lock. This was not finished because of time constraints but the natural step is to modify the mutex structure to have a “next” field that is a 16 wide array instead of a single int. Then the extra space allocated for “slots” can be used to implement 16 locks within the same mutex instead of being wasted. The changes to the code necessary for this to happen can be seen in “lockFuture.h”; however, the changes were producing slow results due to lack of debugging and optimization time and so were not implemented in the final lock.

# Appendix A: Compiled Statistics

| | Bench Time Cost | | | | |
|---|---|---|---|---|---|
| | 1 | 2 | 4 | 8 | 16 |
| base | 0.010747 | 0.008793 | 0.006839 | 0.005862 | 0.008793 |
| arrayBadCache | 0.013678 | 0.00977 | 0.010747 | 0.010747 | 0.011724 |
| withJumps | 0.012701 | 0.007816 | 0.006839 | 0.006839 | 0.006839 |
| withoutJumps | 0.012701 | 0.007816 | 0.006839 | 0.006839 | 0.006839 |
| o3base | 0.001954 | FAILED | FAILED | FAILED | FAILED |
| final | 0.005862 | 0.004885 | 0.003908 | 0.003908 | 0.004885 |

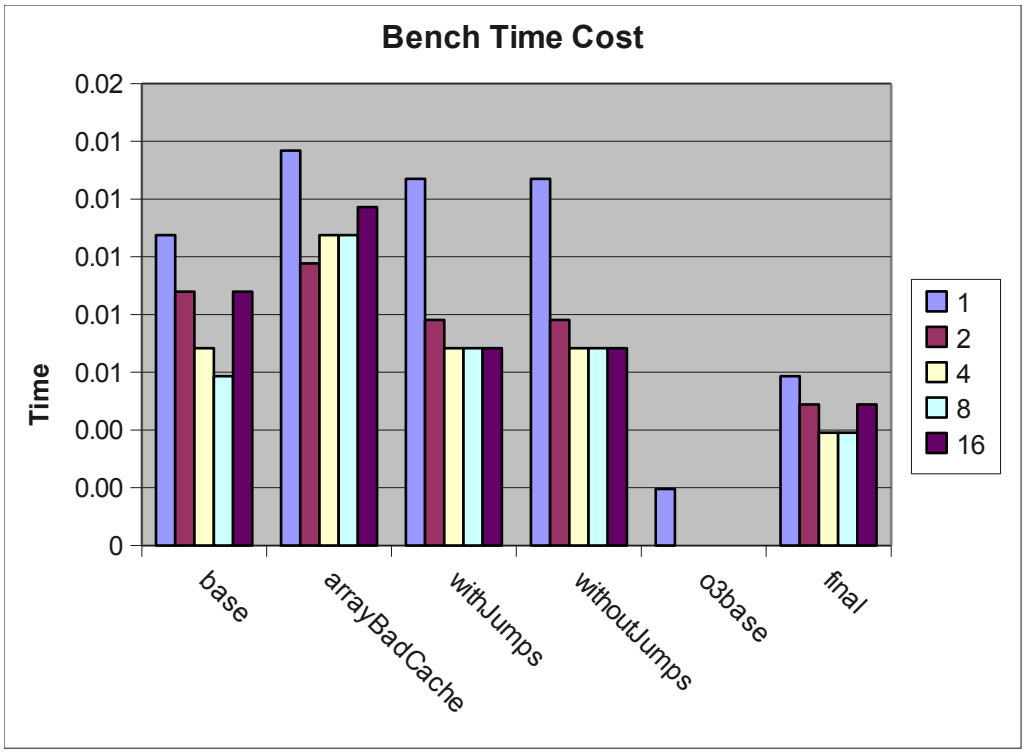
| | Cache Misses (dcache) | | | | |
|---|---|---|---|---|---|
| | 1 | 2 | 4 | 8 | 16 |
| base | 45222 | 175251 | 396252 | 782252 | 1223209 |
| arrayBadCache | 46259 | 229782 | 487227 | 1003935 | 1963852 |
| withJumps | 45881 | 237134 | 241806 | 247681 | 283000 |
| withoutJumps | 45862 | 237111 | 241594 | 257694 | 284431 |
| o3base | 42929 | 189557 | 417243 | 639046 | 995703 |
| final | 44376 | 232914 | 267074 | 290246 | 311666 |

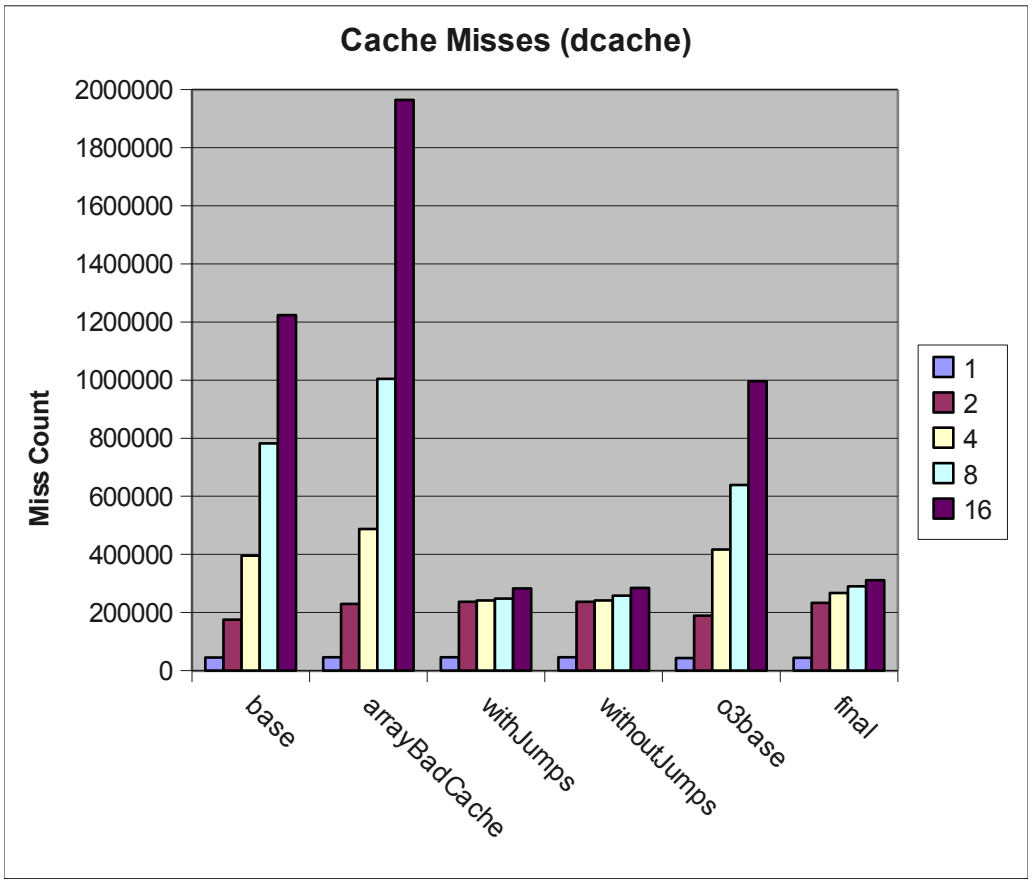
| | LL Issued (dcache) | | | | |
|---|---|---|---|---|---|
| | 1 | 2 | 4 | 8 | 16 |
| base | 76036 | 76928 | 191293 | 423615 | 886253 |
| arrayBadCache | 80325 | 76787 | 77190 | 78407 | 140213 |
| withJumps | 79399 | 76680 | 77135 | 78295 | 105353 |
| withoutJumps | 79401 | 76679 | 77093 | 136503 | 79685 |
| o3base | 75459 | 132718 | 257498 | 462334 | 804638 |
| final | 79060 | 76508 | 76684 | 261168 | 498485 |

| | SC Issued (dcache) | | | | |
|---|---|---|---|---|---|
| | 1 | 2 | 4 | 8 | 16 |
| base | 75659 | 72363 | 168081 | 287733 | 302395 |
| arrayBadCache | 79952 | 76472 | 76886 | 78096 | 139773 |
| withJumps | 79035 | 76401 | 76764 | 77914 | 104980 |
| withoutJumps | 79028 | 76407 | 76732 | 136113 | 79371 |
| o3base | 75129 | 129675 | 226777 | 294261 | 283239 |
| final | 78699 | 76232 | 76436 | 260915 | 498188 |

| | SC Hits (dcache) | | | | |
|---|---|---|---|---|---|
| | 1 | 2 | 4 | 8 | 16 |
| base | 73333 | 5404 | 5253 | 5512 | 6476 |
| arrayBadCache | 77596 | 9287 | 9240 | 9724 | 70622 |
| withJumps | 76715 | 9283 | 9141 | 9559 | 35853 |
| withoutJumps | 76719 | 9290 | 9121 | 67690 | 10364 |
| o3base | 72900 | 8865 | 7617 | 7445 | 8049 |
| final | 76415 | 9943 | 9438 | 38641 | 29284 |

| | SC Misses (dcache) | | | | |
|---|---|---|---|---|---|
| | 1 | 2 | 4 | 8 | 16 |
| base | 2326 | 66959 | 162828 | 282221 | 295919 |
| arrayBadCache | 2356 | 67185 | 67646 | 68372 | 69151 |
| withJumps | 2320 | 67118 | 67623 | 68355 | 69127 |
| withoutJumps | 2309 | 67117 | 67611 | 68423 | 69007 |
| o3base | 2229 | 120810 | 219160 | 286816 | 275190 |
| final | 2284 | 66289 | 66998 | 222274 | 468904 |


Bench Time Cost
Time
0.02
0.01
0.01
0.01
0.01
0.01
0.00
0.00
0
base
arrayBadCache
withJumps
withoutJumps
o3base
final
1
2
4
8
16


Cache Misses (dcache)
Miss Count
2000000
1800000
1600000
1400000
1200000
1000000
800000
600000
400000
200000
0
base
arrayBadCache
withJumps
withoutJumps
o3base
final
1
2
4
8
16