Module 11: "Synchronization"

Lecture 23: "Barriers and Speculative Synchronization"

- Barrier
- Centralized barrier
- Sense reversal
- Centralized barrier
- Tree barrier
- Hardware support
- Hardware barrier
- Speculative synch.
- Why is it good?
- How does it work?
- Why is it correct?
- Performance concerns
- Speculative flags and barriers
- Speculative flags and branch prediction

Previous Next

Module 11: "Synchronization"
Lecture 23: "Barriers and Speculative Synchronization"

Barrier

- High-level classification of barriers
  - Hardware and software barriers
- Will focus on two types of software barriers
  - .Centralized barrier: every processor polls a single count
  - Distributed tree barrier: shows much better scalability
- Performance goals of a barrier implementation
  - Low latency: after all processors have arrived at the barrier, they should be able to leave quickly
  - Low traffic: minimize bus transaction and contention
  - Scalability: latency and traffic should scale slowly with the number of processors
  - Low storage: barrier state should not be big
  - Fairness: Preserve some strict order of barrier exit (could be FIFO according to arrival order); a particular processor should not always be the last one to exit

Centralized barrier

```
struct bar_type {
  int counter;
  struct lock_type lock;
  int flag = 0;
} bar_name;

BARINIT (bar_name) {
  LOCKINIT(bar_name.lock);
  bar_name.counter = 0;
}

BARRIER (bar_name, P) {
  int my_count;
  LOCK (bar_name.lock);
  if (!bar_name.counter) {
    bar_name.flag = 0; /* first one */
  }
  my_count = ++bar_name.counter;
  UNLOCK (bar_name.lock);
  if (my_count == P) {
    bar_name.counter = 0;
    bar_name.flag = 1; /* last one */
  }
  else {
     while (!bar_name.flag);
  }
}
```

Sense reversal

The last implementation fails to work for two consecutive barrier invocations

- Need to prevent a process from entering a barrier instance until all have left the previous instance
- Reverse the sense of a barrier i.e. every other barrier will have the same sense: basically attach parity or sense to a barrier

```
BARRIER (bar_name, P) {
  local sense = !local_sense; /* this is private per processor */
  LOCK (bar_name.lock);
  bar_name.counter++;
  if (bar_name.counter == P) {
    UNLOCK (bar_name.lock);
    bar_name.counter = 0;
    bar_name.flag = local_sense;
  }
  else {
     UNLOCK (bar_name.lock);
     while (bar_name.flag != local_sense);
  }
}
```

Previous Next

Centralized barrier

- How fast is it?
  - Assume that the program is perfectly balanced and hence all processors arrive at the barrier at the same time
  - Latency is proportional to P due to the critical section (assume that the lock algorithm exhibits at most O(P) latency)
  - The amount of traffic of acquire section (the CS) depends on the lock algorithm; after everyone has settled in the waiting loop the last processor will generate a BusRdX during release (flag write) and others will subsequently generate BusRd before releasing: O(P)
  - Scalability turns out to be low partly due to the critical section and partly due to O(P) traffic of release
  - No fairness in terms of who exits first

Tree barrier

- Does not need a lock, only uses flags
  - Arrange the processors logically in a binary tree (higher degree also possible)
  - Two siblings tell each other of arrival via simple flags (i.e. one waits on a flag while the other sets it on arrival)
  - One of them moves up the tree to participate in the next level of the barrier
  - Introduces concurrency in the barrier algorithm since independent subtrees can proceed in parallel
  - Takes log(P) steps to complete the acquire
  - A fixed processor starts a downward pass of release waking up other processors that in turn set other flags
  - Shows much better scalability compared to centralized barriers in DSM multiprocessors; the advantage in small bus-based systems is not much, since all transactions are any way serialized on the bus; in fact the additional log (P) delay may hurt performance in bus-based SMPs

```
TreeBarrier (pid, P) {
  unsigned int i, mask;
  for (i = 0, mask = 1; (mask & pid) != 0; ++i, mask <<= 1) {
    while (!flag[pid][i]);
    flag[pid][i] = 0;
  }
  if (pid < (P - 1)) {
    flag[pid + mask][i] = 1;
    while (!flag[pid][MAX- 1]);
    flag[pid][MAX - 1] = 0;
  }
  for (mask >>= 1; mask > 0; mask >>= 1) {
flag[pid - mask][MAX-1] = 1;
 }
}
```

- Convince yourself that this works
- Take 8 processors and arrange them on leaves of a tree of depth 3
- You will find that only odd nodes move up at every level during acquire (implemented in the first for loop)
- The even nodes just set the flags (the first statement in the if condition): they bail out of the first loop with mask=1
- The release is initiated by the last processor in the last for loop; only odd nodes execute this loop (7 wakes up 3, 5, 6; 5 wakes up 4; 3 wakes up 1, 2; 1 wakes up 0)
- Each processor will need at most log (P) + 1 flags
- Avoid false sharing: allocate each processor's flags on a separate chunk of cache lines
- With some memory wastage (possibly worth it) allocate each processor's flags on a separate page and map that page locally in that processor's physical memory
    - Avoid remote misses in DSM multiprocessor
    - Does not matter in bus-based SMPs

Previous Next

Hardware support

- Read broadcast
  - Possible to reduce the number of bus transactions from P-1 to 1 in the best case
  - A processor seeing a read miss to flag location (possibly from a fellow processor) backs off and does not put its read miss on the bus
  - Every processor picks up the read reply from the bus and the release completes with one bus transaction
  - Needs special hardware/compiler support to recognize these flag addresses and resort to read broadcast

Hardware barrier

- Useful if frequency of barriers is high
  - Need a couple of wired-AND bus lines: one for odd barriers and one for even barriers
  - A processor arrives at the barrier and asserts its input line and waits for the wired-AND line output to go HIGH
  - Not very flexible: assumes that all processors will always participate in all barriers
  - Bigger problem: what if multiple processes belonging to the same parallel program are assigned to each processor?
  - No SMP supports it today
  - However, possible to provide flexible hardware barrier support in the memory controller of DSM multiprocessors: memory controller can recognize accesses to special barrier counter or barrier flag, combine them in memory and reply to processors only when the barrier is complete (no retry due to failed lock)

Speculative synch.

- Speculative synchronization
  - Basic idea is to introduce speculation in the execution of critical sections
  - Assume that no other processor will have conflicting data accesses in the critical section and hence don't even try to acquire the lock
  - Just venture into the critical section and start executing
  - Note the difference between this and speculative execution of critical section due to speculation on the branch following SC: there you still contend for the lock generating network transactions
- Martinez and Torrellas. In ASPLOS 2002.
- Rajwar and Goodman. In ASPLOS 2002.
- We will discuss Martinez and Torrellas

Why is it good?

- In many cases compiler/user inserts synchronization conservatively
  - Hard to know exact access pattern
  - The addresses accessed may depend on input
- Take a simple example of a hash table
  - When the hash table is updated by two processes you really do not know which bins they will insert into
  - So you conservatively make the hash table access a critical section

- For certain input values it may happen that the processes could actually update the hash table concurrently

Previous Next

How does it work?

- Speculative locks
  - Every processor comes to the critical section and tries to acquire the lock
  - One of them succeeds and the rest fail
  - The successful processor becomes the `safe` thread
  - The failed ones don't retry but venture into the critical section speculatively as if they have the lock; at this point a speculative thread also takes a checkpoint of its register state in case a rollback is needed
  - The safe thread executes the critical section as usual
  - The speculative threads are allowed to consume values produced by the safe thread but not by the sp. threads
  - All stores from a speculative thread are kept inside its cache hierarchy in a special "speculative modified" state; these lines cannot be sent to memory until it is known to be safe; if such a line is replaced from cache either it can be kept in a small buffer or the thread can be stalled
- Speculative locks (continued)
  - If a speculative thread receives a request for a cache line that is in speculative M state, that means there is a data race inside the critical section and by design the receiver thread is rolled back to the beginning of critical section
  - Why can't the requester thread be rolled back?
  - In summary, the safe thread is never squashed and the speculative threads are not squashed if there is no cross-thread data race
  - If a speculative thread finishes executing the critical section without getting squashed, it still must wait for the safe thread to finish the critical section before committing the speculative state (i.e. changing speculative M lines to M); why?
- Speculative locks (continued)
  - Upon finishing the critical section, a speculative thread can continue executing beyond the CS, but still remaining in speculative mode
  - When the safe thread finishes the CS all speculative threads that have already completed CS, can commit in some non-deterministic order and revert to normal execution
  - The speculative threads that are still inside the critical section remain speculative; a dedicated hardware unit elects one of them the lock owner and that becomes the safe non-speculative thread; the process continues
  - Clearly, under favorable conditions speculative synchronization can reduce lock contention enormously

Previous Next

Why is it correct?

- In a non-speculative setting there is no order in which the threads execute the CS
  - Even if there is an order that must be enforced by the program itself
- In speculative synchronization some threads are considered safe (depends on time of arrival) and there is exactly one safe thread at a time in a CS
- The speculative threads behave as if they complete the CS in some order after the safe thread(s)
- A read from a thread (spec. or safe) after a write from another speculative thread to the same cache line triggers a squash
  - It may not be correct to consume the speculative value
  - Same applies to write after write

Performance concerns

- Maintaining a safe thread guarantees forward progress
  - Otherwise if all were speculative, cross-thread races may repeatedly squash all of them
- False sharing?
  - What if two bins of a hash table belong to the same cache line?
  - Two threads are really not accessing the same address, but the speculative thread will still suffer from a squash
  - Possible to maintain per-word speculative state

Speculative flags and barriers

- Speculative flags are easy to support
  - Just continue past an unset flag in speculative mode
  - The thread that sets the flag is always safe
  - The thread(s) that read the flag will speculate
- Speculative barriers come for free
  - Barriers use locks and flags
  - However, since the critical section in a barrier accesses a counter, multiple threads venturing into the CS are guaranteed to have conflicts
  - So just speculate on the flag and let the critical section be executed conventionally

Speculative flags and branch prediction

P0: A=1; flag=1;
P1: while (!flag); print A;
Assembly of P1's code

```
Loop:  lw register, flag_addr
       beqz register, Loop
         …
```

- What if I pass a hint via the compiler (say, a single bit in each branch instruction) to the branch predictor asking it to always predict not taken for this branch?
  - Isn't it achieving the same effect as speculative flag, but with a much simpler technique? No.

Previous Next