Special Topics

- Virtually indexed caches
- Virtual indexing
- TLB coherence
- TLB shootdown
- Snooping on a ring
- Scaling bandwidth
- AMD Opteron
- Opteron servers
- AMD Hammer protocol

[From Chapter 6 of Culler, Singh, Gupta]

Previous Next

Virtually indexed caches

- Recall that to have concurrent accesses to TLB and cache, L1 caches are often made virtually indexed
  - Can read the physical tag and data while the TLB lookup takes place
  - Later compare the tag for hit/miss detection
  - How does it impact the functioning of coherence protocols and snoop logic?
- Even for uniprocessor the synonym problem
  - Two different virtual addresses may map to the same physical page frame
  - One simple solution may be to flush all cache lines mapped to a page frame at the time of replacement
  - But this clearly prevents page sharing between two processes

Virtual indexing

- Software normally employs page coloring to solve the synonym issue
  - Allow two virtual pages to point to the same physical page frame only if the two virtual addresses have at least lower k bits common where k is equal to cache line block offset plus log2 (number of cache sets)
  - This guarantees that in a virtually indexed cache, lines from both pages will map to the same index range
- What about the snoop logic?
  - Putting virtual address on the bus requires a VA to PA translation in the snoop so that physical tags can be generated (adds extra latency to snoop and also requires duplicate set of translations)
  - Putting physical address on the bus requires a reverse translation to generate the virtual index (requires an inverted page table)
- Dual tags (Goodman, 1987)
  - Hardware solution to avoid synonyms in shared memory
  - Maintain virtual and physical tags; each corresponding tag pair points to each other
  - Assume no page coloring
  - Use virtual address to look up cache (i.e. virtual index and virtual tag) from processor side; if it hits everything is fine; if it misses use the physical address to look up the physical tag and if it hits follow the physical tag to virtual tag pointer to find the index
  - If virtual tag misses and physical tag hits, that means the synonym problem has happened i.e. two different VAs are mapped to the same PA; in this case invalidate the cache line pointed to by physical tag, replace the line at the virtual index of the current virtual address, place the contents of the invalidated line there and update the physical tag pointer to point to the new virtual index
- Goodman, 1987
  - Always use physical address for snooping
  - Obviates the need for a TLB in memory controller
  - The physical tag is used to look up the cache for snoop decision
  - In case of a snoop hit the pointer stored with the physical tag is followed to get the virtual index and then the cache block can be accessed if needed (e.g., in M state)
  - Note that even if there are two different types of tags the state of a cache line is the same and does not depend on what type of tag is used to access the line
- Multi-level cache hierarchy

- Normally the L1 cache is designed to be virtually indexed and other levels are physically indexed
- L2 sends interventions to L1 by communicating the PA
- L1 must determine the virtual index from that to access the cache: dual tags are sufficient for this purpose

Previous Next

TLB coherence

- A page table entry (PTE) may be held in multiple processors in shared memory because all of them access the same shared page
  - A PTE may get modified when the page is swapped out and/or access permissions are changed
  - Must tell all processors having this PTE to invalidate
  - How to do it efficiently?
- No TLB: virtually indexed virtually tagged L1 caches
  - On L1 miss directly access PTE in memory and bring it to cache; then use normal cache coherence because the PTEs also reside in the shared memory segment
  - On page replacement the page fault handler can flush the cache line containing the replaced PTE
  - Too impractical: fully virtual caches are rare, still uses a TLB for upper levels (Alpha 21264 instruction cache)
- Hardware solution
  - Extend snoop logic to handle TLB coherence
  - PowerPC family exercises a tlbie instruction (TLB invalidate entry)
  - When OS modifies a PTE it puts a tlbie instruction on bus
  - Snoop logic picks it up and invalidates the TLB entry if present in all processors
  - This is well suited for bus-based SMPs, but not for DSMs because broadcast in a large-scale machine is not good

TLB shootdown

- Popular TLB coherence solution
  - Invoked by an initiator (the processor which modifies the PTE) by sending interrupt to processors that might be caching the PTE in TLBs; before doing so OS also locks the involved PTE to avoid any further access to it in case of TLB misses from other processors
  - The receiver of the interrupt simply invalidates the involved PTE if it is present in its TLB and sets a flag in shared memory on which the initiator polls
  - On completion the initiator unlocks the PTE
  - SGI Origin uses a lazy TLB shootdown i.e. it invalidates a TLB entry only when a processor tries to access it next time (will discuss in detail)

Snooping on a ring

- Length of the bus limits the frequency at which it can be clocked which in turn limits the bandwidth offered by the bus leading to a limited number of processors
- A ring interconnect provides a better solution
  - Connect a processor only to its two neighbors
  - Short wires, much higher switching frequency, better bandwidth, more processors
  - Each node has private local memory (more like a distributed shared memory multiprocessor)
  - Every cache line has a home node i.e. the node where the memory contains this line: can be determined by upper few bits of the PA
  - Transactions traverse the ring node by node

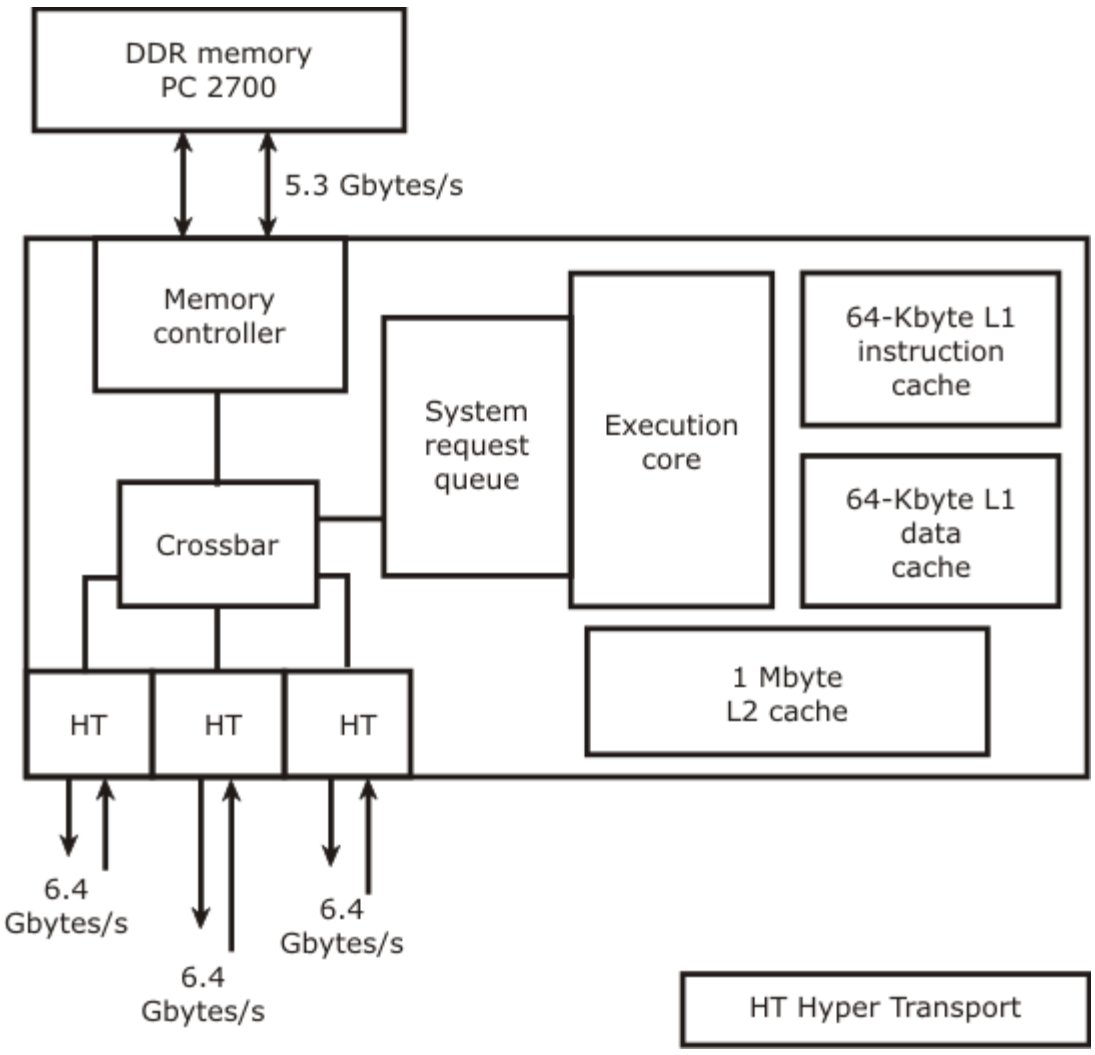
- Snoop mechanism
  - When a transaction passes by the ring interface of a node it snoops the transaction, takes appropriate coherence actions, and forwards the transaction to its neighbor if necessary
  - The home node also receives the transaction eventually and let's assume that it has a dirty bit associated with every memory line (otherwise you need a two-phase protocol)
  - A request transaction is removed from the ring when it comes back to the requester (serves as an acknowledgment that every node has seen the request)
  - The ring is essentially divided into time slots where a node can insert new request or response; if there is no free time slot it must wait until one passes by: called a slotted ring
- The snoop logic must be able to finish coherence actions for a transaction before the next time slot arrives
- The main problem of a ring is the end-to-end latency, since the transactions must traverse hop-by-hop
- Serialization and sequential consistency is trickier
  - The order of two transactions may be differently seen by two processors if the source of one transaction is between the two processors
  - The home node can resort to NACKs if it sees conflicting outstanding requests
  - Introduces many races in the protocol

Previous Next

Scaling bandwidth

- Data bandwidth
  - Make the bus wider: costly hardware
  - Replace bus by point-to-point crossbar: since only the address portion of a transaction is needed for coherence, the data transaction can be directly between source and destination
  - Add multiple data busses
- Snoop or coherence bandwidth
  - This is determined by the number of snoop actions that can be executed in unit time
  - Having concurrent non-conflicting snoop actions definitely helps improve the protocol throughput
  - Multiple address busses: a separate snoop engine is associated with each bus on each node
  - Order the address busses logically to define a partial order among concurrent requests so that these partial orders can be combined to form a total order

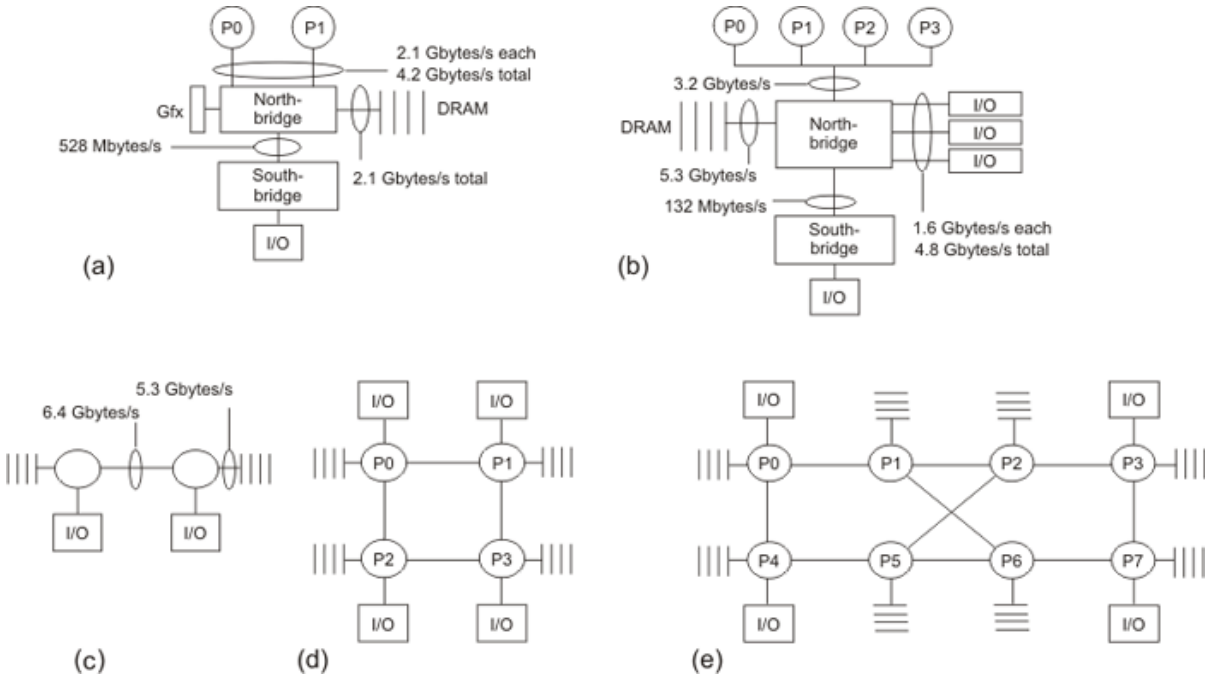
AMD Opteron



- Each node contains an x86-64 core, 64 KB L1 data and instruction caches, 1 MB L2 cache, on-chip integrated memory controller, three fast routing links called hyperTransport, local DDR memory
- Glueless MP: just connect 8 Opteron chips via HT to design a distributed shared memory multiprocessor
- L2 cache supports 10 outstanding misses
- Integrated memory controller and north bridge functionality help a lot
  - Can clock the memory controller at processor frequency (2 GHz)
  - No need to have a cumbersome motherboard; just buy the Opteron chip and connect it to a few peripherals (system maintenance is much easier)
  - Overall, improves performance by 20-25% over Athlon
  - Snoop throughput and bandwidth is much higher since the snoop logic is clocked at 2 GHz
- Integrated hyperTransport provides very high communication bandwidth
  - Point-to-point links, split-transaction and full duplex (bidirectional links)
  - On each HT link you can connect a processor or I/O

Previous Next

Opteron servers



Produced from IEEE Micro

AMD Hammer protocol

- Opteron uses the snoop-based Hammer protocol
  - First the requester sends a transaction to home node
  - The home node starts accessing main memory and in parallel broadcasts the request to all the nodes via point-to-point messages
  - The nodes individually snoop the request, take appropriate coherence actions in their local caches, and sends data (if someone has it in M or O state) or an empty completion acknowledgment to the requester; the home memory also sends the data speculatively
  - After gathering all responses the requester sends a completion message to the home node so that it can proceed with subsequent requests (this completion ack may be needed for serializing conflicting requests)
  - This is one example of a snoopy protocol over a point-to-point interconnect unlike the shared bus

Previous Next

Exercise : 2

These problems should be tried after module 12 is completed.

1. [30 points] For each of the memory reference streams given in the following, compare the cost of executing it on a bus-based SMP that supports (a) MESI protocol without cache-to-cache sharing, and (b) Dragon protocol. A read from processor N is denoted by rN while a write from processor N is denoted by wN. Assume that all caches are empty to start with and that cache hits take a single cycle, misses requiring upgrade or update take 60 cycles, and misses requiring whole block transfer take 90 cycles. Assume that all caches are writeback.

Stream1: r1 w1 r1 w1 r2 w2 r2 w2 r3 w3 r3 w3
Stream2: r1 r2 r3 w1 w2 w3 r1 r2 r3 w3 w1
Stream3: r1 r2 r3 r3 w1 w1 w1 w1 w2 w3

[For each stream for each protocol: 5 points]

2. [15 points] (a) As cache miss latency increases, does an update protocol become more or less preferable as compared to an invalidation based protocol? Explain.

(b) In a multi-level cache hierarchy, would you propagate updates all the way to the first-level cache? What are the alternative design choices?

(c) Why is update-based protocol not a good idea for multiprogramming workloads running on SMPs?

3. [20 points] Assuming all variables to be initialized to zero, enumerate all outcomes possible under sequential consistency for the following code segments.

(a) P1: A=1;
P2: u=A; B=1;
P3: v=B; w=A;

(b) P1: A=1;
P2: u=A; v=B;
P3: B=1;
P4: w=B; x=A;

(c) P1: u=A; A=u+1;
P2: v=A; A=v+1;

(d) P1: fetch-and-inc (A)
P2: fetch-and-inc (A)

4. [30 points] Consider a quad SMP using a MESI protocol (without cache-to-cache sharing). Each processor tries to acquire a test-and-set lock to gain access to a null critical section. Assume that test-and-set instructions always go

on the bus and they take the same time as the normal read transactions. The initial condition is such that processor 1 has the lock and processors 2, 3, 4 are spinning on their caches waiting for the lock to be released. Every processor gets the lock once, unlocks, and then exits the program. Consider the bus transactions related to the lock/unlock operations only.

(a) What is the least number of transactions executed to get from the initial to the final state? [10 points]

(b) What is the worst-case number of transactions? [5 points]

(c) Answer the above two questions if the protocol is changed to Dragon. [15 points]

5. [30 points] Answer the above question for a test-and-test-and-set lock for a 16-processor SMP. The initial condition is such that the lock is released and no one has got the lock yet.

6. [10 points] If the lock variable is not allowed to be cached, how will the traffic of a test-and-set lock compare against that of a test-and-test-and set lock?

7. [15 points] You are given a bus-based shared memory machine. Assume that the processors have a cache block size of 32 bytes and A is an array of integers (four bytes each). You want to parallelize the following loop.

```
for(i=0; i<17; i++) {
for (j=0; j<256; j++) {
A[j] = do_something(A[j]);
}
}
```

(a) Under what conditions would it be better to use a dynamically scheduled loop?

(b) Under what conditions would it be better to use a statically scheduled loop?

(c) For a dynamically scheduled inner loop, how many iterations should a processor pick each time?

8. [20 points] The following barrier implementation is wrong. Make as little change as possible to correct it.

```
struct bar_struct {
LOCKDEC(lock);
int count; // Initialized to zero
int releasing; // Initialized to zero
} bar;

void BARRIER (int P) {
LOCK(bar.lock);
bar.count++;
if (bar.count == P) {
bar.releasing = 1;
bar.count--;
}
```

```
else {
UNLOCK(bar.lock);
while (!bar.releasing);
LOCK(bar.lock);
bar.count--;
if (bar.count == 0) {
bar.releasing = 0;
}
}
UNLOCK(bar.lock);
}
```

Previous Next

Solution of Exercise : 2

[Thanks to Saurabh Joshi for some of the suggestions.]

1. [30 points] For each of the memory reference streams given in the following, compare the cost of executing it on a bus-based SMP that supports (a) MESI protocol without cache-to-cache sharing, and (b) Dragon protocol. A read from processor N is denoted by rN while a write from processor N is denoted by wN. Assume that all caches are empty to start with and that cache hits take a single cycle, misses requiring upgrade or update take 60 cycles, and misses requiring whole block transfer take 90 cycles. Assume that all caches are writeback.

Solution:
Stream1: r1 w1 r1 w1 r2 w2 r2 w2 r3 w3 r3 w3

(a) MESI: read miss, hit, hit, hit, read miss, upgrade, hit, hit, read miss, upgrade, hit, hit. Total latency = 90+1+1+1+2*(90+60+1+1) = 397 cycles
(b) Dragon: read miss, hit, hit, hit, read miss, update, hit, update, read miss, update, hit, update. Total latency = 90+1+1+1+2*(90+60+1+60) = 515 cycles

Stream2: r1 r2 r3 w1 w2 w3 r1 r2 r3 w3 w1

(a) MESI: read miss, read miss, read miss, upgrade, readX, readX, read miss, read miss, hit, upgrade, readX. Total latency = 90+90+90+60+90+90+90+90+1+60+90 = 841 cycles
(b) Dragon: read miss, read miss, read miss, update, update, update, hit, hit, hit, update, update. Total latency = 90+90+90+60+60+60+1+1+1+60+60=573 cycles

Stream3: r1 r2 r3 r3 w1 w1 w1 w1 w2 w3

(a) MESI: read miss, read miss, read miss, hit, upgrade, hit, hit, hit, readX, readX. Total latency = 90+90+90+1+60+1+1+1+90+90 = 514 cycles
(b) Dragon: read miss, read miss, read miss, hit, update, update, update, update, update, update. Total latency=90+90+90+1+60*6=631 cycles

[For each stream for each protocol: 5 points]

2. [15 points] (a) As cache miss latency increases, does an update protocol become more or less preferable as compared to an invalidation based protocol? Explain.

Solution: If the system is bandwidth-limited, invalidation protocol will remain the choice. However, if there is enough bandwidth, with increasing cache miss latency, invalidation protocol will lose in importance.

(b) In a multi-level cache hierarchy, would you propagate updates all the way to the first-level cache? What are the alternative design choices?

Solution: If updates are not propagated to L1 caches, on an update the L1 block must be invalidated/retrieved to the L2 cache.

(c) Why is update-based protocol not a good idea for multiprogramming workloads running on SMPs?

Solution: Pack-rat. Discussed in class.

3. [20 points] Assuming all variables to be initialized to zero, enumerate all outcomes possible under sequential consistency for the following code segments.

(a) P1: A=1;
P2: u=A; B=1;
P3: v=B; w=A;

Solution: If u=1 and v=1, then w must be 1. So (u, v, w) = (1, 1, 0) is not allowed. All other outcomes are possible.

(b) P1: A=1;
P2: u=A; v=B;
P3: B=1;
P4: w=B; x=A;

Solution: Observe that if u sees the new value A, v does not see the new value of B, and w sees that new value of B, then x cannot see the old value of A. So (u, v, w, x) = (1, 0, 1, 0) is not allowed. Similarly, if w sees the new value of B, x sees the old value of A, u sees the new value of A, then v cannot see the old value B. So (1, 0, 1, 0) is not allowed, which is already eliminated in the above case. All other 15 combinations are possible.

(c) P1: u=A; A=u+1;
P2: v=A; A=v+1;

Solution: If v=A happens before A=u+1, then the final (u, v, A) = (0, 0, 1).
If v=A happens after A=u+1, then the final (u, v, A) = (0, 1, 2).
Since u and v are symmetric, we will also observe the outcome (1, 0, 2) in some cases.

(d) P1: fetch-and-inc (A)
P2: fetch-and-inc (A)

Solution: The final value of A is 2.

4. [30 points] Consider a quad SMP using a MESI protocol (without cache-to-cache sharing). Each processor tries to acquire a test-and-set lock to gain access to a null critical section. Assume that test-and-set instructions always go on the bus and they take the same time as the normal read transactions. The initial condition is such that processor 1 has the lock and processors 2, 3, 4 are spinning on their caches waiting for the lock to be released. Every processor gets the lock once, unlocks, and then exits the program. Consider the bus transactions related to the lock/unlock operations only.

(a) What is the least number of transactions executed to get from the initial to the final state? [10 points]

Solution: 1 unlocks, 2 locks, 2 unlocks (no transaction), 3 locks, 3 unlocks (no

transaction), 4 locks, 4 unlocks (no transaction). Notice that in the best possible scenario, the timings will be such that when someone is in the critical section no one will even attempt a test-and-set. So when the lock holder unlocks, the cache block will still be in its cache in M state.

(b) What is the worst-case number of transactions? [5 points]

Solution: Unbounded. While someone is holding the lock, other contending processors may keep on invalidating each other indefinite number of times.

(c) Answer the above two questions if the protocol is changed to Dragon. [15 points]

Solution: Observe that it is an order of magnitude more difficult to implement shared test-and-set locks (LL/SC-based locks are easier to implement) in a machine running an update-based protocol. In a straightforward implementation, on an unlock everyone will update the value in cache and then will try to do test-and-set. Observe that the processor which wins the bus and puts its update first, will be the one to enter the critical section. Others will observe the update on the bus and must abort their test-and-set attempts. While someone is in the critical section, nothing stops the other contending processors from trying test-and-set (notice the difference with test-and-test-and-set). However, these processors will not succeed in getting entry to the critical section until the unlock happens.

Least number is still 7. A test-and-set or an unlock involves putting an update on the bus.

Worst case is still unbounded.

5. [30 points] Answer the above question for a test-and-test-and-set lock for a 16-processor SMP. The initial condition is such that the lock is released and no one has got the lock yet.

Solution:MESI:

Best case analysis: 1 locks, 1 unlocks, 2 locks, 2 unlocks, ... This involves exactly 16 transactions (unlocks will not generate any transaction in the best case timing).

Worst case analysis: Done in the class. The first round will have (16 + 15 + 1 + 15) transactions. The second round will have (15 + 14 + 1 + 14) transactions. The last but one round will have (2 + 1 + 1 + 1) transactions and the last round will have one transaction (just locking of the last processor). The last unlock will not generate any transaction. If you add these up, you will get (1.5P+2)(P-1) + 1. For P=16, this is 391.

Dragon:

Best case analysis: Now both unlocks and locks will generate updates. So the total number of transactions would be 32.

Worst case analysis: The test & set attempts in each round will generate updates. The unlocks will also generate updates. Everything else will be cache hits. So the number of transactions is (16+1)+(15+1)+...+(1+1) = 152.

6. [10 points] If the lock variable is not allowed to be cached, how will the traffic of a test-and-set lock compare against that of a test-and-test-and-set lock?

Solution:In the worst case both would be unbounded.

7. [15 points] You are given a bus-based shared memory machine. Assume that the processors have a cache block size of 32 bytes and A is an array of integers (four bytes each). You want to parallelize the following loop.

```
for(i=0; i<17; i++) {
for (j=0; j<256; j++) {
A[j] = do_something(A[j]);
}
}
```

(a) Under what conditions would it be better to use a dynamically scheduled loop?

Solution: If runtime of do_something varies a lot depending on its argument value or if nothing is known about do_something.

(b) Under what conditions would it be better to use a statically scheduled loop?

Solution: If runtime of do_something is roughly independent of its argument value.

(c) For a dynamically scheduled inner loop, how many iterations should a processor pick each time?

Solution: Multiple of 8 integers (one cache block is eight integers).

8. [20 points] The following barrier implementation is wrong. Make as little change as possible to correct it.

```
struct bar_struct {
LOCKDEC(lock);
int count; // Initialized to zero
int releasing; // Initialized to zero
} bar;

void BARRIER (int P) {
LOCK(bar.lock);
bar.count++;
if (bar.count == P) {
bar.releasing = 1;
bar.count--;
}
else {
UNLOCK(bar.lock);
while (!bar.releasing);
LOCK(bar.lock);
bar.count--;
if (bar.count == 0) {
bar.releasing = 0;
```

```
}
}
UNLOCK(bar.lock);
}
```

Solution: There are too many problems with this implementation. I will not list them here. The correct barrier code is given below which requires addition of one line of code. Notice that the releasing variable nicely captures the notion of sense reversal.

```
void BARRIER (int P) {
while (bar.releasing); // New addition
LOCK(bar.lock);
bar.count++;
if (bar.count == P) {
bar.releasing = 1;
bar.count--;
}
else {
UNLOCK(bar.lock);
while (!bar.releasing);
LOCK(bar.lock);
bar.count--;
if (bar.count == 0) {
bar.releasing = 0;
}
}
UNLOCK(bar.lock);
}
```

Previous Next