Module 18: "TLP on Chip: HT/SMT and CMP"

Lecture 41: "Case Studies: Intel Montecito and Sun Niagara"

TLP on Chip: HT/SMT and CMP

- Intel Montecito
- Features
- Overview
- Dual threads
- Thread urgency
- Core arbiter
- Power efficiency
- Foxton technology
- Die photo
- Sun Niagara OR Ultrasparc T1
- Features
- Pipeline details
- Cache hierarchy
- Thread selection

Previous Next

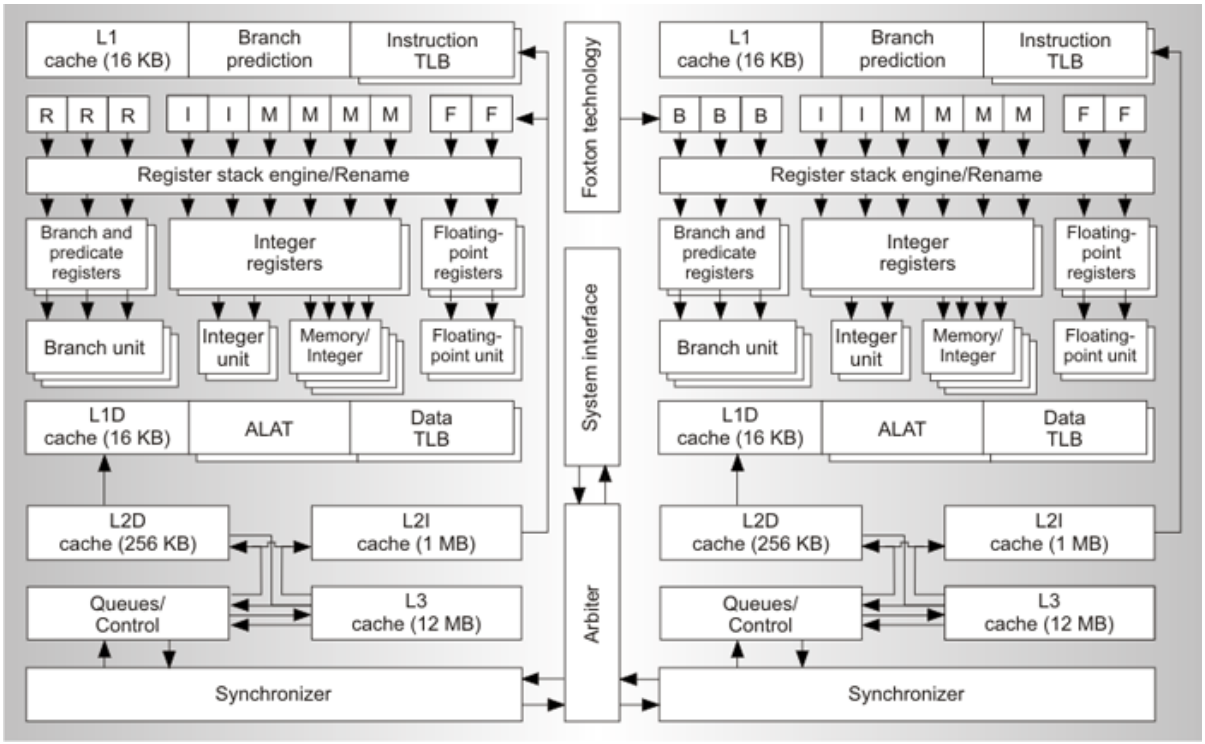
Module 18: "TLP on Chip: HT/SMT and CMP"
Lecture 41: "Case Studies: Intel Montecito and Sun Niagara"

Intel Montecito

Features

- Dual core Itanium 2, each core dual threaded
- 1.7 billion transistors, 21.5 mm x 27.7 mm die
- 27 MB of on-chip three levels of cache
  - Not shared among cores
- 1.8+ GHz, 100 W
- Single-thread enhancements
  - Extra shifter improves performance of crypto codes by 100%
  - Improved branch prediction
  - Improved data and control speculation recovery
  - Separate L2 instruction and data caches buys 7% improvement over Itanium2; four times bigger L2I (1 MB)
  - Asynchronous 12 MB L3 cache

Overview



Reproduced from IEEE Micro

Dual threads

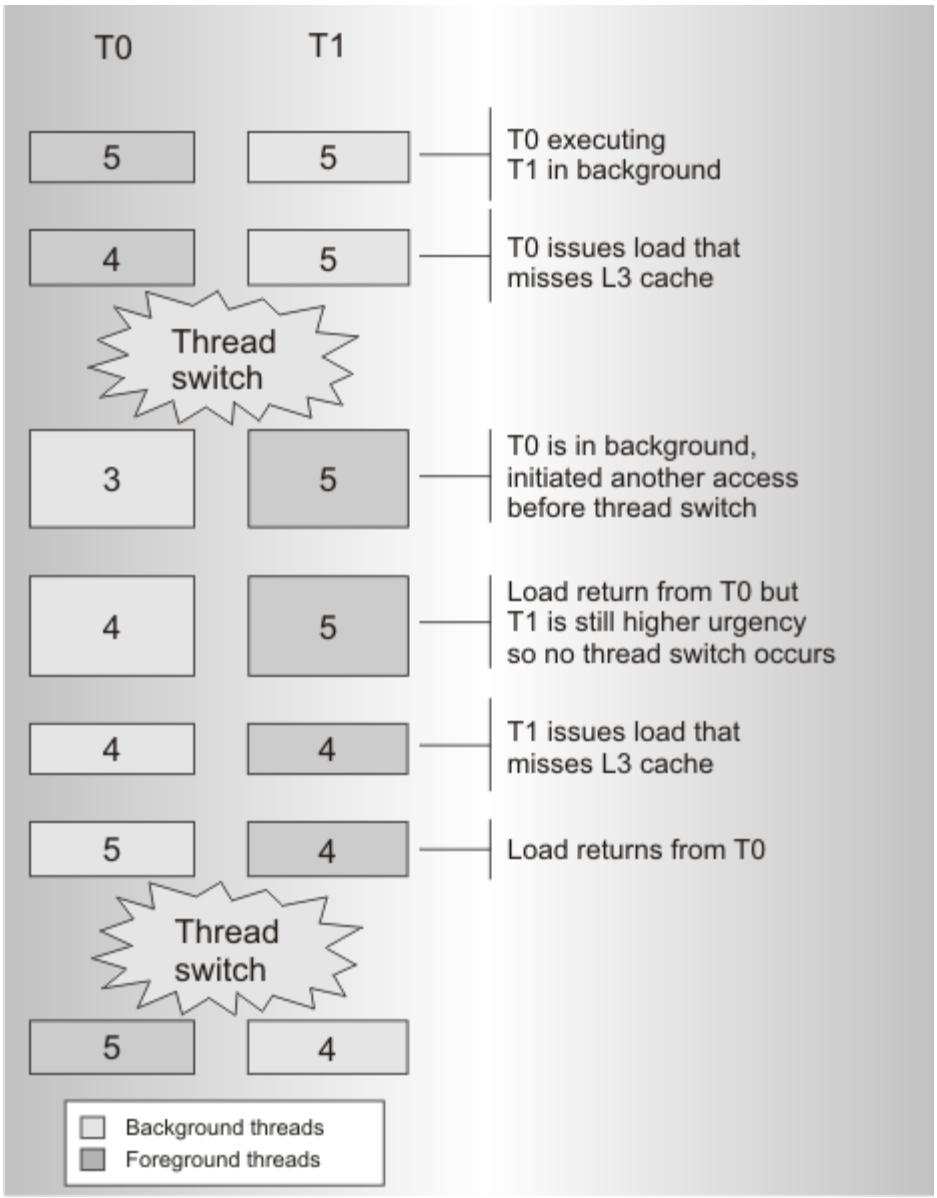
- SMT only for cache, not for core resources
  - Simulations showed high resource utilization at core level, but low utilization of cache
  - Branch predictor is still shared but use thread id tags
  - Thread switch is implemented by flushing the pipe
    - More like coarse-grain multithreading
  - Five thread switch events

- L3 cache miss (immense impact on in-order pipe)/ L3 cache refill
- Quantum expiry
- Spin lock/ ALAT invalidation
- Software-directed switch
- Execution in low power mode

Previous Next

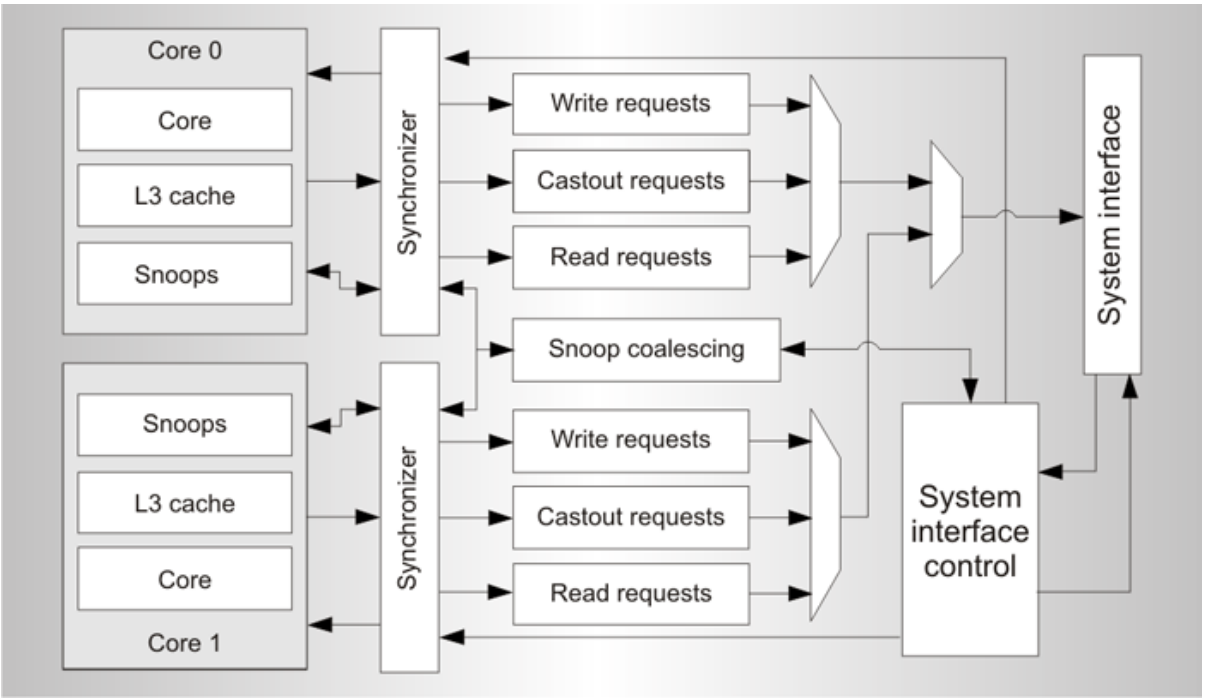
Thread urgency

- Each thread has eight urgency levels
  - Every L3 miss decrements urgency by one
  - Every L3 refill increments urgency by one until urgency reaches 5
  - A switch due to time quantum expiry sets the urgency of the switched thread to 7
  - Arrival of asynchronous interrupt for a background thread sets the urgency level of that thread to 6
  - Switch from L3 miss requires urgency level to be compared also



Reproduced from IEEE Micro

Core arbiter



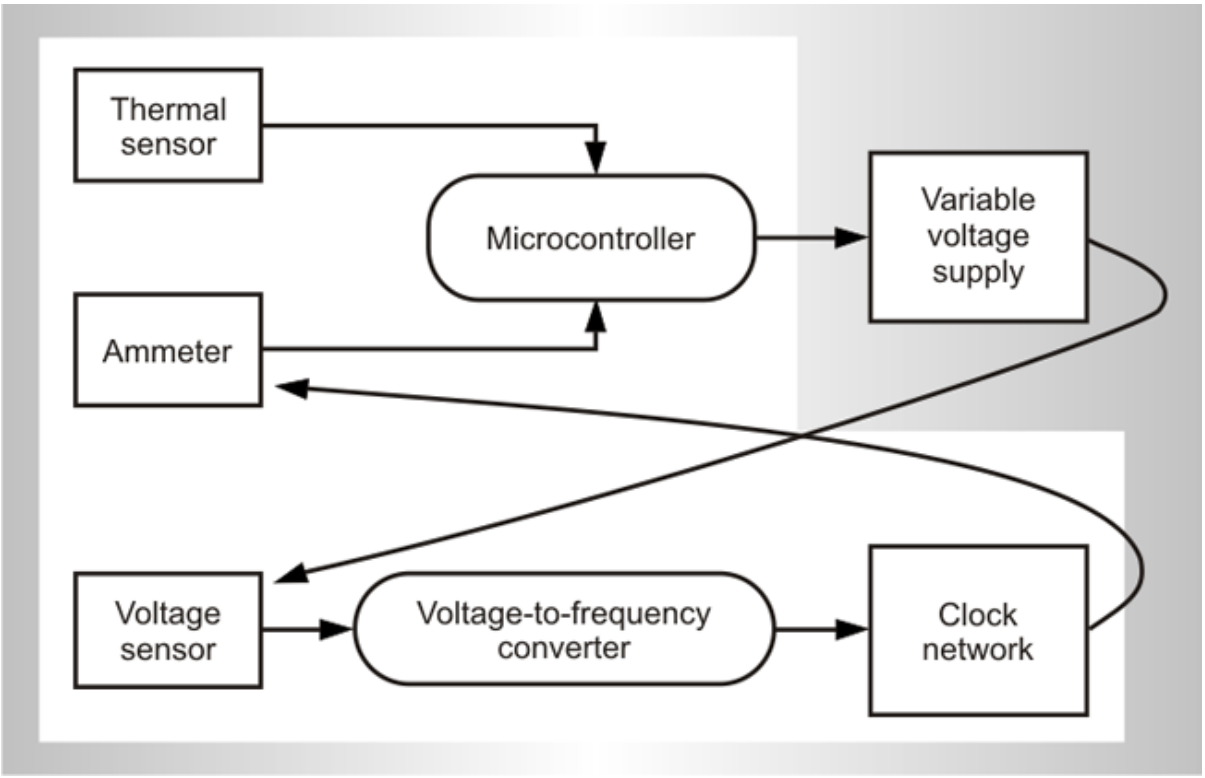
Reproduced from IEEE Micro

Previous Next

Power efficiency

- Foxton technology
  - Blind replication of Itanium 2 cores at 90 nm would lead to roughly 300 W peak power consumption (Itanium 2 consumes 130 W peak at 130 nm)
  - In case of lower than the ceiling power consumption, the voltage is increased leading to higher frequency and performance
- 10% boost for enterprise applications
  - Software or OS can also dictate a frequency change if power saving is required
  - 100 ms response time for the feedback loop
  - Frequency control is achieved by 24 voltage sensors distributed across the chip: the entire chip runs at a single frequency (other than asynchronous L3)
  - Clock gating found limited application in Montecito

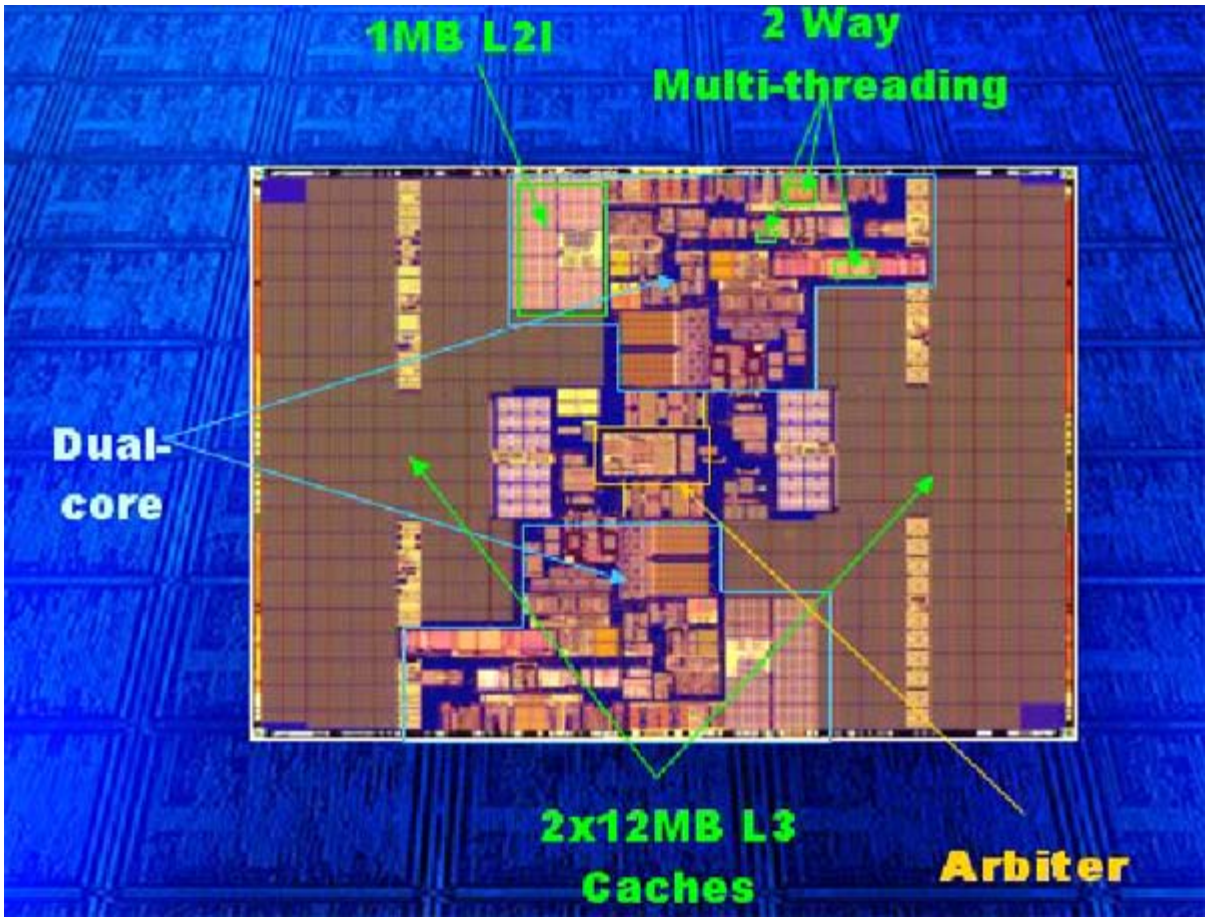
Foxton technology



Reproduced from IEEE Micro

- Embedded microcontroller runs a real-time scheduler to execute various tasks

Die photo



Previous Next

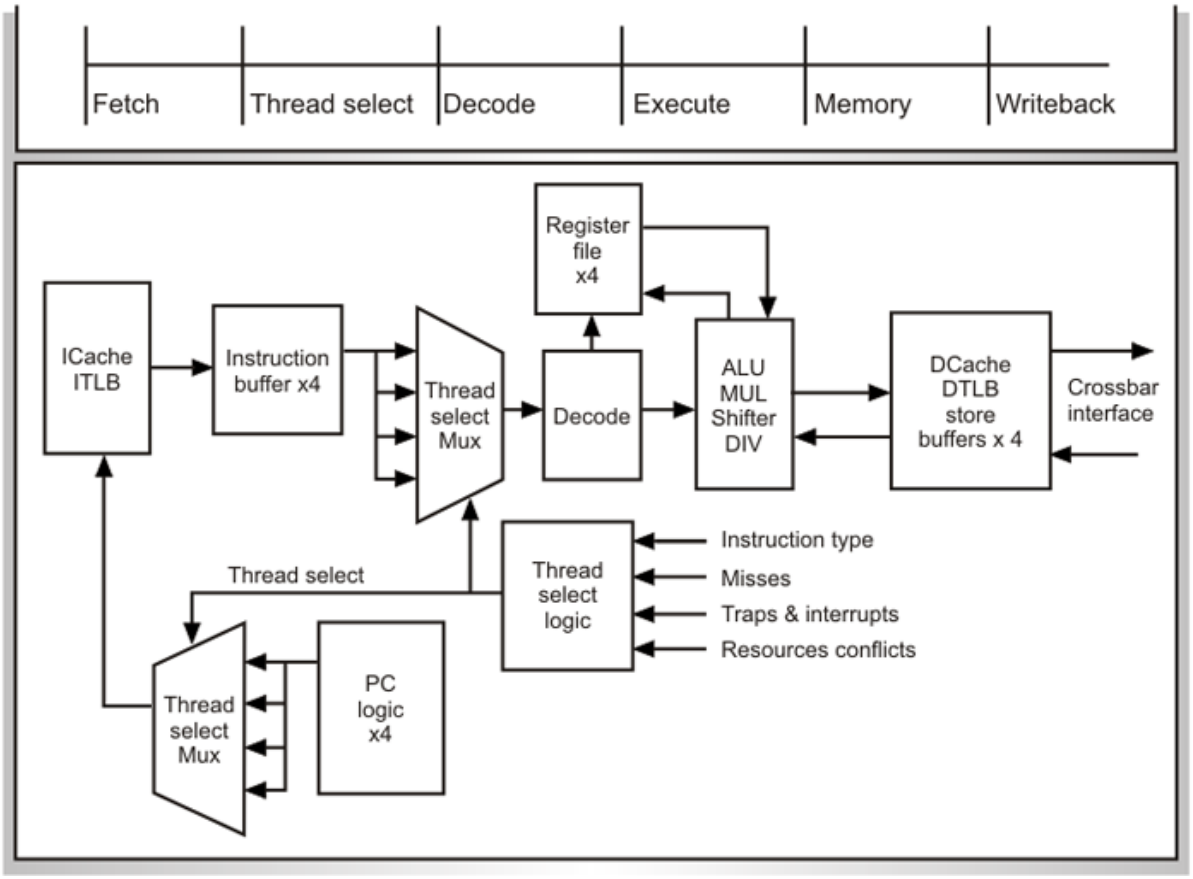
Module 18: "TLP on Chip: HT/SMT and CMP"
Lecture 41: "Case Studies: Intel Montecito and Sun Niagara"

Sun Niagara OR Ultrasparc T1

Features

- Eight pipelines or cores, each shared by 4 threads
  - 32-way multithreading on a single chip
  - Starting frequency of 1.2 GHz, consumes 60 W
  - Shared 3 MB L2 cache, 4-way banked, 12-way set associative, 200 GB/s bandwidth
  - Single-issue six stage pipe
  - Target market is web service where ILP is limited, but TLP is huge (independent transactions)
    - Throughput matters

Pipeline details



Reproduced from IEEE Micro

- Four threads share a six-stage pipeline
  - Shared L1 caches and TLBs
  - Dedicated register file per thread
  - Fetches two instructions every cycle from a selected thread
  - Thread select logic also determines which thread's instruction should be fed into the pipe
  - Although pipe is in-order, there is an 8-entry store buffer per thread (why?)
  - Threads may run into structural hazards due to limited number of FUs

- Divider is granted to the least recently executed thread

Previous Next

Cache hierarchy

- L1 instruction cache
  - 16 KB / 4-way / 32 bytes / random replacement
  - Fetches two instructions every cycle
  - If both instructions are useful, next cycle is free for icache refill
- L1 data cache
  - 8 KB / 4-way / 16 bytes/ write-through, no-allocate
  - On avearge 10% miss rate for target benchmarks
  - L2 cache extends the tag to maintain a directory for keeping the core L1s coherent
- L2 cache is writeback with silent clean eviction

Thread selection

- Based on long latency events such as load, divide, multiply, branch
- Also based on pipeline stalls due to cache misses, traps, or structural hazards
- Speculative load dependent issue with low priority

Previous

Solution of Exercise : 1

1. [10 points] Suppose you are given a program that does a fixed amount of work, and some fraction s of that work must be done sequentially. The remaining portion of the work is perfectly parallelizable on P processors. Derive a formula for execution time on P processors and establish an upper bound on the achievable speedup.

Solution: Execution time on P processors, T(P) = sT(1) + (1-s)T(1)/P. Speedup = 1/(s + (1-s)/P). Upper bound is achieved when P approaches infinity. So maximum speedup = 1/s. As expected, the upper bound on achievable speedup is inversely proportional to the sequential fraction.

2. [40 points] Suppose you want to transfer n bytes from a source node S to a destination node D and there are H links between S and D. Therefore, notice that there are H+1 routers in the path (including the ones in S and D). Suppose W is the node-to-network bandwidth at each router. So at S you require n/W time to copy the message into the router buffer. Similarly, to copy the message from the buffer of router in S to the buffer of the next router on the path, you require another n/W time. Assuming a store-and-forward protocol total time spent doing these copy operations would be (H+2)n/W and the data will end up in some memory buffer in D. On top of this, at each router we spend R amount of time to figure out the exit port. So the total time taken to transfer n bytes from S to D in a store-and-forward protocol is (H+2)n/W+(H+1)R. On the other hand, if you assume a cut-through protocol the critical path would just be n/W+(H+1)R. Here we assume the best possible scenario where the header routing delay at each node is exposed and only the startup n/W delay at S is exposed. The rest is pipelined. Now suppose that you are asked to compare the performance of these two routing protocols on an 8x8 grid. Compute the maximum, minimum, and average latency to transfer an n byte message in this topology for both the protocols. Assume the following values: W=3.2 GB/s and R=10 ns. Compute for n=64 and 256. Note that for each protocol you will have three answers (maximum, minimum, average) for each value of n. Here GB means 10^9 bytes and not 2^30 bytes.

Solution: The basic problem is to compute the maximum, minimum, and average values of H. The rest is just about substituting the values of the parameters. The maximum value of H is 14 while the minimum is 1. To compute the average, you need to consider all possible messages, compute H for them, and then take the average. Consider S=(x0, y0) and D=(x1, y1). So H = |x0-x1| + |y0-y1|. Therefore, average H = (sum over all x0, x1, y0, y1 |x0-x1| + |y0-y1|)/(64*63), where each of x0, x1, y0, y1 varies from 0 to 7. Clearly, this is same as (sum over x0, x1 |x0-x1| + sum over y0, y1 |y0-y1|)/63, which in turn is equal to 2*(sum over x0, x1 |x0-x1|)/63 = 2*(sum over x0=0 to 7, x1=0 to x0 (x0-x1)+ sum over x0=0 to 7, x1=x0+1 to 7 (x1-x0))/63 = 16/3.

3. [20 points] Consider a simple computation on an nxn double matrix (each element is 8 bytes) where each element A[i][j] is modified as follows. A[i][j] = A[i][j] - (A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1])/4. Suppose you assign one matrix element to one processor (i.e. you have n^2 processors). Compute the total amount of data communication between processors.

Solution: Each processor requires the four neighbors i.e. 32 bytes. So total amount of data communicated is 32n^2.

4. [30 points] Consider a machine running at 10^8 instructions per second on some workload with the following mix: 50% ALU instructions, 20% load instructions, 10% store instructions, and 20% branch instructions. Suppose the instruction cache miss rate is 1%, the writeback data cache miss rate is 5%, and the cache line size is 32 bytes. Assume that a store miss requires two cache line transfers, one to load the newly updated line and one to replace the dirty line at a later point in time. If the machine

provides a 250 MB/s bus, how many processors can it accommodate at peak bus bandwidth?

Solution: Let us compute the bandwidth requirement of the processor per second. Instruction cache misses 10^6 times transferring 32 bytes on each miss. Out of 20*10^6 loads 10^6 miss in the cache transferring 32 bytes on each miss. Out of 10^7 stores 5*10^5 miss in the cache transferring 64 bytes on each miss. Thus, total amount of data transferred per second is 96*10^6 bytes. Thus at most two processors can be supported on a 250 MB/s bus.

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

Solution of Exercise : 3

1. [0 points] Please categorize yourself as either "CS698Z AND CS622" or "CS622 ONLY".

2. [5+5] Consider a 512-node system. Each node has 4 GB of main memory. The cache block size is 128 bytes. What is the total directory memory size for (a) bitvector, (b) DiriB with i=3?

Solution: (a) Number of cache blocks per node = 2^25 and 512 bits per directory entry. So total directory memory size = (2^25)*64*512 bytes = 1 TB.
(b) Each directory entry is 27 bits. So total directory memory size = 54 GB.

3. [5+5+5+10] For a simple two-processor NUMA system, the number of cache misses to three virtual pages X, Y, Z is as follows.

Page X: P0 has 14 misses, P1 has 11 misses. P1 takes the first miss.
Page Y: P0 has zero misses, P1 has 18 misses.
Page Z: P0 has 15 misses, P1 has 9 misses. P0 takes the first miss.

The remote to local miss latency ratio is four. Evaluate the aggregate time spent in misses by the two processors in each of the following policies. Assume that a local miss takes 400 cycles.

(a) First touch placement.
(b) All three pages on P0.
(c) All three pages on P1.
(d) Best possible application-directed static placement i.e. a one-time call to the OS to place the three pages.

Solution: (a) First touch: Page X is local to P1, page Y is local to P1, page Z is local to P0. P0 spends (14*1600+15*400) cycles or 28400 cycles. P1 spends (11*400+18*400+9*1600) cycles or 26000 cycles. Aggregate: 54400 cycles.

(b) All three pages on P0: P0 spends (14*400+15*400) cycles or 11600 cycles. P1 spends (11*1600+18*1600+9*1600) cycles or 60800 cycles. Aggregate: 72400 cycles.

(c) All three pages on P1: P0 spends (14*1600+15*1600) cycles or 46400 cycles. P1 spends (11*400+18*400+9*400) cycles or 15200 cycles. Aggregate: 61600 cycles.

(d) To determine the best page-to-node affinity, we need to compute the latency for both the choices for each page. This is shown in the following table. Since P0 doesn't access Y, it should be placed on P1. Therefore, Y is not included in the following table.

Page	Home	Latency of P0	Latency of P1	Aggregate
X	P0	5600	17600	23200
X	P1	22400	4400	26800
Z	P0	6000	14400	20400
Z	P1	24000	3600	27600

Thus X and Z both should be placed on P0. Y should be placed on P1. The aggregate latency of this is 50800 cycles. As you can see, this is about 7% better than first touch.

4. [30] Suppose you want to transpose a matrix in parallel. The source matrix is A and the destination matrix is B. Both A and B are decomposed in such a way that each node gets a chunk of consecutive rows. Application-directed page placement is used to map the pages belonging to each chunk in the local memory of the respective nodes. Now the transpose can be done in two ways. The first algorithm, known as "local read algorithm", allows each node to transpose the band of rows local to it. So naturally this algorithm involves a large fraction of remote writes to matrix B. Assume that the band is wide enough so that there is no false sharing when writing to matrix B. The second algorithm, known as "local write algorithm", allows each node to transpose a band of columns of A such that all the writes to B are local. Naturally, this algorithm involves a large number of remote reads in matrix A. Assume that the algorithms are properly tiled so that the cache utilization is good. In both the cases, before doing the transpose, every node reads and writes to its local segment in A and after doing the transpose every node reads and writes to its local segment in B. Assuming an invalidation-based cache coherence protocol, briefly but clearly explain which algorithm is expected to deliver better performance. How much synchronization does each algorithm require (in terms of the number of critical sections and barriers)? Assume that the caches are of infinite capacity and that a remote write is equally expensive in all respects as a remote read because in both cases the retirement is held up for a sequentially consistent implementation.

Solution: Analysis of local read algorithm: Before the algorithm starts, the band of rows to be transposed by a processor is already in its cache. Each remote write to a cache block of B involves a 2-hop transaction (requester to home and back) without involving any invalidation. After transpose each processor has a large number of remote cache blocks in its cache. Here a barrier is needed before a processor is allowed to work on its local rows of B. After the barrier each cache read or write miss to B involves a 2-hop intervention (local home to owner and back). So in this algorithm, each processor suffers from local misses before transpose, 2-hop misses during transpose, and 2-hop misses after transpose.

Analysis of local write algorithm: Before the algorithm starts, the band of columns to be transposed by a processor is quite distributed over the system. In fact, 1/P portion of the column would be in the local cache. Before the transpose starts, a barrier is needed. In the transpose phase, each cache miss of A involves a 2-hop transaction (requester to home's cache and back). Each cache miss to B is a local miss. After the transpose the local band of rows of B is already in the cache of each processor. So they enjoy hits in this phase. So in this algorithm, each processor suffers from local misses before transpose, 2-hop misses and local misses during transpose, and hits after transpose.

Both the algorithms have the same number of misses, but the local write algorithm has more local misses and less remote misses. Both have the same synchronization requirement. Local write algorithm is better and that's what SPLASH-2 FFT implements. Cheers!

5. [5+5] If a compiler reorders accesses according to WO and the underlying processor is SC, what is the consistency model observed by the programmer? What if the compiler produces SC code, but the processor is RC?

Solution: WO compiler and SC hardware: programmer sees WO because an SC processor will execute whatever is presented to it in the intuitive order. But since the instruction stream presented to it is already WO, it cannot do any better.

SC compiler and RC hardware: programmer sees RC.

6. [5+5] Consider the following piece of code running on a faulty microprocessor system which does not preserve any program order other than true data and control dependence order.

LOCK (L1)
load A

```
store A
UNLOCK (L1)
load B
store B
LOCK (L1)
load C
store C
UNLOCK (L1)
```

(a) Insert appropriate WMB and MB instructions to enforce SC. Do not over-insert anything.
(b) Repeat part (a) to enforce RC.

Solution:
(a)

```
MB
LOCK(L1)
MB
load A
store A
WMB                 // Need to hold the unlock back before the store to A is done
UNLOCK (L1)
MB
load B
store B
MB
LOCK (L1)
MB
load C
store C
WMB                 // Need to hold the unlock back before the store to C is done
UNLOCK (L1)
MB
```

(b)

```
LOCK (L1)
MB                      // This MB wouldn't be needed if the processor was not faulty; you lose some RC advantage
load A
store A
WMB
UNLOCK (L1)
load B
store B
LOCK (L1)
MB                      // This MB wouldn't be needed if the processor was not faulty; you lose some RC advantage
load C
store C
WMB
UNLOCK (L1)
```

7. [5+10] Consider implementing a directory-based protocol to keep the private L1 caches of the cores

in a chip-multiprocessor (CMP) coherent. Assume that the CMP has a shared L2 cache. The most natural way of doing it is to attach a directory to the tag of each L2 cache block. An L1 cache miss gets forwarded to the L2 cache. The directory entry is looked up in parallel with the L2 cache block. Does this implementation suit an inclusive hierarchy or exclusive hierarchy? Explain. If the cache block sizes of the L1 cache and the L2 cache are different, explain briefly how you will manage the state of the L2 cache block on receiving a writeback from the L1 cache. Do not assume per sector directory entry in this design.

Solution: This is typical design for an inclusive hierarchy. For exclusive hierarchy, you cannot keep a directory entry per L2 cache block. Instead, you must have a separate large directory store holding the directory states of all the blocks in both L2 and L1 caches. If the L1 and L2 block sizes are different, you need to keep a count of dirty L1 sectors with the owner; otherwise you won't be able to turn off the L1M state even after the last dirty sector is written back by the L1 cache. This has no correctness problem, but would involve unnecessary interventions to L1 caches. Fortunately, you don't need any extra bits for doing this for a medium to large scale CMP if the ratio of L2 to L1 block sizes is not too large. Assume a p-core CMP with L2 to L1 block size ratio of k. So you need log(p) bits to store the owner when L1M state is set,
log(k) bits to store the dirty sector count, and p bits to store the sharer vector if the block is only shared. So as long as p >= log(p)+log(k), we can pack the owner and dirty sector count into the sharer vector. This condition corresponds to k <= 2^{p-log(p)} or k <= (2^p)/p. The number on the right-hand side grows very fast with p. For eight cores onward we are in the safe zone.

Previous Next

Self-assessment Exercise

These problems should be tried after module 05 is completed.

1. Consider the following memory organization of a processor. The virtual address is 40 bits, the physical address is 32 bits, the page size is 8 KB. The processor has a 4-way set associative 128-entry TLB i.e. each way has 32 sets. Each page table entry is 32 bits in size. The processor also has a 2-way set associative 32 KB L1 cache with line size of 64 bytes.

(A) What is the total size of the page table?
(B) Clearly show (with the help of a diagram) the addressing scheme if the cache is virtually indexed and physically tagged. Your diagram should show the width of TLB and cache tags.
(C) If the cache was physically indexed and physically tagged, what part of the addressing scheme would change?

2. A set associative cache has longer hit time than an equally sized direct-mapped cache. Why?

3. The Alpha 21264 has a virtually indexed virtually tagged instruction cache. Do you see any security/protection issues with this? If yes, explain and offer a solution. How would you maintain correctness of such a cache in a multi-programmed environment?

4. Consider the following segment of C code for adding the elements in each column of an NxN matrix A and putting it in a vector x of size N.

```
for(j=0;j<N;j++) {
for(i=0;i<N;i++) {
x[j] += A[i][j];
}
}
```

Assume that the C compiler carries out a row-major layout of matrix A i.e. A[i][j] and A[i][j+1] are adjacent to each other in memory for all i and j in the legal range and A[i][N-1] and A[i+1][0] are adjacent to each other for all i in the legal range. Assume further that each element of A and x is a floating point double i.e. 8 bytes in size. This code is executed on a modern speculative out-of-order processor with the following memory hierarchy: page size 4 KB, fully associative 128-entry data TLB, 32 KB 2-way set associative single level data cache with 32 bytes line size, 256 MB DRAM. You may assume that the cache is virtually indexed and physically tagged, although this information is not needed to answer this question. For N=8192, compute the following (please show all the intermediate steps). Assume that every instruction hits in the instruction cache. Assume LRU replacement policy for physical page frames, TLB entries, and cache sets.

(A) Number of page faults.
(B) Number of data TLB misses.
(C) Number of data cache misses. Assume that x and A do not conflict with each other in the cache.
(D) At most how many memory operations can the processor overlap before coming to a halt? Assume that the instruction selection logic (associated with the issue unit) gives priority to older instructions over younger instructions if both are ready to issue in a cycle.

5. Suppose you are running a program on two machines, both having a single level of cache hierarchy (i.e. only L1 caches). In one machine the cache is virtually indexed and physically tagged while in the other it is physically indexed and physically tagged. Will there be any difference in cache miss rates

when the program is run on these two machines?

Previous