

Overview

- performance issues
 - motivation
 - performance metrics and tools
 - ◆ hardware event (performance) counters
 - ★ issues: event attribution and interpretation
 - ◆ profiling and instruction-based sampling
 - multicore performance considerations
 - ◆ strategies
 - ◆ synchronization
 - case study: pointer chasing
 - tools usage: profiler
- intermission
- scalable synchronization methods
 - locks: ticket, MCS, CLH
 - barriers: central, combining tree, scalable tree
- a look at Assignment 1

Motivation – Why Doesn't My Application Run faster ?!?

- many factors can potentially affect multicore program performance
 - e.g. stalls (CPU dependencies, L1/L2 cache / TLB misses), serial computation fraction, load imbalance, synchronization overheads, memory coherency overheads
 - slightly easier for simple, highly threaded cores (e.g. T2):
 - ◆ can issue only 1 instruction (per group) per cycle: thus maximum instruction throughput @ 1.2 GHz is $16 \times 1.2 \times 10^9 = 1.9 \times 10^{10}$ instrns/sec
 - but:
 - ◆ L1/L2 cache miss penalty may vary due to crossbar/memory controller bank conflicts or use of prefetch
 - ◆ even with sufficient threads, stalls may only be partially hidden
- counts of associated events may give us some insights
- how do I improve programming performance?
 - need then to relate significant events to source code and key data structures

Hardware Performance Counter Issues

- limited number of counter registers; can be solved by statistical multiplexing techniques (or just use multiple runs)
- event counts do not necessarily indicate proportional loss of execution time
 - can also have counters to measure CPU stall cycles lost for various causes (e.g. cache miss, store buffer full)
 - more useful for performance analysis (2006 seminar; p. 7-9, 13–14)
 - however, problem of **attribution** when several causes occur simultaneously
- the *meaning* of some events may be unclear
 - e.g. on the T2, we have an ‘Instr_sw’ event (??)
 - e.g. on a Core 2, L1 cache reference count exceeded total instruction count! (COMP2300 Ass 3, 2007)
 - *Hardware event counters were designed by hardware engineers for the purposes of hardware engineers...*
 - significant efforts to reverse-engineer this information!

Profiling: Attributing Hardware Events to Source Code

- traditional sampling (e.g. `gprof`) uses timer interrupts to sample where the PC was in the application when the interrupt
 - can give a statistically accurate profile of how much time was spent in each function
- recall the counter registers can similarly be programmed to generate an interrupt – can similarly profile any event
- but how can we relate say cache misses to a particular array or lock variable?
 - only possible if we can recover the instruction causing the miss (or recover the virtual address causing it, directly)
 - problem: interrupts take time (e.g. 10–20 cycles) to be delivered, and the PC where program stopped is inaccurate (called `skid`)
 - made worse with deep pipelining and O-O-E (conversely, is possible to do on the T2)
 - x86-64 Barcelona (and newer) processors incorporate instruction-based sampling, where the hardware statically selects an instruction for sampling, and can report associated events that it caused

Multicore Performance Considerations

- constructive sharing: many threads of one binary share stack and other data areas
- destructive sharing: when running many copies of same binaries:
the stacks are aligned and can cause high L1\$ misses
- general application analysis strategy
 - calculate IPC or other suitable metric (e.g. GFLOPs); compare with what is possible (in theory)
 - identify the main causes for stalls (slow (multi-cycle) instructions, L1\$ misses)
 - identify hotspots (functions where most time is spent)
- a stall in a hardware thread of x cycles may be hidden, provided at least 1 of the other 3 in the group have no stalls for x cycles
i.e. an average 'stall budget' of 3 stalls per instruction can still be OK *provided* we have enough software threads as hardware threads
- e.g. Table 4 from Calculating Processor Utilisation ... T2 Performance Counters
in this case, reducing L1\$ misses the only useful things to do
- high number of TLB misses: can reduce by increasing the page size

Multicore Performance Considerations: Synchronization

- memory coherency costs in a single chip (CMP) system typically much less than in a multi-socket system
 - e.g. L1\$ cache lines invalidations within crossbar on T2 are much cheaper than L2\$ invalidations across a memory system backplane
 - however (on T2), cost in cycles of an atomic operation comparable
 - better to spin using a normal load instr'n, rather than on an atomic instruction
- most locks are **adaptive**: spin for a given time interval
 - consider increasing the time interval: it is more likely the lock holder will be running on a CMP
- identifying **hot locks**' (however costly in a single chip (CMP) system, typically much more on a multi-socket)

```
plockstat -A command arg...
```
- for large numbers of cores/threads, use more scalable synchronization algorithms
...

Cast Study: Pointer Chasing

- pointer chasing: `for (p = *p0; p != p0; p = *p);` (unrolled 10×)

ring size	64KB	2MB	128MB
tsc	3,354,371	23,943,498	183,961,584
Instr_cnt	1,200,907	1,204,696	1,204,479
Instr_ld	1,000,642	1,000,852	1,000,806
Instr_other	100,456	100,888	101,771
Br_completed	100,491	100,270	100,150
Br_taken	100,367	100,157	100,105
Idle_strands	1,669,750	16,832,180	142,418,525
DC_miss		1,000,094	1,000,155
L2_dmiss_ld			1,000,028
DTLB_miss		15,643	15,644
DTLB_HWTW_ref_L2		15,641	15,650
TLB_miss		15,641	15,640
CPU_ld_to_PCX		1,000,099	1,000,048
MMU_ld_to_PCX		15,634	15,658

- provided there are few slow instructions, `Instr_cnt + Idle_strands = tsc`

credits: *Hardware Performance Counters* by Richard Smith, Sun

Using a profiling Tool: Solaris Collect on the T2

- e.g.

```
$ collect -h Idle_strands,on,L2_dmiss_ld,on -p on -d /tmp linpack -v 2 2000
Creating experiment database /tmp/test.1.er ...
compute the Linpack benchmark, for N=2000 with NB=32 (alg version 2) using 16 threads
Linpack benchmark with N=2000 in 6.52719s @ 817 MFLOPS
PASSEd residual check: 2.02e-02
```

- some overhead: runs about 5% faster normally

- almost all time in `_${diA14}.matMult`, 12% was in OpenMP waits, 20M L2\$ misses;
33G Idle_strand events

- default sampling frequency 10007 events

- OpenMP support: (work + wait) for each function

pseudo-functions such as `<OMP-expliict-barrier>`

- for hardware event counter information only:

```
cputrack -c Idle_strands,L2_dmiss_ld linpack -v 2 2000
```

shows .3G Idle_strand events and 150K L2\$ misses per thread per sec

Synchronization: Locking

- purpose: ensure that only a single thread gets access to a 'critical region'
 - hence ensures consistent updates to the protected shared data object
- simplest scheme is Test & Set Lock (tight loop with an atomic instruction):
 - initialize: `lockword = 0;`
 - acquire: tight loop until atomic 'test-and-set' instruction on `lockword` returns 1
 - release: `lockword = 0;` (via normal store instruction)

Why does this give poor performance (when many threads try to access)?

- Ticket Lock: analogous to protocol of buying seafood at Woolworths
 - initialize: `ticketNumber = screenNumber = 0;`
 - acquire: (atomic)
 - `myTicket = fetchAndIncrement(&ticketNumber);`
 - wait until (`myTicket == screenNumber`)
 - release: `screenNumber++;`
- can use 'backoff' schemes to reduce number of atomic instructions used, when heavily contended

Synchronization: MCS Lock

- why is the Ticket Lock better than the Test & Set Lock?
 - still problems under contention: large amount of cache line invalidations due to all threads accessing 'scoreboard' variable `screenNumber`
 - can we restrict the threads accessing their 'scoreboard' to just 2?
- MCS lock: Woolworths analogy: instead of watching a screen, the person just ahead tells you when they are served
 - initialize: set a wait queue `L` to empty (nodes have `lockWord` and `next` fields)
 - acquire: obtain next node `I`; `I->next = NULL, I->locked = 0`
(atomic) `L_old, L = L, I;`
`if (L_old != NULL) I->locked = 1, L_old->next = I;`
`wait until (I->locked == 0)`
 - release: uses same `I` as for acquire
`if (I->next == NULL) //currently only one in queue`
(atomic) `if (L==I) L = NULL, return;`
`'wait until (I->next != NULL) //newcomer arrives`
`I->next->locked = 0; //notify next person`
- illustration: see John Mellor-Crummey's Slides, p15-20

Synchronization: CLH Lock

- possible to achieve same effect with a single 'fetch-and-store- atomic':
 - initialize: wait queue **L** and per-thread node **I** point to some non-empty node
 - acquire: `I->wait = 1; (atomic) I->prev, L = L, I`
wait until `(I->prev->wait = 0)`
 - release: `p = I->prev;`
`I->wait = 0;`
`I` now points to `p`
- performance: see Xi Yang's slides, p 16-19 (on 4-socket dual core Opteron
`fremont.anu.edu.au`)

Synchronization: Barriers

- wait for *all* to arrive at same point in computation before proceeding (none may leave until all have arrived)
- central barrier with p threads (initialize **ctr** to p , **sense** to 0)
 - `localSense = sense;`
`if (atomic_decrement(ctr) == 1) ctr = p, sense = !sense;`
`// last to reach`
`wait until (localSense != sense)`
- **sense** is required for repeated barriers; gets toggled between
- caution: deadlock occurs if 1 thread does not participate!
- problems:
 - most processors do not support atomic decrement of a memory location
 - ◆ e.g. on SPARC, must use a tight loop with a compare-and-swap instruction
 - not scalable: p atomic decrements per barrier

Synchronization: Combining Tree Barrier

- each pair of threads points to a leaf node in a tree
 - each node has a **ctr** (initialized to 2), and a **sense** flag
- algorithm: (each thread has **threadSense** flag)
 - `if (atomic_decrement(ctr) == 1)`
 - repeat this step on parent node
 - `ctr = 2, sense = !sense`
 - wait until `threadSense == sense`
 - then as leaving the barrier, `threadSense = !threadSense`
- notes:
 - last thread to reach each node continues up the tree
 - thread that reaches root begins 'wakeup' (reversing **sense**)
 - upon wakeup, a thread releases siblings at each node along path
- performance: $2 \times$ the atomic operations, but can distribute memory locations (e.g. across different L2\$ banks) (see Xi Yang's slides, p 26-28)
 - atomics can be avoided by the scalable tree barrier! (replace **cnt** with 2 flags)
 - note: 8-byte word can be used for atomicless barrier ($p \leq 8$)