1. Performance/Benchmarking

Be familiar with the general idea of performance counters and interpreting perf results. Perf the fastest. Valgrind is 20 times slower. Gprof is as fast as perf. (different results using function entry instead if exact instruction count of sampling.)

Benchmark choice: it should match what you plan to do with the computer.

Know a little about the difference between integer benchmarks and floating point (integer have more random/ unpredictable behavior with lots of conditionals; floating point are often regular looped strides over large arrays or data sets)

Be familiar with concept of skid. Cause of skid is because in general the processor not being able to stop fast enough.

2. Power(lecture 5)

Know the CMOS Power equation P=(1/2)*cv^2*f

Energy, Energy Delay, Energy Delay Squared E(total)=(Pdynamic +Pstatic)*t, ED=Energy*delay, EDD=Energy*Delay*Delay

Idle Power Question

(a) Which benchmark causes the cores to use the highest average power?

Matrix matrix multiply

(b) Which benchmark causes the RAM to use the highest average power?

Stream

(c) Did any of the benchmarks use GPU power? Why not?

No, none of the benchmarks use GPU power.

Because sleep is shutting off the screen, so it would not use GPU power.

The STREAM benchmark is a simple synthetic benchmark program that measures sustainable memory bandwidth (in MB/s) and the corresponding computation rate for simple vector kernels. So the GPU is not used in this benchmark.

The matrix and matrix multiply is mainly using the CPU and does not use the GPU.

IOzone is a filesystem benchmark tool. The benchmark generates and measures a variety of file operations. So it does not use the GPU.

3. Branch Prediction

Static vs Dynamic, static prediction: (backward taken, forward not taken) dynamic branch prediction: (one-bit branch history table)

2-bit up/down counter (two bit saturating counter)

Looking at some simple C constructs say expected branch predict rate

4. Cache

Given some paramaters (size, way, blocksize, addr space) be able to calculate number of bits in tag, index, and offset.

32kB cache (2^15), direct mapped (2^0)

32 Byte linesize (2^5), 32-bit address size (2^32)

offset = log2(linesize) = 5 bits, lines = log2((cachesize/ways)/linesize) = 1024 lines(10 bits), tag = addresssize - (offset bits + line bits) = 17 bits
Know why caches are used, that they exploit temporal and spatial locality, and know the tradeoffs (speed vs nondeterminism)

Be at least familiar with the types of cache misses (cold, conflict, capacity)

Compulsory (Cold) | miss because first time seen Capacity | wouldn't have been a miss with larger cache Conflict | miss caused by conflict with another address (would not have been miss with fully assoc cache) Coherence | miss caused by other processor

Fixing Compulsory Misses Prefetching

Hardware Prefetchers very good on modern machines. Automatically bring in nearby cachelines Software { loading values before needed also special instructions available Large-blocksize of caches. A load brings in all nearby values in the rest of the block.

Fixing Capacity Misses

Build Bigger Caches

Fixing Conflict Misses

More Ways in Cache Victim Cache Code/Variable Alignment, Cache Conscious Data Placement

Fixing Coherence Misses

False Sharing { independent values in a cache line being accessed by multiple cores

Know difference between writeback and write-through

Write-through cache: Every write operation to the cache is accompanied by a write of the same data to main memory. When this is implemented, then the input/output processor need not consult the cache directory when it reads memory, since the state of main memory is an accurate reflection of the state of the cache as updated by the central processor. Although this scheme simplifies the accesses for the input/output processor, it results in fairly high traffic between central processor and memory, and the high traffic tends to degrade input/output performance.

Write-back cache (write-in) : In this scheme, the central processor updates the cache during a write, but actual updating of the memory is deferred until the line that has been changed is discarded from the cache. At that point, the changed data are written back to main memory. Write-back caching yields somewhat better performance than write-through caching because it reduces the number of write operations to main memory. With this performance improvement comes a slight risk that data may be lost if the system crashes.

Be able to work a few simple steps in a cache example

(like in HW#5)

5. Prefetch

Why have prefetchers?

Try to avoid cache misses by bringing values into the cache before they are needed.

Common prefetch patterns?

Software prefetching and hardware prefetching. Hardware prefetching:(icache,dcache,stride prefetching)

disabling prefetch hurt, dramatically so on equake.

6. Virtual Memory

General concept of VM

Original purpose was to give the illusion of more main memory than available, with disk as backing store. Give each process own linear view of memory. Demand paging (no swapping out whole processes).

Execution of processes only partly in memory, effectively a cache. Memory protection

Benefits of VM?

Memory Protection, each program has own address space, allows having more memory than physical memory, demand paging, copy-on-write for fork, less memory fragmentation, etc.

Why is TLB behavior important?

TLB Translation Lookaside Buffer (Lookaside Buffer is an obsolete term meaning cache) Caches page tables Much faster than doing a page-table walk. Historically fully associative, recently multi-level multi-

Way TLB shootdown { when change a setting on a mapping and TLB invalidated on all other processors

Depending on cache config:

worst case: (VIVT) every memory access looked up in TLB best case: (PIPT) every cache miss looked up in TLB

Combinations

PIPT { older systems. Slow, as must be translated (go through TLB) for every cache access (don't know index or tag until after lookup) VIVT { fast. Do not need to consult TLB to find data in cache. VIPT { ARM L1/L2. Faster, cache line can be looked up in parallel with TLB. Needs more tag bits. PIVT { theoretically possible, but useless. As slow as PIPT but aliasing like VIVT.

Large Pages

Another way to avoid problems with 64-bit address space Larger page size (64kB? 1MB? 2MB? 2GB?) Less granularity. Potentially waste space Fewer TLB entries needed to map large data structures Compromise: multiple page sizes. Complicate O/S and hardware. OS have to find free blocks of contiguous memory when allocating large page.Transparent usage? Transparent Huge Pages? Alternative to making people using special interfaces to allocate.

Physical Caches

Location in cache based on physical address Can be slower, as need TLB lookup for each cache access No need to flush cache on context switch (or ever, really) No need to do TLB lookup on writeback If properly sized, the index bits are the same for virt and physical. In this case no need to do TLB lookup on cache hit. If not sized, the extra index bits need to be stored in the cache so they can be passed along with the tag when doing a lookup

Virtual Caches

Location in cache based on virtual address Faster, as no need to do TLB lookup before access Will have to use TLB on miss (for fill) or when writing back dirty addresses Cache might have extra bits to indicate permissions so TLB doesn't have to be checked on write Can have aliasing issues when processes use same virtual addresses. Flush cache on context switch? How to avoid flushing? Have a process-id. Can also implement sharing this way, by both processes mapping to same virt address. Having kernel addresses high also avoids aliasing Operating system has to do more work