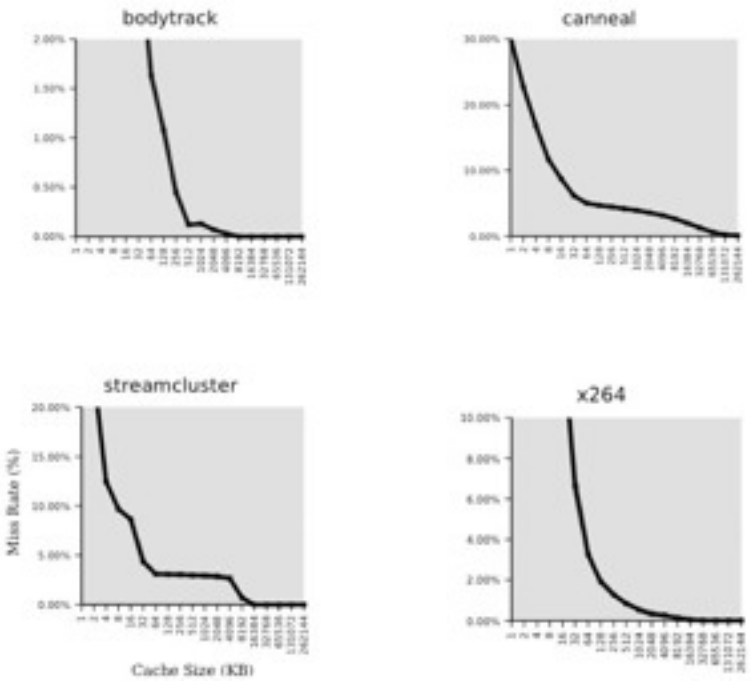
CS152
Caches and the Memory Hierarchy
Section #5 Handout

2011 Feb 25

Problem 1 **Working Set Size**

Below are some plots of cache miss rates from four applications in the PARSEC benchmark suite. The cache modeled has 64B lines, is 4-way set associative, and its capacity is varied.



a) For each of the plots above, determine the application's working set size.
bodytrack - 512 KB; canneal - 64KB; streamcluster - 64KB, 16MB; x264 64KB-4MB Things to notice:

- streamcluster displays two plateaus implying there is even a hierarchy of working sets
- canneal has a long tail, implying that for many of its accesses there is poor locality
- x264 has a gradual slope, indicating that the algorithm has been tuned to perform well on any cache size

b) Why might bodytrack's miss rate have gone up with a bigger cache?
Long running, real world applications can sometimes exhibit strange behavior. For any cache, one can devise a pathologically bad memory access stream. In this case somehow the 512KB cache ended up with the right blocks slightly more of the time than the 1024KB cache. It's quite possible that since a 4-way cache was used, bodytrack may be relying on more than 4 memory values that all aliased to the same line for the 1024KB size.

Problem 2 **Caches**

a) Cache A has a 90% hit rate and cache B has a 95% hit rate. Numerically, how much better is cache B?

A small 5% increase in hit rate halves the miss rate for cache B, so its nearly twice as fast when you consider the AMAT (miss penalty dominates hit time).

b) Can you have a no-write allocate, write-back cache?

Yes. On a write miss the write will go straight to memory and will not be stored in the cache.

c) If you are concerned about memory bandwidth (to DRAM), should you have a write allocate cache?

No. With write-allocate, on a write miss you will be bringing in the rest of cache block using up more bandwidth, so you will probably not allocate to save bandwidth.

Problem 3 **Write Buffers**

For this problem we are considering adding a write buffer to a single issue in-order CPU with one level of cache. The cache is write back, no write allocate, and for this problem you don't have to worry about lines ever getting evicted. By adding a write buffer, if a write misses in the cache, it can go directly into the write buffer rather than stalling the system for the miss.

	Quantity	Time (cycles)
Read	Hit Time Miss Penalty Hit (after adding WB)	2 200 $3+2\log_2$(WB depth)
Write	Hit Time Miss Penalty Hit/miss time (after adding WB)	2 240 6
	Cache read hit rate Cache write hit rate	98% 70%

a) What is the average memory access time (AMAT) for reads before and after a write buffer of depth 4 is added?

before: (hit time) + (miss rate)(miss penalty) = 2 + (1-0.98)(200) = 2 + (.02)(200) = 6 cycles
after: $(3 + 2\log_2(4)) + (.02)(200) = 11$ cycles

b) What is the AMAT for writes before adding the write buffer?

4 + (1 - 0.7)(240) = 76 cycles

c) Assume a program averages a write every 24 cycles and that they are decently uniformly distributed. What is the worst average write miss rate the program can have and expect a finite sized write buffer to not overflow?

For the WB, it needs (rate in) = (rate out) for it to not overflow. A write to memory takes 240 cycles, so a write should miss the cache no more than that. Thus:

$$\text{write miss interval} = \frac{\text{write interval}}{\text{write miss rate}} \quad \text{so} \quad \text{write miss rate} = \frac{\text{write interval}}{\text{write miss interval}} = 10\%$$

```
typedef double real; /* 8 bytes! */

/* Column-major matrices */
#define A(row,column) _A[(row) + ldA*(column)]
#define B(row,column) _B[(row) + ldB*(column)]
#define C(row,column) _C[(row) + ldC*(column)]

void gemm(int M, int N, int K,  real alpha, real* _A, int ldA,
       real* _B, int ldB, real beta, real* _C, int ldC) {
  int m, n, k;
  for(n = 0; n < N; n++) {
    for(m = 0; m < M; m++) {
      real C_mn = beta * C(m,n);
      for(k = 0; k < K; k++)
          C_mn += alpha*A(m,k)*B(k,n);
      C(m,n) = C_mn;
    }
  }
}
```

Above is code from Lab 2. The function *gemm* performs a matrix multiply such that **C**=alpha*__A__*__B__+beta*__C__ (where __A__,__B__, and __C__ are matrices and alpha and beta are constants).

However, if you run this code you will notice it exhibits poor performance (say, the matrix sizes denoted by M,N, and K are all 1024).

a) Why is the performance so bad?

The above code exhibits terrible locality (both spatial and temporal), and thus the miss rate is high.

b) As the programmer in charge of this piece of code, can you come up with two modifications to the software that will make this code run much, *much* faster? (hint: you may want to change or modify one or more of the matrices).

1) For the inner k-loop, Matrix A is accessed by striding through its columns (which are non-contiguous in memory). By **transposing** A, we can instead get spatial locality by accessing A align a cache line.
2) **block** the algorithm by only working on a small corner of the matrices. See the Lab#2 code to see the blocked_gemm() implementation. By picking an appropriate blocking size, you can dramatically improve temporal locality.

Problem 5 **Power vs Energy**

a) What is power? What is energy? How are they different?

Energy (measured in Joules) is a measure of the "amount of work" that can be performed (think of a battery storing energy, that lets me watch X hours worth of video).
Power (measured in Joules / second) is a measure of Energy dissipated in a given time interval (think of the heat generated per unit time that must be dissipated).

The number of transistors a design uses ∝ area the processor takes up ∝ amount of power used. The more transistors, the more energy dissipated to switch each transistor for each cycle.

b) Does an iPhone (or other mobile device) care more about power-efficiency, or energy-efficiency? What your desktop machine at home? What about a server chip at Google?

A mobile device cares most about a long battery life, so energy-efficiency is more important. It's okay if your phone gets hot while loading a web-page (i.e., not very power-efficient), so long as once it finishes it can go to a lower energy state and save battery life.

A desktop machine doesn't have a battery, and the utility bill to run a desktop machine is relatively cheap (power used is around 200-500W, or equivalent to a few lights), so energy-efficiency isn't a primary motivator. However, a desktop user cares A LOT about noise, which is mostly caused by the fans trying desperately to cool the processor while watching movies or playing video games. Also, the processor, which runs around +100W peak, has to worry about *melting*, so the more power-efficient the processor is the more performance that can be squeezed out of it.

A server typically runs at around 10-20% load, so it rarely gets a chance to turn down to a lower energy state. The utility bill to power all of the servers in a building is highly non-trivial, so energy-efficiency is a big concern. However, it also requires considerable effort to *cool* a room full of servers, so power-efficiency is also important. And because servers run at a relatively constant load for all time, being efficient in terms of power directly equates to being efficient in energy, since time becomes immaterial (no going to sleep!).