Memory hierarchy (caches)

- Transparently provide the illusion of a high-speed-high-capacity memory
- Built out of caches: small memory devices that exploit the principle of locality of reference
Spatial locality: In the temporal trace a program tends to access memory locations that are nearby.

Special case: if the nearby memory location is at a distance zero (same location is repeatedly accessed), this is called temporal locality.

Sanjay’s peeve: all locality is temporal, it is relative to a small interval in the temporal trace of execution of program.

Cache operation:

- When CPU accesses (for now assume all accesses are reads) memory location $x$, the cache checks if it has $x$.
  - If so (this is a hit) the value is returned immediately.
  - Otherwise, CPU execution is stalled, the value is fetched from memory, and stored in the cache.

Blocks/lines: granularity of memory transactions, multiple words (to exploit spatial locality, and amortize overheads).
Let's build our first cache

Performance Analysis

- Run simulations/execution traces of many programs in a suite of benchmarks. Measure various events, and aggregate

  - CPI (Cycles per Instruction): Average number of clock cycles it takes to execute an instruction
dynamic instruction count
  # clock cycles in execution

  - Hit rate (at level i): fraction of memory accesses that are found in the ith level of cache

  - Miss rate: 1 - hit rate

  - Hit time: time to service the request at level i, assuming it's a hit

  - Miss penalty: time to service the request at level i, if it's a miss
Example

Miss rate: 3% for instruction cache, and 4% for data cache
CPI is 2 if there are no memory stalls
Miss penalty: 100 cycles
36% of instructions are LD/ST

Determine the effective CPI for

- A perfect cache
- No cache
- The cache with the above specifications

Quiz
Hierarchy of caches

Do the same analysis, but now with an additional level of data cache with:

- 20 cycle access time (for hit or a miss)
- Enough capacity that the miss rate to main memory is reduced to 1%

Key cool idea: multiple levels help significantly

Further improvements

Miss penalty is not in our control, so reduce miss rate as much as possible

What limits the miss rate?

- Cache size (but remember, larger is better, but not necessarily faster)

- Organization/Architecture:
  - For fixed size, reorganize the cache so better exploit resources
Determine the effective CPI for

- A perfect cache
- No cache
- The cache with the above specifications

Cache replacement policy

- Caches are smaller than local memories, they fill up quickly, and therefore a replacement policy is needed.
- The heuristic that it uses to choose the entry to evict is called the replacement policy. The fundamental problem with any replacement policy is that it must predict which existing cache entry is least likely to be used in the near future.
- A popular replacement policy, least-recently used (LRU), replaces the least recently accessed entry.
The replacement policy decides where in the cache a copy of a particular entry of main memory will go. If the replacement policy is free to choose any entry in the cache to hold the copy, the cache is called **fully associative**. At the other extreme, if each entry in main memory can go in just one place in the cache, the cache is **direct mapped**. Many caches implement a compromise in which each entry in main memory can go to any one of \( N \) places in the cache, and are described as **N-way set associative**.

---

**Sources of cache misses: the 3C model**

- **Compulsory**: On the first access to a block; the block must be brought into the cache; also called cold start misses, or first reference misses.
- **Capacity**: Occur because blocks are being discarded from cache because cache cannot contain all blocks needed for program execution (program working set is much larger than cache capacity).
- **Conflict**: In the case of set associative or direct mapped block placement
### Improving cache performance: hardware

- Increased cache capacity
- Higher associativity (without sacrificing speed/energy)
- Hardware prefetching of instructions and data
  - Equidistant locality
- Second-level / third level cache (L2, L3)
  - L3 often shared by multiple cores
  - There is a difference in access time between L1, L2, L3
- Out of order instruction execution
- Branch prediction

All this makes modern CPUs highly complex.

Colorado State University

---

### Improving cache performance: software

- **Merging Arrays**: Improve spatial locality by single array of structs vs. parallel arrays (Fortran).
- **Loop Interchange**: Change nesting of loops to access data in the order stored in memory.
- **Loop Fusion**: Combine 2 or more independent loops that have the same looping and some variables overlap.
- **Blocking or “tiling”**: Improve temporal locality by accessing “blocks” of data repeatedly vs. going down whole columns or rows. (prime sieve)

Colorado State University
Matrix Multiply

Data or loop reordering for improve cache performance

Matrix multiply:

\[
\text{for } i = 1 \text{ to } n \\
\text{for } j = 1 \text{ to } n \\
C[i,j] = 0 \\
\text{for } k = 1 \text{ to } n \\
C[i,j] += A[i,k] \times B[k,j] \\
\]

B is accessed in column order. If arrays are (as in C) stored in row major order, cache lines are not helping, which can cause cache misses for all Bs. Solution: transpose B
Tiling

- Instead of reading a whole row of $A$ and doing $n$ whole row $A$ column $B$ inner products we can read a block of $A$ and compute smaller inner products with sub columns of $B$.
- These partial products are then added up.

Eratosthenes: the problem

- Find all the prime numbers up to a given number $n$
- By “filtering out” the multiples of known primes
- Many strategies
  - Start with a sequential algorithm and systematically parallelize it using our known and trusted approach (Foster’s method)
  - Think out of the box (a completely different approach)
The complexity

- How many primes are there?
  - A: See [http://primes.utm.edu/howmany.shtml](http://primes.utm.edu/howmany.shtml)
- What is the (work) complexity of the sieve?
  - $\Theta(n \ln \ln n)$

Sequential Algorithm
Create an array of numbers 2 ... n, none of which is “marked”

**Invariant**: the smallest unmarked number is a prime

k ← 2       /* k is the “next” prime number */
repeat
mark off all multiples of k as non-primes
set k to the next unmarked number (which must be a prime)
until “done”

---

**Pseudo code**

for (i=1; i<=n; i++) marked[i] = 0;
for (i=2; i<=n; i++) marked[i] = 1;
while (k<=n) {
    k = index = 2;
    while (marked[++index]) ; /* do nothing */
    /* now index has the first unmarked number, so */
    /* ... */
    k = index;
}
Analysis & Improvement

- Where does the program spend its time?

- How to improve?
  - if $x=ab$ is a composite number, then at least one of $a$ or $b$ is less than (or equal to) $\sqrt{x}$ (algorithmic improvement)

Improved code

```c
for (i=0; i<n; i++) marked[i] = 0;
k = index = 2;
marked[0] = marked[1] = 1;
while (k*k<=n) { /* outer loop iterates only until $\sqrt{n}$ */
  for (i=k*k; i<n; i++) if (i%k == 0) marked[i]=1;
  while (marked[++index]) ; /* do nothing */
  /* now index has the first unmarked number, so ... */
  k = index;
}
```

Colorado State University
Sequential performance first (avoid division)

```c
for (i=1; i<=n; i++) marked[i] = 0;
k = index = 2;
marked[0] = marked[1] = 1;
while (k*k<=n) { /* outer loop iterates only until sqrt(n) */
   /* for (i=k*k; i<=n; i++) if (i%k == 0) marked[i]=1; */
   for (i=k*k; i<=n; i+=k) marked[i]=1;
   while (marked[++index]) ; /* do nothing */
   /* now index has the first unmarked number, so ... */
   k = index;
}
```

Colorado State University

Lessons

- Exercise
  - Write the three programs and measure the running time for large values of n
- Priorities:
  - First improve algorithm (asymptotic running time)
  - Next constant factor gains
  - Only then consider parallelization

Colorado State University
**HW2 Clarification**

- Experiment a bit
  - find a “large enough $n$”
  - running time on one processor ~30 sec
- Then, keep $n$ fixed, change the number of threads
  - gather some running time data
  - plot the speedup
  - explain what you see
  - does it jive with your expectations (hypotheses)

---

**Revisit complexity analysis**

- Basic conventions and background:
  - $\log$ (base 10), $\lg$ (base 2), $\ln$ (base $e$)
    - $\log x$ is number of digits to represent $x$
    - $\lg x$ is number of bits to represent $x$
  - #primes no larger than $x$: $x/\ln x$
  - Sum of reciprocals of integers no larger than $x$: $\ln x$
  - Sum of reciprocals of primes no larger than $x$: $\ln \ln x$
Naïve algorithm:
- How many times is the outer loop (i.e., the $k$-loop) executed?
- How many times is the inner loop executed? $T = O(n \ln \ln n)$

Sequential complexity

$T(n) = O(n \ln \ln n)$

Almost linear time complexity
Analysis

- Sequential complexity

\[ T(n) = O(n \ln \ln n) \]
- \( \log_{10} n \): Number of digits in decimal representation of \( n \)
- \( \log_2 n \): Number of bits in the binary representation of \( n \)
- \( \log \log_{10} n \): Digits in decimal representation of that
- \( \log \log_2 n \): Bits in the binary representation of that

- Almost linear time complexity

First parallelization

```
for (i=1; i<=n; i++) marked[i] = 0;
k = index = 2;
marked[0] = marked[1] = 1;
while (k*k<=n) {
    for (i=k*k; i<=n; i+= k)
        marked[i] = 1;
    while (marked[++index]) ; /* do nothing */
    k = index;
}
```

Parallelize this loop
Are we done?

- How much can we improve?
  - Sequential performance first – locality
  - A cache miss costs hundreds of cycles
  - Matrix multiplication:

\[ O(n^3) \text{ or } O(n^5) \]

“The uniform Memory Hierarchy Model of Computation”

Improving sieve locality

- Current scheme re-accesses the marked array many times (once for each prime)
- Solution: “block off” the array and mark off only multiples of \( k \) within the current block for many different primes
  - (Only) then move to the next block
  - Need to save many (why not all?) primes
/* Preamble: In an array primes[] compute and save all the
primes up to sqrt(n), say there are numprimes of them */

for (j=0; j<numprimes; j++) {
    for (i=primes[j]*primes[j]; i<=n; i+= primes[j]) marked[i]=1;
}

/* Preamble (unchanged) */

for (j=0; j<numprimes; j++) {
    for (ii=start; ii<min(start+BKSIZE, n); ii+=BKSIZE){
        for (i=FMIB(ii,j); i<=min(start+BKSIZE, n); i+= primes[j]) marked[i]=1;
    }
}

What is start?

FMIB(ii,j) is First Multiple of primes[j] In current Block

But nothing has changed

/* Preamble: In an array primes[] compute and save all the
primes up to sqrt(n), say there are numprimes of them */

for (j=0; j<numprimes; j++) {
    for (i=primes[j]*primes[j]; i<=n; i+= primes[j]) marked[i]=1;
}

/* Preamble (unchanged) */

for (j=0; j<numprimes; j++) {
    for (ii=start; ii<min(start+BKSIZE, n); ii+=BKSIZE){
        for (i=FMIB(ii,j); i<=min(start+BKSIZE, n); i+= primes[j]) marked[i]=1;
    }
}
**Interchange the loops**

```c
/* Preamble (unchanged) */
for (ii=start; ii<min(start+BKSIZE, n); ii+=BKSIZE)
    for (j=0; j<numprimes; j++)
        for (i=FMIB(ii,j); i<min(start+BKSIZE, n); i+= primes[j]) marked[i]=1;
```

```c
/* Preamble (unchanged) */
for (j=0; j<numprimes; j++)
    for (ii=start; ii<min(start+BKSIZE, n); ii+=BKSIZE)
        for (i=FMIB(ii,j); i<min(start+BKSIZE, n); i+= primes[j]) marked[i]=1;
```

---

**But is this legal?**

- This is the key issue (HW2)
- We must ensure that the modified code does exactly what the original program did