Multithreading Review

• Difficult to continue to extract instruction-level parallelism (ILP) or data-level parallelism (DLP) from a single sequential thread of control

• Multithreaded processors exploit thread-level parallelism (TLP) to improve performance

• Fine-grain vertical multithreading
  – Fixed thread scheduling (possibly avoid bypass/stall logic)
  – Software configurable thread scheduling
  – Dynamic thread scheduling (schedule around stalls)

• Coarse-grain vertical multithreading
  – Only switch threads on cache miss

• Simultaneous multithreading
  – Use OOO execution resources to implement both vertical and horizontal multithreading yet still also enable high single-thread performance
Advanced Cache Optimizations

• Reduce hit time
  – Way prediction
  – Trace caches

• Reduce miss penalty
  – Early restart and critical word first
  – Merging write buffers
  – Victim caches

• Reduce miss rate
  – Compiler optimizations

• Reduce miss rate and penalty via prefetching
  – Software prefetching
  – Hardware prefetching

• Increasing cache bandwidth
  – Multibanked caches
  – Non-blocking caches
Way Prediction

- Use address bits to index into way prediction table
- Way prediction table indicates likely way for the next cache access
- Use the prediction to setup way mux ahead of time and then can truly do a single tag check in parallel with data access
  - Reduces critical path, especially if tag check was on critical path
  - Reduces hit latency in cycles if associative cache was pipelined with tag check in first cycle and data access in second cycle
  - Reduces energy overhead of multiple tag comparisons
  - Can potentially only enable desired data memory set further reducing energy overhead
- If mispredict, then do full parallel tag check on next cycle and update the way prediction table
Way Prediction

Way 0
Way 1

Tag Check
Read Access

Tag Check
Data Access

ECE 4750
L18: Advanced Cache Optimizations
Trace Cache

- Similar motivation to sw trace scheduling
- Dynamically detect hot paths and place probable consecutive basic blocks next to each other in trace cache
- Enables fetching past these branches, but requires verifying that the branches were correctly predicted
Trace Cache Implementation

![Diagram of Trace Cache Implementation]
Advanced Cache Optimizations

• Reduce hit time
  – Way prediction
  – Trace caches

• **Reduce miss penalty (on board)**
  – Early restart and critical word first
  – Merging write buffers
  – Victim caches

• Reduce miss rate
  – Compiler optimizations

• Reduce miss rate and penalty via prefetching
  – Software prefetching
  – Hardware prefetching

• Increasing cache bandwidth
  – Multibanked caches
  – Non-blocking caches
Advanced Cache Optimizations

- Reduce hit time
  - Way prediction
  - Trace caches
- Reduce miss penalty
  - Early restart and critical word first
  - Merging write buffers
  - Victim caches
- **Reduce miss rate**
  - Compiler optimizations
- Reduce miss rate and penalty via prefetching
  - Software prefetching
  - Hardware prefetching
- Increasing cache bandwidth
  - Multibanked caches
  - Non-blocking caches
Compiler Optimizations

• Restructuring code affects the data block access sequence
  – Group data accesses together to improve spatial locality
  – Re-order data accesses to improve temporal locality

• Prevent data from entering the cache
  – Useful for variables that will only be accessed once before being replaced
  – Needs mechanism for software to tell hardware not to cache data (“no-allocate” instruction hints or page table bits)

• Kill data that will never be used again
  – Streaming data exploits spatial locality but not temporal locality
  – Replace into dead cache locations
Loop Interchange

```c
for(j=0; j < N; j++) {
    for(i=0; i < M; i++) {
        x[i][j] = 2 * x[i][j];
    }
}
```

```c
for(i=0; i < M; i++) {
    for(j=0; j < N; j++) {
        x[i][j] = 2 * x[i][j];
    }
}
```

What type of locality does this improve?
Loop Fusion

```
for (i = 0; i < N; i++)
    a[i] = b[i] * c[i];

for (i = 0; i < N; i++)
    d[i] = a[i] * c[i];
```

```
for (i = 0; i < N; i++)
{
    a[i] = b[i] * c[i];
    d[i] = a[i] * c[i];
}
```

What type of locality does this improve?
Matrix Multiply, Naïve Code

for (i=0; i < N; i++)
    for (j=0; j < N; j++) {
        r = 0;
        for (k=0; k < N; k++)
            r = r + y[i][k] * z[k][j];
        x[i][j] = r;
    }

Old access

Not touched

New access
Matrix Multiply with Cache Tiling

```c
for(jj=0; jj < N; jj(jj+B)
    for(kk=0; kk < N; kk=kk+B)
        for(i=0; i < N; i++)
            for(j=j; j < min(jj+B,N); j++) {
                r = 0;
                for(k=k; k < min(kk+B,N); k++)
                    r = r + y[i][k] * z[k][j];
                x[i][j] = x[i][j] + r;
            }
```

Diagram of matrix multiplication with cache tiling.
Matrix Multiply with Cache Tiling
Matrix Multiply with Cache Tiling

\[
\begin{array}{cccc}
  & & z & j \\
  & k & & \\
  y & k & & \\
  i & & & \\
\end{array}
\]

\[
\begin{array}{cccc}
  & & & \\
  & & & x \\
  j & & & \\
  i & & & \\
\end{array}
\]

\[
\begin{array}{cccc}
  & & & \\
  & & & \\
  & & & \\
  & & & \\
\end{array}
\]
Matrix Multiply with Cache Tiling
Matrix Multiply with Cache Tiling

What type of locality does this improve?
Advanced Cache Optimizations

• Reduce hit time
  – Way prediction
  – Trace caches

• Reduce miss penalty
  – Early restart and critical word first
  – Merging write buffers
  – Victim caches

• Reduce miss rate
  – Compiler optimizations

• **Reduce miss rate and penalty via prefetching**
  – Software prefetching
  – Hardware prefetching

• Increasing cache bandwidth
  – Multibanked caches
  – Non-blocking caches
Prefetching

• Speculate on future instruction and data accesses and fetch them into cache(s)
  – Instruction accesses easier to predict than data accesses

• Varieties of prefetching
  – Hardware prefetching
  – Software prefetching
  – Mixed schemes

• What types of misses does prefetching affect?
Software Prefetching

```c
for(i=0; i < N; i++) {
    prefetch( &a[i + 1] );
    prefetch( &b[i + 1] );
    SUM = SUM + a[i] * b[i];
}
```

*What property do we require of the cache for prefetching to work?*
Software Prefetching Issues

• Timing is the biggest issue, not predictability
  – If you prefetch very close to when the data is required, you might be too late
  – Prefetch too early, cause pollution
  – Estimate how long it will take for the data to come into L1, so we can set P appropriately
  – Why is this hard to do?

```c
for(i=0; i < N; i++) {
    prefetch( &a[i + P] );
    prefetch( &b[i + P] );
    SUM = SUM + a[i] * b[i];
}
```

Must consider cost of prefetch instructions
Issues in Prefetching

- Usefulness – should produce hits
- Timeliness – not late and not too early
- Cache and bandwidth pollution
Hardware Instruction Prefetching

Instruction prefetch in Alpha AXP 21064

- Fetch two blocks on a miss; the requested block \( i \) and the next consecutive block \( i+1 \)
- Requested block placed in cache, and next block in instruction stream buffer
- If miss in cache but hit in stream buffer, move stream buffer block into cache and prefetch next block \( i+2 \)
Hardware Data Prefetching

• One Block Lookahead (OBL) schemes
  – Initiate prefetch for block \( b + 1 \) when see miss to block \( b \)
  – Can extend to N-block lookahead
  – Why is this different than increasing block size?
  – Just a hint, can store prefetches in stream buffers, can predict when to actually prefetch

• Strided prefetch
  – If observe sequence of accesses to block \( b, b+N, b+2N, \) then prefetch \( b+3N \) etc.

Example: IBM Power 5 [2003] supports eight independent streams of strided prefetch per processor, prefetching 12 lines ahead of current access
Advanced Cache Optimizations

• Reduce hit time
  – Way prediction
  – Trace caches

• Reduce miss penalty
  – Early restart and critical word first
  – Merging write buffers
  – Victim caches

• Reduce miss rate
  – Compiler optimizations

• Reduce miss rate and penalty via prefetching
  – Software prefetching
  – Hardware prefetching

• Increasing cache bandwidth
  – Multibanked caches (on board)
  – Non-blocking caches (see vector refill unit slides)
Acknowledgements

• These slides contain material developed and copyright by:
  – Arvind (MIT)
  – Krste Asanovic (MIT/UCB)
  – Joel Emer (Intel/MIT)
  – James Hoe (CMU)
  – John Kubiatowicz (UCB)
  – David Patterson (UCB)

• MIT material derived from course 6.823
• UCB material derived from course CS252 & CS152