Announcements

- Milestone meetings
  - Meet with Evangelos, Lavanya, Vivek
  - And, me… especially if you receive(d) my feedback and I asked to meet
Last Time

- Markov Prefetching
- Content Directed Prefetching
- Execution Based Prefetchers
Multi-Core Issues in Prefetching and Caching
Prefetching in Multi-Core (I)

- Prefetching shared data
  - Coherence misses

- Prefetch efficiency a lot more important
  - Bus bandwidth more precious
  - Prefetchers on different cores can deny service to each other and to demand requests
    - DRAM contention
    - Bus contention
    - Cache conflicts
  - Need to coordinate the actions of independent prefetchers for best system performance
    - Each prefetcher has different accuracy, coverage, timeliness
Local-only prefetcher control techniques have no mechanism to detect inter-core interference.
Prefetching in Multi-Core (II)

- Ideas for coordinating different prefetchers’ actions
  - Utility-based prioritization
    - Prioritize prefetchers that provide the best marginal utility on system performance
  - Cost-benefit analysis
    - Compute cost-benefit of each prefetcher to drive prioritization
  - Heuristic based methods
    - Global controller overrides local controller’s throttling decision based on interference and accuracy of prefetchers
Hierarchical Prefetcher Throttling

Local Control's goal: 
Maximize the prefetching performance of core \(i\) independently

Global Control's goal: 
Keep track of and control prefetcher-caused inter-core interference in shared memory system

- Pref. \(i\)
- Local Control
- Core \(i\)
- Global Control
- Memory Controller
- Cache Pollution Feedback
- Bandwidth Feedback
- Shared Cache

Global control's goal: Accepts or overrides decisions made by local control to improve overall system performance.
Hierarchical Prefetcher Throttling Example

- High accuracy
- High pollution
- High bandwidth consumed while other cores need bandwidth

Memory Controller

High BW (i)
High BWNO (i)

Global Control

High Acc (i)

Local Throttling Decision

Core i

Pref. i

Local Control

Enforce Throttling Decision

High Pol (i)

Local Throttle Up

Pol. Filter i

Shared Cache
Multi-Core Issues in Caching

- Multi-core
  - More pressure on the memory/cache hierarchy → cache efficiency a lot more important
  - Private versus shared caching
  - Providing fairness/QoS in shared multi-core caches
  - Migration of shared data in private caches
  - How to organize/connect caches:
    - Non-uniform cache access and cache interconnect design

- Placement/insertion
  - Identifying what is most profitable to insert into cache
  - Minimizing dead/useless blocks

- Replacement
  - Cost-aware: which block is most profitable to keep?
Cache Coherence

- Basic question: If multiple processors cache the same block, how do they ensure they all see a consistent state?
The Cache Coherence Problem

P1

P2

ld r2, x

1000

Interconnection Network

1000

x

Main Memory
The Cache Coherence Problem

ld r2, x

ld r2, x

P1

1000

Interconnection Network

P2

1000

Main Memory

ld r2, x
The Cache Coherence Problem

ld r2, x
add r1, r2, r4
st x, r1

ld r2, x

2000

P1

1000

P2

1000

Interconnection Network

Main Memory
The Cache Coherence Problem

ld r2, x
add r1, r2, r4
st x, r1

ld r2, x
Should NOT load 1000
ld r5, x

ld r2, x

Interconnection Network

P1

2000

P2

1000

Main Memory

x 1000
Cache Coherence: Whose Responsibility?

- **Software**
  - Can the programmer ensure coherence if caches are invisible to software?
  - What if the ISA provided the following instruction?
    - FLUSH-LOCAL A: Flushes/invalidates the cache block containing address A from a processor’s local cache
    - When does the programmer need to FLUSH-LOCAL an address?
  - What if the ISA provided the following instruction?
    - FLUSH-GLOBAL A: Flushes/invalidates the cache block containing address A from all other processors’ caches
    - When does the programmer need to FLUSH-GLOBAL an address?

- **Hardware**
  - Simplifies software’s job
  - One idea: Invalidate all other copies of block A when a processor writes to it
Snoopy Cache Coherence

- Caches “snoop” (observe) each other’s write/read operations
- A simple protocol:

- Write-through, no-write-allocate cache
- Actions: PrRd, PrWr, BusRd, BusWr
Multi-core Issues in Caching

- How does the cache hierarchy change in a multi-core system?
- **Private** cache: Cache belongs to one core
- **Shared** cache: Cache is shared by multiple cores
Shared Caches Between Cores

- Advantages:
  - Dynamic partitioning of available cache space
    - No fragmentation due to static partitioning
  - Easier to maintain coherence
  - Shared data and locks do not ping pong between caches

- Disadvantages
  - Cores incur conflict misses due to other cores’ accesses
    - Misses due to inter-core interference
    - Some cores can destroy the hit rate of other cores
      - What kind of access patterns could cause this?
  - Guaranteeing a minimum level of service (or fairness) to each core is harder (how much space, how much bandwidth?)
  - High bandwidth harder to obtain (N cores $\rightarrow$ N ports?)
Shared Caches: How to Share?

- Free-for-all sharing
  - Placement/replacement policies are the same as a single core system (usually LRU or pseudo-LRU)
  - Not thread/application aware
  - An incoming block evicts a block regardless of which threads the blocks belong to

- Problems
  - A cache-unfriendly application can destroy the performance of a cache friendly application
  - Not all applications benefit equally from the same amount of cache: free-for-all might prioritize those that do not benefit
  - Reduced performance, reduced fairness
Problem with Shared Caches
Problem with Shared Caches

Diagram showing the relationship between Processor Core 1, Processor Core 2, L1 $, and L2 $.
Problem with Shared Caches

Processor Core 1  ← t1  t2 → Processor Core 2

L1 $  

L2 $  

......

L1 $  

......

t2’s throughput is significantly reduced due to unfair cache sharing.
Controlled Cache Sharing

- Utility based cache partitioning
  - Suh et al., "A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning," HPCA 2002.

- Fair cache partitioning

- Shared/private mixed cache mechanisms
Utility Based Shared Cache Partitioning

- Goal: Maximize system throughput
- Observation: Not all threads/applications benefit equally from caching → simple LRU replacement not good for system throughput
- Idea: Allocate more cache space to applications that obtain the most benefit from more space

The high-level idea can be applied to other shared resources as well.

Utility Based Cache Partitioning (I)

Utility $U^b_a = \text{Misses with } a \text{ ways} - \text{Misses with } b \text{ ways}$

![Graph showing utility across different numbers of ways]

- Low Utility
- High Utility
- Saturating Utility
Utility Based Cache Partitioning (II)

Idea: Give more cache to the application that benefits more from cache
Utility Based Cache Partitioning (III)

Three components:

- Utility Monitors (UMON) per core
- Partitioning Algorithm (PA)
- Replacement support to enforce partitions
Utility Monitors

- For each core, simulate LRU using auxiliary tag store (ATS)
- Hit counters in ATS to count hits per recency position
- LRU is a stack algorithm: hit counts ➔ utility
  E.g. hits(2 ways) = H0+H1

(MRU)H0 H1 H2…H15(LRU)

MTS
- Set A
- Set B
- Set C
- Set D
- Set E
- Set F
- Set G
- Set H

ATS
- Set A
- Set B
- Set C
- Set D
- Set E
- Set F
- Set G
- Set H
Utility Monitors

Figure 4. (a) Hit counters for each recency position. (b) Example of how utility information can be tracked with stack property.
Dynamic Set Sampling

- Extra tags incur hardware and power overhead
- DSS reduces overhead [Qureshi+ ISCA’06]
- 32 sets sufficient (analytical bounds)
- Storage < 2kB/UMON
Partitioning Algorithm

- Evaluate all possible partitions and select the best

- With \( a \) ways to core1 and \((16-a)\) ways to core2:
  \[
  \text{Hits}_{\text{core1}} = (H_0 + H_1 + \ldots + H_{a-1}) \quad \text{---- from UMON1}
  \]
  \[
  \text{Hits}_{\text{core2}} = (H_0 + H_1 + \ldots + H_{16-a-1}) \quad \text{---- from UMON2}
  \]

- Select \( a \) that maximizes \((\text{Hits}_{\text{core1}} + \text{Hits}_{\text{core2}})\)

- Partitioning done once every 5 million cycles
Way Partitioning

Way partitioning support:
1. Each line has core-id bits
2. On a miss, count $\text{ways\_occupied}$ in set by miss-causing app if $\text{ways\_occupied} < \text{ways\_given}$

- Yes: Victim is the LRU line from other app
- No: Victim is the LRU line from miss-causing app
Performance Metrics

- Three metrics for performance:

1. **Weighted Speedup (default metric)**
   \[ \text{perf} = \frac{\text{IPC}_1}{\text{Single IPC}_1} + \frac{\text{IPC}_2}{\text{Single IPC}_2} \]
   \(\Rightarrow\) correlates with reduction in execution time

2. **Throughput**
   \[ \text{perf} = \text{IPC}_1 + \text{IPC}_2 \]
   \(\Rightarrow\) can be unfair to low-IPC application

3. **Hmean-fairness**
   \[ \text{perf} = \text{hmean}(\frac{\text{IPC}_1}{\text{Single IPC}_1}, \frac{\text{IPC}_2}{\text{Single IPC}_2}) \]
   \(\Rightarrow\) balances fairness and performance
Utility Based Cache Partitioning Performance

Four cores sharing a 2MB 32-way L2

![Bar chart showing weighted speedup with LRU, UCP(Greedy), UCP(Lookahead), and UCP(EvalAll) for Mix1, Mix2, Mix3, and Mix4.](image)

- **Mix1** (gap-applu-apsi-gzp)
- **Mix2** (swm-glg-mesa-prl)
- **Mix3** (mcf-applu-art-vrtx)
- **Mix4** (mcf-art-eqk-wupw)
Utility Based Cache Partitioning

- Advantages over LRU
  + Better utilizes the shared cache

- Disadvantages/Limitations
  - Scalability: Partitioning limited to ways. What if you have numWays < numApps?
  - Scalability: How is utility computed in a distributed cache?
  - What if past behavior is not a good predictor of utility?