LARGE CACHE DESIGN

Mahdi Nazm Bojnordi
Assistant Professor
School of Computing
University of Utah
Overview

- Upcoming deadline
  - Feb. 1\textsuperscript{st}: project group formation

- This lecture
  - Compiler optimizations
  - Cache replacement policies
  - Cache partitioning
  - Highly associative caches
Software Optimizations
Recall: Cache Power Optimization

- Cache banking and resizing
  - Static and dynamic power

- Way prediction
  - Dynamic power

- Gated Vdd, Drowsy caches
  - Leakage power

Example: FX Processors

What can we do in software?

[source: AMD]
Compiler-Directed Data Partitioning

- Multiple D-cache banks, each with sleep mode
- Lifetime analysis used to assign commonly idle data to the same bank

Variables

Banks

- B1
- B2

Read access
Write access
Idle interval
Compiler Optimizations

- **Loop Interchange**
  - Swap nested loops to access memory in sequential order

```c
/* Before */
for (j = 0; j < 100; j = j+1)
    for (i = 0; i < 5000; i = i+1)
        x[i][j] = 2 * x[i][j];

/* After */
for (i = 0; i < 5000; i = i+1)
    for (j = 0; j < 100; j = j+1)
        x[i][j] = 2 * x[i][j];
```

- **Blocking**
  - Instead of accessing entire rows or columns, subdivide matrices into blocks
  - Requires more memory accesses but improves locality of accesses
/* Before */
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;
        };

$2N^3 + N^2$ memory words accessed
/* After*/
for (jj=0; jj<N; jj = jj+B)
for(kk=0; kk<N; kk = kk+B)
for (i=0; i<N; i++)
    for (j=jj; j < min(jj+B,N); j++)
    {r=0;
     for (k=kk; k < min(kk+B,N); k++)
       r = r + Y[i][k]*Z[k][j];
     X[i][j] = X[i][j] + r;
    };

\[2N^3/B + N^2\]
Replacement Policies
Basic Replacement Policies

- Least Recently Used (LRU)
- Least Frequently Used (LFU)
- Not Recently Used (NRU)
  - every block has a bit that is reset to 0 upon touch
  - a block with its bit set to 1 is evicted
  - if no block has a 1, make every bit 1
- Practical pseudo-LRU
Common Issues with Basic Policies

- Low hit rate due to cache pollution
  - streaming (no reuse)
    - A-B-C-D-E-F-G-H-I-...
  - thrashing (distant reuse)
    - A-B-C-A-B-C-A-B-C-...

- A large fraction of the cache is useless – blocks that have serviced their last hit and are on the slow walk from MRU to LRU
Basic Cache Policies

- Insertion
  - Where is incoming line placed in replacement list?

- Promotion
  - When a block is touched, it can be promoted up the priority list in one of many ways

- Victim selection
  - Which line to replace for incoming line? (not necessarily the tail of the list)

Simple changes to these policies can greatly improve cache performance for memory-intensive workloads
Inefficiency of Basic Policies

- About 60% of the cache blocks may be dead on arrival (DoA)
Adaptive Insertion Policies

- MIP: MRU insertion policy (baseline)
- LIP: LRU insertion policy

MRU: a → b → c → d → e → f → g → h

Traditional LRU places ‘i’ in MRU position.

LIP places ‘i’ in LRU position; with the first touch it becomes MRU.

(Qureshi’07)
Adaptive Insertion Policies

- LIP does not age older blocks
  - A, A, B, C, B, C, B, C, ...

- BIP: Bimodal Insertion Policy
  - Let $\varepsilon = \text{Bimodal throttle parameter}$
    
    ```
    if ( rand() < \varepsilon )
        Insert at MRU position;
    else
        Insert at LRU position;
    ```

[Qureshi’07]
Adaptive Insertion Policies

- There are two types of workloads: LRU-friendly or BIP-friendly
- DIP: Dynamic Insertion Policy
  - Set Dueling

Read the paper for more details.

[Qureshi’07]
Adaptive Insertion Policies

- DIP reduces average MPKI by 21% and requires less than two bytes storage overhead

[Qureshi’07]
Re-Reference Interval Prediction

- **Goal**: high performing scan resistant policy
  - DIP is thrash-resistance
  - LFU is good for recurring scans
- **Key idea**: insert blocks near the end of the list than at the very end
- **Implement** with a multi-bit version of NRU
  - zero counter on touch, evict block with max counter, else increment every counter by one

Read the paper for more details.

[Jaleel’10]
Shared Cache Problems

- A thread’s performance may be significantly reduced due to an unfair cache sharing
- Question: how to control cache sharing?
  - Fair cache partitioning [Kim’04]
  - Utility based cache partitioning [Qureshi’06]
Utility Based Cache Partitioning

- Key idea: give more cache to the application that benefits more from cache

![Graph showing misses per 1000 instructions (MPKI) for different cache partitioning strategies. The graph compares equake and vpr with UTIL and LRU.]
Utility Based Cache Partitioning

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

[Qureshi’06]
Highly Associative Caches

- Last level caches have ~32 ways in multicores
  - Increased energy, latency, and area overheads

[Sanchez’10]
Recall: Victim Caches

- Goal: to decrease conflict misses using a small FA cache

Can we reduce the hardware overheads?
The ZCache

- Goal: design a highly associative cache with a low number of ways
- Improves associativity by increasing number of replacement candidates
- Retains low energy/hit, latency and area of caches with few ways
- Skewed associative cache: each way has a different indexing function (in essence, $W$ direct-mapped caches)

[Sanchez’10]
The ZCache

- When block A is brought in, it could replace one of four (say) blocks B, C, D, E; but B could be made to reside in one of three other locations (currently occupied by F, G, H); and F could be moved to one of three other locations.

Read the paper for more details.

[Sanchez’10]