LARGE CACHE DESIGN

Mahdi Nazm Bojnordi

Assistant Professor
School of Computing
University of Utah
Overview

- Upcoming deadline
  - Feb. 3rd: project group formation

- This lecture
  - Gated Vdd/ cache decay, drowsy caches
  - Compiler optimizations
  - Cache replacement policies
  - Cache partitioning
  - Highly associative caches
Main Consumers of CPU Resources?

- A significant portion of the processor die is occupied by on-chip caches

- Main problems in caches
  - Power consumption
    - Power on many transistors
  - Reliability
    - Increased defect rate and errors

Example: FX Processors

[source: AMD]
Leakage Power

- dominant source for power consumption as technology scales down

\[ P_{\text{leakage}} = V \times I_{\text{leakage}} \]

[src of data: ITRS]
Gated Vdd

- Dynamically resize the cache (number of sets)
- Sets are disabled by gating the path between Vdd and ground ("stacking effect")

Other possibilities, e.g., virtual Vdd (see paper)

Shared among cells in same row (5% total area cost)

[Powell00]
Gated Vdd Microarchitecture

threshold above/below which cache is upsize/downsized

number of instructions between resizings

[Powell00]
Gated-Vdd I$ Effectiveness
due to additional misses

High mis-predication costs!

[Powell00]
Exploits generational behavior of cache contents

Access Interval

M : Miss  H : Hit

Live time

Dead time

NEW Generation

Last Access

NEW Generation

100-500 cycles

1,000-500,000 cycles

[Kaxiras01]
Cache Decay

- Fraction of time cache lines that are “dead”

32KB L1 D-cache

[Kaxiras01]
Cache Decay Implementation

High mis-predication costs!

[State Diagram for 2-bit (S1,SO), saturating, Gray-code counter with two inputs (WRD, T)]

[Kaxiras01]
Drowsy Caches

- Gated-Vdd cells lose their state
  - Instructions/data must be refetched
  - Dirty data must be first written back

- By **dynamically scaling** Vdd, cell is put into a **drowsy** state where it retains its value
  - Leakage drops superlinearly with reduced Vdd ("DIBL" effect)
  - Cell can be fully restored in a few cycles
  - Much lower misprediction cost than gated-Vdd, but noise susceptibility and less reduction in leakage
Drowsy Cache Organization

Keeps the contents (no data loss)

[Kim04]
Drowsy Cache Effectiveness

32KB L1 caches
4K cycle drowsy period

[Kim04]
Drowsy Cache Performance Cost

[Kim04]
Software Techniques
Compiler-Directed Data Partitioning

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

Diagrams showing read access, write access, and idle intervals for variables, with an illustration of data banks and their idle times.
Chapter Two
Memory Hierarchy Design

Compiler Optimizations

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

  /* 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
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)

[Qureshi’07]
Adaptive Insertion Policies

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

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 $\varepsilon = \text{Bimodal throttle parameter}$

```java
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-resistant
  - 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

Misses per 1000 instructions (MPKI)

- equake
- vpr
- UTIL
- LRU

[Qureshi’06]
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]
A Cache for Approximate Computing

Two data blocks are *approximately similar* (i.e., *doppelgängers*) if replacing the values of one with the other still results in acceptable application output in the end.

[Image of a highway with numbers and values]

[Miguel’15]
Doppelgänger

- Conventional Cache

[Image of a diagram showing a conventional cache with a tag array and a data array. The diagram includes an arrow labeled "address from L2" pointing to the tag array, and another arrow labeled "data from memory" pointing to the data array.]

[Miguel’15]
Doppelgänger

Approximate Blocks

- Address 6 from L2

- Tag array:
  - tag 0: map X
  - tag 6: map X
  - tag 5: map Y
  - tag 1: map X
  - tag 2: map X
  - tag 3: map X

- Approximate data array:
  - map X: data block A
  - map Y: data block B

- Generate map X from data C

- Data C to L2

- Data C from memory

[Miguel’15]
The **map** value represents the signature (or **likeness**) of a block. Blocks that generate the same map value are approximately similar.

**Diagram:**

- **Data block A:** `A[0]` to `A[n]`
- **Hash function:** `hash`
- **Mapping:** `map`
- **Aggregates values in block:**
  
  \[ \text{hash} = \text{AVG}(A[0], ..., A[n]) \]

- **Discretizes hash value:**

  
  - **All possible hash values**
  - **map (M-bit)**

---

[Michael’s 15]