Imbalanced Cache Partitioning for Balanced Data-Parallel Programs

Abhisek Pan & Vijay S. Pai
Electrical and Computer Engineering
Purdue University
MICRO-46, 2013
Motivation

• Last level cache partitioning heavily studied for multiprogramming workloads
• Multithreading ≠ multiprogramming
  ▫ All threads have to progress equally
  ▫ Pure throughput maximization is not enough
• Data-parallel threads are similar to each other in their data access patterns
• However equal allocation => suboptimal cache utilization
Motivation

• Last level cache partitioning heavily studied for multiprogramming workloads
• Multithreading ≠ multiprogramming
  ▫ All threads have to progress equally
  ▫ Pure throughput maximization is not enough
• Data-parallel threads are similar to each other in their data access patterns
• However equal allocation => suboptimal cache utilization

Balanced threads need highly imbalanced partitions
Contributions

- Shared LLC partitioning for balanced data-parallel applications
- Increasing allocation for one thread at a time improves utilization
- Prioritizing each thread in turn ensures balanced progress
- 17% drop in miss rate, 8% drop in execution time on average for 4-core 8MB cache
- Negligible overheads
Outline

• Motivation
• Contributions
• Background
• Memory Reuse Behavior of Threads
• Proposed Scheme
• Evaluation
• Overheads & Limitations
• Conclusion
Way-partitioning

• **N-way set-associative cache** = > each set has N ways or blocks

• **Unpartitioned cache**
  ▫ Least recently used entry among all ways replaced on a miss
  ▫ Thread-agnostic LRU

• **Way-partitioning**
  ▫ Each way is owned by one core at a time
  ▫ On a miss, a core replaces the LRU entry among the ways owned by it
  ▫ No restriction on access, only on replacement
Per-thread Miss Rate Curves

- Miss-rate vs. ways in a single set
- Each thread considered in isolation
Per-thread Miss Rate Curves

- Miss-rate vs. ways in a single set
- Each thread considered in isolation

![Graph showing miss rate vs. ways for different working sets and threads. The graph indicates inefficient allocation.](image-url)
Symmetric Memory Access

- Miss-curves symmetric across threads
- Seen for all benchmarks & cache sizes
Utilization through Imbalance

**WS too small**

**WS too large**

**Improvement opportunity**

- 32 way cache, 4 threads – default allocation = 8 ways / thread
- Prioritize one thread at a time
- Vary preferred thread identity
Utilization through Imbalance

**WS too small**

**WS too large**

- 32 way cache, 4 threads – default allocation = 8 ways / thread
- Prioritize one thread at a time
- Vary preferred thread identity

Imbalance in partitions benefits the preferred thread
High Imbalance & Unpreferred threads

- Each thread switches between preferred and un-preferred
- Unpreferred thread data remains in preferred partition
- Continues to benefit un-preferred thread even as its partition shrinks
- Imbalance magnifies benefits by reducing pressure on preferred partition
High Imbalance & Unpreferred threads

- Each thread switches between preferred and un-preferred
- Unpreferred thread data remains in preferred partition
- Continues to benefit un-preferred thread even as its partition shrinks
- *Imbalance magnifies benefits by reducing pressure on preferred partition*

Large preferred partition benefits unpreferred threads too
Proposed Strategy

- Default allocation is inefficient
- Allocate extra ways to a single thread by equally penalizing all other threads
- Select the preferred thread in round-robin manner
  - Ensure balanced progress
- Allocation changes at pre-set execution intervals
Two-Stage Partitioning

• Evaluation Stage
  ▫ Triggers at the start of a new program phase
  ▫ Divide the cache sets into equal-sized segments
  ▫ Each segment is partitioned into a different level of imbalance
  ▫ 32 way cache shared among 4 cores – configurations from 8-8-8-8 -> 29-1-1-1
  ▫ Each core is prioritized in turn
  ▫ Configuration with least number of misses chosen
Evaluation Stage Cache

- Each segment has multiple sets
- Each thread becomes the preferred thread in turn
Evaluation Stage Cache

Ways

Segments

1 2 3 4 5 6 7 8

Thread 1  Thread 2  Thread 3  Thread 4

- Each segment has multiple sets
- Each thread becomes the preferred thread in turn

Capture effects of imbalance on preferred and unpreferred threads
Considering Unpartitioned Cache

- An unpartitioned (thread-agnostic LRU) segment included in evaluation

- Replace a low-imbalance configuration

- Benefits of partitioning are obtained through high levels of imbalance
Unpartitioned Segment

Ways

Segments

1 2 3 4 5 6 7 8

Thread 1  Thread 2  Thread 3  Thread 4

No partitions
Stable Stage

- Maintain the chosen configuration till the next program phase change
- Choose preferred thread in round-robin manner
- Basic-block vector tracking used to identify changes in program phase (based on previous work)
Evaluation Framework

- **Simulator:** Simics-GEMS
- **Target:** 4-core CMP with 32 way shared L2 cache, and 2 way private L1 caches
  - 1 thread per core, 64 byte line size, LRU replacement
- **Workload:**
  - 9 data-parallel workloads
  - Mix of parsec (pthread build) and SPEC OMP suite
  - Parsec - Blackscholes, Canneal, Fluidanimate, Streamcluster, Swaptions
  - SPEC OMP – Art, Equake, Swim, Wupwise
Baselines

- Unpartitioned cache (thread-agnostic LRU)
- Statically equi-partitioned cache
- A CPI-based adaptive partitioning scheme
  \(\text{(Muralidhara et al., IPDPS 2010)}\)
  - Starts with equal partition
  - Proportional partitioning (ways proportional to CPI)
  - Store <ways, CPI> to build a runtime model to predict CPI variations with change in allocation
  - Accelerate critical thread
**Misses vs size**

*4-core 32-way cache with equal partitions*

- Art
- Blackscholes
- Canneal
- Equake
- Fluidanimate
- Streamcluster
- Swaption
- Swim
- Wupwise

**Cache Size (Bytes)**

- 128K
- 256K
- 512K
- 1M
- 2M
- 4M
- 8M
- 16M
- 32M
- 64M
- 128M
- 256M
- 512M

**Misses**

- $1.0 \times 10^3$
- $1.0 \times 10^4$
- $1.0 \times 10^5$
- $1.0 \times 10^6$
- $1.0 \times 10^7$
- $1.0 \times 10^8$
- $1.0 \times 10^9$
Results

• Benefits of partitioning strongly tied to cache size

• Partitioning beneficial only when per-thread working set is between the default allocation and the cache capacity

• Proposed method outperforms the baselines where there is potential for benefit
Comparison with Unpartitioned: 8 MB cache, 4 cores, 32 ways

![Misses and Execution Time Graphs]

MISSES

Execution Time

MICRO-46, 2013
Comparison with Unpartitioned: 32 MB cache, 4 cores, 32 ways

![Bar chart showing misses and execution time for different benchmarks.](chart.png)

- **Misses**: STATIC-EQ, CPI, IMB-RR
- **Execution Time**: STATIC-EQ, CPI, IMB-RR

**Benchmarks**:
- Swappons
- Blackscholes
- Art
- Streamcluster
- Canneal
- Fluidanimate
- Equake
- Swim
- Wupwise

MICRO-46, 2013
Comparison with Unpartitioned:  
128 MB cache, 4 cores, 32 ways

Comparison with Unpartitioned:

- 128 MB cache, 4 cores, 32 ways

Comparison with Unpartitioned:

- 128 MB cache, 4 cores, 32 ways
Across the Board

• Outperforms the CPI-based in most cases where there is potential for benefit
  ▫ Proportional partitioning generates data points near the default allocation
  ▫ From these starting points the search fails to find the high-utility (high-imbalance) configurations

• No partitioning is best in some cases (Equake)
  ▫ Constructive interference
  ▫ Proposed scheme chooses global LRU appropriately
  ▫ Worst-case 5% increase in time due to evaluation
Overheads

• Space overhead negligible
  ▫ Way partitioning for each segment

• Program phase detection overhead
  ▫ Basic block vector tracking

• For small cache sizes, evaluation stage can increase execution time
  ▫ <1 % on average, 5 % maximum
Limitations

• Scalability
  ▫ Fine-grained barriers would mean smaller intervals

• Limited exploration of solution space
  ▫ One preferred thread at a time
  ▫ The benefits of high imbalance makes the scheme practical
Conclusion

• Simple runtime partitioning for balanced data-parallel programs
• Effective cache utilization and balanced progress achieved through
  A. High Imbalance in partitions and
  B. Prioritizing each thread in turn
• High imbalance allows un-preferred threads to benefit from the large preferred partition
Thank You!

Questions...
Injecting Extra Imbalance

- Over-allocation in preferred thread protects long distance accesses of unpreferred thread

Long RD accesses not served by shrinking partition

Long RD accesses are protected in preferred partition
Effect of Over-allocation

- Benefits to preferred thread saturate at 14 ways
- Benefits to un-preferred thread increase as allocation falls
- Hits for un-preferred thread are in preferred thread partition

Thread in preferred state  Thread in un-preferred state

- Benefits to preferred thread saturate at 14 ways
- Benefits to un-preferred thread increase as allocation falls
- Hits for un-preferred thread are in preferred thread partition
Adapting to Phase Changes

• Changes in program phase need to be identified to trigger evaluation
• Per-thread binary basic block vectors are used to identify the basic blocks touched in each interval
• Hamming distance between the BBVs of current and last intervals are compared to identify phase changes
Considering Unpartitioned Cache

- Time spent in various imbalance configurations for runs showing benefits of partitioning

![Bar chart showing time spent in various allocation configurations.]

MICRO-46, 2013
Round-robin vs. Critical-thread

- Prioritize the critical thread instead of using round robin
- No significant difference – Accelerating critical thread has the same effect as giving each thread a fair share

Performance of critical-thread normalized to round-robin