Generic talk structure

- Model and motivation
- Results
- Open problems
Can theory help?

- We need a well defined question to work on
- This talk:
  - A new model for multicore machines
    - Abstracts a different aspect
  - A good question
    - No good answers – I have come to ask for help
Previous theoretical work

- Most theoretical work dealt with dividing a single task (e.g. matrix multiplication) between many cores

- We are interested in the other end of the spectrum
  - Each core runs its own task
  - Models a server with multiple clients, or an OS which has multiple applications

- May look trivial, but:
  - The cores share the same motherboard, and thus dependencies are formed
  - This work focuses on the cache
Cores make requests to the L1 cache. If the data is not there, a miss occurs and they fetch the data from the L2 cache. If another miss occurs, the data is fetched from main memory.

Cache architecture in an Intel machine

- Each core has a private L1 cache.
- All cores share an L2 cache.
- L2 cache implements Least Recently Used (LRU).

This work

Goal: Develop an efficient replacement policy for the L2 cache.

Simplifying assumption: Each core runs an independent process → no coherency

Simplest case: Can be used with a mechanism which ensures coherency [LL08], or with different architectures.
Input: n lists of requests. Different cores request different letters (memory is disjoint)

- \( t=1 \): All cores make requests
- \( t=2 \): All cores which received a memory address request a new one
- \( t=\tau+1 \): Cores who suffered a miss at time 1 make a request
- Bandwidth between L2 cache and memory is unbounded
- Goal: Minimize makespan (last core to finish)

\[
\tau = \frac{\text{miss time}}{\text{hit time}}
\]
Challenge: Design a good online algorithm for this caching problem

Bad news: LRU is inefficient (as well as other algorithms)

Every solution for caching when some memory addresses are shared must solve this as a special case
Competitive analysis

- Competitive analysis considers the ratio between the performance of an online algorithm and that of the optimal algorithm.
- Consider the behavior of Least Recently Used, with a single core and a cache of size k:

\[ x_1, x_2, \ldots, x_{k+1} \]

- LRU misses all requests. The optimal algorithm for a cache of size k saves \( x_1, \ldots, x_k \), and misses only requests for \( x_{k+1} \).
- The ratio between the number of misses is k, and thus LRU is k competitive
Resource augmentation

- In practice (for single core), LRU performs well.
- This motivated Sleator and Tarjan (ST85) to consider the ratio between LRU’s performance with a cache of size k, to the optimal algorithm with a cache of size (say) k/2.
- In the previous example, the optimal algorithm saves \(x_1 \ldots x_{k/2}\) and misses on \(x_{k/2+1} \ldots x_{k+1}\).
- ST85: This is the worst sequence. LRU with double the cache takes at most twice the time.
Thm: LRU has competitive ratio $O(\tau)$ even with resource augmentation (for any constant factor)

A competitive ratio of $\tau$ can be obtained by missing all the requests

$\Rightarrow$ LRU is not performing well.
LRU is inefficient

- Proof: Assume the adversary has a cache of size $k$, and LRU has a cache of size $ck$. Let $n=c+1$ cores.

LRU answers the requests column by column. When LRU gets to column $kc/c+1$ it throws away the first column. Thus LRU misses all the time, total time $k\tau^2$.

The optimal algorithm takes care of one core at a time. Runtime for each core is $2k\tau$. Total runtime is $2(c+1)k\tau=O(k\tau^2)$.

The result also holds when we try to optimize different measures (e.g. number of misses), and when the bandwidth between the cache and the memory is bounded.
Where does the problem come from?

- Suppose each core has a working set which is slightly smaller than size of the cache.
- If all cores try to share the cache, they will all thrash (no working set will be in the cache)
- If each core uses the entire cache part of the time, it can make progress.
- Same example (and intuition) show that MARK is bad (and other algorithms as well)
More results

- The offline problem of finding the optimal allocation is **NP complete**
- There is a PTAS, for a limited choice of parameters
- An optimal solution is just a partition of the cache between different cores (given this partition it is easy to know which page to evict).
Open problems

- A good online algorithm
  - A better offline approximation algorithm (see the paper)
  - NP complete when the number of cores is constant?

- Understanding the practical significance of the theoretical results

- Other architecture questions. What else happens assuming that different cores run different processes?
Thanks!