Lab 3 – Interference in the Memory System

18740: Shravani Dhote, Simrit Kaur, Vins Sharma

# Task 2: Baseline, Stat! (Again)

We can compute maximum slowdown as:

The weighted speedup can be computed as:

# Task 3: Way Partitioning

The base code emulates an 8-way memory structure. In order to modify this, we noticed that the list was a bounded list of lists, which themselves were bounded to objects in .

Our implementation strategy was simple – In order to add in core-locked ways *without* modifying the cache’s overall functionality, associativity, block sizing, set sizing, etc, and still reuse the base code, we cut down the number of ways to and duplicated the list of cache line sets by . This gave us -way associativity overall, where every set of ways was specified to a particular core, without changing the number of sets, block sizings, or any significant functionality.

This required us to change some of the function headers to ensure that core information would be passed down through various call chains such that proper eviction was possible.

Our way partitioning implementation was significantly fairer than the baseline implementation, receiving a much larger weighted speedup. However, the performance was significantly worse.

# Task 4: BLISS

We implemented BLISS by tracking information in the controller and utilizing that information to compare two requests in the scheduler.

Interestingly, we found that (by functional specification from the PDF) it was possible for all cores to be locked up with no memory access until the next cycle quanta. If one core performed requests back-to-back, it would be blacklisted, and the other cores were capable of sending requests much faster, increasing their chances of being blacklisted, etc. As such, the “blacklisting” part of BLISS would stop being relevant after three cores lock up – After the next consecutive requests from one core, all cores would be blacklisted, and would be served in a first-come first-serve basis until the start of the next cycle quanta (and the blacklisting would have no significant effect on performance).

# Task 5: A custom scheduler

The following are a series of graphs of all of our scheduling mechanisms, overlaid upon each other to compare performance.

## Equity scheduler

Our original idea was to evolve BLISS’s mechanism to better improve performance. In particular, having blacklists was an interesting idea to solve fairness, but as mentioned earlier, it did not significantly allow us to be “fairer” to threads – Instead, with two blacklisted threads or two non-blacklisted threads, it simply boiled down to first-come first-serve in most cases. Threads that were memory hogs would compete with other threads that were memory hogs quickly.

We wanted to take this blacklisting concept and flip it – That is, instead of deprioritizing memory hogs, inversely prioritize stingy threads that did not make many requests. This led to our implementation of equity scheduler, which computes the number of requests a thread would make and allows it access correspondingly. Any ties are still broken with first-come first-serve.

This resulted in a decent amount of performance, but it’s not as fair as way partitioning.

## EquiBLISS scheduler

Our next idea was to leverage BLISS’s performance gains but still add another key of further incentivizing low usage to particular cores. As such, we implemented BLISS alongside our equity scheduler, with our equity scheduler being the tiebreaker for any BLISS operations and having FR-FCFS as our final tiebreaker.

The results of this showed improvements on our prior Equity scheduler overall, but less performance gain than BLISS alone and less fairness than way partitioning alone.

## EquiBLISSPart Scheduler

We wanted to leverage some of the extreme fairness of way partitioning with the significant performance increases of BLISS and Equity. As such, we combined the implementation structures together, partitioning our cache into equal sets of ways per program.

Our results showed that this implementation was significantly fairer than all other prior implementation schemes, but our performance suffered significantly (being even worse than way partitioning).

## Uneven Ways Scheduler

We know that our programs , , , and had particular program counts. Looking at the percentage composition of these values, we noticed that we could partition particular numbers of ways to each program. Ideally, this would reduce the cache miss ratio significantly, offering noticeable increases of performance, and would be “fairer” as programs that are more demanding would be given more memory to play with (but still be blacklisted in scheduling, so memory hog programs would still be deprioritized).