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:

$$\max_i \frac{IPC_i^{alone}}{IPC_i^{shared}}$$

The weighted speedup can be computed as:

$$\sum_i \frac{IPC_i^{shared}}{IPC_i^{alone}}$$

Task 3: Way Partitioning

The base code emulates an 8-way memory structure. In order to modify this, we noticed that the $cache\_lines[]$ list was a bounded list of $Line[]$ lists, which themselves were bounded to 8 objects in $need\_eviction[]$.

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 2 and duplicated the $cache\_lines$ list of cache line sets by 4. This gave us 8-way associativity overall, where every set of 2 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.

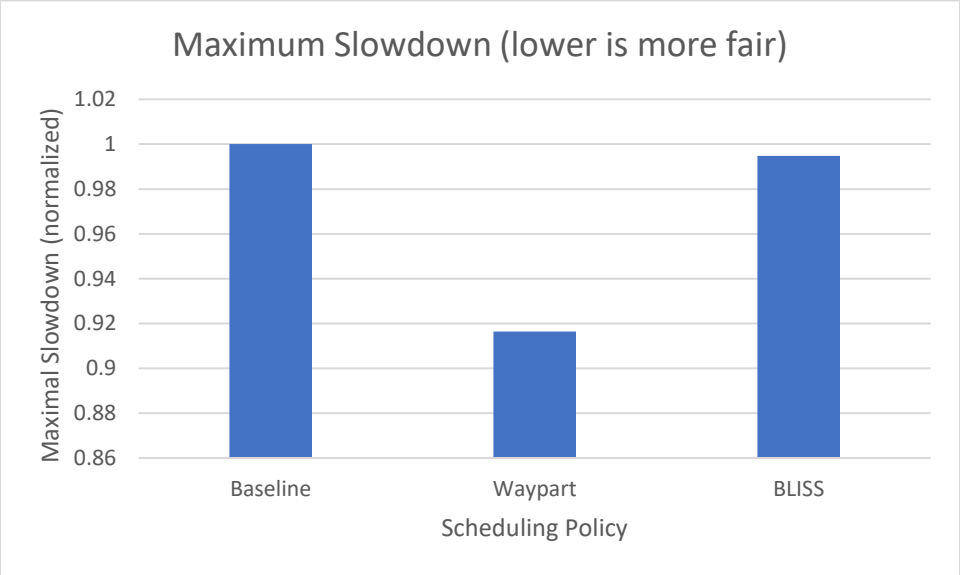
Task 4: BLISS

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 $10{,}000$ cycle quanta. If one core performed 5 requests back-to-back, it would be blacklisted, and the other 3 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 5 consecutive requests from one core, all cores would be blacklisted, and would be served in a first-come first-

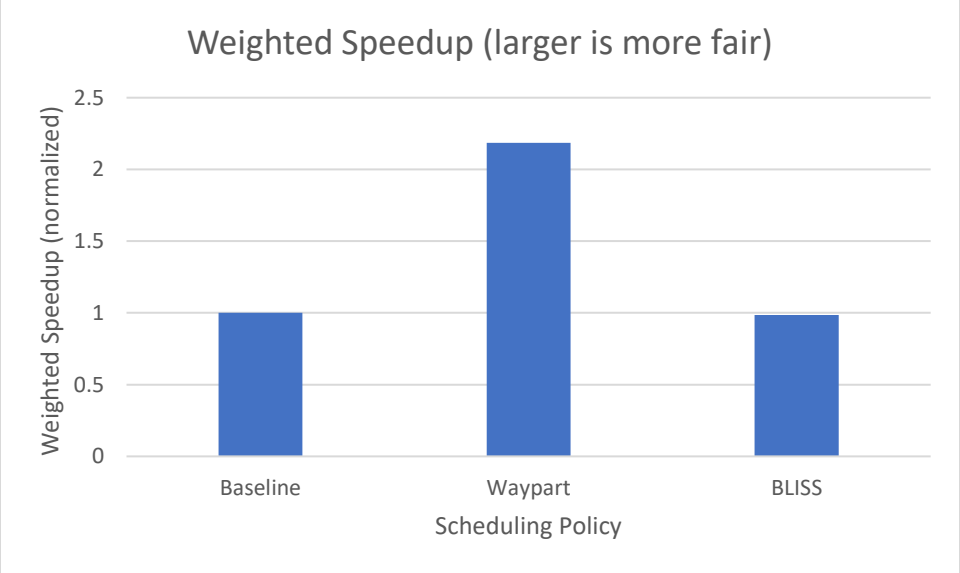
serve basis until the start of the next $10{,}000$ cycle quanta (and the blacklisting would have no significant effect on performance).

The following is a graph of our maximum slowdown:



BLISS is mildly fairer than the baseline, but way partitioning is significantly more fair than both.

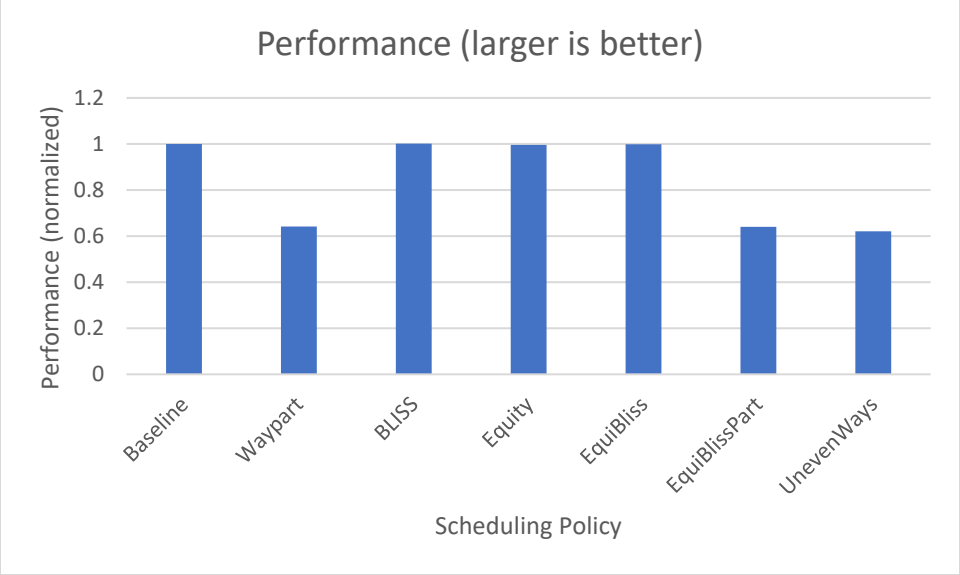
The following is a graph of our weighted speedup:



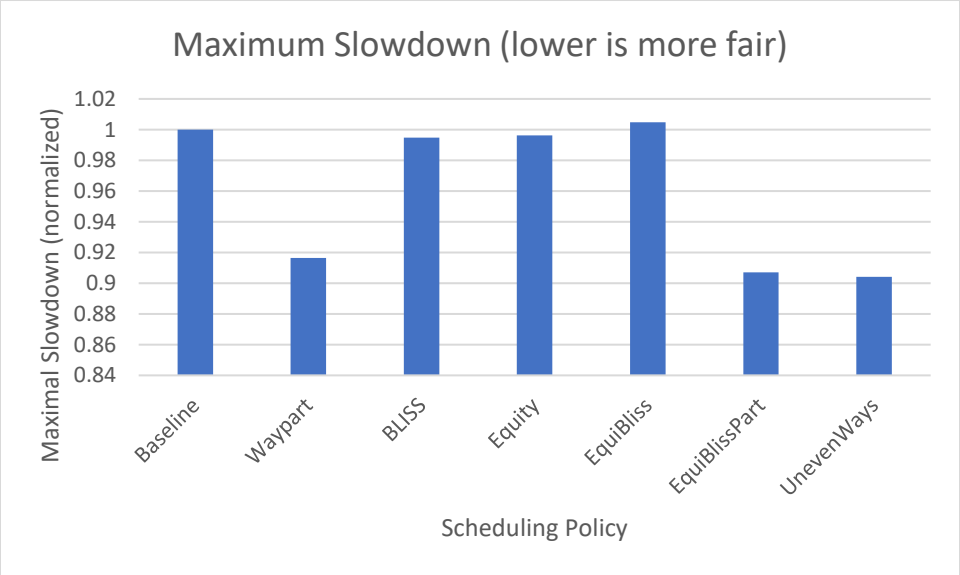
BLISS has mildly more interference from other threads (due to the blacklisting of memory hog threads causing other threads to overwork), whereas way partitioning has significantly lower interference between threads.

Task 5: A custom scheduler

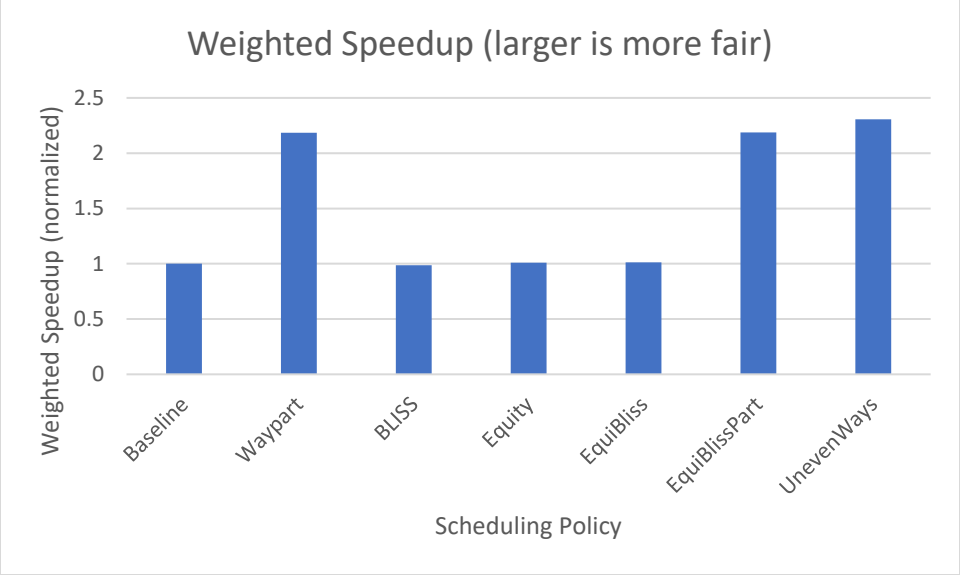
The following are a series of graphs of all of our scheduling mechanisms, overlaid upon each other to compare performance and fairness. First, our performance graph:



Our later implementations sacrificed performance heavily in favor of fairness of memory scheduling. We can see significant improvements on our slowdown graph:



As well as our weighted speedup graph:



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 2 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).

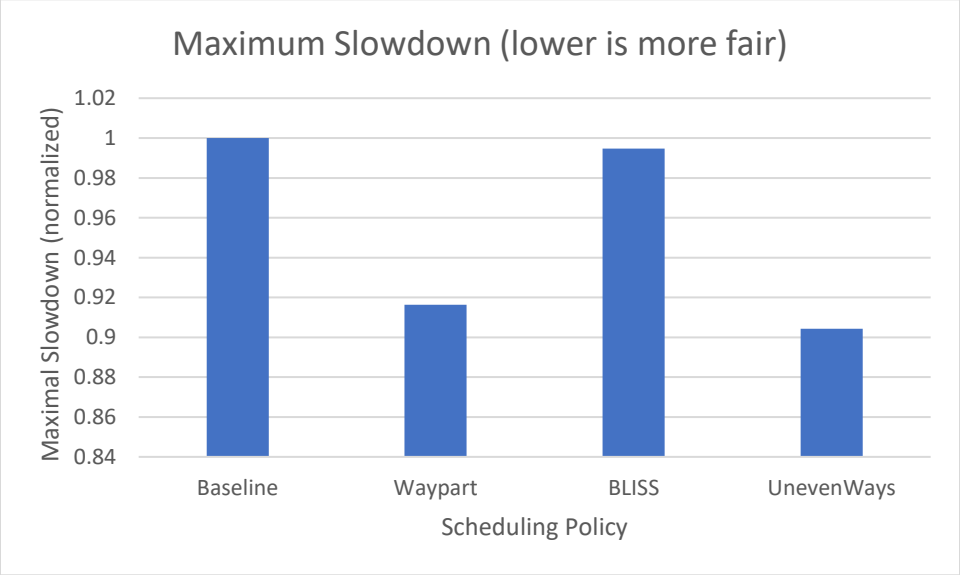
Uneven Ways Scheduler

We know that our programs gcc, mcf, $milc$, and $omnetpp$ 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).

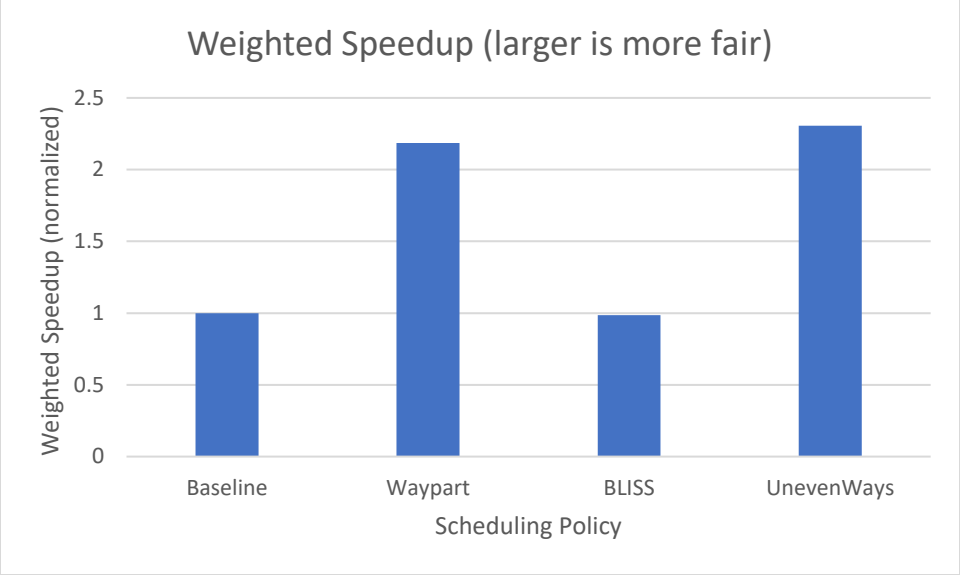
This implementation is significantly less performant than all prior implementations. However, it is noticeably fairer than all of our implementations as well.

Our final implementation

Our final custom implementation that we went with was based entirely on fairness, and *not* in any significant way off performance. As such, here is the graph of our maximum slowdown compared to tasks 3 and 4:



And the following is our weighted speedup:



These results demonstrate that our custom implementation Uneven Ways Scheduler is fairer than our other implementations.

Part A

Our fairness mechanism has significant components of cache structure and scheduler configuration.

Part B

On the cache configuration level, we noticed that our programs are significantly weighted such that one program utilizes significantly more resources than the other. In order to reduce cache miss ratios for that larger load and increase fairness, we decided to partition more ways to that particular core as opposed to other cores. As such, gcc gets 1 way, mcf gets 4 ways, $milc$ gets 2 ways, and $omnetpp$ gets 1 way as well.

The scheduler program utilizes a triple-layered check structure. First, BLISS is used to blacklist memory hog threads. Any ties will then move on to our equity scheduler, which counts the number of memory requests serviced to each individual core and simply prioritizes the cores that have not been served as much (fairly). If two cores have been served the exact same amount, the request that comes first gets served first.

Part C

On the cache structure: The amount of effective memory is partitioned such that each program gets more if they demonstrably need more, so ideally smaller programs will take a hit (that is hopefully insignificant) and larger programs will incur less of a penalty from having too little cache memory.

Our scheduler protocol works seemingly against this – Cores that hog too much memory in terms of requests will get rate-limited against cores that do not. Alongside the cache structure, this means that cores which encounter more misses (the cores with lesser ways partitioned) will hog more, and allow the larger cores (with larger caches) to process more, which is fairer, since they are in fact

larger (and need to do more work). By blacklisting certain cores that hog bandwidth, we increase the capabilities of other cores by allowing them to get more work done in the same period of time. This leads to more fairness on its own.

Part D

The majority of our changes had to deal with propagation of data throughout our structure.

We noticed that the function $need\_eviction()$ managed the number of ways by utilizing the $assoc$ parameter. As such, we modified this check to cut down the number of ways for our cache design, and quadrupled the number of cache sets (effectively, we modified an 8-way n-set structure to four k_i-way, n-set structures, where $\sum_i k_i = 8$.) This does not functionally change the memory, but it *does* allow for per-core partitioning. It's important to note that this **does not change our design's overall associativity**, or the sizing of the cache, in any significant way – It just simulates the splitting of cores in a more manageable way. The cache is still one singular memory block of the exact same sizing and associativity.

In order to do this, we had to modify our cache retrieval. We created a separate function from $get\_lines()$ in the headerfile ($get\_lines\_waypart()$) to pick the particular cache set group for a particular core. This physical separation of virtual memory ensured that, throughout the simulation, no cores could ever access any information from other cores.

We also had to modify other functions to ensure that our new $cache\_lines\_wp$ lists were being utilized in eviction, etc. To do this, we needed to modify the function signatures of:

- $need\_eviction()$,
- $allocate\_line()$,
- $evict()$,
- $invalidate()$,
- And $evictline()$.

Finally, our $get\_lines\_wp()$ call replaced any $get\_lines()$ calls in our custom cache structure.

In our scheduling mechanism, we simply needed to add trackers to the controller class and compute them (in the $tick()$ function). From there, our actual scheduling policy was done in the scheduler, all contained within the custom function of the list.