Cache Simulator Project Report

Group 11: Bilal **Siddiqui**, Zhongxiu **Yang**, Trung **Dinh**

CS 5513
05/09/2019

• A brief description of the implementation of the simulator

The simulator uses vanilla python3 to implement basic address translations. The address translations can be configured using command-line arguments that specify the attributes of a L1 cache.

For example:

`> ./sim.py <sequence_file>`
will run the simulator with it's default configuration that involves:

- cache size = 1MB
- cache line size = 64B
- associativity = 16 ways

All these parameters can be configured to the needs of an individual tester. The **sim.py** script will require new command-line inputs when/if *more* than one parameter is passed to the script.
For example:
`> ./sim.py <sequence_file> [line_sz] [cache_sz] [n_ways]`
where:

- **`sequence_file`** is the path to the memory trace file
- **`line_sz`** is the block size in the L1 cache
- **`cache_sz`** is the size of the whole L1 cache
- **`n_ways`** is set associativity of the L1 cache

note: all parameters must be specified when changing from the default configuration

The core logic of the simulator is located in the **`cache.py`**, with **`sim.py`** and **`parser.py`** acting as helper modules that drive and parse the trace data respectively.

<sim.py>

The sim module servers as the driver for the L1 cache simulator. As briefly shown above, execution begins in the main function of sim.py where user input is checked to determine which trace file to utilize for testing, and whether the cache should be configured in a manner different from the default settings.

Once the cache is configured either to it's default or to a user specification, a list of cache objects is created for the simulation purposes. Next, a path to the memory sequence file (which is the first cmd-line parameter to the module) is passed to a parser that yields back results, one line at a time. The results are then translated and sent to the **`cache.py`** module for lookup.

We use simple counters to determine the number of total cache accesses and misses. These counters are updated on each cache lookup and are finally used to compute the **miss ratio**.

<parser.py>

The parser module takes in raw data from a sequence file and then utilizes a generator to yield results, line-by-line, to the caller of the **`parser.py`** module. The module's generator function, **`get_data`** is used to send back data as three tokens: **`program counter`** `(as an int)`, **`mode`** `(as a char)`, and **`virtual address`** `(as a int)`. Using a generator allows us to comfortably accommodate large trace files while leaving the OS to implicitly take care of prefetching data for better cache locality on the host system which runs the simulator.

<cache.py>

The cache module was initially designed with a few objects the implemented python's `OrderedDict()` in that it was a linked list of dictionaries where the dictionary corresponded to a set and the position in the linked list corresponded to the cache replacement policy.

We decided to scrap the effort into our custom data structure implementation and use Python's `OrderedDict()` for managing tags inside an object that includes fields for replacement policy and set associativity. The object is called cache_set, here is a snippet:

```
class cache_set:
	def __init__(self, r_policy, n_way):
		self.n_way = n_way;
```

```
        self.r_policy = r_policy;
        self.tags = OrderedDict();
```

The driver module, (sim.py), we use the cmd-line parameters to determine the number of sets are will be in the L1 cache and then create a normal list of set **`cache_set`** objects accordingly. Thereafter, we simply use bitwise arithmetic to extract the offset, set index, and tag for each address so that we can access the cache in a **write back** strategy. That is, each access to the cache can possibly alter/update the contents of the cache according to the memory access. The set index provides as an index into the cache_set object list and each object has a **`search()`** function that checks:

1. is the tag present in the cache? if, so update its 'position' in the set based on the replacement policy
2. is the cache empty? if so, allocate a block for the memory access
3. is the cache full and no tag hit? evict a block based on the replacement policy and update the set with the new block (tag)

Note: The entries inside each **`OrderedDict()`** are of a **cache_line()** object that simply mirrors the contents of a typical cache line: data and control bits. For the purposes of our simulator, although the **cache_line()** object is updated as it would be irl, the updates are only really needed for the implementation of the Second Chance algorithm because it needs the SC bit to be changed for blocks in the set. The other algorithms only need tag info and last modification info and **`OrderedDict()`** inherently remembers positional information of its members and we used tags for it's keys.

Source Code

https://github.com/slmatrix/cache

• A description of your experiments and the experiment results with figures and/or tables

The cache simulator is tested with the aid to two bash scripts[1], each over 800 lines long, and run for a combined total of fifteen hours. The scripts essentially go over **every** sequence file provided for the project and simulate memory accesses with a different L1 cache configurations.

The first script keeps a control on the *block* size by fixing it to 64 bytes and changing all other parameters (the cache size, associativity and eviction policy). The second script somewhat similarly puts a control on the *cache* size by fixing it to 1 mega-bytes and changing all other parameters such as cache-line size, associativity and eviction policy.

The testing serves both as a comparison between different caching techniques and as a sanity check on the correctness of our implementation.

Synthetic Benchmark Evaluations

The synthetic benchmarks simulate sequential memory accesses. Therefore, a cache size does not matter much since the misses are due to compulsory misses; the memory needs to be in the cache for subsequent accesses.

Interestingly, when the block size is increased to sizes greater than 64 bytes, misses go down significantly. The idea is that, with larger block sizes, more data can be prefetched and therefore increasing cache locality which in turn reduces misses.

Normal Applications' Benchmark Evaluations

Common programs such as GCC and LS exhibit behaviour similar to the synthetic process, albeit with a much less exaggerated compulsory miss rate. Real world applications tend to reuse already allocated memory blocks more frequently, and thus have fewer misses as long as the L1 cache size is large enough.

In our testing, increasing associativity did not significantly reduce cache misses and we found that larger block sizes were the most impactful for reducing cache misses.

[1] provided in the source code on the previous page

1KB_64B: (line size=64B)

Cache size	2-ways				4-ways				8-ways				16-ways			
	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO
1 KB	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%
32 KB	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%
256 KB	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%
1 MB	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%
4 MB	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%

1KB_64B: (cache size=1MB)

Line size	2-ways				4-ways				8-ways				16-ways			
	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO
16 Bytes	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%
64 Bytes	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%
128 Bytes	25.0%	25.0%	25.0%	25.0%	25.0%	25.0%	25.0%	25.0%	25.0%	25.0%	25.0%	25.0%	25.0%	25.0%	25.0%	25.0%
512 Bytes	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%
1024 Bytes	3.12%	3.12%	3.12%	3.12%	3.12%	3.12%	3.12%	3.12%	3.12%	3.12%	3.12%	3.12%	3.12%	3.12%	3.12%	3.12%

notes:

only changes in the block size are reflected in different miss rates due to the sequential memory accesses being in 64 byte strides. once the block size is increased beyond 64 bytes, we end up with better cache locality since more data is being brought into the L1 cache.

4MB_4: (line size=64B)

Cache size	2-ways				4-ways				8-ways				16-ways			
	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO
1 KB	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%
32 KB	6.25%	6.25%	6.25%	6.23%	6.25%	6.25%	6.25%	6.23%	6.25%	6.25%	6.25%	6.22%	6.25%	6.25%	6.25%	6.22%
256 KB	6.25%	6.25%	6.25%	6.12%	6.25%	6.25%	6.25%	6.05%	6.25%	6.25%	6.25%	6.02%	6.25%	6.25%	6.25%	6.01%
1 MB	6.25%	6.25%	6.25%	5.73%	6.25%	6.25%	6.25%	5.47%	6.25%	6.25%	6.25%	5.34%	6.25%	6.25%	6.25%	5.27%
4 MB	2.08%	2.08%	2.08%	2.08%	2.08%	2.08%	2.08%	2.08%	2.08%	2.08%	2.08%	2.08%	2.08%	2.08%	2.08%	2.08%

4MB_4: (cache size=1MB)

Line size	2-ways				4-ways				8-ways				16-ways			
	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO
16 Bytes	25.0%	25.0%	25.0%	22.92%	25.0%	25.0%	25.0%	21.88%	25.0%	25.0%	25.0%	21.35%	25.0%	25.0%	25.0%	21.09%
64 Bytes	6.25%	6.25%	6.25%	5.73%	6.25%	6.25%	6.25%	5.47%	6.25%	6.25%	6.25%	5.34%	6.25%	6.25%	6.25%	5.27%
128 Bytes	3.12%	3.12%	3.12%	2.86%	3.12%	3.12%	3.12%	2.73%	3.12%	3.12%	3.12%	2.67%	3.12%	3.12%	3.12%	2.64%
512 Bytes	0.78%	0.78%	0.78%	0.72%	0.78%	0.78%	0.78%	0.68%	0.78%	0.78%	0.78%	0.67%	0.78%	0.78%	0.78%	0.66%
1024 Bytes	0.39%	0.39%	0.39%	0.36%	0.39%	0.39%	0.39%	0.34%	0.39%	0.39%	0.39%	0.33%	0.39%	0.39%	0.39%	0.33%

notes:

changes in cache size do affect the miss ration for this benchmark since larger cache will result in fewer *capacity* misses. Interestingly, LIFO eviction policy works better to reduce misses and is due to the nature of accessed being sequential meaning that the most recently placed block will be the least recently used one once accesses to it are completed. furthermore, using larger blocks increases locality and therefore reduces misses.

32MB_4B: (line size=64B)

Cache size	2-ways				4-ways				8-ways				16-ways			
	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO
1 KB	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%
32 KB	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%
256 KB	6.25%	6.25%	6.25%	6.23%	6.25%	6.25%	6.25%	6.23%	6.25%	6.25%	6.25%	6.22%	6.25%	6.25%	6.25%	6.22%
1 MB	6.25%	6.25%	6.25%	6.18%	6.25%	6.25%	6.25%	6.15%	6.25%	6.25%	6.25%	6.14%	6.25%	6.25%	6.25%	6.13%
4 MB	6.25%	6.25%	6.25%	5.99%	6.25%	6.25%	6.25%	5.86%	6.25%	6.25%	6.25%	5.79%	6.25%	6.25%	6.25%	5.76%

32MB_4B: (cache size=1MB)

Line size	2-ways				4-ways				8-ways				16-ways			
	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO
16 Bytes	25.0%	25.0%	25.0%	24.74%	25.0%	25.0%	25.0%	24.61%	25.0%	25.0%	25.0%	24.54%	25.0%	25.0%	25.0%	24.51%
64 Bytes	6.25%	6.25%	6.25%	6.18%	6.25%	6.25%	6.25%	6.15%	6.25%	6.25%	6.25%	6.14%	6.25%	6.25%	6.25%	6.13%
128 Bytes	3.12%	3.12%	3.12%	3.09%	3.12%	3.12%	3.12%	3.08%	3.12%	3.12%	3.12%	3.07%	3.12%	3.12%	3.12%	3.06%
512 Bytes	0.78%	0.78%	0.78%	0.77%	0.78%	0.78%	0.78%	0.77%	0.78%	0.78%	0.78%	0.77%	0.78%	0.78%	0.78%	0.77%
1024 Bytes	0.39%	0.39%	0.39%	0.39%	0.39%	0.39%	0.39%	0.38%	0.39%	0.39%	0.39%	0.38%	0.39%	0.39%	0.39%	0.38%

notes:

changes in cache size do not affect the miss ratio much since we would need an L1 cache of 32 megabytes to reduce the number of *capacity* misses. Using a larger block and a LIFO eviction policy help reduce misses due better cache locality and block replacement mirroring the needs of a circular memory pattern.

bw_mem: (line size=64B)

Cache size	2-ways				4-ways				8-ways				16-ways			
	LRU	**SC**	**FIFO**	**LIFO**	**LRU**	**SC**	**FIFO**	**LIFO**	**LRU**	**SC**	**FIFO**	**LIFO**	**LRU**	**SC**	**FIFO**	**LIFO**
1 KB	1.57%	1.57%	1.57%	1.57%	1.57%	1.57%	1.57%	1.57%	1.57%	1.57%	1.57%	1.57%	1.57%	1.57%	1.57%	1.57%
32 KB	1.57%	1.57%	1.57%	1.57%	1.57%	1.57%	1.57%	1.57%	1.57%	1.57%	1.57%	1.57%	1.57%	1.57%	1.57%	1.57%
256 KB	1.57%	1.57%	1.57%	1.57%	1.57%	1.57%	1.57%	1.57%	1.57%	1.57%	1.57%	1.56%	1.57%	1.57%	1.57%	1.56%
1 MB	1.56%	1.56%	1.56%	1.56%	1.56%	1.56%	1.56%	1.56%	1.56%	1.56%	1.56%	1.56%	1.56%	1.56%	1.56%	1.56%
4 MB	1.56%	1.56%	1.56%	1.56%	1.56%	1.56%	1.56%	1.56%	1.56%	1.56%	1.56%	1.56%	1.56%	1.56%	1.56%	1.56%

bw_mem.traces: (cache size=1MB)

Line size	2-ways				4-ways				8-ways				16-ways			
	LRU	**SC**	**FIFO**	**LIFO**	**LRU**	**SC**	**FIFO**	**LIFO**	**LRU**	**SC**	**FIFO**	**LIFO**	**LRU**	**SC**	**FIFO**	**LIFO**
16 Bytes	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%	6.25%
64 Bytes	1.56%	1.56%	1.56%	1.56%	1.56%	1.56%	1.56%	1.56%	1.56%	1.56%	1.56%	1.56%	1.56%	1.56%	1.56%	1.56%
128 Bytes	0.78%	0.78%	0.78%	0.78%	0.78%	0.78%	0.78%	0.78%	0.78%	0.78%	0.78%	0.78%	0.78%	0.78%	0.78%	0.78%
512 Bytes	0.2%	0.2%	0.2%	0.2%	0.2%	0.2%	0.2%	0.2%	0.2%	0.2%	0.2%	0.2%	0.2%	0.2%	0.2%	0.2%
1024 Bytes	0.1%	0.1%	0.1%	0.1%	0.1%	0.1%	0.1%	0.1%	0.1%	0.1%	0.1%	0.1%	0.1%	0.1%	0.1%	0.1%

notes:

the application has a largely unchanged number of cache misses with only block sizes being the main discriminator. It appears the bw_mem is confined to a small range of memory accesses and larger block size increases locality immensely.

ls: (line size=64B)

Cache size	2-ways				4-ways				8-ways				16-ways			
	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO
1 KB	17.3%	18.43%	18.43%	28.8%	16.25%	16.93%	17.38%	37.46%	15.97%	16.52%	17.25%	50.3%	16.18%	16.71%	17.18%	62.8%
32 KB	2.52%	2.66%	2.66%	4.33%	2.24%	2.34%	2.43%	4.4%	2.19%	2.28%	2.41%	5.41%	2.17%	2.22%	2.39%	7.09%
256 KB	1.65%	1.69%	1.69%	1.75%	1.61%	1.63%	1.63%	1.74%	1.6%	1.61%	1.62%	1.86%	1.59%	1.61%	1.61%	1.85%
1 MB	1.59%	1.59%	1.59%	1.59%	1.59%	1.59%	1.59%	1.59%	1.59%	1.59%	1.59%	1.59%	1.59%	1.59%	1.59%	1.59%
4 MB	1.59%	1.59%	1.59%	1.59%	1.59%	1.59%	1.59%	1.59%	1.59%	1.59%	1.59%	1.59%	1.59%	1.59%	1.59%	1.59%

ls.traces: (cache size=1MB)

Line size	2-ways				4-ways				8-ways				16-ways			
	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO
16 Bytes	5.5%	5.5%	5.5%	5.5%	5.5%	5.5%	5.5%	5.5%	5.5%	5.5%	5.5%	5.5%	5.5%	5.5%	5.5%	5.5%
64 Bytes	1.59%	1.59%	1.59%	1.59%	1.59%	1.59%	1.59%	1.59%	1.59%	1.59%	1.59%	1.59%	1.59%	1.59%	1.59%	1.59%
128 Bytes	0.87%	0.87%	0.87%	0.87%	0.87%	0.87%	0.87%	0.87%	0.87%	0.87%	0.87%	0.87%	0.87%	0.87%	0.87%	0.87%
512 Bytes	0.28%	0.28%	0.28%	0.28%	0.27%	0.27%	0.27%	0.27%	0.27%	0.27%	0.27%	0.27%	0.27%	0.27%	0.27%	0.27%
1024 Bytes	0.16%	0.16%	0.16%	0.17%	0.16%	0.16%	0.16%	0.16%	0.16%	0.16%	0.16%	0.16%	0.16%	0.16%	0.16%	0.16%

notes:

changes in cache size help reduce capacity misses while LRU appears to be slightly better of the eviction policies. once again, using larger blocks increases locality and has a decent impact on locality.

gcc: (line size=64B)

Cache size	2-ways				4-ways				8-ways				16-ways			
	LRU	**SC**	**FIFO**	**LIFO**	**LRU**	**SC**	**FIFO**	**LIFO**	**LRU**	**SC**	**FIFO**	**LIFO**	**LRU**	**SC**	**FIFO**	**LIFO**
1 KB	13.15%	13.9%	13.9%	25.43%	11.94%	12.6%	13.06%	35.67%	11.46%	12.02%	12.74%	47.78%	11.34%	11.87%	12.83%	60.61%
32 KB	2.06%	2.16%	2.16%	3.15%	1.94%	2.02%	2.08%	3.8%	1.91%	1.97%	2.07%	5.91%	1.89%	1.93%	2.04%	7.93%
256 KB	1.34%	1.36%	1.36%	1.4%	1.31%	1.33%	1.34%	1.4%	1.3%	1.32%	1.32%	1.47%	1.29%	1.31%	1.32%	1.54%
1 MB	1.29%	1.29%	1.29%	1.29%	1.29%	1.29%	1.29%	1.29%	1.29%	1.29%	1.29%	1.29%	1.29%	1.29%	1.29%	1.29%
4 MB	1.29%	1.29%	1.29%	1.29%	1.29%	1.29%	1.29%	1.29%	1.29%	1.29%	1.29%	1.29%	1.29%	1.29%	1.29%	1.29%

gcc.traces: (cache size=1MB)

Line size	2-ways				4-ways				8-ways				16-ways			
	LRU	**SC**	**FIFO**	**LIFO**	**LRU**	**SC**	**FIFO**	**LIFO**	**LRU**	**SC**	**FIFO**	**LIFO**	**LRU**	**SC**	**FIFO**	**LIFO**
16 Bytes	4.6%	4.6%	4.6%	4.6%	4.6%	4.6%	4.6%	4.6%	4.6%	4.6%	4.6%	4.6%	4.6%	4.6%	4.6%	4.6%
64 Bytes	1.29%	1.29%	1.29%	1.29%	1.29%	1.29%	1.29%	1.29%	1.29%	1.29%	1.29%	1.29%	1.29%	1.29%	1.29%	1.29%
128 Bytes	0.69%	0.69%	0.69%	0.69%	0.69%	0.69%	0.69%	0.69%	0.69%	0.69%	0.69%	0.69%	0.69%	0.69%	0.69%	0.69%
512 Bytes	0.22%	0.22%	0.22%	0.23%	0.21%	0.22%	0.22%	0.22%	0.21%	0.21%	0.21%	0.21%	0.21%	0.21%	0.21%	0.21%
1024 Bytes	0.13%	0.13%	0.13%	0.16%	0.13%	0.13%	0.13%	0.13%	0.13%	0.13%	0.13%	0.13%	0.13%	0.13%	0.13%	0.13%

notes:

changes in cache size do affect the miss ration for this benchmark but after a 32KB cache, the impact has diminishing results for *gcc*. Using a larger block size does reduce misses but the impact is not nearly as pronounced as it is in *bw_mem* which is likely due to gcc relying only on core set of blocks that store the needed information for compilation on most portions of code.

native_dgemm: (line size=64B)

Cache size	2-ways				4-ways				8-ways				16-ways			
	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO
1 KB	56.44%	58.75%	58.75%	61.55%	56.44%	56.45%	58.0%	66.69%	56.44%	56.44%	56.44%	77.21%	56.44%	56.44%	56.44%	99.31%
32 KB	50.27%	50.27%	50.27%	49.23%	50.27%	50.27%	50.27%	48.61%	50.27%	50.27%	50.27%	48.5%	50.27%	50.27%	50.27%	49.15%
256 KB	50.25%	50.26%	50.26%	38.36%	50.25%	50.25%	50.25%	32.24%	50.25%	50.25%	50.25%	29.2%	50.25%	50.25%	50.25%	27.72%
1 MB	0.87%	0.88%	0.88%	0.93%	0.87%	0.87%	0.87%	0.87%	0.87%	0.87%	0.87%	0.87%	0.87%	0.87%	0.87%	0.87%
4 MB	0.87%	0.87%	0.87%	0.87%	0.87%	0.87%	0.87%	0.87%	0.87%	0.87%	0.87%	0.87%	0.87%	0.87%	0.87%	0.87%

native_dgemm.traces: (cache size=1MB)

Line size	2-ways				4-ways				8-ways				16-ways			
	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO
16 Bytes	3.48%	3.51%	3.51%	3.54%	3.48%	3.48%	3.48%	3.48%	3.48%	3.48%	3.48%	3.48%	3.48%	3.48%	3.48%	3.48%
64 Bytes	0.87%	0.88%	0.88%	0.93%	0.87%	0.87%	0.87%	0.87%	0.87%	0.87%	0.87%	0.87%	0.87%	0.87%	0.87%	0.87%
128 Bytes	0.43%	0.44%	0.44%	0.49%	0.43%	0.43%	0.43%	0.43%	0.43%	0.43%	0.43%	0.43%	0.43%	0.43%	0.43%	0.43%
512 Bytes	0.11%	0.11%	0.11%	0.16%	0.11%	0.11%	0.11%	0.11%	0.11%	0.11%	0.11%	0.11%	0.11%	0.11%	0.11%	0.11%
1024 Bytes	0.05%	0.06%	0.06%	0.11%	0.05%	0.05%	0.05%	0.05%	0.05%	0.05%	0.05%	0.05%	0.05%	0.05%	0.05%	0.05%

notes:

changes in cache size have a dramatic impact on the miss ratio once the cache size is increased to 1MB. This is likely due to the matrix multiplications not being optimized such that the operands are subdivided into small matrices that can be computed in smaller caches. *native_dgemem* most likely stores the matrix elements in long sequential arrays. Lastly, larger block increase locality for the matrix elements and therefore help to squash misses.

native_dgemm_full: (line size=64B)

Cache size	2-ways				4-ways				8-ways				16-ways			
	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO
1 KB	55.48%	57.74%	57.74%	60.82%	55.47%	55.48%	57.0%	66.15%	55.47%	55.47%	55.47%	76.81%	55.47%	55.47%	55.47%	98.12%
32 KB	49.4%	49.4%	49.4%	49.56%	49.4%	49.4%	49.4%	49.73%	49.4%	49.4%	49.4%	50.06%	49.39%	49.4%	49.4%	51.1%
256 KB	49.38%	49.38%	49.38%	49.4%	49.38%	49.38%	49.38%	49.4%	49.37%	49.38%	49.38%	49.43%	49.37%	49.37%	49.37%	49.49%
1 MB	0.13%	0.15%	0.15%	0.64%	0.13%	0.15%	0.17%	24.95%	0.13%	0.15%	0.17%	25.6%	0.13%	0.15%	0.17%	21.0%
4 MB	0.08%	0.08%	0.08%	0.08%	0.08%	0.08%	0.08%	0.08%	0.08%	0.08%	0.08%	0.08%	0.08%	0.08%	0.08%	0.08%

native_dgemm_full: (cache size=1MB)

Line size	2-ways				4-ways				8-ways				16-ways			
	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO
16 Bytes	0.5%	0.6%	0.6%	1.11%	0.51%	0.6%	0.69%	23.76%	0.51%	0.6%	0.68%	23.95%	0.51%	0.6%	0.67%	19.78%
64 Bytes	0.13%	0.15%	0.15%	0.64%	0.13%	0.15%	0.17%	24.95%	0.13%	0.15%	0.17%	25.6%	0.13%	0.15%	0.17%	21%
128 Bytes	0.06%	0.08%	0.08%	0.57%	0.06%	0.08%	0.09%	25.86%	0.07%	0.08%	0.09%	26.85%	0.07%	0.08%	0.09%	21.99%
512 Bytes	0.02%	0.02%	0.02%	0.55%	0.02%	0.02%	0.02%	29.23%	0.02%	0.02%	0.02%	30.8%	0.02%	0.02%	0.02%	25.35%
1024 Bytes	0.01%	0.01%	0.01%	0.58%	0.01%	0.01%	0.01%	30.68%	0.01%	0.01%	0.01%	32.66%	0.01%	0.01%	0.01%	27.68%

notes:

changes in cache size have a dramatic impact on the miss ratio once the cache size is increased to 1MB. This is likely due to the matrix multiplications not being optimized such that the operands are subdivided into small matrices that can be computed in smaller caches. *native_dgemem* most likely stores the matrix elements in long sequential arrays. Eviction policies do have a noticeable impact here with LIFO having disastrous results when the cache size is set to 1KB. Lastly, larger block increase locality for the matrix elements and therefore help to squash misses.

openblas_dgemm: (line size=64B)

Cache size	2-ways				4-ways				8-ways				16-ways			
	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO
1 KB	62.75%	62.84%	62.84%	67.19%	61.72%	61.8%	61.85%	79.71%	61.76%	61.85%	61.95%	81.49%	61.78%	61.8%	61.93%	83.08%
32 KB	18.43%	19.45%	19.45%	17.94%	13.44%	21.47%	22.2%	19.81%	16.33%	23.11%	23.16%	16.2%	22.13%	25.05%	25.01%	14.38%
256 KB	8.34%	9.12%	9.12%	7.75%	8.3%	8.39%	9.08%	7.43%	8.3%	8.37%	9.06%	7.71%	8.3%	8.35%	9.1%	7.98%
1 MB	4.03%	4.08%	4.08%	3.38%	5.17%	4.95%	5.04%	3.41%	6.57%	6.19%	6.34%	3.32%	8.06%	7.97%	8.01%	3.13%
4 MB	2.27%	2.27%	2.27%	2.27%	2.27%	2.27%	2.27%	2.27%	2.27%	2.27%	2.27%	2.27%	2.27%	2.27%	2.27%	2.27%

openblas_dgemm: (cache size=1MB)

Line size	2-ways				4-ways				8-ways				16-ways			
	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO
16 Bytes	7.39%	7.54%	7.54%	6.61%	8.93%	8.76%	9.0%	6.35%	10.54%	10.15%	10.43%	6.17%	14.06%	13.52%	13.78%	6.09%
64 Bytes	4.03%	4.08%	4.08%	3.38%	5.17%	4.95%	5.04%	3.41%	6.57%	6.19%	6.34%	3.32%	8.06%	7.91%	8.01%	3.13%
128 Bytes	2.03%	2.05%	2.05%	1.71%	2.61%	2.49%	2.54%	1.73%	3.31%	3.11%	3.19%	1.69%	4.04%	3.96%	4.01%	1.6%
512 Bytes	0.54%	0.53%	0.53%	0.46%	0.68%	0.65%	0.66%	0.48%	0.86%	0.8%	0.82%	0.49%	1.02%	1.0%	1.02%	0.49%
1024 Bytes	0.28%	0.28%	0.28%	0.24%	0.36%	0.36%	034%	0.27%	0.45%	0.45%	0.42%	0.28%	0.52%	0.51%	0.52%	0.3%

notes:

changes in cache size have a dramatic impact on the miss ratio once the cache size is increased to 1KB and continue to roughly reduce the miss rate in half as the cache size is doubled. This is again likely due to the matrix multiplications not being optimized such that the operands are subdivided into smaller matrices that can be computed in smaller caches. Interestingly, LIFO does not do nearly as much harm here as it did in *native_dgmm.* This is likely due to a somewhat better implementation for the linear algebra, we assume. Lastly, larger block sizes increase locality for the matrix elements and therefore help to squash misses.

openblas_dgemm_full: (line size=64B)

Cache size	2-ways				4-ways				8-ways				16-ways			
	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO
1 KB	51.49%	51.66%	51.66%	56.7%	50.65%	50.75%	50.88%	68.36%	50.39%	50.48%	50.64%	72.74%	50.41%	50.43%	50.63%	75.65%
32 KB	15.52%	16.39%	16.39%	26.03%	11.58%	18.05%	18.58%	30.88%	13.83%	19.21%	19.33%	33.95%	18.42%	20.76%	20.83%	38.64%
256 KB	7.54%	8.17%	8.17%	9.36%	7.5%	7.53%	8.13%	11.09%	7.5%	7.51%	8.13%	14.38%	7.5%	7.51%	8.15%	25.99%
1 MB	3.02%	3.15%	3.15%	7.92%	4.26%	4.16%	4.32%	8.35%	5.66%	5.41%	5.71%	9.3%	7.2%	6.83%	7.11%	11.02%
4 MB	0.88%	0.93%	0.93%	1.4%	0.94%	0.94%	0.93%	2.17%	0.96%	0.95%	0.97%	7.7%	0.98%	0.96%	0.98%	8.12%

openblas_dgemm_full: (cache size=1MB)

Line size	2-ways				4-ways				8-ways				16-ways			
	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO	LRU	SC	FIFO	LIFO
16 Bytes	6.98%	7.29%	7.29%	17.52%	8.69%	8.7%	9.0%	18.15%	10.48%	10.46%	10.79%	19.67%	13.71%	13.64%	13.98%	22.24%
64 Bytes	3.02%	3.15%	3.15%	7.92%	4.26%	4.16%	4.32%	8.35%	5.66%	5.41%	5.71%	9.3%	7.2%	6.83%	7.11%	11.08%
128 Bytes	1.53%	1.59%	1.59%	4.02%	2.15%	2.09%	2.18%	4.28%	2.86%	2.72%	2.87%	4.85%	3.61%	3.42%	3.56%	3.91%
512 Bytes	0.4%	0.42%	0.42%	1.17%	0.57%	0.55%	0.57%	1.35%	0.76%	0.71%	0.75%	1.7%	0.93%	0.87%	0.92%	2.37%
1024 Bytes	0.22%	0.22%	0.22%	0.72%	0.3%	0.29%	0.3%	0.9%	0.41%	0.37%	0.4%	1.27%	0.48%	0.45%	0.48%	1.96%

notes:

changes in cache size have a dramatic impact on the miss ratio once the cache size is increased to 1KB and continue to roughly reduce the miss rate in half as the cache size is doubled. This is again likely due to the matrix multiplications not being optimized such that the operands are subdivided into smaller matrices that can be computed in smaller caches. Interestingly, LIFO does not do nearly as much harm here as it did in *native_dgmm.* This is likely due to a somewhat better implementation for the linear algebra, we assume. Lastly, larger block sizes increase locality for the matrix elements and therefore help to squash misses.

• Individual contribution summary

Bilal authored the python3 code. He also wrote the introduction to the report including the description of how to use the simulator code and an overview of the implementation's logic. He also contributed in editing the final draft of the report.

Zhongxiu provided the initial code for parsing the sequence file and authored the first 800+ line bash script to run tests on the simulator (with the block size fixed). He also wrote the contributed to writing the initial draft of the report.

Trung Dinh wrote the second 800+ line bash script to test the effects of differing eviction policies, associtivities, and block sizes when cache size is fixed. He also contributed to the initial draft of the report.