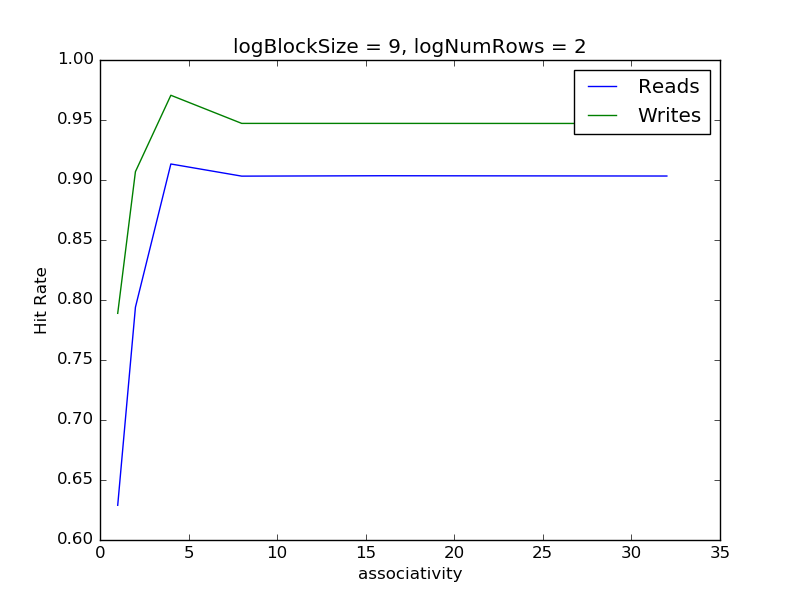
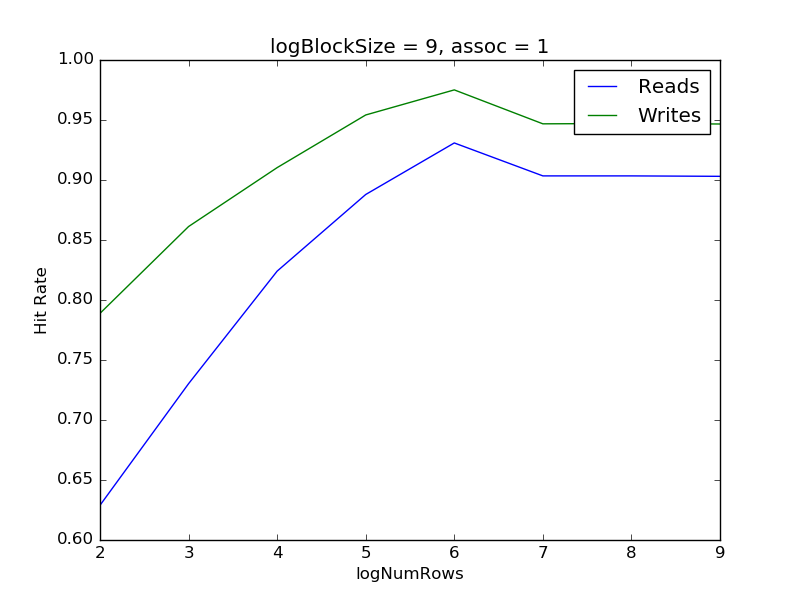
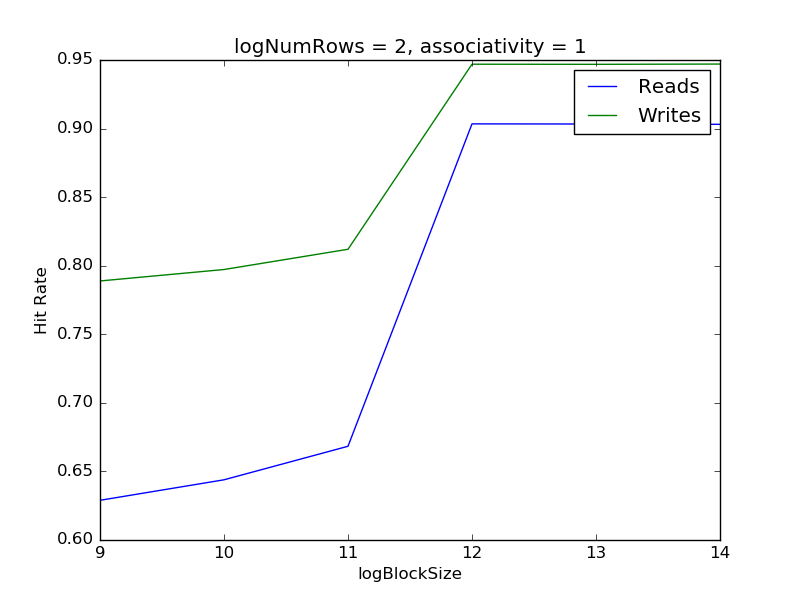
Note: The virtually indexed physically tagged was not analyzed when logBlockSize + logNumRows > logPageSize. This essentially makes this cache the same as a physically indexed physically tagged cache, therefore results are not reported.

1.





For numRows variation, we saw that there was a peak and then the hit rate flattened.

For associativity variation, we saw the same behavior for associativity also.

For block size though, the plot doesn’t go down. There is no peak, only the graph flattens. We also see that between 11 and 12, there was a big leap in the rates. This suggests that most programs had a working size on that order.

The writes are showing a higher hit rate then reads, this is probably because a recently read entry is most likely to be written to so that entry is probably already in the cache when it gets written.

Just looking at the numbers from the different types of caches, there doesn’t seem to be a lot of difference. This is probably mainly because we are running single processes and so the flushing requirement of virtually tagged cache is not apparent here. Also the caches differ mostly in terms of their access time, and so we can conclude that the hit rates are not much affected among the different types of caches.

2. Analyzing the hit rates (Lab1.xlsx), the SPECFP have a higher hit rate as the number of rows were increasing however from numRows of 128, the hit rate of SPECINT was higher with more cache lines. Similar behavior of SPECINT performing better when cache sizes are larger is observed with other parameters. This suggests that SPECINT has a higher working set than FP? And once it gets a larger cache to work with, it performs a lot better. Increasing associativity or block size is nicer for SPECINT whereas, increasing the same makes SPECFP perform a bit worse.

3. To have physically indexed, virtually tagged cache, we will first need to do a translation to get the physical address in order to get the physical index. Only then will we able to index into the cache using the physical index and compare the tags. Timing wise, the access to the cache will have to wait until the address has been translated which can potentially make the clock period longer.

4. The virtually tagged cache will not work correctly with multiple processes. This is because every process has its own translation table. So the cache will have to be flushed and invalidated every time a context switch happens, because since the mappings have changed, the data in the cache may no longer be correct for a given virtual address as that virtual address may point into some other physical process for the new process.

We can have a process id metadata field in the cache and every memory access can have this field. So we will only need to flush a cache line if the process id of that line doesn’t match to that of the current process. A simpler way of using this kind of asid tagging can be utilized by not flushing the cache lines belonging to the operating system as these pages are usually mapped (for e.g. in linux the upper quarter of the memory is reserved for the kernel is always mapped between context switches).

For other kind of caches, since they are indexed and tagged by physical address, switching processes hence changing the mapping doesn’t affect us at all.

5. I modified the caches.cpp in the following way:

if (virtualAddr & 0x3) numMisalignedLoads (or Stores)++;

The percentage of unaligned access vary a lot between benchmark to benchmark. On average, about 10% of the accesses were unaligned, with a standard deviation of 15%. The bzip2 test had a 50% misaligned access rate (maybe because decompression/compression a lot of byte level/half word level accesses?). But for most programs, the rate was low and hence it does make sense. It all comes down if the program is operating with data of the machine word size then it does a lot less unaligned access.