Report for part F

For implementing non static partitioning of shared cache the approach proposed by Quershi et al.[1] has been followed. Each core is assigned a utility monitoring (UMON) block that tracks the utility information of the application executing on it. The UMON circuit is separated from the shared cache, which allows the UMON circuit to obtain utility information about an application for all the ways in the cache, independent of the contention from the application executing on the other core. The partitioning algorithm uses the information collected by the UMON to decide the number of ways to allocate to each core.

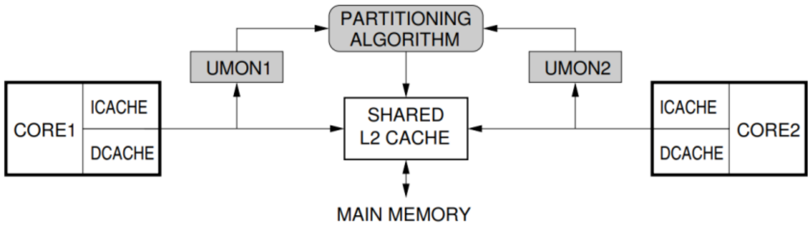
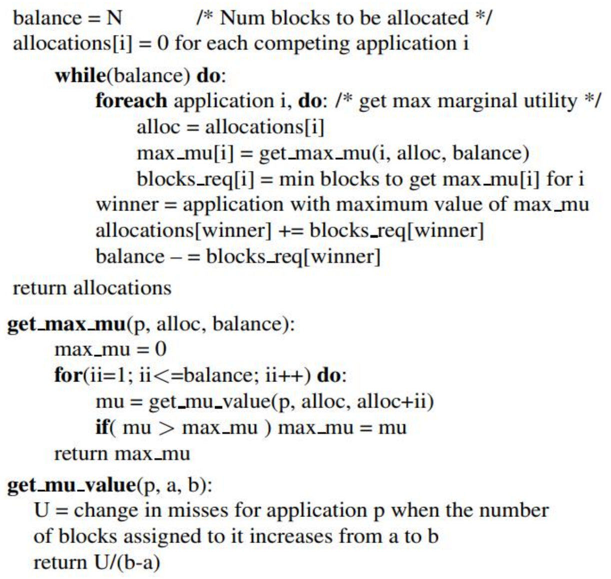


Fig: Memory system with added structure [1]

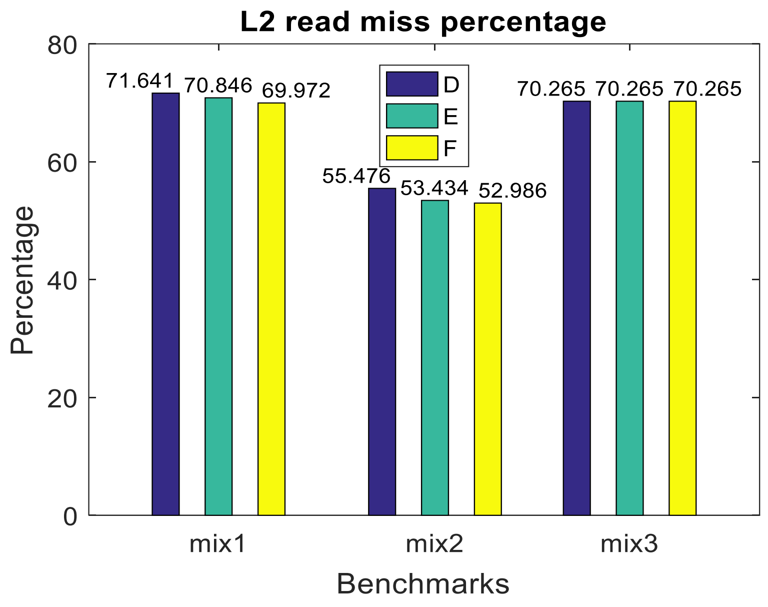
Each UMON block has a 16 way set associative cache. To reduce hardware overhead only every 33rd set of the L2cache has been included in it which is enough to estimate the resources used by the running applications. This method is known as dynamic set sampling. Each cacheline of UMON consists of a counter tag which holds the value of number of hits at that particular position. LRU replacement policy is used in UMON block. Since every UMON block is 16 way set associative and it follows LRU replacement policy, it can provide the number of misses that would incur if any number of ways less than 16 was allocated for the running application. This change in number of misses with change in allocated ways is the measure of utility of the running application. For partitioning of cache following lookahead utility based algorithm is followed:

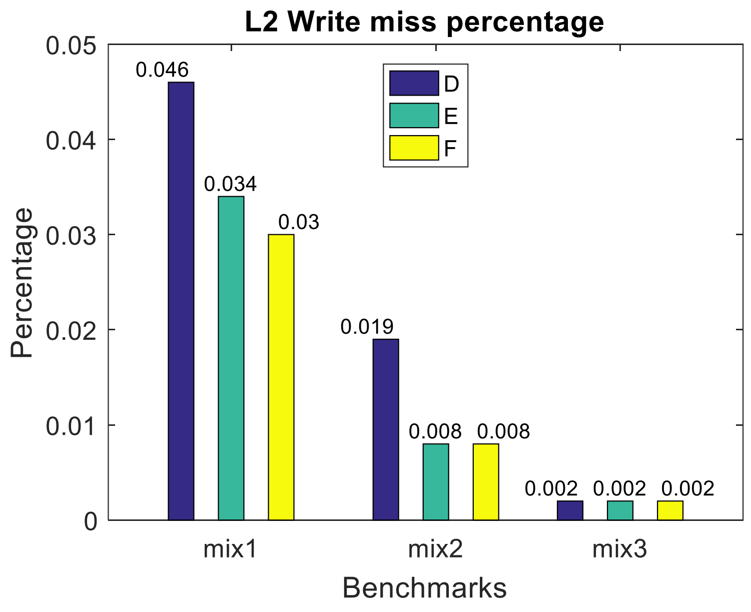


```
balance = N            /* Num blocks to be allocated */
allocations[i] = 0 for each competing application i
    while(balance) do:
        foreach application i, do: /* get max marginal utility */
            alloc = allocations[i]
            max_mu[i] = get_max_mu(i, alloc, balance)
            blocks_req[i] = min blocks to get max_mu[i] for i
        winner = application with maximum value of max_mu
        allocations[winner] += blocks_req[winner]
        balance – = blocks_req[winner]
return allocations

get_max_mu(p, alloc, balance):
    max_mu = 0
    for(ii=1; ii<=balance; ii++) do:
        mu = get_mu_value(p, alloc, alloc+ii)
        if( mu > max_mu ) max_mu = mu
    return max_mu

get_mu_value(p, a, b):
  U = change in misses for application p when the number
  of blocks assigned to it increases from a to b
  return U/(b-a)
```

Fig: Utility based cache partiioning algorithm[1]

Number of ways allocated to a particular core in shared L2 cache is updated after every 5 million instructions. Since the algorithm tries to maximize the total utility of the all running application reducing overall number of misses it should decrease the read/write miss percentage in shared L2 cache. The results obtained by using utility based cache partitioning on the given benchmarks are shown below. It can be observed that for mix1 benchmarking instruction set the read/write miss ratio has decreased more than 1%.





REFERENCE:

1. Qureshi, Moinuddin K., and Yale N. Patt. "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches." 2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06). IEEE, 2006.