Gem5 Multicore Assignment

N.Sukumar 1242065, Vigneshwar Madras Elango 1282409

***Abstract*—The goal of the assignment is to port a single threaded EEG application to a multi threaded application and map it on a heterogeneous multi core platform. This is implemented with the help of OpenMP to create a multi threaded version of the application and GEM5 to simulate various processor configurations. Also, to evaluate the cache configurations and its impact on the performance.**

***Keywords*—Single core, Multi-core, OpenMP, GEM5, cache, performance.**

I. Introduction

The first assignment deals with the caches and the impact of the caches on the overall performance of ARM A9. The second assignment deals with upgrading the ARM A9 to A15 model and evaluate the impact of caches on the overall performance performance of A15 model. The third assignment is to convert this single threaded application to a multi threaded version to be executed on multiple cores with the help of OpenMP.

II. Motivation

The motivation behind this assignment is as follows which also forms the main part of this assignment.

1) Cache Size & associativity : How does cache size and associativity level influence the performance of the processor? Or is there any point that the enlargement of cache size above a certain point makes little improvement in the performance. Also, cache is one of the important factors in deciding the power consumption. How to obtain a trade off between the cache configuration and power consumption? Hence, we have dealt with different cache configurations on single core as well as multicore to decide an optimal cache size and associativity levels. We will consider the impact of L1 cache and L2 cache configuration for both the single core and multicore platform.
2) Parallelism : On employing OpenMP API, how can we a convert single threaded application to a multi threaded application and parallelize the given application to run on multicores and improve the performance.
3) GEM5: To learn how to use GEM5 and simlulate various processor configurations and evaluate the results using the generated *stats.txt* [4].

III. Cache Assignments

In this assignment, a single thread implementation of the EEG application is run on the supplied A9 core. First, the ARM binary is built using *make* and then it is simulated on GEM5 by executing the command *gem5.opt /gem5/config/example/armA9.py -n 1 -c eeg.arm*. The goal of this assignment is to modify the caches and evaluate the overall performance on the ARM A9 processor configuration.

A. Assignment 1.1 L1 cache size

In this assignment, the L2 cache is disabled and the L1 instruction and data cache sizes are varied ranging from 8kb to 64kb. The caches are directly mapped which means the associativity level is set to 1.

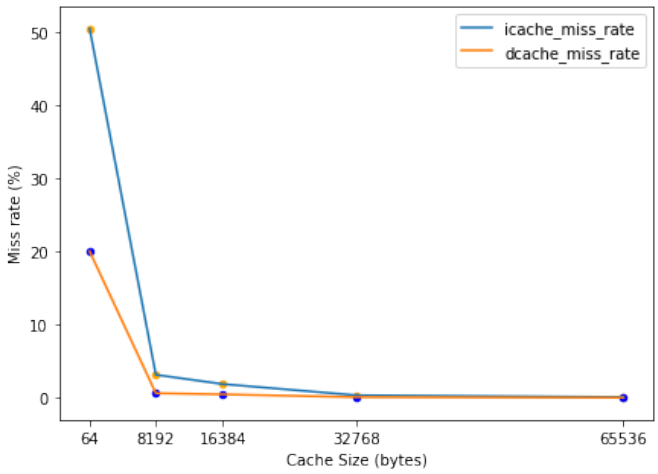

Fig. 1. Impact of L1 cache size on miss rate

From that stats generated using the gem5 simulation, we evaluate the impact of cache size on the miss rate and hit ratio. *Miss rate* is the number of cache misses divided by the total number of accesses. From the figure 1, we observe that as the capacity of the cache increases, the miss rate of both the instruction cache and data cache decreases. As the misses reduce, the hit ratio should increase on increasing the cache size. *Hit ratio* is the number to hits to the total number of hits and misses. The results met our expectation which is shown in figure 2. We also noticed that the hit ratio of data cache is higher than that of the instruction cache. The misses that are reduced can be classified as *capacity misses* since they arise due to the insufficient capacity of caches. On the other hand, larger cache size can have longer hit latency which is not favourable. This is dealt in the final part of the assignment to obtain the optimal cache configuration. Similarly, we evaluate the impact of the L1 cache size on the miss latency. This is shown in figure 3. In the figure 4, we evaluate the impact of cache size on the simulation time of the application. As the cache size increases, the simulation time reduces. This is due to the fact that, as the size increases, the capacity misses reduce and that the availability of data is closer to the processor which exploits the locality property for fetching both the instructions and data. Hence the performance improves on increasing the cache size.

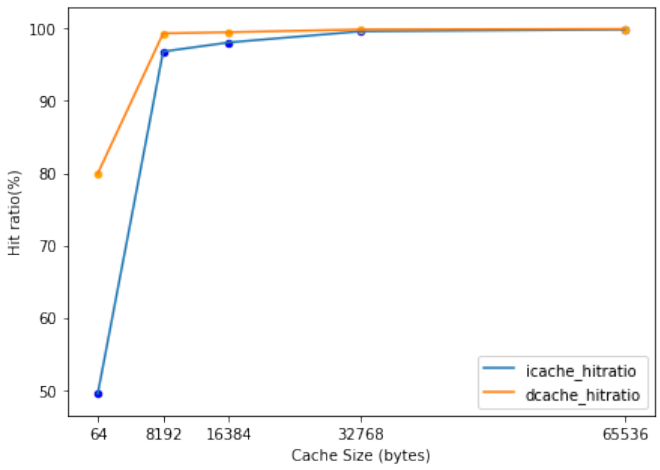

Fig. 2. Impact of L1 cache size on hit ratio

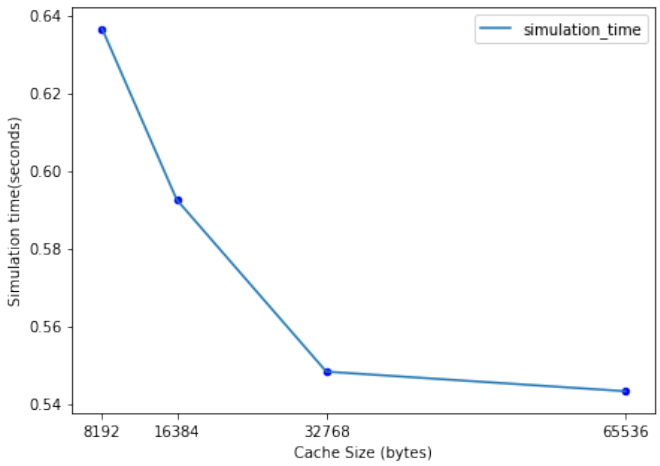

Fig. 4. Impact of L1 cache size on simulation time

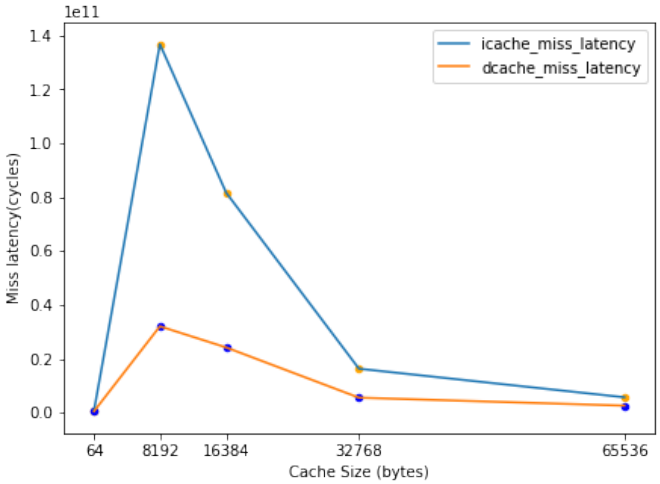

Fig. 3. Impact of L1 cache size on miss latency

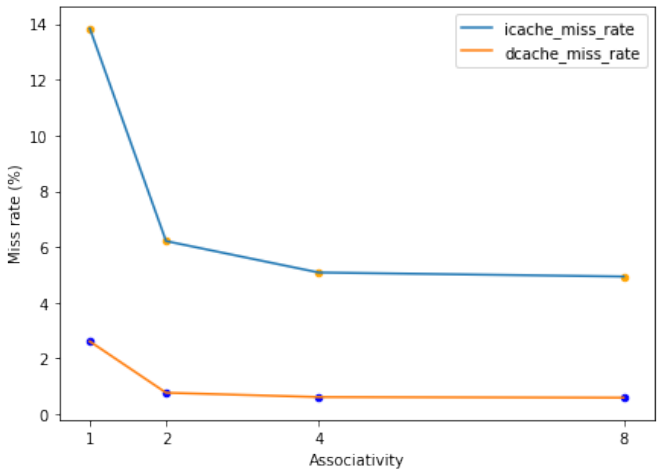

Fig. 5. Impact of associativity on miss rate

B. Assignment 1.2 Associativity level

In this assignment, L2 cache is disabled and the L1 instruction and data cache size are set to 2 kB. Then, the associativity levels for both L1 caches are varied from 1 to 8(fully associative) and its impact on the overall performance is measured. From the figure 5, we notice that as the L1 cache asssociaivity increases, the miss rate decreases. This is because higher associativity can reduce the conflict misses. *Conflict Misses* occur in set associative or direct mapped caches because a block can be discarded and later retrieved if too many blocks map to its set. Also called collision misses or interference misses. This potentially will lead to the increase of hit ratio which can be shown in figure 6. Hence, both the instruction cache and data cache hit ratio increases as the associativity is increased. Also, we can find that there is not much of improvement from increasing the associativity level from 2 to 8. This holds for both the instruction hit ratio data cache hit ratio. We also compared the influence of associativity levels on miss latency and simulation time of the program which are shown in figure 7 and 8.

C. Assignment 1.3 L2 cache size

In this assignment, the L1 instruction cache and data cache size is to be 1kB and they are directly mapped meaning the associativity is set to 1. Then, we change the direct mapped L2 cache size from 2 kB to 16 kB and measure the impact on its overall performance. Apart from evaluating the impact of cache size on miss rate, hit ratio, miss latency and simulatiom time, one can compare other important metrics such as AMAT(Average Memory Access time) and *Processor Performance(CPU) time*.
$AMAT = Hittime + (Missrate * Misspenalty)$. Inorder to derive the miss penalty for an out of order processor [1], one need to calculate the $\frac{MemoryStallcycles}{Instruction} = \frac{Misses}{Instruction}(Totalmisslatency - OverlappedMisslatency)$. The CPU time can be calculated as

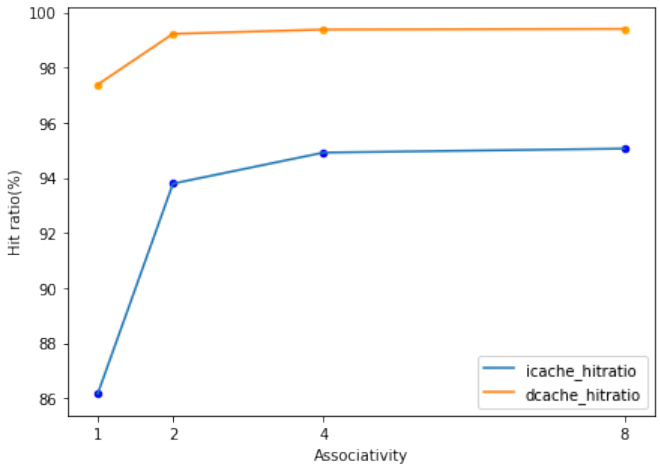

Fig. 6. Impact of associativity on hit ratio

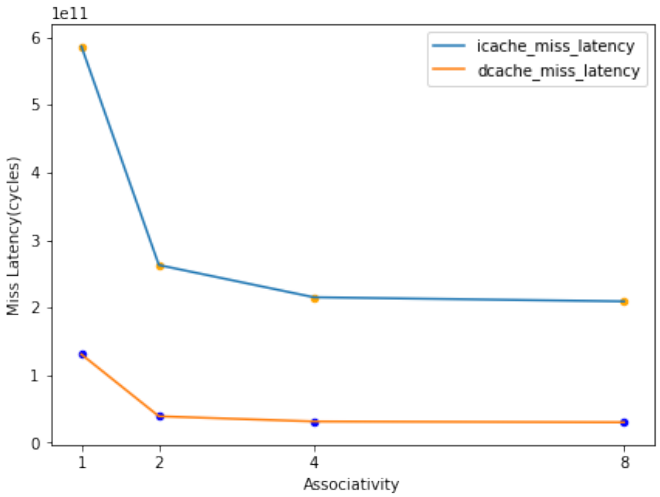

Fig. 7. Impact of associativity on miss latency

$$CPUtime = IC*(CPI_{exceution} + \frac{Memoryaccess}{Instruction} * Missrate * Misspenalty) * ClockCycleTime.$$

IV. Upgrade A9 to A15

A. Assignment 2.1 ARM 15

In this assignment, we modified the ARM A9 processor to ARM 15 by making the following changes.

1) An Out-of-Order CPU capable of fetching, decoding, issuing, and dispatching 3 instructions per cycle. 2 integer ALUs
2) 1 integer multiplier
3) 1 integer divider
4) 1 memory read/write port

This is done by changing the configuration in armA9.py to adopt to ARM 15.

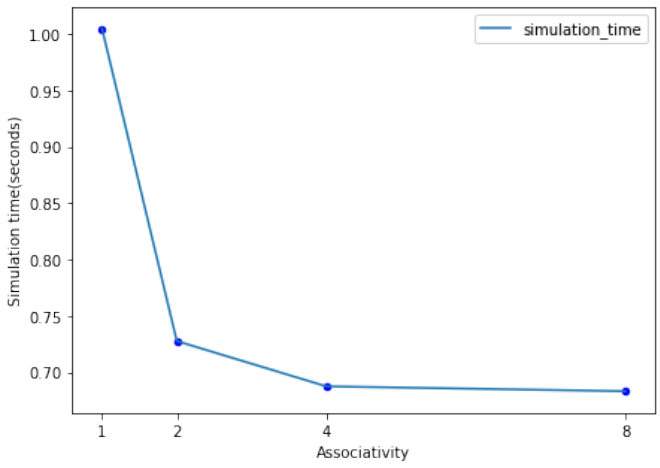

Fig. 8. Impact of associativity on simulation time

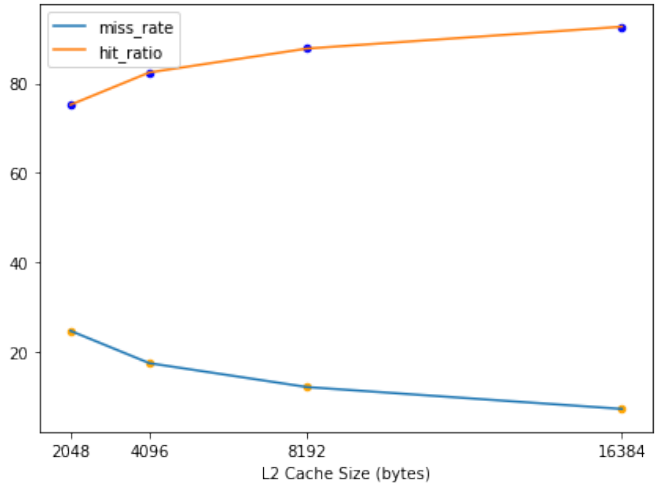

Fig. 9. Impact of L2 cache on miss rate and hit ratio

B. Assignment 2.2 L1 cache size

In this assignment, the L2 cache is disabled and the L1 instruction cache and data cache sizes are changed to evaluate their impact on the overall performance for ARM A15.The caches are directly mapped meaning the associative is set to 1. On varying the L1 instruction cache and data cache, we evaluated the impact of cache size on miss rate, hit ratio and miss latency similar to the a9 processor. These are shown in the figure 12 to. 14 We also compared the performance of the A9 and A15 processor. From the figure 15, we can infer that the performance of A15 is much better than the A9 processor. The simulation time is reduced by 20%. The number of cycles per instruction also reduces when the performance of the processor improves. This is evident from the graph 16.

V. Multicore implementation

In this section we extend our configuration to a multicore system. We have used OpenMP to create a simple multi

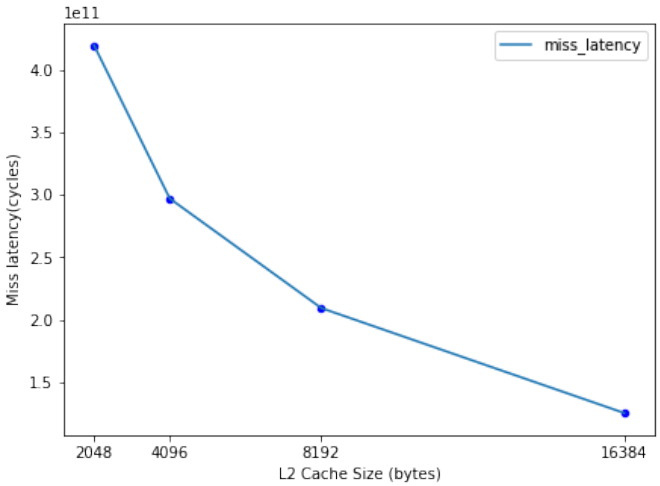

Fig. 10. Impact of L2 cache on miss latency

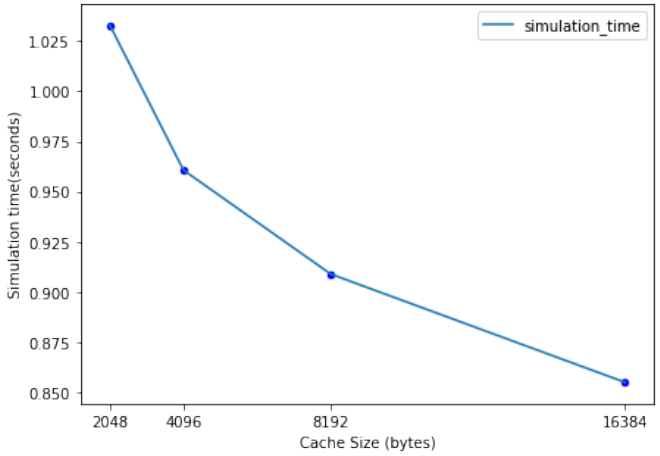

Fig. 11. Impact of L2 cache on simulation time

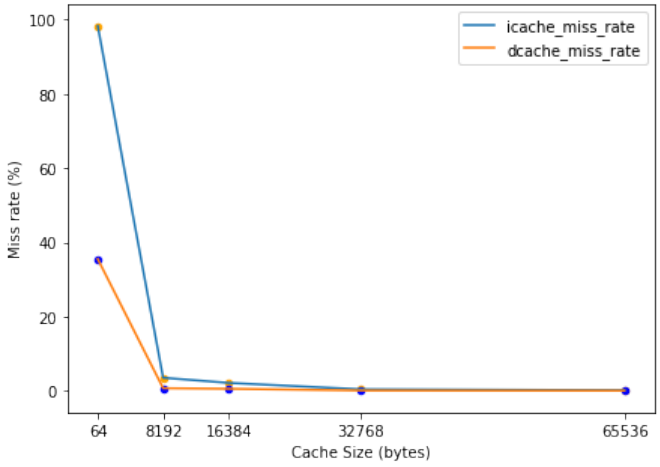

Fig. 12. Impact of L1 cache size on miss rate on A15

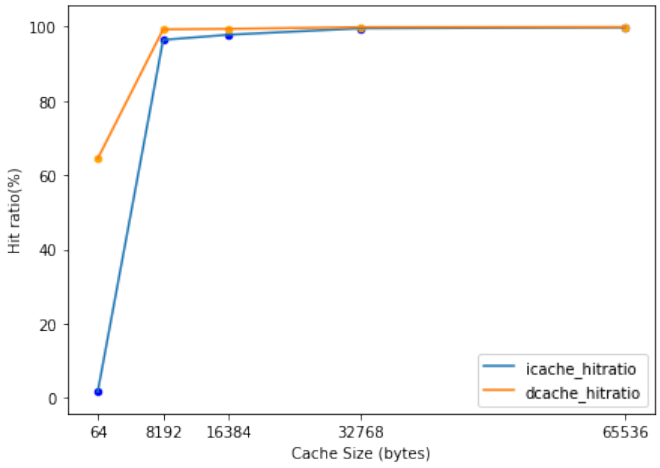

Fig. 13. Impact of L1 cache size on hit ratio on A15

threaded version of the given eeg application. OpenMP is an application programming interface that supports multi-platform shared memory multiprocessing programming on different platforms [3]. It is an implementation of multithreading, a method of parallelizing where by a master thread forks a specified number of slave threads and the system divides a task among them. The threads then run concurrently, with the run time environment allocating threads to different processors. The core elements of OpenMP as the constructs for thread creation, workload distribution, data environment management, thread synchronization, user-level run time routines and environment variables.

In the following sections, we explain how these features are exploited in the optimization of the application.

1) Assignment 3 Heterogeneous platform: The goal of this assignment section is to create a multicore platform consisting of two A15 cores and two A9 cores. The L2 cache of 16kiB is kept as shared and a private L1 cache of 1iKB is set with the associativity level as 1(direct mapped). We execute this configuration on 4 threads `NUM_THREADS` = 4.We have set the number of threads using the OpenMP construct *omp_set_num_threads(NUM_THREADS)* and the splitting of the master thread to multiple threads (forking) starts at line25 *#pragma omp parallel for*. This is shown in figure 17.

VI. Final Assignment

In the final step of the assignment, we optimized the heterogeneous platform V-1 further using more advanced parallelization strategies uing OpenMP. We also improved the algorithm of eeg application, optimized the cache sizes and increased the number of cores for our implementation.

A. OpenMP strategies

As we observe from the figure 17, *#pragma omp parallel*(open mp construct) is used for splitting the for loop. In

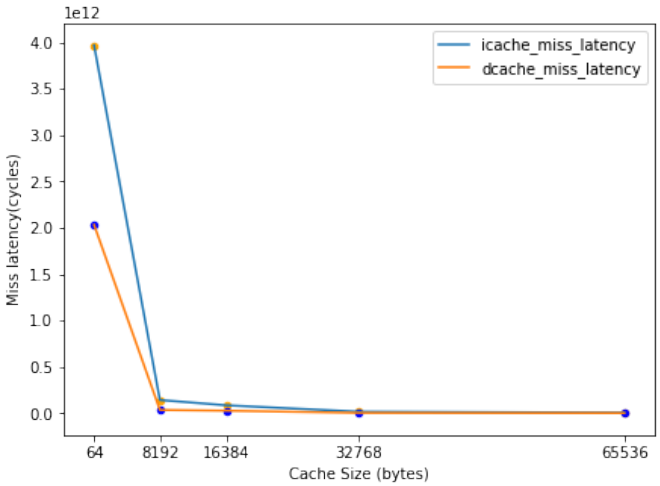

Fig. 14. Impact of L1 cache size on miss latency on A15

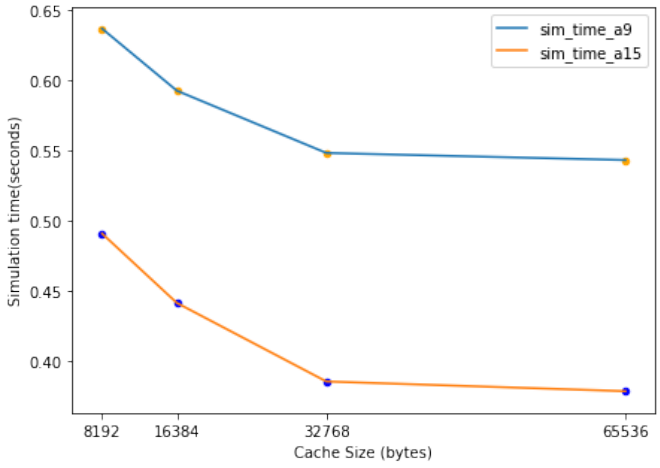

Fig. 15. Simulation time A9 vs A15

the single threaded version of the application, each of the 23 channels will be executed in sequence. Whereas, in the parallel version, we split the load to run on multithreads using OpenMP. Similar strategies have been implemented in other *.c files of the eeg application to exploit maximum parallelism. However, when applying these strategies, one needs to watch out for the private variables which are specific to a particular block of code. Hence we included the *private* clauses in those cases. Each of the inner loop creates its own copy of the specified variable to avoid race condition that is mentioned inside the private clause. The figure 17 shows one such usage from the given eeg implementation. This has been taken care everywhere in our implementation.

#pragma omp parallel sections clause combined with *#pragma omp section* is used for executing independent blocks of the code in parallel. The *#pragma omp parallel sections* defines the region where parallelism can be exploited and the *#pragma omp section* clause defines the sections that can be

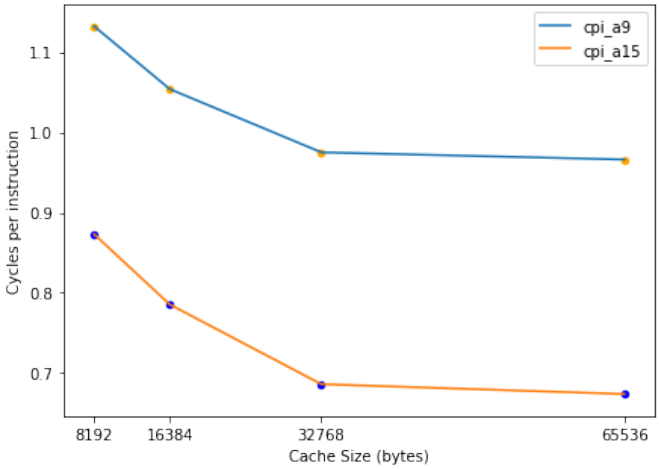

Fig. 16. Cycles per instruction- A9 vs A15

```
16  int main(int argc, char *argv[]) {
17      float features[CHANNELS][FEATURE_LENGTH];
18      float favg[FEATURE_LENGTH] = {0};
19          int32_t x[CHANNELS][DATAPOINTS];
20          uint32_t i, j;
21
22      // void open_set_num_threads(NUM_THREADS);
23      read_data(x, CHANNELS, DATAPOINTS);
24          omp_set_num_threads(NUM_THREADS);
25       #pragma omp parallel for
26      for (i = 0; i < CHANNELS; i++) {
27          printf("Running channel %d...\n", i);
28          run_channel(DATAPOINTS, x[i], features[i]);
29      }
30
31
32      // Averaging channels
33       #pragma omp parallel for private(i)
34   for (j = 0; j < FEATURE_LENGTH; j++) {
35      for (i = 0; i < CHANNELS; i++) {
36              favg[j] += features[i][j] / FEATURE_LENGTH;
37          }
38      }
39
40      printf("\n");
41          for (i=0; i<FEATURE_LENGTH; i++)
42          printf("Feature %d: %.6f\n", i, favg[i]);
43
44      return 0;
45  }
```

Fig. 17. Implementation of OpenMP in *eeg.c*

executed in parallel. A sample snippet of the code where this strategy is employed is shown in figure 18.

The `run_channel` function shown in figure 18 is called by other functions to calculate features which can be executed in parallel. Hence, we used *#pragma omp parallel sections* and

```
void run_channel(int np, int32_t *x, float *features)
{
    // Butterworth returns np + 1 samples
          int32_t *X = malloc((np + 1) * sizeof(int32_t));

    #pragma omp parallel sections
    {
      #pragma omp section
      {
        printf("    Butterworth filter...\n");
        bw0_int(np, x, X);
      }

      #pragma omp section
      {
        printf("    Standard features...\n");
        stafeature(np, X, &features[0]);
      }

      #pragma omp section
      {
        printf("    Peak 2 peak features...\n");
        p2p(np, X, &features[4], 7);
      }
```

Fig. 18. Parallel sections of the code

#pragma omp section in this function method. The reduction

```
float average(int np, int32_t *x)
{
    int i;
    int32_t s = 0;
    #pragma omp parallel for reduction(+:s)
    for (i = 0; i < np; i++) {
        s += x[i];
    }

    return ((float) s) / ((float) np);
}
```

Fig. 19. Usage of reduction clause

clause is a special directive that instructs the compiler to generate code that accumulates values from different loop iterations together. This clause is combined with *#pragma omp parallel for* to perform various arithmetic operations more efficiently. The figure 19 shows one of the instances where the reduction clause was used in the application. Thus in figure 19, each thread calculates its own *s* value and finally the value of *s* from the team of threads are added together to get the final result.

B. Cache size optimization

For the heterogeneous multi-core platform, we conducted tests similar to the single core, by disabling the L2 cache and varying the L1 instruction and data caches sizes.

1) L1 cache size tests: L1 instruction and data cache size of 1kb, 4kb, 16kb and 3kb were tested on a 8-core heterogeneous multicore platform. The associativity level is set to be fully associative(8). From the graph 20, one can see that there is a great improvement in performance when going from 1kB to 16kB. Any extra L1 cache did not give very big of an improvement.

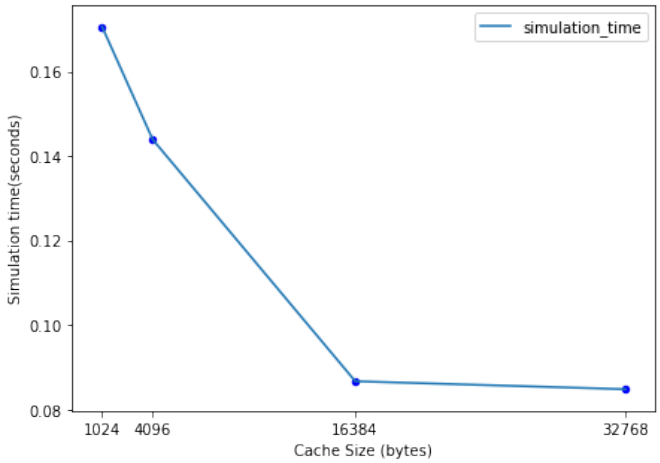

Fig. 20. Impact of L1 cache size on simulation time in multi-core system

2) L1 associativity tests: In order to find the optimal associativity level for the L1 cache, we disabled the L2 cache and set the size of both the L1 instruction and data cache to be 8kB and varied the associativity level from 1 to 8. From this

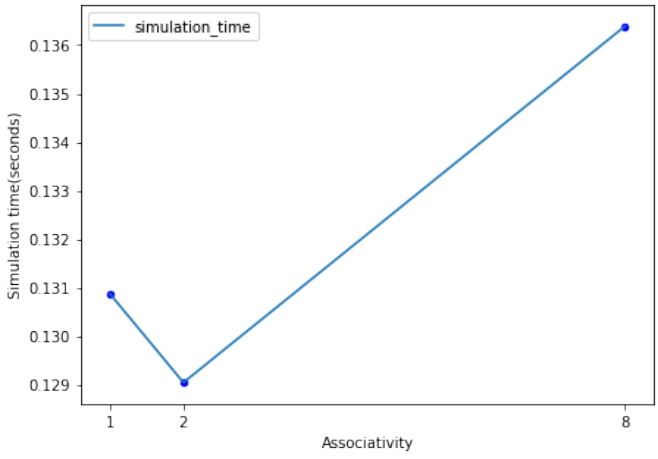

Fig. 21. Impact of L1 cache associativity on simulation time

test, we observe that the optimal associativity would be 2 or 4. Because for the fully associative L1 cache , the hit latency will increase which is not favourable. The ideal solution is to set the minimum cache size for L1 and associativity level which can reduce the hit time. Thus, the optimal solution is to set 16kb of L1 cache size with the associativity level of 4.

3) Multilevel-cache hierarchy tests: In this section, we enabled the L2 shared cache of 256kB with associativity level of 8 and varied the levels of l1 instruction and data cache. We tried the cache configuration similar to that *Intel Core i7 processor* where the associativity level of data cache is higher than that of the instruction cache. Hence, we have set the associativity level of 4 for the instruction cache and fully associative data cache in the L1 cache configuration. This is simulated on a 8 core platform and we evaluated the impact on the simulation time which is shown in figure 22. From

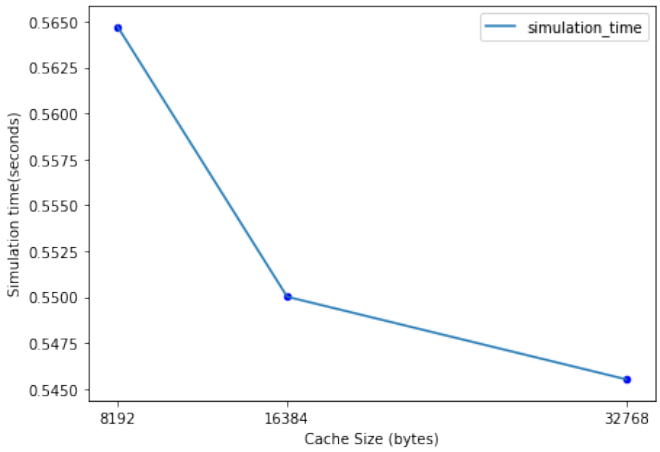

Fig. 22. L1 cache size vs simulation time (with L2 cache enabled) in 8 core

the previous testings of L2 cache, we set up maximum l2 cache size of 256 and fully associative inorder to reduce both capacity misses and conflict misses. Also, we infered that in a multilevel cache hierarchy, l1 cache size should be small to reduce the hit time. Hence, on considering the overall miss rate and the power consumption factors, we finalised the following cache configuration.

```
options.cpu_type = "detailed"   # The A9 is an OutOfOrder CPU
options.cpu_clock = "2GHz"
options.num_cpus = 8

options.caches = 1              # Symmetric, L1 caches
options.cacheline_size = 64

options.l1i_size = "16kB"
options.l1i_assoc = 4

options.l1d_size = "16kB"
options.l1d_assoc = 4

options.l2cache = 1
options.l2_size = "256kB"
options.l2_assoc = 8
```

Fig. 23. Cache configuration

C. Number of cores

Now, with the optimized cache configuration, we simulated the parallelized version of eeg application on different cores ranging from 2 to 8. We have setup four A9 processors and four A15 processors for the 8 core implementation. We expected that as the number of cores increases, the simulation time will reduce. Indeed, it was evident from the results shown in figure 24. It is also interesting to find that the simulation time reduced to a great extent from core 2 to core 6 and for further increase on the number of cores, there was not significant improvement. We also compared the multicore implementation

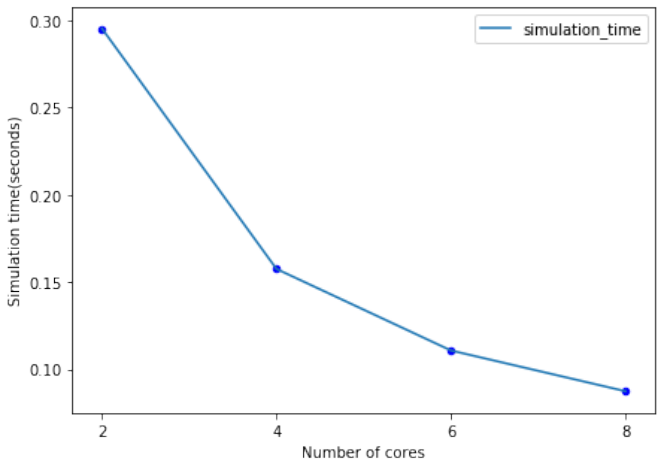

Fig. 24. Number of cores vs simulation time

with the original native version (single core) and evaluated the speed up. The simulation time of each core is compared with the native version and the percentage of improvement(speed up %) is calculated. It is shown in figure 25. As we can see, the multi core implementation has been speeded up by almost *85%* compared to the original implementation.

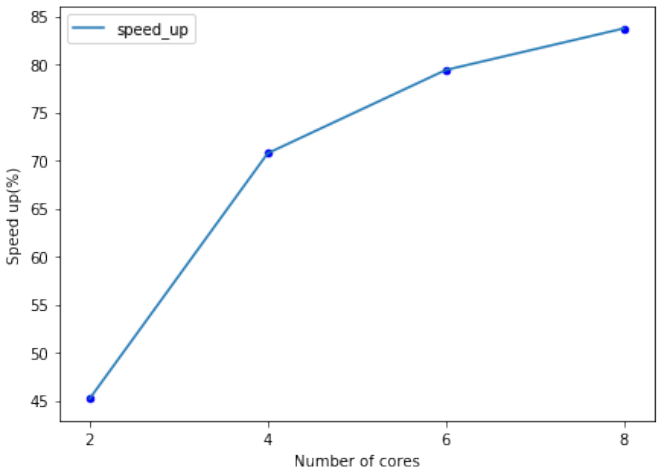

Fig. 25. Number of cores vs speed up

D. Power Consumption

The addition of extra cores and increase in cache size also contributes to the overall power consumption of the system.

As the cache sizes are increases, the static as well as dynamic power consumption increases. The static power consumption increases because of the increase in the number of transistors due to increase in cache size which results in the increase of leakage current. The dynamic power consumption also increases due to the increase in the number of read and write accesses. Though the increase in cache size increases the power consumption, on reducing that, will increase the cache misses. Hence we tried to reach a best trade off between the cache size and power consumption without much comprises on the miss rate and hit latency.

VII. Conclusion

This assignment wasintended to understand the memory hierarchy and performance of a single core vs multi core implementation. For the single core, we conducted several cache tests to understand the impact of cache optimization on performance. We conducted similar tests on multicore implementation to achieve an optimal solution. Also, we tried different parallelization strategies using OpenMP and obtained the best result possible. Many results were expected, like a smaller L1 cache will reduce the hit time and larger L2 cache helps in reducing the miss rate. However, we could not optimize beyond a certain level as we severely faced the *CORE DUMPED ERROR*. But, we still managed to find the best optimal solutions possible. Finding good cache configurations certainly helped increase performance at a significant level.

VIII. Acknowledgment

The authors would like to thank all the technical assistants and research assistants who helped us fixing some of the major bugs. Also, we would like to thank all the students who helped us with the issues on the oncourse discussion forum.

References

[1] J. L. Hennessy, D. A. Patterson, Computer Architecture: A Quantitative Approach, 5th Edition, Morgan Kaufmann Publishing Co. 1996.

[2] Michel Dubois, Murali Annavaram, Per Stenstrom, Parallel Computer Organization and Design

[3] OpenMP Wikipedia: https://en.wikipedia.org/wiki/OpenMP [Online]

[4] Gem5 Website : https://gem5.org [Online]

[5] Julian Bui, Chenguang Xu, Sudhanva Gurumurthi, *Understanding Performance Issues on both Single Core and Multi-core Architecture*, Computer Organizatione07, Month 912, 2007, Charlottesville, VA, Copyright 2007 ACM 1-58113-000-0/00/0007