Project 3 Report

Christina Pavlopoulou cpavl001@ucr.edu

Andres Calderon acald013@ucr.edu

June 8, 2016

1 Implementation

1.1 Victim Cache

To implement this part, we firstly added a new cache. To add the new cache we changed the following files:

1. `sim-outorder.c`: There we checked if some requirements were satisfied. We have to check if there is data level 1 cache. If it exists, then we can create it. Then, we had to register the victim cache with its default values as it is requested in the assignment. Then we created the cache itself. Finally, we had to create a cache access function to declare what happens in every miss in the victim cache.
2. `power.h` and `power.c`: In these files, we added variables and functions to calculate the power consumption for victim cache. We based on the calculations for data cache level 1.
3. `cache.c` and `cache.h`: Here, we created the `vic_cache_access` to define what will happen with replacements in a victim cache. According to the assignment, if we have a hit, the block must be transferred to dl1 cache, and the evicted block to victim cache, to replace the hit block.

1.2 Stream Buffer

In this part of the assignment, we had to create two stream buffers, one for the dl1 and the other for the il1 cache. We observed that to add a stream buffer, we had to follow the same steps as we did for the above section. In the previous part, we only had to add one cache between dl1 and dl2. Now, we had to also add one cache between il1 and il2. We did the same procedure but instead of following dl1 cache, we followed the il1 cache.

1. `cache.c`: We added a function called `cache_prefetch_block` to manage what is happening in a hit or miss block. In this case on a miss, the stream buffer prefetches all the 4 blocks after the miss block.

1.3 Pseudo LRU replacement

In this part, we used the FIFO replacement policy as guide to create the PLRU policy. We changed the following files:

1. `sim-outorder.c`: We added the option for the PLRU policy in the replacement parameter that already existed for each cache.
2. `cache.c` and `cache.h`: In the `cache_access` function, we create the PLRU case both in misses and in hits. Inside these cases, we use two functions to implement the code for the tree-based pseudo LRU replacement algorithm.

2 Simulations

We run simulations using the GCC and Anagram applications. For both cases, we run first a baseline evaluation of `sim-outourder` with the default parameters without the deployment of neither victim or stream buffers caches. Then, each new cache implementation was evaluated. First, we test `sim-outourder` with a victim cache (without the stream buffers), and then with both, instruction and data, stream buffers (without the victim cache). Finally, we run `sim-outourder` setting the replacement policy of the data cache level 1 to the new implementation of Pseudo-LRU.

2.1 Results for GCC

We compare two metrics to measure performance: the total simulation time in cycles and CPI. Figures 1a and 1b show the performance results on these two metrics for victim caches and stream buffers compared to the baseline. We can see a considerable improvement of both implementations, but the use of an instruction and data stream buffer seems to bring the best results.

In the case of power consumption we measure the total power per cycle and the average total power per instruction and we compare them to the baseline. Figures 1c and 1d show a similar trend already seen with the performance metrics. Both caches reduce the power consumption with a greater reduction in the case of the stream buffers.

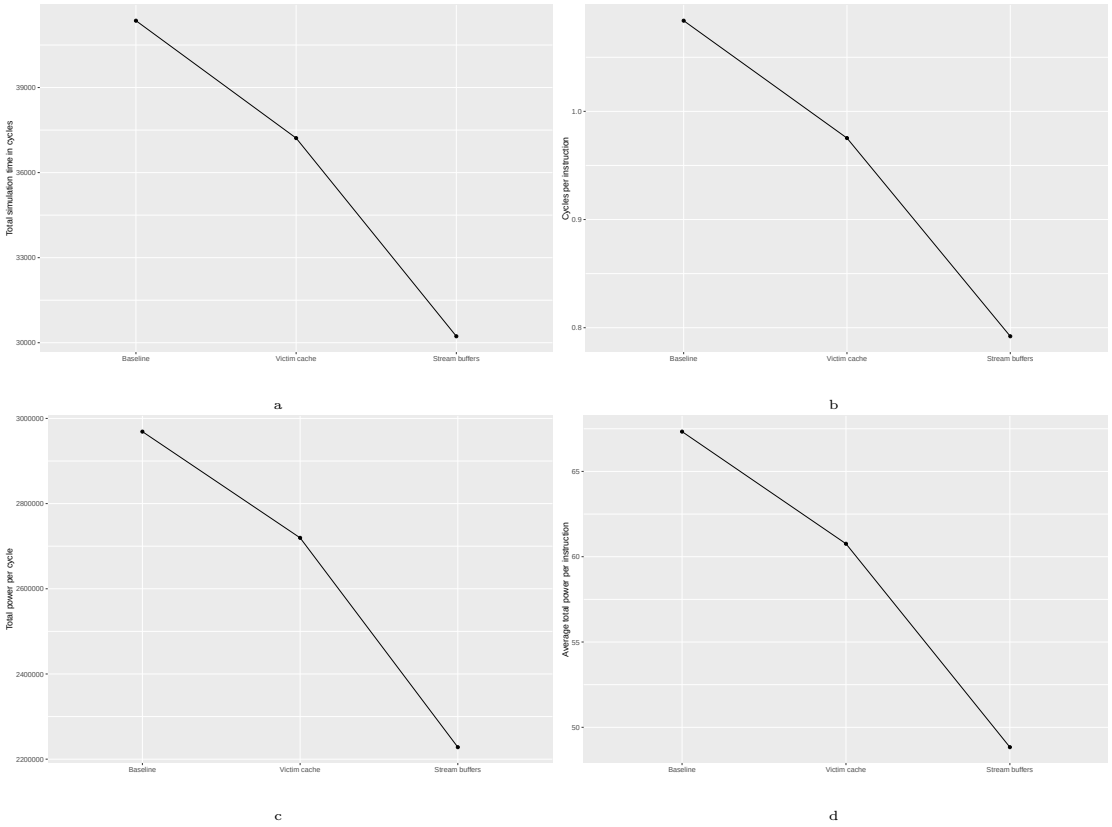

Figure 1: GCC metrics comparison. (a) Total time in cycles for the GCC simulation. (b) Cycles per instruction (CPI) for the GCC simulation. (c) Total power per cycle for the GCC simulation. (d) Average power per instruction for the GCC simulation.

2.2 Results for Anagram

We used the same metrics to measure performance (CPI and time in cycles). Figures 2a and 2b show the performance results. Although the new cache implementations still gave important improvements, for this application the victim cache offered the best performance.

Similarly, the power consumption follows the same trend. Figures 2c and 2d show the results for the total power per cycle and average power per instruction respectively. We can see that for this application, the victim cache seems to bring better consumption rate that the stream buffers. Overall, both implementation are good alternatives to improve performance and power consumption.

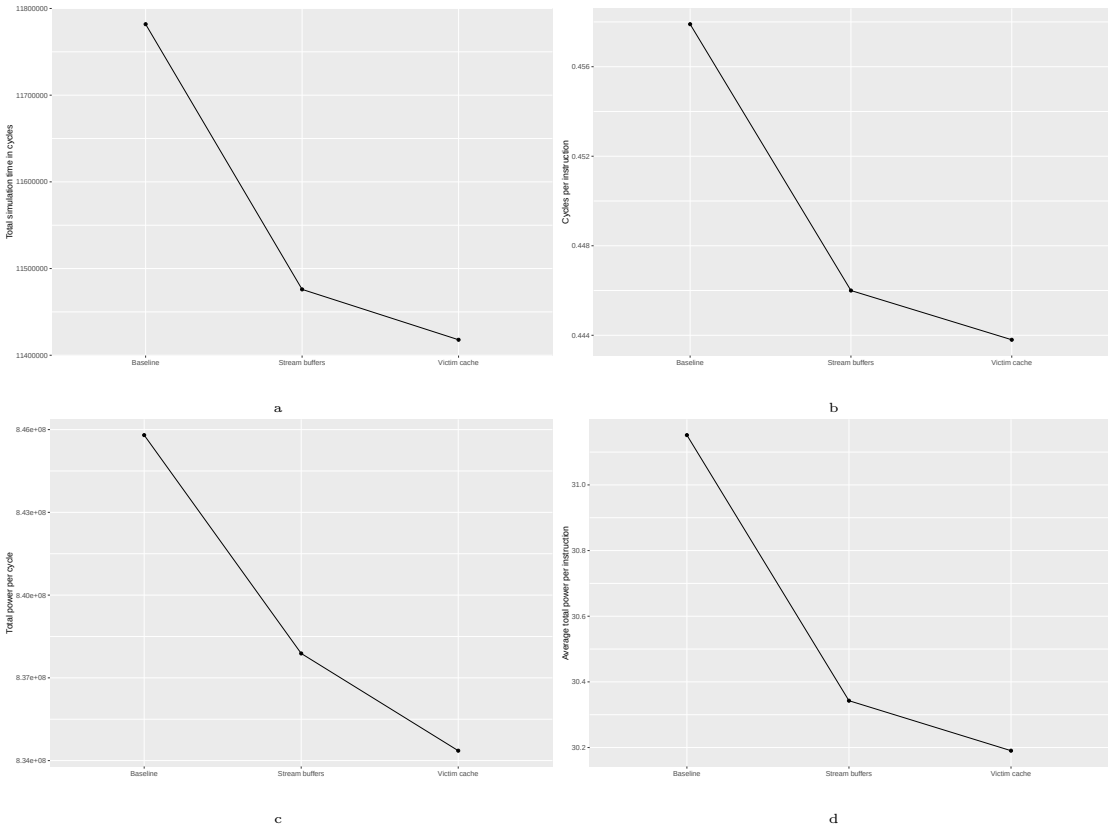

Figure 2: Anagram metrics comparison. (a) Total time in cycles for the Anagram simulation. (b) Cycles per instruction (CPI) for the Anagram simulation. (c) Total power per cycle for the Anagram simulation. (d) Average power per instruction for the Anagram simulation.

2.3 Pseudo LRU

Table 1 shows the comparison after the implementation of PLRU. We can see that the result are slightly worse than the baseline. However, since LRU is more expensive, PLRU is a nice alternative with similar performance.

Metric	Replacement policy	Value
Cycles per instruction	Baseline	1.0838
	PLRU	1.084
Total power per cycle	Baseline	2969016.2831
	PLRU	2969447.0122
Average total power per instruction	Baseline	67.3323
	PLRU	67.3359

Table 1: PLRU comparison with baseline.