ASSIGNMENT-1: CACHE PROFILING OF SORTING ALGORITHMS

CS6600, Computer Architecture
Sai Gautham Ravipati, EE19B053

1 Introduction

Caches perform an important task in the modern processors. They try to solve the memory wall problem and provide a method to serve the data requests faster. The access patterns of the memory vary from algorithm to algorithm, so does the cache performance. Hence, it is useful to analyse the cache performance by varying the design parameters like associativity, block-size and cache-size for different algorithms. In this assignment, different sorting algorithms like bubble sort, selection sort, merge sort, quick sort and radix sort have been analysed for the cache performance by varying the input size. Further, cache parameters were tuned in order to improve the hit-rate. The IPCs for the case of original processor were estimated using the execution time and data from the cache simulator. All of the experiments were performed on an Intel i5-8265U processor using the Cachegrind profiler.

2 Profiling of Sorting Algorithms

Algorithms of different complexity have been analysed for the cache performance for four different lengths of test vectors, $[250k, 500k, 750k, 1000k]$. Bubble sort and Selection sort being $O(n^2)$ take significant time for execution even for the starting test length of $250k$. While rest of the algorithms, merge sort, radix sort and quick sort being less than quadratic in complexity work sufficiently faster in comparision to the former. The profiling data obtained for each of the cases has been presented in Figure-1. The execution time (not cachegrind simulation) of the code obtained from time command is also presented. For the case of selection sort and bubble sort, the smallest case was taking around 50 min for simulation owing to the complexity of algorithm. Hence, profiling is done only for the case of $250k$ for these two cases. For this simulation the cache parameters are taken to be same as that of the Intel core. The I1 cache is 32 KB (8-way associative, 64 B block), D1 cache is 32 KB (8-way associative, 64 B block) and the LL cache is 6144 KB (12-way associative, 64 B block).

N: 250000								
Algorithm	**Instruction Refs.**	**I1 misses**	**Data Refs.**	**D1 misses**	**LL refs.**	**LLd misses**	**LLi misses**	**Execution Time**
Bubble	1,093,973,162,972	1,090	531,361,737,320	1,950,995,863	1,950,996,953	18,250	1,064	177.81 s
Selection	562,535,418,947	1,090	281,271,169,086	1,951,102,400	1,951,103,490	18,251	1,063	57.53 s
Merge	358,564,072	1,125	185,279,738	443,981	445,106	41,825	1,101	0.055 s
Quick	182,665,590	1,103	100,091,408	203,577	204,680	18,250	1,080	0.037 s
Radix	363,442,259	1,105	120,313,665	981,763	982,868	33,882	1,081	0.057 s
N: 500000								
Algorithm	**Instruction Refs.**	**I1 misses**	**Data Refs.**	**D1 misses**	**LL refs.**	**LLd misses**	**LLi misses**	**Execution Time**
Merge	745,418,715	1,125	385,495,189	1,009,720	1,010,845	80,885	1,101	0.108 s
Quick	367,065,824	1,103	201,140,462	424,875	425,978	33,875	1,080	0.072 s
Radix	726,676,129	1,105	240,563,665	1,966,106	1,967,211	65,132	1,081	0.105 s
N: 750000								
Algorithm	**Instruction Refs.**	**I1 misses**	**Data Refs.**	**D1 misses**	**LL refs.**	**LLd misses**	**LLi misses**	**Execution Time**
Merge	1,145,683,980	1,125	592,667,689	1,650,403	1,651,528	119,983	1,113	0.160 s
Quick	604,182,320	1,103	331,116,748	802,179	803,282	49,500	1,080	0.116 s
Radix	1,089,910,001	1,105	360,813,665	2,950,534	2,951,639	96,430	1,091	0.166 s
N: 1000000								
Algorithm	**Instruction Refs.**	**I1 misses**	**Data Refs.**	**D1 misses**	**LL refs.**	**LLd misses**	**LLi misses**	**Execution Time**
Merge	1,547,618,389	1,125	800,921,671	2,266,278	2,267,403	409,921	1,113	0.225 s
Quick	808,046,224	1,103	442,903,006	1,051,209	1,052,312	65,126	1,080	0.154 s
Radix	1,453,143,890	1,105	481,063,666	3,934,901	3,936,006	1,547,702	1,092	0.214 s

Figure 1: Cache Profiling data obtained using different input sizes along with code execution times

Of the given algorithms, the bubble sort takes the most instructions. This is because not only that the algorithm is $O(n^2)$ but this method also uses swap operations between adjacent elements and in multiple passes. On contrast, the selection sort tracks the maximum element unlike the bubble sort algorithm and performs relatively lower number of data swaps. Although the best case complexity of Bubble sort is $O(n)$, for random test vectors, the selection sort is better for the aforementioned reason. But since, both the algorithms have a worst case complexity of $O(n^2)$, hence, they take relatively more instructions to perform the sorting in comparision to the other set of algorithms. On checking the relative number of cache misses, the instruction misses are negligible when compared to the number of hits. Similarly, the LL data misses are less than 0.01%, hence these 2 types of misses

aren't the common case. On the contrary, for the case of Bubble sort the L1 data misses are 0.4% in comparision to the hits and account for the significant number of cache misses. Therefore, only L1 data cache has been modified at a later point for analysis of cache performance. On close observation, the L1 data misses for both Bubble-sort and Selection-sort are almost the same, this is because, although the bubble sort algorithm makes more access to memory, the data access pattern is almost the same for both the algorithms. While the bubble sort access contiguous elements and swaps immediately, the selection sort also accessing contiguous elements finds the maximum before swapping.

Further, the complexity of the other three algorithms is evident from the total number of instructions. Since the three faster algorithms have lower complexity when compared to the other two algorithms and thus have lower number of instructions. Within the three fast algorithms, quick sort has the least number of data misses. This is because, quick sort uses an in-place sorting algorithm which takes advantage of the locality of reference unlike the other two algorithms which use separate arrays to perform the sorting. The merge sort algorithm although exhibits some temporality, still uses separate arrays for the left and right sub-arrays. Further, the indices of left and right sub-arrays are not incremented continuously but are dependent on the data they hold and hence this influences the access pattern. While this is the case of merge sort, radix sort uses a separate array to store the output. On observation, the radix sort algorithm seems to have the highest number of D1 misses and this could be because of the non-regular data accesses. The access pattern is dependent on the data and could also involve a strided access because of dividing the array index by the exponent. Moreover, the radix sort is different from the other set of algorithms as the radix parameter is to be tuned for performance. On increasing the radix parameter from 10 to 20, the number of data access reduced from 240 M to 168 M, while the number of instructions reduced from 727 M to 501 M. Hence, for the random test vector used, the quick sort algorithm performs better on terms of cache performance.

3 Inference of IPC

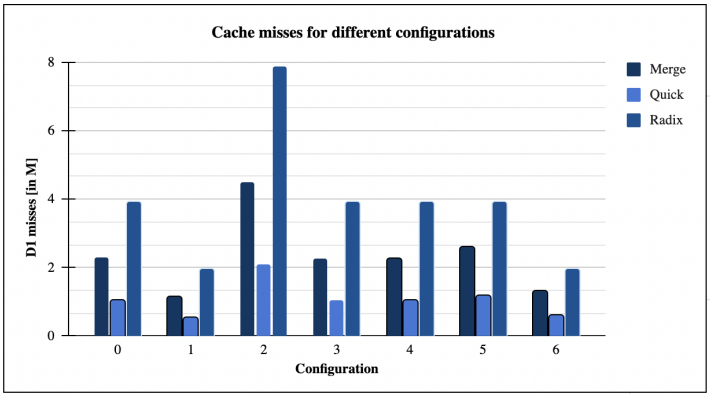
For the inference of IPC using the configuration of original processor, the data from the perf tool was used along with that of the Instruction count from the cachegrind tool. Since cachegrind doesn't provide any cycle count, estimation of IPC for cache configurations different from that of the base processor, requires other information like access latencies or the cycle accurate simulation of code for the required design parameters. Hence, in this case, to obtain the cycle counts for the code running on the base processor, the same is timed using the time command and the respective execution times are presented in Figure-1. The average frequency obtained from perf tool is used to estimate the cycle counts. Using the cycle counts inferred and the instruction count from the cachegrind tool the IPCs were estimated for several different sizes of test vectors. These values are approximately close to those obtained from the perf tool and this provides an insight about the accuracy of cachegrind tool. Further, it can be inferred that the IPC for a given algorithm varies slightly as we change the input test vector but not significantly.

N: 250000				N: 500000			
Algorithm	**Frequency**	**Cycles**	**IPC**	**Algorithm**	**Frequency**	**Cycles**	**IPC**
Bubble	3.394 GHz	604 G	1.81	**Bubble**	-	-	-
Selection	3.766 GHz	217 G	2.59	**Selection**	-	-	-
Merge	3.535 GHz	194 M	1.84	**Merge**	3.705 GHz	400 M	1.86
Quick	3.669 GHz	136 M	1.34	**Quick**	3.775 GHz	272 M	1.35
Radix	3.746 GHz	214 M	1.71	**Radix**	3.829 GHz	402 M	1.81
N: 750000				**N: 1000000**			
Algorithm	**Frequency**	**Cycles**	**IPC**	**Algorithm**	**Frequency**	**Cycles**	**IPC**
Merge	3.777 GHz	604 M	1.89	**Merge**	3.712 GHz	835 M	1.85
Quick	3.753 GHz	435 M	1.39	**Quick**	3.779 GHz	582 M	1.39
Radix	3.727 GHz	619 M	1.76	**Radix**	3.825 GHz	818 M	1.78

Figure 2: IPCs obtained for the cache configuration same as that of the Intel core

4 Optimising cache design parameters

From the previous profiling data, it is evident that L1 data misses constitute a significant portion in comparision to the other type of misses. Hence, we consider this to be the common case and this is to be optimised by Amadahl's law. To reduce the number of L1 misses only the L1 cache configuration was tuned, by changing the cache size, associativity and block size. The resultant profiling data for the number of L1 data misses for different fast algorithms has been presented in Figure-3. For all the three algorithms it can be seen that increasing the block size is reducing the number of D1 misses. This could be because of more accesses to contiguous words in the memory by the algorithms and reduced cold misses. While the number of misses are almost reduced by a factor of two, this comes at the cost of increased latency to bring larger and larger blocks to the cache. Hence indefinitely increasing the block size is not feasible. Clearly reducing the block size to 32 B has led to increased number of cache misses in most of the cases, and this could be because, both the increased number of cold misses (for say, we need to load two words, a 32 B line incurs two misses, while a 64 B line incurs only a single miss) as well as conflicts because of multiple words going to the same line. The number of cache misses is almost invariant with respect to the associativity, hence reducing the same is beneficial as it could reduce the search time. Changing the cache size has very minimal effect on the cache misses and we could possibly use a lower size cache for the case of sorting applications, and this could possibly reduce the access latency. In the final configuration, the size of the cache and associativity were reduced to half and the block size was doubled to reduce the miss rate by nearly 2x. While this could be the case for sorting applications, it might not be the case in general and for designing the same multiple algorithms are to be considered.



Config.	Size	Assoc.	Block Size
0	32 KB	8	64 B
1	32 KB	8	128 B
2	32 KB	8	32 B
3	32 KB	4	64 B
4	32 KB	16	64 B
5	16 KB	8	64 B
6	16 KB	4	128 B

Figure 3: Variation of D1 miss rate w.r.t different cache configurations

5 Conclusion

In this assignment, multiple sorting algorithms were analysed for their performance with respects to cache effects. The $O(n^2)$ algorithms take significant time for execution and this is because of their large number of access to the memory as well as the cache misses in comparision to the other algorithms. Within the fast algorithms, the quick sort has both lower number of access as well as misses owing to the in-place sorting algorithm while the other two use separate arrays to perform the same. However with the other 2 algorithms, merge sort makes more access to memory. But since access are more continuous in merge sort than the case of radix sort, the latter incurs more D1 misses. Thus quick sort performs better in comparision to other algorithms on terms of cache performance. The IPC for the configuration of base processor were estimated by timing the code and obtaining the frequency from the perf tool. For this case, IPC for a given algorithm was almost the same for different length test vectors. The cache parameters were modified to reduce the number of D1 misses since they constitute the majority of the misses. Hit rate was improved by using a cache block of higher size. Therefore, the cachegrind tool provides important insights related to the cache performance of a given algorithm. However, it doesn't provide any information about the cycle counts and addition of the same shall further provide valuable insights.

6 Processor Configuration

All the experiments were run on an 8 core, Intel i5-8265U CPU with an L1d cache of 32K and L1i cache of 32K, L2 cache of 256K, L3 cache of 6144K.