Introduction

A multi-level direct-mapped cache, L1 and L2, was implemented in Java, and a set of experiments was run with different configurations to determine the optimal configuration, along with analysis.

Two varying cache parameters were used for testing: block size in bytes/block and number of blocks, both assumed to be powers of 2 specified by the user.

Main memory addresses are parsed into the following format: Tag bits | Index bits | Offset bits.
If the block required is present in cache, it is returned from there and the number of hits is incremented. Else, the block and data associated with it is obtained from main memory, incrementing the number of misses.

Technical Overview

OOP was used to structure my code so that the same interface is used for a block in cache and main memory, each having its own set of blocks associated with some data.

The block class is responsible for defining a block object, created by specifying in the constructor the index, tag and data associated with it. These parameters are determined in the cache class.
When a block is created with some data, the valid bit is set to 1. Two important getters are implemented, getTag() and getIndex(), to return the tag and index associated with a block.

The cache class is more comprehensive. An object of this class is created by specifying the number of blocks in cache, and block size in bytes/block. Creating this object implies forming a separate set of blocks of one fixed size in cache, along with allocating the data size for each block.
The read() method forms the crux of this class; it reads addresses from a file passed into the argument, adding the blocks of addresses to main memory in their allocated indices.

Address partitioning is used to specify the index and tag for each block, which is done as follows using the getIndexBits(), getITagBits() and getOffsetBits() methods.
After converting the addresses to binary, the index bits are determined by $\log_2$(No. of blocks), offset bits by $\log_2$(Block size), and the tag bits are what is left over: 16 – [(getIndexBits()+getOffsetBits)]. These are extracted from the address in the format, Tag bits | Index bits | Offset bits.
The rest of the methods in the cache class are used to increment the hits, misses, or getters returning the index, tag, block number, block size, hits and misses.

Results and Analysis

A set of experiments was devised across a range of configuration parameters for analysis. The results obtained for the assignment.addr file are tabulated below.

Config	No. of blocks	Block size	Hits	Misses	Cycles	CPI	Performance ratio
1	8	8	48333	23269	23752330	332	1
2	8	16	58278	13324	13906780	195	1.7
3	16	16	60098	11504	12104980	170	2.0
4	16	32	65420	6182	6836200	96	3.5
5	16	64	67628	3974	4650280	65	5.1
6	16	8	50379	50379	21726790	304	1.1
7	32	8	51387	20215	20728870	290	1.1
8	64	16	64690	6912	7558900	106	3.1
9	128	32	64944	6658	7307440	59	5.6
10	128	128	70506	1096	1801060	26	12.8

Table 1: L1 results for assignment.addr

Config	No. of blocks		Block size		Hits		Misses		Cycles	CPI	Performance ratio
	L1	L2	L1	L2	L1	L2	L1	L2			
1	8	16	8	16	48333	12173	23269	11096	12796630	179	1
2	8	8	16	8	58278	1820	13324	11504	12268780	172	1
3	8	64	16	64	58278	11548	13324	1776	3513580	50	3.6
4	16	64	32	128	65420	5232	6182	950	2127400	30	6.0
5	64	64	64	64	69432	0	69432	2170	2864320	41	4.4
6	16	8	16	8	60098	610	11504	10894	11555980	162	1.1
7	32	64	8	64	51387	18463	20215	1752	4112170	58	3.1
8	64	128	16	128	64690	6076	6912	836	2090500	30	6.0
9	128	32	32	128	68128	2400	3474	1074	1995280	28	6.4
10	128	256	128	256	70506	893	1096	203	997360	14	12.8

Table 2: L2 results for assignment.addr

A similar but smaller set of experiments was devised for the heapsort.addr file, which has a much larger data set than assignment.addr.

Config	No. of blocks	Block size	Hits	Misses	Cycles	CPI	Performance ratio
1	8	16	923738	196875	206112380	184	1
2	16	16	975926	144687	154446260	138	1.3
3	16	32	1033707	86906	97243070	87	2.1
4	128	32	1091636	28977	39893360	36	5.1
5	128	128	1114645	5968	17114450	16	11.5

Table 3: L1 results for heapsort.addr

Config	No. of blocks		Block size		Hits		Misses		Cycles	CPI	Performance ratio
	L1	L2	L1	L2	L1	L2	L1	L2			
1	8	16	8	16	850137	137749	270476	132727	155003270	139	1
2	16	64	32	128	1033707	74574	86906	12332	30126470	27	5.1
3	16	8	16	8	975926	17099	144687	127588	139057160	125	1.1
4	64	64	64	64	1095788	0	24825	24825	35782880	32	4.3
5	128	256	128	256	1114645	5811	5968	157	11884550	11	12.6

Table 4: L2 results for heapsort.addr

Evaluation

The calculations performed in obtaining the comparisons made in the Results and Analysis section are shown.

Taking 10 cycles for a L1 hit and 1000 cycles for a main memory fetch,
No. of cycles for L1 cache = (L1 hits * 10) + (L1 misses * 1000)

Taking 10 cycles for a L1 hit , 100 cycles for a L2 hit and 1000 cycles for a main memory fetch,
No. of cycles for L1 cache = (L1 hits * 10) + (L2 hits * 100) + (L2 misses * 1000)

Cycles per Instruction (CPI) = Number of cycles / Number of addresses

Finally, the performance ratios were calculated by comparing the configuration with the highest CPI (worst performing) to the subsequent configurations.

Two data sets of address instructions, assignment.addr and heapsort.addr, were experimented with different configurations to check for consistency. Analyzing Tables 1 – 4, it can be seen that both the files produced results with a similar pattern explained below.

Performance	**Cache level**	**No. of blocks**		**Cache size**		**CPI**
		L1	L2	L1	L2	
Worst	L1	8		8		332
	L2	8	16	8	16	179
Best	L1	128		128		26
	L2	128	256	128	256	14

Table 5: Best and worst performance for assignment.addr file. An identical pattern is seen for heapsort.addr

The optimal configuration required maximum block and cache sizes, while the configuration with the smallest cache and block sizes performed the worst. This is applicable for both the data sets that were tested.

Larger block sizes take advantage of spatial locality to reduce the *compulsory misses* i.e on the first access to a block. Similarly, a larger cache size means the cache can now hold more blocks at any one time. This in turn increases performance by reducing the number of *capacity misses* – when the cache is not large enough to hold all the blocks required by a program.

Along with tweaking the cache/block sizes, the number of caches can also impact the performance since main memory access is avoided to lower the miss rate.
It can be seen from Table 5 that the performance is significantly increased when using a multi-level cache. The best performance configuration for L1 was executed with a 26 CPI, while the same L1 configuration together with a larger L2 cache was executed with 14 CPI. The same effect of lower CPI for L2 is seen in the worst-case performance.

In general, L1 caches are faster than L2 caches, which is a trade-off considering the much higher cost of L1. This is achieved by using a different SRAM design for L2, which is optimized to increase its capacity and hence, lower the miss rate. In contrast, the SRAM for L1 is optimized to focus on minimal hit time.
Although L2 is slower than L1, it is still faster than main memory, which explains its presence.

Although increasing the block size decreases capacity misses as explained above, this may increase the miss penalty since more data needs to be fetched at every miss. In addition, increasing the block size means fewer blocks can be stored in cache, thereby increasing the tendency of conflicts between two blocks.
Similarly, a larger cache would increase the time taken to deliver data in the cache to the processor (hit time).

From the experiments conducted, increasing both the block and cache sizes would give the best performance. However, generalizing this does not take into account the main focus of each cache level.
Since the primary L1 cache focuses on minimal hit time, increasing the block size more than the cache size would ensure lower hit time. L2 cache aims at increasing the hit rate, which is ideal for a larger cache size. However, a larger cache size would mean a rapid increase in the cost.

Future Work

Integrating the support for associativity into the cache simulator would enable a greater control over the parameters to boost performance. Higher associativity would reduce miss rate by preventing *conflict misses*, where the cache has sufficient space, but the block cannot be maintained in a set because another block maps to the same set.

In order to expand the simulator to support an N-way associative cache, I would implement a separate class named ‘Set’. A cache would consist of a collection of sets, while a set would consist of a collection of N number of blocks. Therefore, when creating a cache object, one would need to pass in the value of N, the associativity, along with the number of blocks and block size, as implemented currently. Within this same constructor, the complete Set would be initialized to a size of (Number of blocks)/N, while each set within would hold N number of blocks. This would hold the blocks containing the data.

When searching for a block in cache, the index is now used to find the set containing the block, and the tag to find the block within the set. To facilitate this, read() and write() methods would be implemented in the Set class. The read() method finds the block in the set based on the tag, while the write() method inserts a block into the set.

To put this together, the code used to search for the block in the cacheSimulator main() method would be changed to first finding the set within the cache corresponding to the index bits, and then searching for the block within the cache corresponding to the tag bits. If the block is not found in cache, get the set corresponding to the index bits, and insert the block into the set.

It is worth noting that direct-mapped cache and fully-associative cache are special cases of an n-way associative cache. The former is the case N = 1, while the latter is the case N = number of blocks. These could be implemented.

Conclusion

Creating this basic cache simulator gave me a hands-on insight into the general working of the cache, facilitated by extensive research and reading. It was a good exercise to plan the overall structure of the code.

By verifying experimentally, it was interesting to note that careful design choices were needed to enhance cache performance.