**EGCR5101-Computer Architecture**

**Parametric Cache Simulator**

**By**

**Sathyanarayanan Ramesh Babu**

**800846311**

**Problem Statement**: Design and implement a parametric cache simulator and use it to design data caches suited for FFT. Study the data access pattern and include this study as part of your report. The simulator should model a memory hierarchy with data and victim cache. Assume two levels of cache, the LRU replacement policy, and

(i) write back with write allocate.

(ii) write through with no-write allocate policies.

**Simulator Inputs:**

* Date cache size in bytes.
* Associativity.

Direct mapped

2-way set associative

4-way set associative.

* Block size in bytes.
* Write policy.

Write back with write allocate

Write through with

* Victim cache size in blocks. (if 0 the system does not have a Victim cache)

**Fast Furiour Transform:** A fast Fourier transform (FFT) is an algorithm to compute the discrete Fourier transform (DFT) and its inverse. Fourier analysis converts time (or space) to frequency and vice versa; an FFT rapidly computes such transformations by factorizing the DFT matrix into a product of sparse (mostly zero) factors.

The “.c” file and the compiled executable are attached with file that is being submitted.

**Creating a Tracefile:** Tracefiles return all the memory accesses done by our program. Hence allowing us to have peek into the memory access pattern and other dynamic operations (such as cache) inside a computing system.

The tool used to create the tracefile in the format used by my cache simulator is valgrind’s “lackey”.

valgrind --tool=lackey --trace-mem=yes FFTarg.c >>trace.txt

**Input file format:** The tracefile generated is of the following format.

<type of memory access> (<runtime address generated>, <size of the datatype accesed>)

Under type of memory access we have:

* L-Load
* S-Store
* M-Modify
* I-Immediate data

For the design, 2 of such tracefiles were used. Few line example of a tracefile is as follows:

* S 7ff000390,8
* I 0040051f,3
* I 00400522,4

The generated tracefile was copied or saved in the form of the “text” file and interfaced with the program.

**Design challenges and Code Walkthrough:**

**Stage1: Address Parsing:**

The given trace file format has to be parsed and we have to break the given addresses into 3 parts.

The input addresses are 32 bits. Which have to be divided as :

<TAG><INDEX><OFFSET>

The number of bits that constitutes the <INDEX> and <OFFSET> of given address is generated at run time and is dependent on user input.

Number of bits in block offset is log to the base 2 of (block size)

Number of bits in index is log to the base 2 of (number of sets(S)).

S can be calculated from the equation:

Cache size= (number of sets (S))\*(Associvity)\*(Block size).

3 of the 4 variables are given by the user and hence find S.

Log to the base 2 of (B) and (S) are calculated by :

int temp=1,b1,s1;

b1=b;

while(1)

{

if(b1==temp)

{

break;

}

else

{

temp=temp<<1;

B++;

}

}

//cout<<B<<"\n"; //number of bits in block offset

temp=1;

s1=s;

while(1)

{

if(s1==temp)

{

break;

}

else

{

temp=temp<<1;

S++;

}

}

Hence B and S hold the number of bits to be extracted as block offset and index respectively.

Hence the parsing is done by the following “pharse\_address()” function .

void pharse\_address()

{

int y;

int z;

for(int i=0;i<n;i++)

{

z=B+S;

tag\_data[i]=memory[i]>>z;//right shifted by (B+S) number of bits to remove the tag

y = (e\*(tag\_data[i]&((1<<S) - 1)));//y is our index.

if(y>s)

{

y=s;

}

idx\_data[i]=y;

//printf("%d\n",idx\_data[i]);

//y=i;

}

}

**Stage 2: Memory allocation**

**Nested structure approach**

By the virtue of my design I am using nested structures. Cache block is the smallest of them all which is inside a cacheline many cachelines form a set and many sets form the whole cache. Each set has its adjoining variables inside it.

The number of sets inside the cache is given by (S) and the number of cache lines inside a set is given by the associvity. Hence the number of such entities are initialized and allocated memory at runtime.

A victim cache which is fully associative and having 8 blocks is initialized and allocated memory at compile time.

L2 cache with all the dimensions doubled such as L1 is initialized and allocated.

**Multidimensional Pointer approach**

In the pursuit of writing an optimized code, I started off by declaring my cache as a 2 dimensional pointer such as

Cache[set number][block number]

instead I access like:

Cache.set[set index].cacheline[n<associvity].block

The malloc statement goes something like this :

struct cache\_line\*\*cache=(cache\_line\*\*)malloc(set\_number\*sizeof(struct cache\_line\*));

struct cache\_line \*current\_set;

But due to my limited programming exposure, I faced a lot of problems in pointer access. Especially during the later stages of the project. Hence I had to abandon this idea stick to nested structures. But I still maintain, this approach is more optimal with very limited number of “for loops” and “if” statements when compared to the approach that I have taken.

**Stages 3: Hits, misses and optimization**

Multiple nested “for” loops and if statements are used to iterate and traverse within a cache. Each set has a set index that is compared with the index of each of the address from the tracefile. If they match, that addressed is mapped into that set. Within a set we iterate to find if the tag of the fetched instruction is present. If it is present it is a “hit”. If it is not present we have to find a place for this tag using LRU and this is a “miss”.

When LRU is done and the evicted block is written into the victim cache.

**Results and inferences:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Size | associvity | blocksize | no. of sets | hits | misses | read hits | read misses | reads |
| 32 | 1 | 4 | 8 | 197 | 37 | 154 | 2 | 156 |
| 32 | 2 | 4 | 4 | 294 | 52 | 213 | 9 | 222 |
| 32 | 4 | 8 | 1 | 233 | 831 | 173 | 140 | 313 |
| 64 | 2 | 4 | 8 | 584 | 42 | 420 | 4 | 424 |
| 64 | 2 | 4 | 8 | 91 | 30 | 90 | 4 | 94 |
| 64 | 2 | 8 | 4 | 305 | 38 | 222 | 2 | 224 |
| 64 | 4 | 4 | 4 | 574 | 44 | 416 | 6 | 442 |
| 128 | 4 | 8 | 4 | 590 | 44 | 427 | 1 | 428 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Size | associvity | blocksize | no. of sets | hits | misses | write hits | write misses | writes |
| 32 | 1 | 4 | 8 | 197 | 37 | 43 | 19 | 62 |
| 32 | 2 | 4 | 4 | 294 | 52 | 81 | 11 | 192 |
| 32 | 4 | 8 | 1 | 233 | 831 | 60 | 62 | 122 |
| 64 | 2 | 4 | 8 | 584 | 42 | 164 | 10 | 174 |
| 64 | 2 | 4 | 8 | 91 | 30 | 1 | 10 | 11 |
| 64 | 2 | 8 | 4 | 305 | 38 | 83 | 11 | 94 |
| 64 | 4 | 4 | 4 | 574 | 44 | 158 | 6 | 164 |
| 128 | 4 | 8 | 4 | 590 | 44 | 163 | 11 | 174 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Size | associvity | blocksize | no. of sets | hits | misses | victim hits | victim misses | L1 miss rate |
| 32 | 1 | 4 | 8 | 197 | 37 | 34 | 184 | 4.829268 |
| 32 | 2 | 4 | 4 | 294 | 52 | 146 | 72 | 5.285714 |
| 32 | 4 | 8 | 1 | 233 | 831 | 17 | 201 | 0.282479 |
| 64 | 2 | 4 | 8 | 584 | 42 | 146 | 72 | 12.73913 |
| 64 | 2 | 4 | 8 | 91 | 30 | 34 | 184 | 2.735294 |
| 64 | 2 | 8 | 4 | 305 | 38 | 150 | 68 | 6.673913 |
| 64 | 4 | 4 | 4 | 574 | 44 | 146 | 72 | 12.04167 |
| 128 | 4 | 8 | 4 | 590 | 44 | 150 | 68 | 11.42308 |

Graphs:

L1 miss rate is the y axis and associvity in the x axis.

Hits in y axis and associvity in x axis.

**Inferences:**

**Large block size to reduce miss rate:**

Changing the block size shows that, larger block size leads to lower miss rate. It is evident when containing more bytes in a block, the tag size and index size will decrease, so that address lines in trace file has more identical index and tag bits, which means higher hit rate and better performance. Table 1 shows results of various block sizes.

The miss ratio may be substantially improved by using slightly larger transfer blocks, in which case bus traffic does not increase greatly. Using larger address blocks reduces the cost of the tag memory considerably. It initially has only a minor effect on miss ratio, which is more than offset by the savings in writes due to the more efficient reservation of modified blocks.

**Large caches to reduce miss rate:**

It is obvious that larger cache size enables more address/data to be cached in caches, as a result hit rate and performance will increase due to increase in the average memory access time. Table 2 shows results of various number of blocks to verify this.

**Higher associativity to reduce miss rate:**

Increasing associativity boosts cache performance by relieving conflict miss. Comparing with Table 1, for one way set associative cache hit rate decreases when increasing associativity. Hit rate will be as high as 99.91% in 8 way and 16 way configurations.

**Limitations:**

1. L2 cache is a static entity, none of its parameters or memory accesses are measured.
2. Inputs have to perfectly divisible and scalable.
3. Number of data accesses in the tracefile should not be above 1000.
4. Is not linked to a dynamic tracefile, it is linked to a static text file that holds data from the trace file.
5. I have used visual studio to develop the code. Consider using the same while executing it.