Readme.pdf
Robert Henderson
TA: Binh Pham
CS211 Cache simulation

DESIGN

```
struct Line {
	int tag;
	char valid;
	char dirty;
	int timeStamp;/*marker of when used last*/
};
```

Each line in a set, has a tag (derived from the memory address). The data in that line is either valid or invalid (not yet initialized). A dirty marker is needed for the “wb” write policy since the modified data is not being written back to memory unless it has to (when that address gets kicked out).

```
typedef struct Line* LinePtr;/*array lines*/
```

Each Set can have many lines for assoc and nway assoc so we need to have an array of lines. I just typedef'd a Line* to be LinePtr.

```
struct Set {
	LinePtr* set;
	unsigned int nextIndex;/*used for FIFO replacement*/
};
```

Each set is an array of LinePtrs

```
typedef struct Set* SetPtr;/*array sets*/
An array of sets

typedef SetPtr* Cache;
```

Here, our cache is an array of sets

```
/*Contains information about the cache simulation*/
struct SimInfo {

	/*Input*/
	char* associativity;
	int blockSize;
	char* replacementAlgorithm;
	char* writePolicy;
	char* traceFile;
	int cacheSize;


	/*Calculated info*/
	int numberOfSets;
	int numberOfLinesPerSet;

	int cacheBits;/*needed for calculations*/
	int tagBits;
	int blockBits;/*needed for calc of the bits*/
	int setBits;
```

```
    int lineBits;


    /*Counters*/
    int cacheHits;
    int cacheMisses;
    int memoryReads;
    int memoryWrites;
    int count;


    /*Input from file pinfile*/
    unsigned int instructionLocation;
    unsigned int referencedLocation;
    char readOrWrite;

    /*Our cache*/
    Cache cache;

    int addressBits;

    int powerFlag;

};
```

The simInfo struct keeps track of all of the information needed for the simulation.

```
typedef struct SimInfo* SimInfoPtr;
```
This is just the address of a SimInfo struct

```
int SimInit(SimInfoPtr si, char* cache_size, char* assoc, char* line_size,
            char* replace_algo, char* write_policy, char* trace_file);
```
This initializes the simulation, assigning the inputs to the variables. Calls IsValidInput to checks for valid input, if invalid, then just exits with an error message.

```
int RunSim(char* cache_size, char* assoc, char* line_size, char* replace_algo,
            char* write_policy, char* trace_file);
```
This runs the simulation in a loop. This calls simInit. Opens up a file, reads in the mem address, calculates the tag and lineNumber (with the help of FindReplacementIndex.

```
int isValidInput(SimInfoPtr si);
```
Checks for valid input.

```
int LogBase2(SimInfoPtr si,int number,int flagNumber);
```
* Takes in a siminfo ptr, number of whiches power of 2 needs to be computed
* Calculates the power of 2 and also modifies the flag checking to see if num is power of 2

```
void ToLowerCase(char* myString);
```
Converts a string to uppercase using pointers to modify the original

```
int CalculateAndMallocIfValid(SimInfoPtr siPtr);
Calculates and mallocs if input is valid

void Free(SimInfoPtr si);
Frees space malloc'd for

void ReadWritePolicy(SimInfoPtr siPtr,int setNumber,int lineNumber,int tag);
 * Will modify the cache hits, cache misses, memory reads, and memory writes thru the use of pointers
 * Takes in a siminfo pointer, a set number calculated for a mem address, a line number determined by
 * FindReplacementIndex, and a tag number generated by the address
 *
 * Side effects: modifies cache hits, misses, memory reads, writes for a specific sim ptr
 * Returns nothing;void
 */

FILE* OpenFile();
Opens a pintrace file

void PrintResults(SimInfoPtr si);
prints the final tally of hits, misses, reads, and writes

int CheckForHelpAndArgCountError(char* cache_size,int argc);
1st check for input—invalid count or -h
Program wont even go into RunSim if -h or invalid argc
```

ALGORITHMS

fifo was implemented by if the memory address was not found (if the tag wasnt found in the set), the tag for the memory address was placed into the next index of the set (line) in a round robbin manner using a division mod algorihm.

Lru was implemented by using a timeStamp for that line, initialized to 0 and assigned the running count when accessed. The least recently used in a set would be the lowest numbered timeStamp in the set. If no match was found, the tag index with the lowest timeStamp would be ousted and the new data would be written over it.

wb was implemented with a dirty bit. If a 'W' was made to the memory address dirty would be '1'. If that line needs to be removed, we must first write that change back to memory. If no change is made, there is no need to write that back to memory.

Wt was simple. I 'W' was made to the memory address, a write would be made to memory immediately.

ANALYSIS

If I was designing a cache that only had L1, it would seem like a bad idea to make a direct map cache. A direct map is nice because it's simple, but a direct map cache would cause too many replacements if memory addresses that map to the same set(and line too since there is only 1 line per set) keep replacing each other. Too many Sets(or lines for that matter, since #lines = #sets) may be empty, not being utilized.

A fully associative cache would also seem like a bad idea, eventhough we would get

a lot of cache hits (resulting in fewer cache misses leading to fewer memory reads). The draw back to the associative is the overhead needed. In a fully associative cache, there is only one set and a memory address can get mapped to any of the lines in the 1 set. This needs a data structure (or linear search, which isn't very effective for a large size) to keep track of which memory addresses (or tags) are currently in the cache to check for a hit. Here, every line in the 1 set would be utilized to its potential, it would contain much overhead to check for a hit. As a result, a fully associative would be very expensive to implement.

A n-way associative cache is most appropriate. It seems to be a middle ground between the simple direct map and the dumb fully associative, where an address can map to any line in the 1 set, causing lots of overhead (either in the hardware or software, or both).

Int my n-way associative cache, A block_offset any bigger than 16 or 32 bytes would defeat the concept of spatial locality. Knowing that the typical cache size for a cpu is 512, I would probably keep my **cache size to 512 bytes**. In keeping the associativity constant, it seems as though a **block size of 16** would maximize the number of cache hits (see below).

./sim 512 assoc:4 32 fifo wb pinatrace.out
Cache Hits: 648282
Cache Misses: 94104
Memory Reads: 94104
Memory Writes: 36410

[**rhender@localhost sim]$ ./sim 512 assoc:4 16 fifo WB pinatrace.out**
./sim 512 assoc:4 16 fifo wb pinatrace.out
Cache Hits: 651520
Cache Misses: 90866
Memory Reads: 90866
Memory Writes: 42211

[rhender@localhost sim]$ ./sim 512 assoc:4 8 fifo WB pinatrace.out
./sim 512 assoc:4 8 fifo wb pinatrace.out
Cache Hits: 646614
Cache Misses: 95772
Memory Reads: 95772
Memory Writes: 47744

As the number of lines per set increases in the n-way associativity, the number of sets decreases. In keeping my cache size at 512 bytes and 16 for my block_Offset. I found that an **8-way associativity** maxmizes the number of cache hits.

[rhender@localhost sim]$ ./sim 512 assoc:4 16 fifo WB pinatrace.out
./sim 512 assoc:4 16 fifo wb pinatrace.out
Cache Hits: 651520
Cache Misses: 90866
Memory Reads: 90866
Memory Writes: 42211

[rhender@localhost sim]$ ./sim 512 assoc:8 16 fifo WB pinatrace.out
./sim 512 assoc:8 16 fifo wb pinatrace.out
Cache Hits: 653064
Cache Misses: 89322

Memory Reads: 89322
Memory Writes: 42515

```
[rhender@localhost sim]$ ./sim 512 assoc:16 16 fifo WB pinatrace.out
./sim 512 assoc:16 16 fifo wb pinatrace.out
Cache Hits: 652038
Cache Misses: 90348
Memory Reads: 90348
Memory Writes: 43135
```