/home/diana/Documents/Comp Org/Project/cache.c

```
1 #include "cache.h"
2
3
4 #include<stdlib.h>
5 #include<math.h>
6 #include<stdio.h>
7 #include <sys/ioctl.h>
8 #include <unistd.h>
9
10 //#define CACHE_DEBUG
11 //#define CACHE_POINTER_DEBUG
12 //#define CACHE_INITIALIZE_DEBUG
13
14 void updateLRU(struct Cache* cache, struct Block * tempBlock,
15       unsigned long long targetIndex);
16
17 int debugFlag = 0, pointerFlag = 0;
18
19 /********************** Initialization **********************/
20 struct Cache * initialize(int newCacheSize, int newBlockSize, int newAssociativity, int newMissTime, int newHitTime) {
21    struct Cache *cache;
22
23    // Create in memory
24    cache = (struct Cache*) malloc(sizeof (struct Cache));
25
26    // Initialize cache variable fields based on configuration files
27    cache->cacheSize = newCacheSize;
28    cache->blockSize = newBlockSize;
29    cache->associativity = newAssociativity;
30    cache->lengthOfWay = (newCacheSize / newBlockSize) / newAssociativity;
31
32    // log(base2)(lengthOfWay)
33    cache->indexFieldSize = log(cache->lengthOfWay) / log(2);
34
35    // log(base2)(newBlockSize)
36    cache->byteOffsetSize = log(newBlockSize) / log(2);
37    cache->missTime = newMissTime;
38    cache->hitTime = newHitTime;
39
40    // Initialize tracking variables to 0
41    cache->hits = 0;
42    cache->misses = 0;
43    cache->writeRefs = 0;
44    cache->readRefs = 0;
45    cache->insRefs = 0;
46    cache->instructionTime = 0;
47    cache->readTime = 0;
48    cache->writeTime = 0;
49    cache->flushTime = 0;
50    cache->transfers = 0;
51    cache->kickouts = 0;
52    cache->dirtyKickouts = 0;
53    cache->flushKickouts = 0;
54
55
56    // Initialize array of blocks
57    cache->blockArray = (struct Block **) malloc(sizeof (struct Block *) * (cache->lengthOfWay));
58
59    int i = 0, j = 0; // used for iteration
60    for (i = 0; i < cache->lengthOfWay; i++) {
61       // First column in blockArray will be a dummy pointer
62       // => add an extra "way"(column)
63       // Go through each row and allocate space for amount of blocks
64       cache->blockArray[i] = (struct Block *)
65             malloc(sizeof (struct Block) * (newAssociativity + 1));
66
67       // Go through and initialize blocks (columns)
68       for (j = 0; j < newAssociativity; j++) {
69
70          // Set valid, dirty, and tag fields
71          cache->blockArray[i][j].valid = 0;
72          cache->blockArray[i][j].dirty = 0;
```

```
73          cache->blockArray[i][j].tag = 0;
74
75          // Point to next block in LRU chain
76          cache->blockArray[i][j].nextBlock =
77                  &(cache->blockArray[i][j + 1]);
78      }
79
80      // Initialize last block to null
81      cache->blockArray[i][newAssociativity].nextBlock = NULL;
82      cache->blockArray[i][newAssociativity].valid = 0;
83      cache->blockArray[i][newAssociativity].dirty = 0;
84      cache->blockArray[i][newAssociativity].tag = 0;
85  }
86
87  // Done
88  return cache;
89 }
90 //
91 ///********************* Read Trace Function *************************/
92
93 unsigned long long moveBlock(struct Cache* cache, unsigned long long targetTag, unsigned long long targetIndex, int isDirty) {
94  cache->transfers++; // Transferring data into cache
95
96  struct Block *tempBlock; // Used for LRU policy implementation
97
98  // Point to the first block in the index
99  tempBlock = cache->blockArray[targetIndex][0].nextBlock;
100
101  while (tempBlock->nextBlock != NULL && tempBlock->valid) { // Find the LRU
102      tempBlock = tempBlock->nextBlock;
103  }
104
105  if ((!tempBlock->valid) || (!tempBlock->dirty)) {
106      if (debugFlag) {
107          printf("~~~> Block was invalid or clean: good to overwrite.\n");
108      }
109
110      if (tempBlock->valid && !(tempBlock->dirty)) {
111          cache->kickouts++; // Kicking out a previously valid, clean block
112      }
113
114      // Block is invalid or clean, feel free to write over it
115      tempBlock->tag = targetTag;
116      tempBlock->valid = 1;
117      tempBlock->dirty = 0;
118
119      // May have been written back due to a dirty/flush kick-out
120      if (isDirty) {
121          tempBlock->dirty = 1; // Needs to be written as dirty
122      }
123
124      cache->writeRefs++; // Seen as a write request
125      cache->writeTime += cache->hitTime;
126
127      // Update LRU chain
128      // Move block to front of LRU chain
129      if (cache->blockArray[targetIndex][0].nextBlock != tempBlock) {
130          updateLRU(cache, tempBlock, targetIndex);
131      }
132      return 0;
133  }
134
135  if (debugFlag) {
136      printf("~~~> Block was dirty, need to return overwritten tag...");
137  }
138
139  // If you made it to this point, you're pointing to the last block in chain
140  // All blocks before it would have been valid, and this block is dirty
141  unsigned long long returnedTag = tempBlock->tag; // track dirty tag
142  cache->dirtyKickouts++;
143  cache->writeRefs++;
144  cache->writeTime += cache->hitTime;
145
146
147  tempBlock->tag = targetTag; // Put in new targetTag into block
```

```
148     if (isDirty)
149         tempBlock->dirty = 1; // mark as clean
150     else
151         tempBlock->dirty = 0; // mark as clean
152 
153     // Update LRU chain
154     // Move block to front of LRU chain
155     if (cache->blockArray[targetIndex][0].nextBlock != tempBlock) {
156         updateLRU(cache, tempBlock, targetIndex);
157     }
158 
159     return returnedTag; // signal that there was a dirty kick out
160 }
161 
162 int scanCache(struct Cache* cache, unsigned long long targetTag, unsigned long long targetIndex, char op) {
163     struct Block * tempBlock; // Used for LRU policy implementation
164 
165     // Point to the first block in the index
166     tempBlock = cache->blockArray[targetIndex][0].nextBlock;
167 
168     while (tempBlock != NULL) {
169         if (tempBlock->valid) { // Only check blocks if they are valid
170             if (tempBlock->tag == targetTag) {
171                 // Found the tag! Increment hits
172                 cache->hits++;
173 
174                 if (op == 'W') {
175                     // valid targetTag found in cache, write operation
176                     // => mark block as dirty
177                     tempBlock->dirty = 1;
178 
179                     // increase write time by hit penalty
180                     cache->writeTime += cache->hitTime;
181                     cache->writeRefs++;
182                 } else if (op == 'R') { //  Read instructions
183                     // increase read time by hit penalty
184                     cache->readTime += cache->hitTime;
185                     cache->readRefs++;
186                 } else if (op == 'I') {
187                     // increase instruction time by hit penalty
188                     cache->instructionTime += cache->hitTime;
189                     cache->insRefs++;
190                 }
191 
192                 // Move block to front of LRU chain
193                 if (cache->blockArray[targetIndex][0].nextBlock != tempBlock) {
194                     updateLRU(cache, tempBlock, targetIndex);
195                 }
196 
197                 return 1;
198 
199             } else { // Tag didn't match
200                 tempBlock = tempBlock->nextBlock;
201             }
202         } else {
203             tempBlock = tempBlock->nextBlock;
204         }
205     }
206     // Target tag not found in cache
207     cache->misses++;
208 
209     // increase times by miss penalty
210     if (op == 'W') {
211         cache->writeTime += cache->missTime;
212         cache->writeRefs++;
213     } else if (op == 'R') {
214         cache->readTime += cache->missTime;
215         cache->readRefs++;
216     } else if (op == 'I') {
217         cache->instructionTime += cache->missTime;
218         cache->insRefs++;
219     }
220     return 0;
221 }
222 
```

```
223 // Purpose: move tempBlock to the front of the LRU chain with index targetIndex
224
225 void updateLRU(struct Cache* cache, struct Block * tempBlock,
226         unsigned long long targetIndex) {
227     // firstBlock points to the old start of the chain
228     struct Block * firstBlock = cache->blockArray[targetIndex][0].nextBlock;
229
230     // Put block at start of chain
231     cache->blockArray[targetIndex][0].nextBlock = tempBlock;
232
233     tempBlock = firstBlock;
234     while (tempBlock->nextBlock != cache->blockArray[targetIndex][0].nextBlock) {
235         // Find block that used to be before tempBlock
236         tempBlock = tempBlock->nextBlock;
237     }
238     // Point tempBlock at start of chain
239     struct Block * temp2 = cache->blockArray[targetIndex][0].nextBlock;
240     tempBlock->nextBlock = temp2->nextBlock; //
241     temp2->nextBlock = firstBlock;
242     return;
243 }
244
245 void printCacheStatus(struct Cache * cache, int * cacheIndexCounter) {
246     if (debugFlag) {
247         printf("************************************************************\n");
248         printf("Cache length: %d, index field size: %d, byte field size: %d\n",
249                 cache->lengthOfWay, cache->indexFieldSize, cache->byteOffsetSize);
250
251         printf("Number of hits: %llu, misses: %llu, kick-outs: %llu, dirty kick-outs: %llu\n",
252                 cache->hits, cache->misses, cache->kickouts, cache->dirtyKickouts);
253
254         printf("Number of flush kick-outs: %llu, invalidates: %llu, flush time: %llu\n",
255                 cache->flushKickouts, cache->invalidates, cache->flushTime);
256
257         printf("Reference Counts: writes = %llu, reads = %llu, instructions = %llu\n",
258                 cache->writeRefs, cache->readRefs, cache->insRefs);
259
260         printf("Times: writes = %llu, reads = %llu, instructions = %llu\n",
261                 cache->writeTime, cache->readTime, cache->instructionTime);
262
263         // Go through each row of the cache
264         printf("------ Current Row Status ------\n");
265     }
266     int i = 0, j = 0;
267     for (i = 0; i < cache->lengthOfWay; i++) {
268         if (cacheIndexCounter[i] == 1) {
269             printf("Index %llx:\t", (unsigned long long) i);
270             for (j = 1; j < (cache->associativity + 1); j++) {
271                 // Skip the first column (dummy pointer)
272                 if (cache->blockArray[i][j].valid != 0) // Block has a tag
273                     printf("| V: %d, D: %d Tag: \t%llx|",
274                         cache->blockArray[i][j].valid,
275                         cache->blockArray[i][j].dirty,
276                         cache->blockArray[i][j].tag);
277                 else // Block doesn't have a tag in it
278                     printf("| V: %d, D: %d Tag: - |",
279                         cache->blockArray[i][j].valid,
280                         cache->blockArray[i][j].dirty);
281             }
282             printf("\n");
283         }
284     }
285     printf("\n");
286
287     if (pointerFlag) {
288         printf("~~~> ---------- Pointer Contents ----------\n\n");
289         int row = 0, col = 0;
290         for (row = 0; row < cache->lengthOfWay; row++) {
291             if (cacheIndexCounter[row] == 1) {
292                 printf("~~~> Index: %llx", (unsigned long long) row);
293                 for (col = 0; col < cache->associativity + 1; col++) {
294                     struct Block* thisBlock = &cache->blockArray[row][col];
295                     printf("||This block:%p, points to %p||\t", thisBlock,
296                             thisBlock->nextBlock);
297                 }
```

```
298                 printf("\n");
299             }
300         }
301     }
302
303     if (debugFlag) {
304         printf("End of Cache Status\n\n");
305         printf("**********************************************************\n");
306     }
307 }
308
309 void freeCache(struct Cache * cache) {
310     int row;
311     // Free all allocated memory
312     for (row = 0; row < cache->lengthOfWay; row++) {
313         free(cache->blockArray[row]);
314     }
315     free(cache->blockArray);
316     free(cache);
317 }
318
319 void flushCaches(struct Cache * iCache, struct Cache * dCache,
320         struct Cache * l2Cache, int mainMemoryTime, int transferTime,
321         int busWidth) {
322     // Go through L1 block, flush dirty blocks to L2 Cache
323     struct Block * tempBlock;
324
325     // Used if L2 needs to have a tag pushed back
326     unsigned long long addressTemp, indexTemp, tagTemp;
327
328     int index;
329     if (debugFlag) {
330         printf("~~~> Starting to flush the iCache\n");
331     }
332     for (index = 0; index < iCache->lengthOfWay; index++) {
333         tempBlock = iCache->blockArray[index][0].nextBlock;
334         // Start at the beginning of LRU chain, go all the way through
335         while (tempBlock != NULL) {
336             // Found a valid block
337             if (tempBlock->valid) {
338                 tempBlock->valid = 0;
339                 tempBlock->tag = 0;
340
341                 iCache->invalidates++; // increment invalidate counter
342                 if (debugFlag) {
343                     printf("Block is now invalid, moving on\n");
344                 }
345
346                 // Check if it was also dirty
347                 if (tempBlock->dirty) {
348                     if (debugFlag) {
349                         printf("~~~> Block was dirty, incrementing iCache flush kickouts\n");
350                     }
351
352                     iCache->flushKickouts++;
353                     tempBlock->dirty = 0;
354
355                     // Get ready to write it back to the L2 Cache
356                     // Start with 0-filled variables
357                     tagTemp = 0;
358                     indexTemp = 0;
359                     addressTemp = 0;
360
361                     // Get the L1 tag and shift it over to the right spot
362                     tagTemp = tempBlock->tag;
363                     tagTemp = tagTemp << (iCache->indexFieldSize +
364                             iCache->byteOffsetSize);
365
366                     // Get the L1 index and shift it over to the right spot
367                     indexTemp = index;
368                     indexTemp = indexTemp << iCache->byteOffsetSize;
369
370                     // Combine for the original address, minus the byte size
371                     addressTemp = indexTemp + tagTemp;
372
```

```
373                 // Pull out the L2 index/tag
374                 indexTemp = (~0) << (l2Cache->indexFieldSize +
375                         l2Cache->byteOffsetSize);
376                 tagTemp = indexTemp;
377                 indexTemp = ~indexTemp;
378                 indexTemp = indexTemp & addressTemp;
379                 // Pull out the L2 index
380                 indexTemp = indexTemp >> iCache->byteOffsetSize;
381
382                 tagTemp = tagTemp & addressTemp;
383                 tagTemp = tagTemp >> (iCache->indexFieldSize +
384                         iCache->byteOffsetSize); // Pull out the L2 tag
385
386                 // Ready to write back to the L2 cache
387                 addressTemp = 0; // Reset to 0
388
389                 if (debugFlag) {
390                     printf("~~~> Writing dirtyBlock back to L2 cache.\n");
391                 }
392                 addressTemp = moveBlock(l2Cache, tagTemp, indexTemp, 1);
393
394                 // Move dirty block from L1->L2
395                 iCache->flushTime += transferTime *
396                         (iCache->blockSize / busWidth);
397
398                 if (addressTemp) {
399                     // Kicked out another block, add to the flush time
400                     if (debugFlag) printf("~~~> Kicked out another dirty block in L2 cache, writing it back to main memory");
401                     l2Cache->flushKickouts++; // Increment L2 flushKickouts
402                     iCache->flushTime += mainMemoryTime;
403                 }
404             }
405         }
406         tempBlock = tempBlock->nextBlock;
407     }
408 }
409
410 // Have completed flushing the iCache, repeat for the dCache
411 if (debugFlag) {
412     printf("~~~> FINISHED FLUSHING ICACHE!!! \n\n"
413             "~~~> Starting to flush the dCache\n");
414 }
415 for (index = 0; index < dCache->lengthOfWay; index++) {
416     tempBlock = dCache->blockArray[index][0].nextBlock;
417     // Start at the beginning of LRU chain and go all the way through
418     while (tempBlock != NULL) {
419         // Found a valid block
420         if (tempBlock->valid) {
421             tempBlock->valid = 0;
422             tempBlock->tag = 0;
423             dCache->invalidates++; // increment invalidate counter
424             if (debugFlag) {
425                 printf("Block is now invalid, moving on\n");
426             }
427
428             // Check to see if it was also dirty
429             if (tempBlock->dirty) {
430                 if (debugFlag) {
431                     printf("~~~> Block was dirty, incrementing dCache flush kickouts\n");
432                 }
433
434                 dCache->flushKickouts++;
435                 tempBlock->dirty = 0;
436
437                 // Get ready to write it back to the L2 Cache
438                 // Start with 0-filled variables
439                 tagTemp = 0;
440                 indexTemp = 0;
441                 addressTemp = 0;
442
443                 // Get the L1 tag and shift it over to the right spot
444                 tagTemp = tempBlock->tag;
445                 tagTemp = tagTemp << (dCache->indexFieldSize +
446                         dCache->byteOffsetSize);
447
```

```
448             // Get the L1 index and shift it over to the right spot
449             indexTemp = index;
450             indexTemp = indexTemp << dCache->byteOffsetSize;
451
452             // Combine for the original address, minus the byte size
453             addressTemp = indexTemp + tagTemp;
454
455             // Pull out the L2 index/tag
456             indexTemp = (~0) << (l2Cache->indexFieldSize +
457                     l2Cache->byteOffsetSize);
458             tagTemp = indexTemp;
459             indexTemp = ~indexTemp;
460             indexTemp = indexTemp & addressTemp;
461             // Pull out the L2 index
462             indexTemp = indexTemp >> l2Cache->byteOffsetSize;
463
464             tagTemp = tagTemp & addressTemp;
465             tagTemp = tagTemp >> (l2Cache->indexFieldSize +
466                     l2Cache->byteOffsetSize); // Pull out the L2 tag
467
468             // Ready to write back to the L2 cache
469             addressTemp = 0; // Reset to 0
470
471             if (debugFlag) {
472                 printf("~~~> Writing dirtyBlock back to L2 cache. Check to see if it's in there\n");
473             }
474             if (!scanCache(l2Cache, tagTemp, indexTemp, 'W')) {
475                 if (debugFlag) {
476                     printf("~~~> Block wasn't in the L2 Cache, Adding miss time (+%d).\n",
477                             l2Cache->missTime);
478                 }
479                 l2Cache->flushTime += l2Cache->missTime;
480                 // Desired index/tag is now in L2 as dirty
481                 addressTemp = moveBlock(l2Cache, tagTemp, indexTemp, 1);
482
483                 if (addressTemp) {
484                     // Kicked out another block, add to the flush time
485                     if (debugFlag) printf("~~~> Kicked out another dirty block in L2 cache, adding L2 -> main memory time(+%d)\n", mainMemoryTime);
486                     // Increment L2 flushKickouts
487                     l2Cache->flushKickouts++;
488                     l2Cache->flushTime += mainMemoryTime;
489                 }
490
491                 if (debugFlag) {
492                     printf("~~~> Bringing desired block into L2 Cache, adding main memory -> L2 time (+%d)\n",
493                             mainMemoryTime);
494                     printf("~~~> Adding L2 replay time (+%d)\n",
495                             l2Cache->hitTime);
496                 }
497                 l2Cache->flushTime += mainMemoryTime;
498                 l2Cache->flushTime += l2Cache->hitTime;
499             } else {
500                 if (debugFlag) {
501                     printf("~~~> Block was in the L2 Cache, Adding hit time (+%d).\n", l2Cache->hitTime);
502                 }
503                 l2Cache->flushTime += l2Cache->hitTime;
504             }
505             if (debugFlag)
506                 printf("Transferring dirty block from L1->L2, adding transfer time (+%d)\n",
507                     transferTime * (dCache->blockSize / busWidth));
508             // Move dirty block from L1->L2
509             dCache->flushTime += transferTime *
510                     (dCache->blockSize / busWidth);
511         }
512     }
513     tempBlock = tempBlock->nextBlock;
514   }
515 }
516
517 // Have completed flushing the dCache, repeat for the l2Cache
518 if (debugFlag) {
519   printf("~~~> FINISHED FLUSHING DCACHE!!! \n\n"
520         "~~~> Starting to flush the l2Cache\n");
521 }
522 for (index = 0; index < l2Cache->lengthOfWay; index++) {
```

```
523         tempBlock = l2Cache->blockArray[index][0].nextBlock;
524         // Start at the beginning of the LRU chain and go all the way through
525         while (tempBlock != NULL) {
526             // Found a valid block
527             if (tempBlock->valid) {
528                 tempBlock->valid = 0;
529                 tempBlock->tag = 0;
530                 l2Cache->invalidates++; // increment invalidate counter
531                 if (debugFlag) {
532                     printf("Block is now invalid, moving on\n");
533                 }
534 
535                 // Check to see if it was also dirty
536                 if (tempBlock->dirty) {
537                     if (debugFlag) {
538                         printf("~~~> Block was dirty, incrementing l2Cache flush kickouts\n");
539                     }
540 
541                     l2Cache->flushKickouts++;
542                     tempBlock->dirty = 0; // Reset to clean
543 
544                     // Write it back to main memory
545                     l2Cache->flushTime += mainMemoryTime;
546                 }
547             }
548             tempBlock = tempBlock->nextBlock;
549         }
550     }
551 
552     if (debugFlag) {
553         printf("~~~> L2 Cache flushed, all caches flushed, all blocks invalidated\n"
554                 "~~~> Completed flushCache function");
555     }
556 }
557 
558 void setDebugStatus(int status) {
559     debugFlag = status;
560     return;
561 }
562 
```