Elec/Comp 526
Spring 2018
Project 2

Overview

In this project we will implement and evaluate a **lockup-free cache**. Such a cache continues to field memory requests while a cache miss is being serviced. In contrast, our earlier design stalled while a missed cache line was being fetched from memory. If the cache only serves cache hits while waiting for the cache miss to complete, it is said to support **hit-under-miss.** If multiple misses can be outstanding at the cache it is said to support **miss-under-miss**.

The design here implements a **miss-under-miss lockup-free** cache. The number of outstanding memory requests will be limited only by the amount of available work (active threads with memory requests) and the size of the writeback queue. We model a multi-threaded processor that switches threads when the currently executing thread is blocked on a cache miss; the newly activated thread will keep the processor until it also encounters a cache miss. The memory controller keeps track of the thread associated with a miss and restarts the thread after the missed cache line has been read and installed in the cache.

Several subtle correctness and performance issues arise in this scenario. What if the running thread wants to access a block that has been recently evicted (you handled this in Project 1)? What if it makes a request for the same memory block as another waiting thread? Does it matter if one request is a read and the other is a write? What if two outstanding requests are to different memory blocks that occupy the same location in the cache? In this last scenario, a careless design can result in livelock: a block installed in the cache by one thread is immediately accessed by the block of the other thread, and the two threads keep evicting the other's block without making progress!!

Different subsystems can simultaneously access the cache – the processor threads looking up the cache and evicting blocks to be replaced, and the back-end memory controller handling completion of memory accesses. The design needs to make sure they work cooperatively and do cause destructive races or deadlock.

Simulation Model

The system is modeled by three types of Yacsim processes: (i) multiple application threads that run on the CPU; each thread makes a sequence of memory requests and performs some (simulated) computation between successive requests (Yacsim process: **processor**) (ii) memory controller that processes requests one-at-a-time from the queue of memory requests at the thread-cache interface (Yacsim process: **memorycontroller**), and (iii) a background memory writer program that writes back dirty blocks evicted from the cache to memory (Yacsim process: **memorywriter**).

Only one thread at a time can execute on the CPU; threads use the semaphore **sem_cpu** to serialize CPU access. A thread that gets the CPU continues execution until it encounters a cache miss that requires an access to main memory: at that time it releases the CPU. The cache miss will be served by the memory system and the thread woken up when the missed block has been installed in cache. At that time it needs to contend again for the CPU and continue.

To service a cache miss, the *memory controller* uses the function **ServiceMemoryRequest**. It first checks if the request can be satisfied from the **WRITEBACK_QUEUE.** If so, it is read from there with a delay **WRITE_BUFFER_SEARCH_DELAY**. Otherwise it needs to get exclusive access to the memory subsystem (using **sem_memory**) and simulate the memory read by delaying for an additional time **MEMORY_READ_TIME**. When the missed block is available, it should be *installed* in the cache.

Note that as part of a cache miss, the current block from the cache should also be *flushed* (written back to memory) if it is dirty; this is done by simply adding it to the tail of the **WRITE_BACK_QUEUE**. The write controller (**memorywriter**) process is signaled when a new request has been added to the queue.

An additional semaphore **sem_cacheaccess** is used to serialize accesses to the cache: only one entity (**processor**, **memorycontroller**, or **memorywriter**) should be accessing the cache at any time. An additional semaphore **sem_writebackQueue** is used to serialize accesses to the write-back queue. Also, **sem_writeQueueFull** is used to limit the number of entries in the write-back queue to **MAX_WRITE_BUFFER_SIZE**. Be sure not to create deadlocks and avoid livelock situations. The **memorywriter** process simulates the writing of the block at the head of the write-back queue to memory by delaying for **MEMORY_WRITE_TIME,** and then deletes the block from the write-back queue.

Workloads

To create memory traces a program **maketrace.c** is provided. You can edit the file to create different sequential traces. You can change the size of the trace by changing the constant TOTALSIZE in **global.h**. Also the source code can be easily modified to set the accesses to *read* or *write,* and to set the *starting address* of the generated trace.

The number of trace files (one per thread) is set using the **MAX_NUM_THREADS** constant in **global.h**. Currently it has been set up to handle 1 or 2 trace files only (respectively **memtrace0** or **memtrace1**). The range of memory addresses in each of these trace files do not overlap. The constant **CPU_DELAY** sets the number of cycles spent by an application thread computing between *successive memory requests*.

A total of four workloads (RA, RB, WA, WB) will be simulated. The "R workloads" consist of **all read requests** and "W workloads" consist of **all write requests**. The read workloads are generated by code in the **READ** directory and the write workloads from that in the **WRITE** directory. The only difference is a change of an argument from READ to WRITE as indicated in the comments in the body of the **dochunk( )** function in **maketrace.c**.

For each workload type (READ or WRITE), workloads **A** and **B** generate traces with different address ranges. The default setting (generates workload **A**) can be flipped by changing an argument in the call to **dochunk ( )** in the file **maketrace.c.** See the comment in the file on how to change the argument. Workload **B** will use the starting address **b** instead of **b+16**.

You will have a total of four experiments. All experiments involve two CPU threads. In each experiment you must record different performance metrics, plot them in various ways and discuss the results.

Assignment Details

1. **Preliminaries**
    a. The programs for the project can be obtained from the tar file **project2.tar.gz** on Canvas.
    b. Uncompress and untar the file.
    c. See file **README** in each of the subdirectories **READ** and **WRITE**

2. Four workloads will be used for the experiments. Two read workloads called **RA** and **RB** and two write workloads called **WA** and **WB** respectively.

3. To generate **RA** and **RB**: connect to the READ directory. Use the default settings and compile and execute **maketrace.c.** This should create two trace files **memtrac0** and **memtrace1** which comprise workload RA.

    a. gcc maketrace.c
    b. ./a.out

4. To create the RB workload edit **maketrace.c** and change the appropriate line in the source code from "**b+16**" to "**b**" as indicated in the comments in the file.

5. To create workloads **WA** and **WB** you must repeat steps 3 and 4 for the files in the **WRITE** directory

Details of Code and Experiments

1. READ directory (R Workloads)

Programming Part: Study the source code **read.c** so you understand the simulation.

Experimental Part
Default settings of **global.h**

Workload RA

1. Create the trace files for workload **RA** using **maketrace.c** provided in the **READ** directory.

2. Compile the provided simulation code files and run the executable **runme**

```
gcc read.c utils.o ./yacsim.o –lm –o runme
```

3. Vary **CPU_DELAY** between 0.0 and 100.0 in increments of 10 and record (or compute from the output) the following:

Total Simulation Time (time at which both threads complete)
Total Memory Read Time
Total Memory Write Time
Total CPU Busy Time
Total Memory Busy Time
Total Number of Hits
Total Number of Misses
Total Number of Evictions
Total Number of Write backs

Workload RB

1. Edit maketrace.c and change the starting address of the trace for memtrace1 from “b+16” to “b”. (See comment in maketrace.c).

2. Create the trace files for the workload **RB**.

3. Repeat steps 2 and 3 of Workload **RA** with the new trace files.

II. W workloads (from WRITE directory)

Programming Part

Study the source code in **write.c**. Compared to the read-only case above it will now be necessary to handle write back requests from dirty evicted cache blocks.[1] Note your solution should be able to handle workloads that are a mixture of reads and writes, and not restricted to the write-only workloads **WA** and **WB**.

In the code you will find a stub for a function **checkAndFlush(req)**. Its task is to check whether the cache block being replaced by the memory request **req** needs to be written back to memory. If so, it must use the provided function **FlushDirtyBlock()** to do the eviction. Detailed steps are specified by comments in the function.

(a) Complete the function **checkAndFlush();**
(b) Call the function **checkAndFlush()** from a suitable point in your code.

Complete the code and run the following experiments. A sample trace output for a trace size of 32 requests per thread (that is, with **TOTALSIZE** set to 32 in **global.h**) is provided in the file **SampleTrace** to help you check your design.

[1] If you go back and check you can see that the read-only implementation never invoked the thread **memorywriter** to do write backs, which remained blocked on the semaphore **sem_writeRequest** throughout the simulation!

Experimental Part

Default settings of global.h

Workload WA

For workload **WA**, repeat the steps performed for workload **RA**. Vary the delay from **0 to 300** cycles in increments of **20**.

Workload WB

For workload **WB**, repeat the steps performed for workload **RB**. Vary the delay from **0 to 300** cycles in increments of **20**.

EXPERIMENTAL DATA and REPORT

READ_ONLY WORKLOADS (I)

x Axis: Delay between **0 and 100** in **increments of 10**. Create plots R1 to R5.

R1: Total Simulation Time vs Delay for workloads RA and RB on the same plot
R2: Total Number of Misses vs Delay for workloads RA and RB on the same plot
R3: Total Memory Read Time vs Delay for workloads RA and RB on the same plot
R4: CPU utilization and Memory utilization for workload A on the same plot
R5: CPU utilization and Memory utilization for workload B on the same plot

WRITE_ONLY WORKLOADS (II)

x Axis: Delay between **0 and 300** in increments of **20**. Create plots W1 to W4.

W1: Total Simulation Time vs Delay for workloads WA and WB on the same plot
W2: Total Number of Misses vs Delay for workloads WA and WB on the same plot
W3: Total Memory Read Time vs Delay for workloads WA and WB on the same plot
W4: Total Memory Write Time vs Delay for workloads WA and WB on the same plot

Discussion

For each plot provide explanations (precise and as quantitative as possible) to analyze the shapes of the curves and their values. Answer the following questions:

1. Compare the total time for workloads A and B in Plots R1. Explain why the curve for RA is flat and then rises smoothly. Also explain the curve for RB shows a staircase like pattern.

2. Explain the relation between plots R2 and R3.

3. Compare plot R2 against the utilizations of plots R3 and R4. Predict what would be the CPU utilization and Memory utilization for both RA and RB as the delay parameter grows very large.

4. Explain why the relation between plots W2 and W3 is different from the relationship you noted between R2 and R3 in Q2 above. Use the relation between in plots W2 and W4 to justify your argument. Also use some other statistic from the output to strengthen your argument.

5. Suppose the two threads were run sequentially on the CPU. Estimate the total running time for CPU DELAY of 100 for WA workload. Compare this with the time needed by the concurrent lockup-free design for both WA and WB.

Submission on Canvas:

Submit (i) the modified source code for the Write Only Workloads (ii) PDF report including plots and discussions

Extra Credit (Optional)

1. Based on the simulation results, describe one *specific* optimization that you feel may improve performance. Justify your suggestion. Do not give a list of possible optimization options but choose the one that you think may be most helpful and argue your case. (up to 5 points)

2. Implement your optimization and show empirical results showing the benefit in at least one of the scenarios of this project. Submit the report and the modified code along with instructions to run it. (up to 10 points depending on the benefits and difficulty of implementation)