Elec/Comp 526
Spring 2018
Project 1

Overview

This project deals with implementing and evaluating a simple optimization for a uniprocessor cache using Yacsim-based simulation. You are provided the complete code for a baseline simulation model that does **synchronous** reads and **synchronous** writes to memory. You must first run the program for different parameter values and plot and explain the results. Then you must extend the model to handle **asynchronous** writes, perform simulation tests and report and analyze the results. It may be worth the effort to think through how to automate the process of doing parameter sweeps and generating outputs that feed directly to a plotting program (almost all the projects will have this requirement and may save you time in the long run).

Cache Model

The baseline implementation is that of a **direct-mapped**, **write back** processor cache exercised by a single CPU thread. Memory requests generated by the processor first look up the cache to check whether it holds the requested memory word. If it does (**cache hit**) the requested word from the cache block is returned to the CPU thread; if the access is a **write** request, the **dirty** bit (D) in the cache must be set and the appropriate bytes of the cache block updated. If the requested word is not in the cache, a **miss** occurs and the block needs to be fetched from main memory. The block currently in that cache location is evicted to make space for the new block. If the evicted block is dirty it must be **written back** to memory. In the base model, the processor **stalls** while it is waiting for the block to be fetched from memory. When the memory controller receives the requested block from memory, it is installed in the cache, and the waiting thread is awakened. The thread tries its request again and should normally have a hit this time around.

In the baseline implementation (**synchronous read write back** and **synchronous read**) the processor stalls while a dirty block is written back to the memory and the missing block is read from the memory. See the code in **sync.c** that models this system. The time to look up the cache (either a hit or miss) is 1 cycle and the time to read or write a block to memory is (optimistically) set at 50 cycles. See **global.h** for the parameters. The default cache size is set to 2 blocks to force evictions even with small trace sizes. You can change it by setting the constant **CACHESIZE**. However, use size 2 for the experiments unless otherwise stated in the instructions.

In an improved design (**async_template.c**) the writing back of dirty blocks is done by a separate background thread when the memory is idle. The aim is to *overlap* the writing back to memory with continuing CPU activity by the thread. When a dirty block needs to be written back it is placed in a **Writeback Queue** -- a first-come-first-served (FCFS) queue of dirty blocks that need to be flushed to memory, and is written to memory asynchronously by the background thread.

Workload Model

For this experiment we use a simple sequential trace of memory requests. The trace consists of a sequence of consecutive word addresses; addresses are 4 bytes long. The trace file is created by compiling and executing **maketrace.c**. By default it will create one trace file called **memtrace0** of 8192 write requests. You can change the constant **TOTALSIZE** in **global.h** to create different-sized traces. Think of the workload as generated by a loop that updates consecutive elements of an array of 8192 integers. On a cache miss, a cache line of **BLKSIZE** bytes is fetched from memory. The default **BLKSIZE** (in file **global.h**) is 32 bytes.

Assignment

1. **Preliminaries**
    a. The supporting project code is available as **project1.tar.gz** on **Canvas**
    b. You can use your own accounts on CLEAR (login remotely using **ssh ssh.clear.rice.edu –l <netid>**) to perform your assignment. It may also be possible to use a 64-bit Linux machine (physical or VM) as your platform.
    c. Copy the tarball into your own working directory
    d. Uncompress the file using **gunzip project1.tar.gz**
    e. Use **tar -xvf project1.tar** to create the **Project1** directory
    f. See file **README** for a description of the other files.in the directory

2. Create a trace file by compiling and executing **maketrace.c**
    a. gcc maketrace.c
    b. ./a.out

Part 1A: Synchronous Writes and Reads (using file **sync.c**)

1. Compile the baseline synchronous simulation model
    gcc sync.c utils.o ./yacsim.o -lm -o runme
    This creates the executable file **runme** which you can execute as **./runme**

2. Analyze the output and relate it to the system parameters. Change system parameters like **CPU_DELAY** (the number of cycles between consecutive memory requests made by the thread) and see if you can correctly predict the execution time. You can also change the *size of the trace file* to create different-sized traces. You can change the trace to be *all read* requests by simply changing the argument **WRITE** to **READ** in the function call in **dochunk** of **maketrace.c**. This system is entirely predictable and you should get familiar with its behavior. You can also change the **CACHESIZE** and the **BLKSIZE** parameters and check your understanding. The **NUMWAYS** parameter *should not be changed*, so it is always a direct-mapped (1-way associative) cache organization.

3. You can get a detailed trace of events by setting the **TRACE** flag in **global.h** to **TRUE**.

4. Study the code in **sync.c**. On a cache miss the processor stalls until the access is complete. If a dirty block is evicted it is first written to memory before the read is initiated. What is time required for a cache hit? For a miss? The parameter **CPU_DELAY** simulates the time spent by the CPU in computing between memory requests. By default it is set at 0.0 meaning memory requests occur back-to-back; by increasing its value (1.0, 2.0 etc) the number of cycles between consecutive requests can be increased simulating a more compute-intensive workload.

EXPERIMENTS

Part A: Synchronous Writes

- Start with all default values of **global.h. CACHESIZE=2** and **BLKSIZE = 32**

- **Step S1**: Vary **CPU_DELAY** from **0.0 to 10.0** cycles (in steps of 1.0). Record the total simulation time and memory idle time for a trace of 8192 writes (default values) for each delay value. Graph the memory utilization versus delay in **plot 1**.

- **Step S2**: Set **CACHESIZE = 2048** and **BLKSIZE = 32 bytes**. Repeat part S1 above. Graph the memory utilization on the same plot as step S1.

- **REPORT**: (**i**) Explain the individual and results of S1 and S2 and comparison between them. (**ii**) What would be the asymptotic value for the memory utilization as the CPU_DELAY continues to increase indefinitely. (**iii**) How would the plot change if the workload is all read requests rather than all write requests?

Part B: Asynchronous Writes (using file **async_template.c**)

1. Understand the structure of the simulation model implemented in **async_template.c**. The idea is to have a separate writeback thread to flush the dirty blocks to memory. This thread is implemented as a Yacsim process; dirty blocks evicted from the cache are placed in a FIFO queue and the writeback thread writes them to memory.

2. You need to complete the code in two functions: **FlushDirtyBlock** and **MemoryRead**. Comments in the code will help you along. Try and get a global view of the operation of the system and not simply use a peephole approach based on the provided comments! Compile, execute and debug till satisfied. Use small files and set TRACE to TRUE to see the detailed execution.

EXPERIMENTS (Using file **async_template.c with your completed changes**)

- **Steps A1, A2**: Repeat the parts S1 and S2 that you previously did for the synchronous implementation with asynchronous writeback. Graph the total execution times of A1 and S1 together in **plot 2** and in **plot 3** graph the total execution times of A2 and S2. There is no need to plot memory utilizations.

- **Step A3**: Set all parameters to their default (initial) values. Vary **BLKSIZE** through the following values: **16, 32, 64, 128, 256** bytes. For each **BLKSIZE** graph the **Execution Time** against **CPU_DELAY** (from 0.0 to 10.0 in steps of 1.0). All 5 graphs should be on the same plot called **plot 4**.

- **REPORT: Comment on the graphs** – explain clearly their individual shapes as delay increases and how the shape of **plot 4** changes with **BLKSIZE**.

SUBMISSION: Submit (i) a total of four plots, (ii) brief report with complete and precise discussions, and (iii) the source code for your completed **async_template.c** on CANVAS by the due date.