SANTA CLARA UNIVERSITY

Multi-level Caching Lab

Rick Sullivan

December 10, 2014

1 INTRODUCTION

To justify the logic behind my generated input file, I will use the GNU Image Manipulation Program (GIMP) as an example. When running in Windows 8.1, GIMP uses about 34 MB of memory when only open, without any editor windows or images loaded. Opening an image of about 2 MB, the memory used by GIMP increases to just under 37 MB - an increase of 3 MB. Opening any of the modular toolbox windows increases memory usage by about another 2 MB.
A single page in memory is typically around 4096 bytes (tested on my system using `getconf PAGESIZE`). Therefore, GIMP's 34 MB usage will request data from about 8,300 pages of memory. These pages will likely be physically close in memory, as they are all part of the foundation of GIMP. They will be requested throughout the entire time the program is being used. Loading a 2 MB image would require loading approximately 500 pages from memory. This data is likely stored in a completely different location than the GIMP core. Opening a 2 MB toolbar would similarly require 500 pages. These would logically be close to the GIMP core code.

1.1 WORKLOAD GENERATION

To mimic a real world program like GIMP, I will create an input file that requests a 'program core' more often than any other memory pages. Using the requested total number of pages requested of 10,000, I will assume that the most often requested part of our program is the first 1,000 pages. The 'program' being run also has 2,000 more pages that make up the program itself, which take up the next 2,000 pages. The workload also makes periodic requests to a chunk of external memory: it makes 100,000 consecutive requests for a page in a 1,000 page range in the remaining 7,000 pages. These requests mimic locality of data accessed from

		Second-level cache			
		LRU	LFU	Sec. Chance	Random
First-level cache	LRU	77.8%	73.2%	77.4%	72.6%
	LFU	72.0%	92.0%	72.2%	72.2%
	Sec. Chance	79.7%	73.4%	79.7%	73.0%
	Random	72.6%	73.3%	72.4%	79.8%

Figure 2.1: Miss rates for two-level cache combinations

loading a large external file. In between each of these large external reads, the 'program' accesses the location of its code as well.
The script for generating this input file can be seen in section 3.1.

2 Analysis

I tested each combination of two-level caches assuming a combined cache size of 500 pages. The script for automating these tests can be found in section 3.2, with the output in section 3.3. I tested combinations of four algorithms: LRU, LFU, Second Chance, and a purely random page replacement algorithm. The miss rates for each two-level combination can be seen in figure 2.1. For the sake of brevity in this report, I will focus on analyzing the most successful combination of caches: LFU as a first cache, and LRU as the second cache.
LFU attempts to keep any pages that are accessed many times in memory as long as possible. With it as the first cache, it manages to keep many pages from the 'core' of my simulated programs cached. This is because my simulated page requests begin with many requests for pages in the 'program core.' Between large external memory accesses, my workload returns to request these same pages. LFU likely has kept many of those pages loaded.

2.1 LFU's approach

Interestingly, LFU is the least accurate algorithm when considered on its own. It only has an 8% hit rate when used as a single 250-page cache, which increases to 16% when the size is doubled. As mentioned above, this is likely because it loads many pages at the beginning, yet never releases them from memory. Therefore, it will never allow any of the externally loading pages to be cached, and will miss nearly every request during these periods. Its single advantage is being able to keep some of the program core in memory.

2.2 LRU's approach

LRU fares much better on its own. At a cache size of 250 pages, it has a hit rate of 20%, and reaches a 40% hit rate at 500 pages. LRU, unlike LFU, is able to better adapt to large external memory accesses. When the program loads many pages from a new range of memory, LRU will flush all of the program execution pages from cache, allowing pages from this new range to be kept in memory. However, when the program returns to request its core program memory, LRU will have replaced every page, resulting in many misses.

2.3 Combination results

In some ways, LFU and LRU take nearly opposite approaches; LFU tries to hold on to a page that used to be accessed many times in the hope that it is accessed again, while LRU tries to pay attention to the location of memory the program is currently accessing. These approaches complement each other when used in tandem. LFU holds much of the program core in memory throughout execution, while LRU supports caching of external areas of memory.

This combination of LFU and LRU resulted in a 28% hit rate with each level of cache having 250 pages in memory. Using LRU as the first-level cache and LFU as the second-level also resulted in decent results, with a hit rate of 26.8%. This is slightly less ideal for the workload we have generated, because LFU will not have the ability to immediately start caching the pages that are used most frequently. It would have to wait for spillover from the LRU cache.

2.4 Notes on experiment accuracy

It is worth noting that using a purely random algorithm as either of the caches led to some of the most accurate test results. This is likely because my generated workload is not completely indicative of a real-world program. While the combination of LFU and LRU worked for this generated workload, results could be very different using an actual program.

3 Appendix

3.1 Input generation

```
import sys
import random
import math

NUM_INPUT_LINES = 1000000
PAGE_RANGE = 10000

FREQUENTLY_REQUESTED = int(PAGE_RANGE/10)
FREQUENTLY_RATE = 80 # 80%
RARELY_REQUESTED = int(2 * (PAGE_RANGE/10))

EXTERNAL_READ_SIZE = int(PAGE_RANGE/10)

def getProgramPage():
    r = random.randint(0, 100)
    if r < FREQUENTLY_RATE:
        pageNum = random.randint(0, FREQUENTLY_REQUESTED)
    else:
        pageNum = random.randint(FREQUENTLY_REQUESTED,
            FREQUENTLY_REQUESTED + RARELY_REQUESTED)
    return pageNum

def getExternalPage(externalReadCount):
    offset = FREQUENTLY_REQUESTED + RARELY_REQUESTED
    previousReads = EXTERNAL_READ_SIZE * externalReadCount
    return random.randint(offset + previousReads, offset + previousReads
        + EXTERNAL_READ_SIZE)

def writeInputFiles(numInputLines=NUM_INPUT_LINES, pageRange=PAGE_RANGE)
    :
    # Input file with page requests mimicking a real program.
    f = open("workload.txt", 'w')
    externalRead = False
    externalReadCount = 0
    for i in range(0, numInputLines):
        if i % (int(numInputLines/14) * 2) == 0:
            externalRead = False
            externalReadCount += 1
        # Seven times, do an external read
        elif i % int(numInputLines/14) == 0:
            # Switch to an external read of 100000 pages
            externalRead = True

        if externalRead:
            pageNum = getExternalPage(externalReadCount)
        else:
            pageNum = getProgramPage()

        f.write(str(pageNum) + '\n')

writeInputFiles(NUM_INPUT_LINES, PAGE_RANGE)
```

3.2 Test automation

```
import os
import subprocess

from createInputs import *

progs = ['lru', 'lfu', 'secondChance', 'random']
infile = 'workload.txt'
infile = 'cat␣' + infile + '␣'
linecount = '␣wc␣-l␣'

# Print number of lines in input
cmd = infile + '␣|␣' + linecount
print cmd
p = subprocess.Popen(cmd, shell=True)
p.communicate()

for i in range(len(progs)):
    progA = progs[i]
    cmds = [infile + '␣|␣./' + progA + '␣250␣|␣' + linecount]
    for j in range(len(progs)):
        progB = progs[j]
        cmds.append(infile + '␣|␣./' + progA + '␣250␣|␣./' + progB + '␣
            250␣|␣' + linecount)

    cmds.append(infile + '␣|␣./' + progA + '␣500␣|␣' + linecount)

    for cmd in cmds:
        print(cmd)
        p = subprocess.Popen(cmd, shell=True)
        p.communicate()
```

3.3 TEST OUTPUT

```
cat workload.txt | wc -l
1000000
cat workload.txt | ./lru 250 | wc -l
796063
cat workload.txt | ./lru 250 | ./lru 250 | wc -l
777819
cat workload.txt | ./lru 250 | ./lfu 250 | wc -l
731611
cat workload.txt | ./lru 250 | ./secondChance 250 | wc -l
773692
cat workload.txt | ./lru 250 | ./random 250 | wc -l
726302
cat workload.txt | ./lru 500 | wc -l
595959
cat workload.txt | ./lfu 250 | wc -l
919925
cat workload.txt | ./lfu 250 | ./lru 250 | wc -l
720134
cat workload.txt | ./lfu 250 | ./lfu 250 | wc -l
919925
cat workload.txt | ./lfu 250 | ./secondChance 250 | wc -l
721846
cat workload.txt | ./lfu 250 | ./random 250 | wc -l
721969
cat workload.txt | ./lfu 500 | wc -l
844745
cat workload.txt | ./secondChance 250 | wc -l
797346
cat workload.txt | ./secondChance 250 | ./lru 250 | wc -l
797346
cat workload.txt | ./secondChance 250 | ./lfu 250 | wc -l
732608
cat workload.txt | ./secondChance 250 | ./secondChance 250 | wc -l
797346
cat workload.txt | ./secondChance 250 | ./random 250 | wc -l
729732
cat workload.txt | ./secondChance 500 | wc -l
601501
cat workload.txt | ./random 250 | wc -l
797886
cat workload.txt | ./random 250 | ./lru 250 | wc -l
726371
```

```
cat workload.txt  | ./random 250 | ./lfu 250 |  wc -l
733034
cat workload.txt  | ./random 250 | ./secondChance 250 |  wc -l
724516
cat workload.txt  | ./random 250 | ./random 250 |  wc -l
797933
cat workload.txt  | ./random 500 |  wc -l
603236
```

3.4 Page replacement algorithm source code

This appendix section contains the sources used for implementing each page replacement algorithm. Each algorithm uses the Page class, which is shown in this section. Simple STL data structures were used to implement each algorithm. Stack and array-based algorithms were implemented using std::vector, while queue-based algorithms were implemented using std::deque.

3.4.1 Page.h

```cpp
#ifndef __Page_H
#define __Page_H

#include <vector>

class Page {
public:
    int id;
    bool referenced;
    int counter;

    Page();
    Page(int id);
    Page &operator=(const Page &other);
};

/* Naively implementing table as an array for simplicity. */
bool tableContainsPage(const std::vector<Page> &pages, const int pageID)
    ;

/* Used for debugging. */
void printPages(const std::vector<Page> &pages);

#endif
```

3.4.2 Page.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include <cstdlib>

#include "Page.h"

/* Simple class to encapsulate page IDs and referenced bits. */
Page::Page() {
    /* Invalid page ID. */
    this->id = -1;
    /* Invalid counter. */
    this->counter = -1;
    this->referenced = false;
}

Page::Page(int id) {
    this->id = id;
    this->counter = 0;
    this->referenced = true;
}

Page& Page::operator=(const Page &other) {
    if (this == &other)
        return *this;

    this->id = other.id;
    this->counter = other.counter;
    this->referenced = other.referenced;

    return *this;
}

/* Naively implementing table as an array for simplicity. */
bool tableContainsPage(const std::vector<Page> &pages, const int pageID)
     {
    for (int j = 0; j < pages.size(); j++) {
        if (pages[j].id == pageID)
            return true;
    }

    return false;
}
```

```
/* Used for debugging. */
void printPages(const std::vector<Page> &pages) {
    std::cout << "Pages:\t";
    for (int i = 0; i < pages.size(); i++) {
        std::cout << pages[i].id << '\t';
    }
    std::cout << std::endl;
}
```

3.4.3 Least Recently Used

```
#include <iostream>
#include <string>
#include <deque>
#include <cstdlib>

#include "Page.h"

/* Simulates the least recently used page replacement algorithm.
 * Uses a deque to maintain the queue of pages, with the most
 * recently used page at the front of the deque.
 * */
void leastRecentlyUsed(int numPFrames) {
    std::deque<Page> pages(numPFrames);

    int pageID;
    while(std::cin >> pageID) {
        bool found = false;
        for (int i = 0; i < pages.size(); i++) {
            if (pages[i].id == pageID) {
                /* If we found the page, move it to the front of the
                   queue. */
                pages.erase(pages.begin() + i);
                Page newPage(pageID);
                pages.push_front(newPage);
                found = true;
                break;
            }
        }

        if (found)
            continue;

        /* Else we have a page fault. */
        std::cout << pageID << std::endl;

        /* If the page table is full, get rid of the last page. */
        if (pages.size() == numPFrames)
            pages.pop_back();

        /* Add to front of queue. */
        Page newPage(pageID);
        pages.push_front(newPage);

        //std::cout << "Adding page " << newPage.id << std::endl;
        //printPages(pages);
    }
}

int main(int argc, char **argv) {
    /* Should be given a single command-line argument for the number of
       available page frames. */
    if (argc != 2) {
        std::cout << "Provide␣a␣single␣command-line␣argument␣for␣the␣
            number␣of␣page␣frames␣available." << std::endl;
        return 0;
    }

    int numPFrames = atoi(argv[1]);

    leastRecentlyUsed(numPFrames);
}
```

3.4.4 Least Frequently Used

```
#include <iostream>
#include <string>
#include <deque>
#include <cstdlib>
#include <climits>

#include "Page.h"

/* Checks for referenced pages and increments counters of
    referenced pages. */
void pageTableScan(std::vector<Page> &pages) {
    for (int i = 0; i < pages.size(); i++) {
        if (pages[i].referenced) {
                pages[i].referenced = false;
                pages[i].counter++;
            }
        }
}

/* Simulates the least frequently used page replacement algorithm.
 * Uses a vector to maintain the unordered list of pages.
 * */
void leastFrequentlyUsed(int numPFrames) {
    std::vector<Page> pages(numPFrames);

    int pageID;
    while(std::cin >> pageID) {
        bool found = false;
        for (int i = 0; i < pages.size(); i++) {
            if (pages[i].id == pageID) {
                /* If we found the page, set it as referenced. */
                pages[i].referenced = true;
                found = true;
            }
        }

        if (found)
            continue;

        /* Else we have a page fault. */
        std::cout << pageID << std::endl;

        pageTableScan(pages);

        Page newPage(pageID);
        /* If the page table is full, get rid of the LFU page. */
        if (pages.size() == numPFrames) {
            int minCount = INT_MAX;
            int minIndex = 0;
            for (int i = 0; i < pages.size(); i++) {
                if (pages[i].counter < minCount) {
                    minCount = pages[i].counter;
                    minIndex = i;
                }
            }

            pages[minIndex] = newPage;
        } else {
            /* Table is not full. Just add page. */
            pages.push_back(newPage);
        }

        //std::cout << "Adding page " << newPage.id << std::endl;
        //printPages(pages);
    }
}

int main(int argc, char **argv) {
    /* Should be given a single command-line argument for the number of
        available page frames. */
    if (argc != 2) {
        std::cout << "Provide␣a␣single␣command-line␣argument␣for␣the␣
            number␣of␣page␣frames␣available." << std::endl;
        return 0;
    }

    int numPFrames = atoi(argv[1]);

    leastFrequentlyUsed(numPFrames);
}
```

3.4.5 Second Chance

```
#include <iostream>
#include <string>
#include <vector>
#include <cstdlib>

#include "Page.h"

/* Simulates the second-change page replacement algorithm.
 * Uses a vector as a circular queue.
 * */
void secondChance(int numPFrames) {
    std::vector<Page> pages(numPFrames);

    int pageID, i = 0;
    while(std::cin >> pageID) {
        if (tableContainsPage(pages, pageID))
            continue;
        /* Else we have a page fault, so we want to replace a page. */
        std::cout << pageID << std::endl;

        while(pages[i].referenced == true) {
            /* Set referenced value to false. */
            pages[i].referenced = false;
            i = (i+1) % numPFrames;
        }

        /* Iterator is pointing to a non-referenced page. */
        Page newPage(pageID);
        pages[i] = newPage;
        //std::cout << "Adding page " << newPage.id << std::endl;
        //printPages(pages);
    }
}

int main(int argc, char **argv) {
    /* Should be given a single command-line argument for the number of
       available page frames. */
    if (argc != 2) {
        std::cout << "Provide␣a␣single␣command-line␣argument␣for␣the␣
           number␣of␣page␣frames␣available." << std::endl;
        return 0;
    }

    int numPFrames = atoi(argv[1]);

    secondChance(numPFrames);
}
```

3.4.6 Random

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>

#include "Page.h"

/* Simulates a random page replacement algorithm.
 * Uses a vector to maintain the unordered list of pages.
 * */
void random(int numPFrames) {
    std::vector<Page> pages(numPFrames);

    int pageID;
    while(std::cin >> pageID) {
        /* Do nothing if page is found. */
        if (tableContainsPage(pages, pageID))
            continue;

        /* Else we have a page fault. */
        std::cout << pageID << std::endl;

        Page newPage(pageID);
        int i = rand() % numPFrames;
        pages[i] = newPage;

        //std::cout << "Adding page " << newPage.id << std::endl;
        //printPages(pages);
    }
}

int main(int argc, char **argv) {
    srand(time(NULL));
    /* Should be given a single command-line argument for the number of
       available page frames. */
    if (argc != 2) {
        std::cout << "Provide␣a␣single␣command-line␣argument␣for␣the␣
           number␣of␣page␣frames␣available." << std::endl;
        return 0;
    }

    int numPFrames = atoi(argv[1]);

    random(numPFrames);
}
```