© 2017 Suraj Venkat

DEVELOPMENT AND EVALUATION OF GRAPHICAL USER INTERFACE AND BENCHMARK CREATION FOR CACHE MANAGEMENT ON MULTI-CORE SYSTEMS

BY

SURAJ VENKAT

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Adviser:

Professor Marco Caccamo

ABSTRACT

There is a constant need to improve processor performance on any system. It is vital to be able to visualize performance owing to a caching strategy and to use custom benchmarks to study the changes in performance. This thesis details the development and the evaluation of a graphical user interface to study the performance of a caching strategy detailed in [1]. Further, the process of creating synthetic benchmarks as well as integrating existing benchmarks to run with the tool are detailed. This thesis also proposes a synchronization mechanism which identifies the core carrying out 'prefetch and lock' of this framework and stalls other online and present cores temporarily during this phase of 'colored lockdown', to provide support on systems in which 'lockdown by master' is not available.

To my family,friends and teachers.

ACKNOWLEDGMENTS

I would like to thank a number of people who have encouraged and supported me throughout my education. I would like to thank my father, Mr.Hemant Kumar Sinha and my mother, Mrs.Seema Sinha for placing a high value on education and being very encouraging in all my academic endeavors.

I would like to thank Dr.Marco Caccamo, who is my adviser for the thesis for giving me the opportunity to work with him and for guiding me. I have been working with Renato Mancuso, a brilliant computer scientist who I have learned much from. A lot of the code in the base was built by him. A big thank you to Renato for all his support and help.

I would also like to thank Dr.Lui Sha who has been a mentor in my education and encouraged and guided me throughout my masters degree.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1
1.1 Motivation 1
1.2 Requirements 3
1.3 Overview 3

CHAPTER 2 BACKGROUND 5
2.1 Importance of Cache 5
2.2 Multi-level Caching 6
2.3 Existing Framework-Colored Lockdown 7

CHAPTER 3 INTERFACING WITH EMBEDDED BOARD AND USING L-2 PROFILER 9
3.1 Specifications of Embedded Board 9
3.2 Interfacing with Embedded Board 10
3.3 Using L-2 Profiler 12

CHAPTER 4 SYNTHETIC AND CUSTOM BENCHMARKS 15
4.1 Compiling and Running Synthetic Benchmark 15
4.2 Process of Compiling and Running a Custom Benchmark 24

CHAPTER 5 SUPPORT FOR SYSTEMS WITH LOCKDOWN BY WAY 29
5.1 Basis 29
5.2 Process of Synchronizing Cores during Prefetch and Lock 30
5.3 Colored Lockdown Performance before and after Synchronizing Cores during Prefetch and Lock 35

CHAPTER 6 DESIGN AND DEVELOPMENT OF THE GRAPHICAL USER INTERFACE 37
6.1 Design 37
6.2 Development 41

CHAPTER 7 USER EXPERIENCE STUDY AND EVALUATION OF GUI 49
7.1 User Experience Study 49
7.2 Evaluation 52

CHAPTER 8 RELATED WORK 55

CHAPTER 9 CONCLUSION 56

REFERENCES 57

APPENDIX A: IRB APPROVAL FORM FOR RECRUITING HUMAN PARTICIPANTS 59

CHAPTER 1

INTRODUCTION

A growing number of embedded systems are employing multi-core architectures. It is becoming increasingly important to present concepts of multi-core systems in academia. This thesis details the development and evaluation of a user friendly Graphical User Interface (GUI) for use in an educational setting. By GUI, we indicate graphical user interface. This GUI can be used to study the framework [1] with custom or synthetic benchmarks.

1.1 Motivation

Any program running on a system is profiled most importantly to understand it's spatial and temporal complexity, among other information. It is important to profile applications so the information gathered can be used for optimizing the program. The framework [1] deals with cache profiling, which is finding the number of cache hits of pages in different memory regions of a program, once they are stored in the Last-Level of Cache (LLC). By LLC we indicate last level of cache. The profile for the different memory regions and associated pages is generated by locking the pages in LLC and measuring the number of cache hits during the execution of the program. It is important to understand the cache profile of a program, because using this information it becomes possible to optimize program execution in general-purpose computing systems; and to improve predictability in real-time systems. A dynamic real-time system is said to be predictable if we are able to satisfy the timing requirements of critical tasks with 100% guarantee over the life of the system, and we are able to assess overall system performance over various time frames, and we are able to assess individual task and task group performance at different times and as a function of the current system state [19].

The framework [1] proposes an effective cache management strategy. The "hottest" pages (those with the highest number of cache hits in the LLC) from the cache profile are assigned a color and cache way number, and then these colored pages are locked down in LLC. Lockdown means to store them in cache and to ensure that these pages are not evicted during the execution of a program. There is a significant performance improvement as more of the "hot"(frequently accessed) pages are locked down in the LLC and task execution becomes more predictable on multi-core systems.

It is important to visualize results from any cache management framework for any end user to understand the effects and appreciate the benefits of the framework. In our case, the end user could be anyone working with the framework or a student using it as an educational tool. The main contributions of this thesis are:

1. A GUI design to automate cache profiling and colored lockdown for a program from the aforementioned framework [1]; display data visualization for the cache profile as well as performance results after performing colored lockdown on some of the "hottest" (most frequently accessed) pages (see chapter 6).

2. Extension of colored lockdown to a new synchronization mechanism (see chapter 5).

There was already a mechanism in place (L-2 profiler) for profiling pages in the L-2 (second level of cache) cache and finding out the number of cache hits for different pages during the execution of a program and based on which colored lockdown is performed, and subsequently performance evaluation. However, the data had to be stored and plotted to measure the performance.

So it was decided that a front end tool to automate the plotting of the results from the profiling, measured in terms of 'pages in different memory regions vs respective cache hits' as well as plots of performance after performing colored lockdown, measured by 'number of pages locked in cache vs cycles(time taken)' could be developed.

The process of creating benchmarks for the tool, either by generating a user's own benchmark or using an existing benchmark to work with the existing L-2 profiler was also something that was required to iron out and test the existing system in place.

Further, several Cortex A-9 processors do not support 'lockdown by master' but only 'lockdown by way', so a mechanism to run the core exclusively (during the prefetch and lock phase of colored lockdown), by stalling other cores until it is done performing this task, was required.

1.2 Requirements

There was a need for a front end to simplify the process of studying cache management using the existing framework (more info in chapter 2.3). So the first requirement was to build a GUI to achieve this.

Since the tool may be used by students it was needed to evaluate the tool and the usability of the system, hence a user experience study was required to help understand the usability of the tool to the intended audience.

Different benchmarks are used in testing a caching strategy. So it was required to compile,run and test two different types of benchmarks, synthetic (user created) as well as custom (already created) with the L-2 profiler and see the results in the GUI.

Finally it was required to implement a mechanism for synchronizing the cores to stall the unimportant cores temporarily during 'prefetch and lock', and study the changes in performance.

1.3 Overview

The remainder of the thesis can be summarized as follows:-

- Chapter 2 deals with existing concepts that were essential such as the concept of caching, multi-level caching and the existing framework of colored lockdown to understand the tasks at hand and begin development.
- Chapter 3 deals with information relating to the embedded board on which the work is performed. Specifications of the board, process of interfacing with the board as well

as how to use L-2 profiler are dealt with.

- Chapter 4 deals with the process of compiling and running user created benchmarks (synthetic) and the process of compiling and running existing benchmarks (custom) with the L-2 profiler and tool.

- Chapter 5 deals with a support mechanism for processors that do not support 'lock-down by master' but only 'lockdown by way', detailing the basis for the mechanism, explanation and implementation of two mechanisms (conservative approach and optimized approach) as well as a comparing the performance results before any synchronization was implemented for this goal and after implementing the optimized approach of synchronizing cores.

- Chapter 6 goes into the details of the design process, development process of the GUI, and features of the GUI.

- Chapter 7 details the methodology for the user experience study and evaluation of the GUI.

- Chapter 8 is to inform the reader of similar tools that exist.

- Chapter 9 is the conclusion chapter for the thesis and summarizes the results from the work.

CHAPTER 2

BACKGROUND

This chapter serves to highlight the background information from readings of technical books and research work that were essential to understand for the purpose of performing the tasks at hand. This chapter serves as a synopsis of concepts detailed in the book [3] as well as the concepts from the research paper [1].

2.1 Importance of Cache

Some factors to consider in any storage device are:

- Access Time: measured as the duration between the instant at which an instruction control unit initiates a request for particular data or a request to store a data, and the instant at which delivery of the data is completed or the storage begins [25].
- Cost Per Bit: measure of how much the technology involved in a storage device costs per unit (bit).

The basis for the mechanism of cache is locality of reference [2]:

- Temporal Locality: if a datum is used once by a program, it is highly probable for it to be used again.
- Spatial Locality: if a particular datum is used by a program, it is highly probable that other data close to it in the memory address space are used by the program.

The primary purpose of cache is to provide faster look-up of data. It works on the principle of locality of reference, which is detailed above. On a computer system, a cache is used to store a small piece of data, originally found in main memory which may be used again by the

program. Since the access time for cache is smaller than the access time for main memory, it provides faster look-up.

The technology used to make the cache usually has a faster access time and typically is more expensive, having a higher cost per bit. To be effective the cache only has to be large enough to hold the applications working set–the set of instructions and/or data items the application is currently using to perform its computations. Due to locality of reference, most of the application accesses will be satisfied out of the cache instead of the main memory, and so most of the time the access characteristics will be that of the cache: far faster and consuming less energy than the main memory behind the cache [3].

CPU caches in multi-core systems, embody instances of the following three items [3]:

- The cache organization: logical arrangement of data stored within a cache's context. The cache may be direct mapped, fully associative, or set associative.
- Content-management heuristics: design choice of what data to store and what data not to store as well as when to store data in cache.
- Cache coherance protocol: it is possible for a coherence problem to arise if multiple cores have access to multiple copies of a datum (in multiple caches) and at least one access is a write. Access to a piece of stale data (a piece of data that is not updated) is prevented using a coherence protocol, which is a set of rules implemented by the distributed set of actors (cores) within a system [20].

2.2 Multi-level Caching

Cache on a computer system may be split into N cache levels: L1, ..., LN, with LN = Last Level Cache (LLC). By LLC we indicate last level cache. The different levels may be in specific relations with one-another, they could follow either an inclusive, exclusive or non-inclusive non-exclusive relationship. Each level in the cache typically has a smaller size and has a smaller cache latency than the next level. For instance, L2 may have a larger size than L1 but a larger cache latency.

While checking for the contents in the cache, a program first looks to the contents in the L1 cache, if not present in L1, the program may then look into the larger L2 level with a higher cache latency, and if not present in L2, it looks at the next level, so on and so forth. If not found in any of the N-1 levels, it looks at the LLC, and a miss in the LLC means it looks into main memory. It must be noted that if speculation or prefetching is enabled, even LLC cache hits can trigger main memory accesses.

The main advantage to such a multi-level cache organization is that there is a decrease in Average Access Time (AAT) with every additional level of cache.

2.3 Existing Framework-Colored Lockdown

The four most common types of cache interferences are [1]:

1. Inter-Task Interference: This is when tasks operating on the same CPU interfere with one another, evicting the other task's cached blocks.

2. Intra-Task Interference: This is when a task evicts it's own cached blocks.

3. Cache Pollution: This is caused by asynchronous kernel mode activity such as Interrupt Service Routine, for instance.

4. Inter-Core Interference: This occurs when tasks running on different cores evict each other's blocks on a shared cache. This is exclusive to multi-core systems, while the above three are possible in both uniprocessor and multi-core systems.

In multi-core systems, it is not accurate to use Worst Case Execution Time, estimated by measurements in a single core configuration to analyze schedulabilty of a complete system. Owing to inter-core interference, there is inter-core dependency. Thus traditional methods of analyzing the schedulability do not work. In order to improve the predictability of a given task set on multi-core systems, [1] proposes :

1. a framework to analyze and profile task memory access patterns which is done in an execution-independent manner.

2. the above framework is then used to perform colored lockdown, exploiting “coloring” of most accessed pages of memory areas to lock in the cache and “lockdown” to override the cache replacement policy, to make it more efficient.

2.3.1 The Memory Profiling Framework

The mechanism of memory profiling framework is split into three stages [1]:

1. Accesses collection: During the access collection phase, all virtual addresses accessed during task execution are collected which is done using a Valgrind subtool named Lackey; then, a list of all accessed memory pages sorted by the number of times they were accessed is created. Finally, the profiler records the list of memory regions assigned by the kernel in the profiling environment.

2. Areas detection: In this phase, the task is executed without valgrind under normal conditions and the memory areas assigned by the kernel are stored in a format similar to those recorded during the accesses collection phase.

3. Profile generation: In this final phase, the actual profile is generated by merging all the pieces of information recorded during accesses collection and area detection. The lists in previous phases are linked, then compared to find the correlations in memory regions to generate an execution-independent memory profile of the task.

2.3.2 Colored Lockdown

Colored Lockdown can be broken down into two phases [1]:

1. During the first phase – start-up color/way assignment, the system assigns every “hot” (frequently accessed) memory page of every considered task a color and a way number, depending on cache parameters and blocks that are available.

2. During the second phase – dynamic lockdown, the system prefetches a portion or all of the ”hot” memory areas of a given task and locks them in LLC.

CHAPTER 3

INTERFACING WITH EMBEDDED BOARD AND USING L-2 PROFILER

3.1 Specifications of Embedded Board

The Embedded Board used for the project was the Open Exynos4 Quad Mobile Development Platform,the ODROID-X, a development platform which is based on Exynos 4412 ARM Cortex-A9 Quad Core [11].

Some of the most important specifications for the purposes of the project, are summarized below [11]:

Processor	Samsung Exynos 4412 Cortex-A9 Quad Core 1.7Ghz
Size of L2 Cache	1MB
3D Accelerator	Mali-400 Quad Core
Memory	1024MB (1GB) LP-DDR2 800 Mega data rate
LAN	10/100Mbps Ethernet with RJ-45 Jack (Auto-MDIX support)
USB 2.0 Device	ADB/Mass storage (Micro USB)
UART	System console monitoring for development (1.8V interface)
System Software	Linux v 3.8.13

Table 3.1: Specifications of the Embedded Board

The Cache Hierarchy on the embedded board is shown in Figure 3.1.

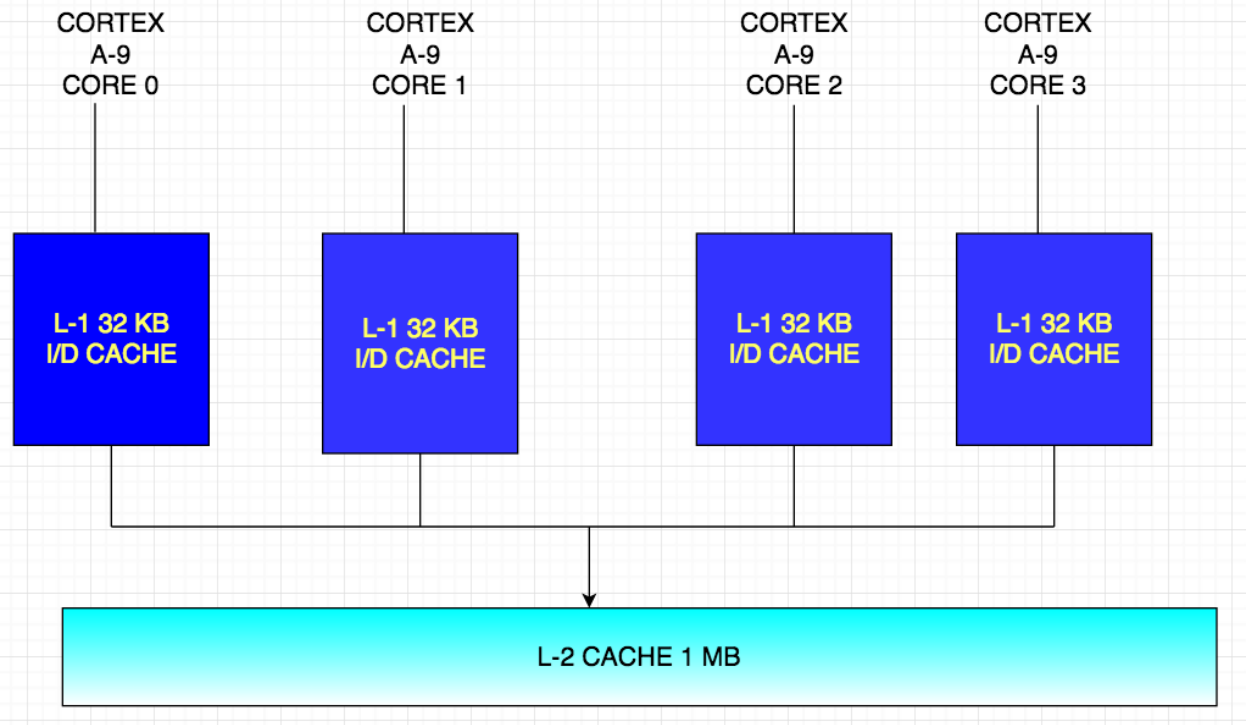


Figure 3.1: Cache Hierarchy on the Samsung Exonys 4412 Cortex-A9 Quad Core

3.2 Interfacing with Embedded Board

The Process of Interfacing with the board can be broken down into three parts: configuring minicom, assigning embedded board an Internet Protocol(IP) address and connecting to IP via ssh. By IP, we indicate internet protocol.

3.2.1 Configuring Minicom

"Minicom is a text-based serial port communication program" [13] and specific to our use, it allows a user to access files, edit files and run programs on an embedded board which is connected to the user's machine via a serial port, `ttyX` (where X represents the serial port example USB0).

The steps to connect to the embedded board are listed:

1. Once the embedded board is powered on and then connected to the user's machine through a USB-UART module kit, which is a system console interface board that

facilitates platform development and debugging [26], the command `dmesg` is issued on the terminal to reveal the port on the user's machine to which the embedded board is connected.

2. The user is granted superuser permission and minicom is opened using the command `sudo minicom`.

3. Once inside minicom, the user has to set up the serial port parameters based on step 1 and turn off hardware control flow from the serial port settings, in order to communicate with the embedded board through a command line interface.

However, we can not display graphical applications through minicom because XServer can not proxy windows through a serial port interface. Hence we need to remotely connect to the embedded board through ssh.

3.2.2 Assigning Embedded Board an IP Address

The goal is to be able to display the GUI on a remote machine. In order to achieve this, it is required to assign a IP address through minicom to the embedded board. To assign the embedded board a IP address on a LAN (Local Area Network) connection, we need to assign a MAC (Media Access Control) address and provide an IP address from available IP addresses to the client(the ethernet port) as a lease, through DHCP (Dynamic Host Configuration Protocol).

3.2.3 Connecting to IP via ssh

SSH stands for secure shell. The SSH protocol provides a method for secure remote login for a remote machine to connect to the embedded board and it provides communications security and protects integrity with strong encryption [12]. From a terminal of the same machine/another machine (on the same LAN), we connect to the IP via ssh. The `-X` option is used, in order to be able to display the graphical application. After this, the user has to grant video memory mapping permissions for the machine in order to display the GUI. The

location of the `$DISPLAY` environment variable is identified and necessary permissions are granted using the command, `xauth add`. This allows a user to display graphical applications through ssh on the embedded board.

3.3 Using L-2 Profiler

The L-2 profiler is a tool which performs accesses collection, areas detection and profile generation for a given program. Based on this profile, colored lockdown of the top N number of pages is performed. The first step in using the L-2 profiler is compiling it with the necessary rules and directives to generate a special executable (more information can be found in section 4.1.1) with the L2 profiler shared library. Once this is done, the sequence of steps are detailed in following subsections.

3.3.1 Snapshot of memory

A system possesses physical memory (RAM) which is responsible for providing memory to different processes for their execution. However, there is only a limited amount of physical memory available. Modern systems provide a 'virtual' memory, a memory management technique to indicate that a larger memory is available to processes on the system than what is available through physical memory. The virtual address space is larger than the physical address space. Virtual addresses are mapped to physical addresses through indirection. On Linux systems, the virtual address space is divided into the kernel space and the user space.

Any memory-physical or virtual, is divided into discrete units called pages. All of the system's internal handling of memory are done on a page-by-page basis [22]. Though page sizes may vary according to the architecture, we restrict our discussion to 4096-byte pages. Any memory virtual or physical is divisible into a page number and an offset within the page [22].

The address space of a process is composed of sets of linear addresses that the process is allowed to use [21]. The kernel represents these sets of linear addresses as memory regions; each memory region is represented by means of an object of type `vm_area_struct` [23]. An

object of this type holds information about ownership, start and end addresses for that memory region among other things.

The L-2 profiler library also uses a similar implementation to maintain information about the memory regions of a user process (the one we are profiling). The L-2 profiler accesses information about the memory regions of the process being profiled from `/proc/pid/maps`. The L-2 profiler stores the memory descriptors for the memory regions in a red-black-tree data structure, similar to the Linux implementation.

To view a snapshot of memory in terms of what the memory regions represent and number of pages in the memory regions, a shell script `snapshot.sh` followed by name of specially generated executable with optional arguments (more information in 4.1.2), runs the generated executable, performs a dumping of `proc/pid/maps` and sets up the environment variables and virtual memory area red-black-tree data structure, provided by the L-2 profiler shared object library. The snapshot script uses the environment variables to display a snapshot of the memory regions. This script is invoked in the the following way:

```
snapshot.sh (generated executable with optional arguments)
```

An example of a line of output showing a snapshot of one memory region is shown below:

```
chunk id:28 start:be92e000 end:be94f000 pages:33 flags:RW S! name:[stack]
```

The first parameter `chunk id` indicates the number given to the memory region, since the framework constructs the cache memory profile in an execution independent manner. The second and third paramter display the start and end addresses of the memory region, respectively. The third parameter displays the number of pages. The fourth parameter displays the permissions for that memory region and finally, the fifth displays the name of the memory region.

3.3.2 Profiling of Memory Regions

Once accesses and areas detection has been performed (on running the specially generated executable with `snapshot.sh`), a memory profile for the executable can be generated. The L-2 profiler provides a method to check for cache hits in the L-2 level of cache for all the

pages in a program. A kernel module, `cachecl.ko` performs lockdown (storing it in cache and ensuring there is no eviction) of a given page in the L-2 level of cache and measures the cache hits for the page during one execution of the program. This process is repeated for all the pages in all possible memory regions. It is to be noted that the kernel module trashes all the contents in the L-2 cache after each page is profiled. Profiling the benchmark program comprises of profiling of all pages in all memory regions and this provides one sample of the memory profile. For an aggregate memory profile for all the pages in all memory regions using N number of samples, batch profiling may be performed using the following script:

```
batch_profile.sh arg0 arg1 arg2
```

where, `arg0` is the number of samples of memory profiling to aggregate, `arg1` is the prefix of the name of `.csv` file to be generated and `arg2` is the generated executable with optional arguments.

CHAPTER 4

SYNTHETIC AND CUSTOM BENCHMARKS

Synthetic (user-created) benchmarks or custom benchmarks (from other sources) may be compiled to work with the L-2 profiler and be used with the GUI tool to study the memory profile of a program and colored lockdown performance.

4.1 Compiling and Running Synthetic Benchmark

The main intention in creating a synthetic benchmark is to mimic the memory allocation and memory access characteristics of a large program. A simple synthetic benchmark program is illustrated in Figure 4.1.

Figure 4.1: C program for synthetic benchmark

```
/* Synthetic Benchmark file - play.c */
#include <stdio.h>
#include <stdlib.h>

/* number of bytes to allocate in heap memory */
#define BUF_SIZE 80*1024

/* number of bytes to allocate in stack memory */
#define STACK_BUF_SIZE 100*1024

/* Upper Limit for access iterations */
#define ACCESSES_ITER 100*1024
```

```
int main()/* main function */
{
int i = 0, c;
        char * buf = malloc(BUF_SIZE); //Perform allocation on heap
        unsigned long crc = 0;
        char stackbuf [STACK_BUF_SIZE]; //Allocate space on stack

        /* Access every byte of memory in heap or stack that is a multiple of
           16 */
        for(c=0; c<ACCESSES_ITER; c+=16)
        {
                buf[c%(BUF_SIZE)] = 10;
                if(c<STACK_BUF_SIZE)
                        stackbuf[c] = 15;
        }

        /* Perform a number of memory accesses */
        for(i=0; i<50; ++i)
        {
                for(c=0; c<ACCESSES_ITER; c+=32)
                {
                        if (c%5 == 0)
                                crc += stackbuf[((c/5)%STACK_BUF_SIZE)];
                        crc += buf[c%(BUF_SIZE)];
                }
        }
        return crc;

}
```

Figure 4.1(contd): C program for synthetic benchmark

The program in Figure 4.1 allocates memory in the stack and heap, and accesses them a number of times. In order to study any benchmark with the L-2 profiler and display the results from the framework [1] on the GUI, it is necessary to compile it with necessary rules and libraries.

4.1.1 Compiling a Benchmark with L-2 Profiler

The `Makefile` for the synthetic benchmark to generate an executable that uses the L-2 profiler library, is illustrated in Figure 4.2.

Figure 4.2: Makefile for synthetic benchmark

```
1  MAKEFLAGS +=--no-builtin-rules --no-builtin-variables \
2              --no-print-directory
3  INCDIR=/root/profiler/l2_prof
4  LIBRELDIR=/root/profiler/l2_prof
5  TARGET=play
6
7  PROFILED=$(TARGET:%=%_l2p)
8
9  CC=gcc
10
11 CFLAGS=-Wall -Wextra -O2 -g -I$(INCDIR)
12
13 LDFLAGS=-Wl,--no-as-needed -L$(LIBRELDIR) -rdynamic  -Wl,-rpath,\
14         '$(LIBRELDIR)' -Wl,-z,now -Wl,--wrap=main
15
16 DYNLIBS=-lm
17
18 all: clean compile library
```

```
19
20
21
22
23  $(PROFILED): $(PROFILED).o
24          $(CC) $(LDFLAGS) -o $@ $^ $(DYNLIBS) -ldl -lL2p \
25                -DL2P_PROFILE
26
27  %_l2p.o: %.c
28          $(CC) $(CFLAGS) -c -o $@ $^
29
30  $(TARGET): $(TARGET).o
31          $(CC) -o $@ $^ $(DYNLIBS)
32
33  %.o: %.c
34          $(CC) $(CFLAGS) -c -o $@ $^
35
36  compile: $(TARGET) $(PROFILED)
37
38  library:
39          $(MAKE) -C $(LIBRELDIR)
40
41  clean:
42          rm -rf *.o *.mtr out* $(TARGET) \
43          $(PROFILED)* core time_* pinatrace.out pin.log
44
45  .PHONY: clean compile
```

Figure 4.2(contd): Makefile for synthetic benchmark

Using the `Makefile` in Figure 4.2 as example, the following are some guidelines for compiling a benchmark to run with the tool. The documentations [14] and [15] were used to refer to some specifics:

- Lines 1 to 2, add some makeflags to the `Makefile`. `---no-builtin-rules` is to ensure that none of the built-in rules of a `Makefile` are followed. `---no-built-in-variables` to indicate that no built-in variables of a makefile are going to be used and `---no-print-directory` to ensure that messages such as entering a directory, leaving a directory are not printed to the message buffer. Similarly, the user may include or exclude certain makeflags as desired.

- Lines 3 and 4 are variables that contain the path to directories. Line 3 indicates the directory to be included in compilation, Line 4 indicates the directory containing the L-2 profiler library `libL2P.so`. Other directories may be listed as per the user's requirements.

- Line 5 is the variable holding the name of the regular executable to build for the benchmark and line 7 is the variable holding the name of compiled special executable with L-2 Profiler library.

- Line 9 indicates the compiler to use. This is as per the user's requirements, in our example we use `gcc` for compilation.

- Line 11 contain flags for the compiler. Line 11 contains `-Wall` which enables warnings about constructions, `-Wextra` enables warning flags that are not provided by `-Wall`. `-O2` is an optimization level specified for compilation. A variant of this could be used based on the user's requirements. Note that higher levels of optimization take a longer compilation time. `-g` provides debugging option with gdb. And `-I` option is to include a directory during compilation, here to compile with the L-2 profiler, we provide this directory reference.

- Line 13 contains flags containing the library path which are essential to build the special executable. The option `-L`(suffixed with directory name) is used to add the

directory to search for the library specified by `-l`, for the purpose of linking. `-rpath` specifies the run-time library search path, which is used when dynamically linking. The option `---wrap=main` is for a wrapping mechanism which works by first replacing the symbol `main` in the original executable of target program to `_real_main`. It redirects any reference to the symbol `main` to a function `_wrap_main` which in our case, sets up required L-2 profiler environment variables and then calls the `_real_main` symbol.

- Line 16 specifies other libraries to dynamically link with. This is as per the benchmark.
- Lines 18 to 40, hold the rules containing targets and dependencies, using the compiler flags and variables specified by the user. To note is line 20 where the option `-lL2P` is used to add the shared object library, `libL2P.so`. Also a variable named `L2P_PROFILE` is defined in our special executable. The rest are typical of regular makefiles. And based on the benchmark of the user, these can be modified accordingly.

4.1.2 Results From a Few Runs of Synthetic Benchmark

Once, a special executable with the L-2 profiler library has been compiled, we display a snapshot of the memory, which also runs the program and sets up the environment variables in the L-2 profiler (see 3.3.1). We then perform batch profiling of the pages in different memory regions (see 3.3.2). And finally, using the `cachecl.ko` kernel module we perform colored lockdown, a script `get_cl_curve.sh` automates this process of performing colored lockdown on the top 200 "hottest" pages. For the sake of simplicity from here on in the document by "hottest" we refer to the most accessed pages in the L-2 level of cache and by "hot" we refer to frequently accessed pages in the L-2 level of cache. The calls to these scripts were combined in `make_helper.sh` for a user to change parameters. This is the script invoked on the button press `Make Benchmark!` on the GUI (see Figure 6.2 in chapter 6).
For comparison, we consider two cases:

1. Case 1: On performing these steps with the synthetic benchmark with batch profiling samples set to 5.

2. Case 2: On performing these steps with the synthetic benchmark with batch profiling samples set to 10.

We are concerned with the difference in progressive colored lockdown performance between these cases, the results for memory profile of stack and heap for the first case are shown in Figure 4.3 and Figure 4.4 respectively, for reference. The resulting progressive lockdown performance curves, in which each point represents the cycles taken for one execution of the program in the y-axis versus the number of "hottest" pages that are locked down in the L-2 level of cache during that particular execution of the program. The progressive lockdown performance curve for the first case is shown in Figure 4.5. Similarly, the resulting progressive colored lockdown curve for second case is shown in Figure 4.6.

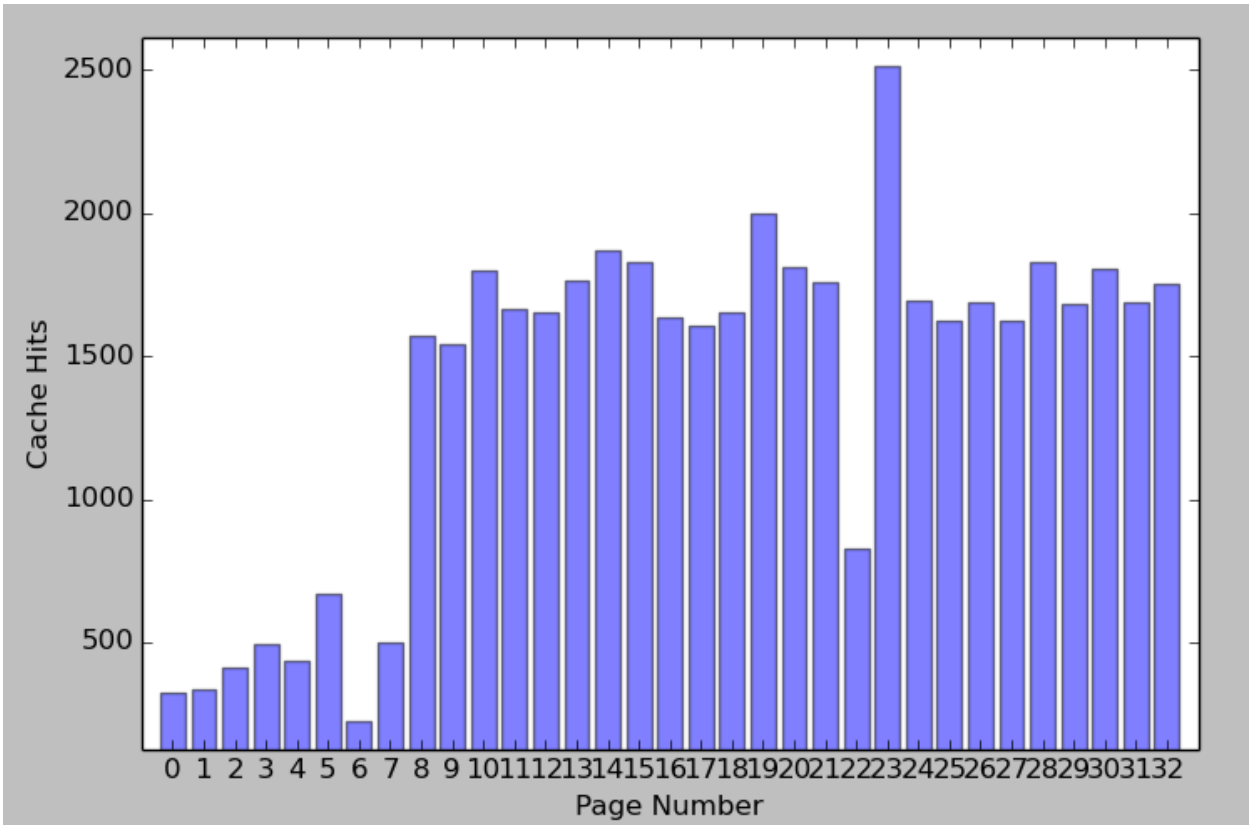


Figure 4.3: memory profile of stack for first case

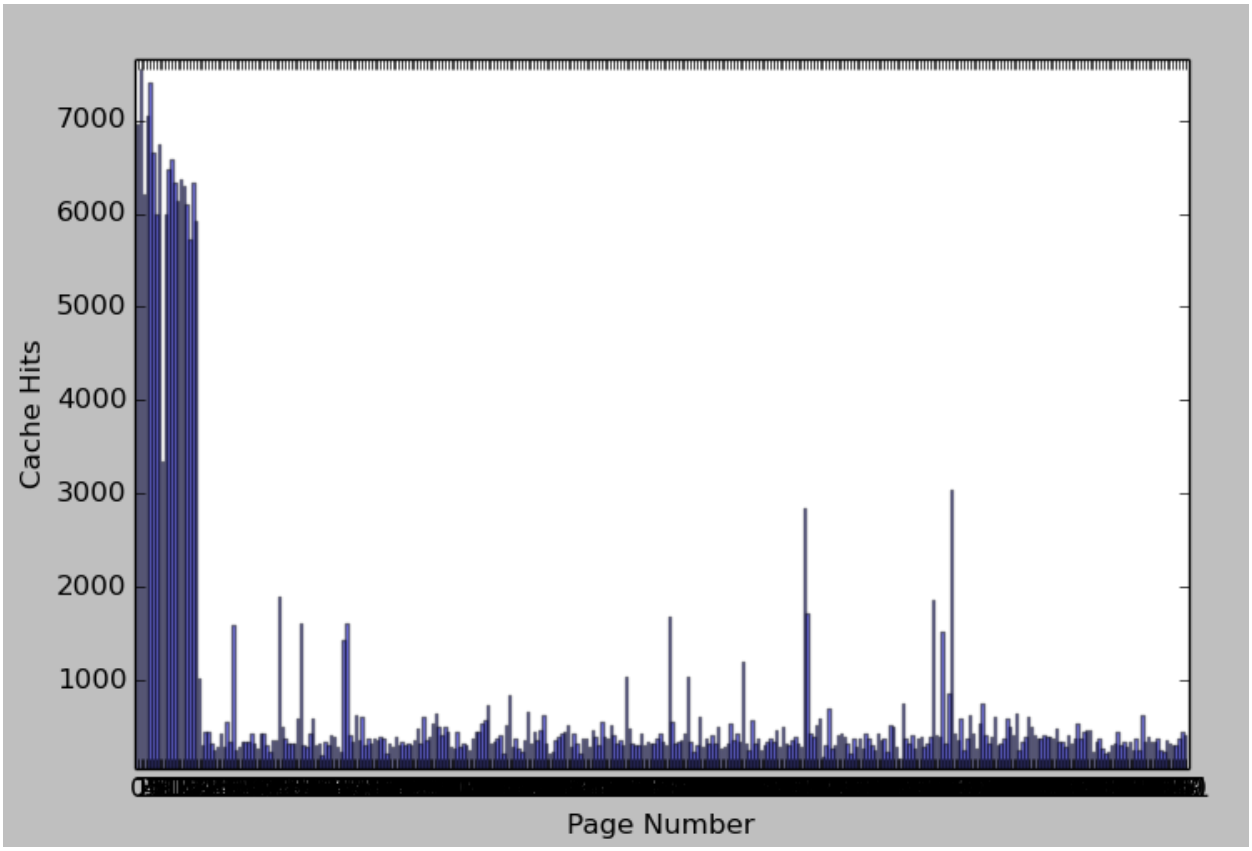


Figure 4.4: memory profile of heap for first case

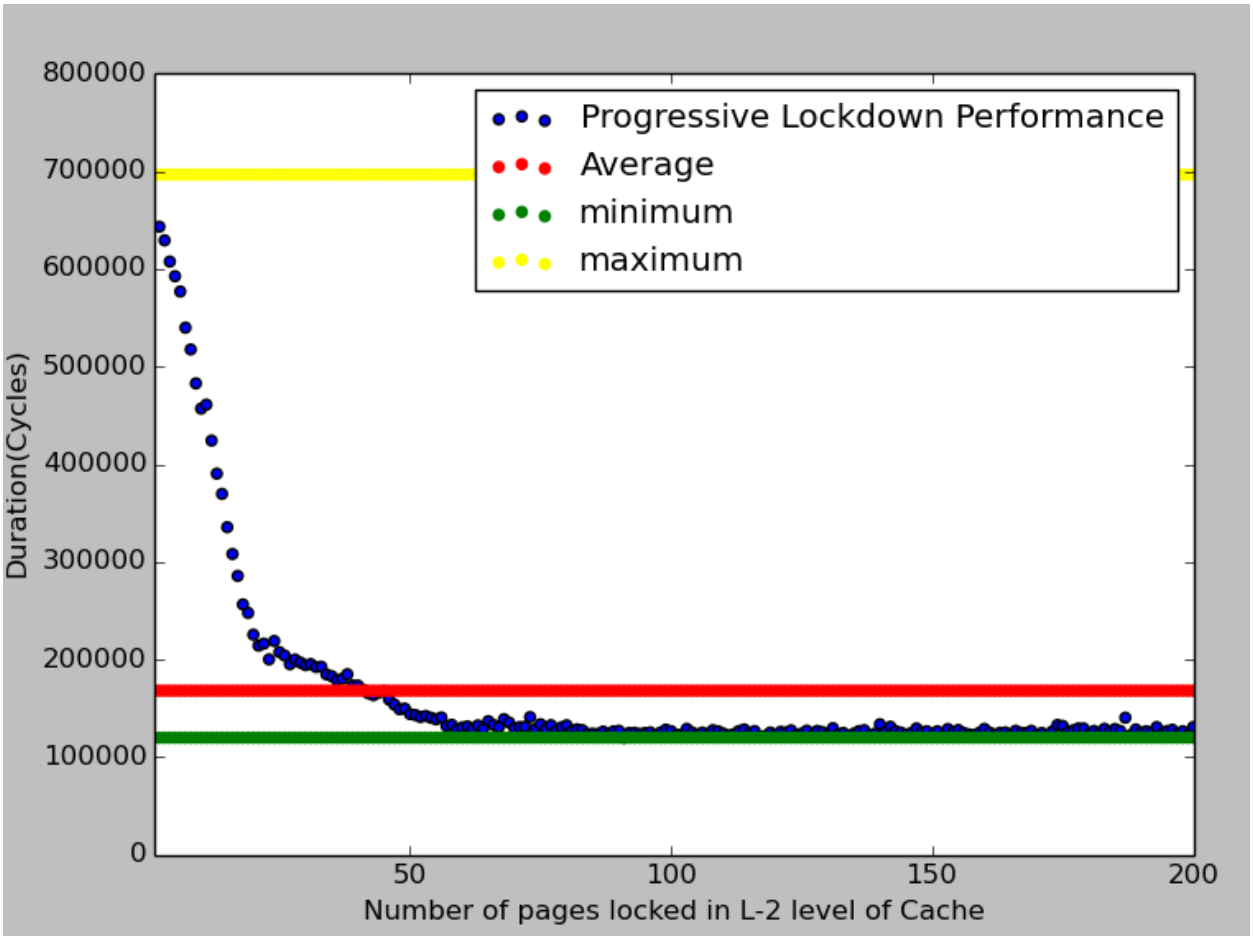


Figure 4.5: progressive colored lockdown performance for first case

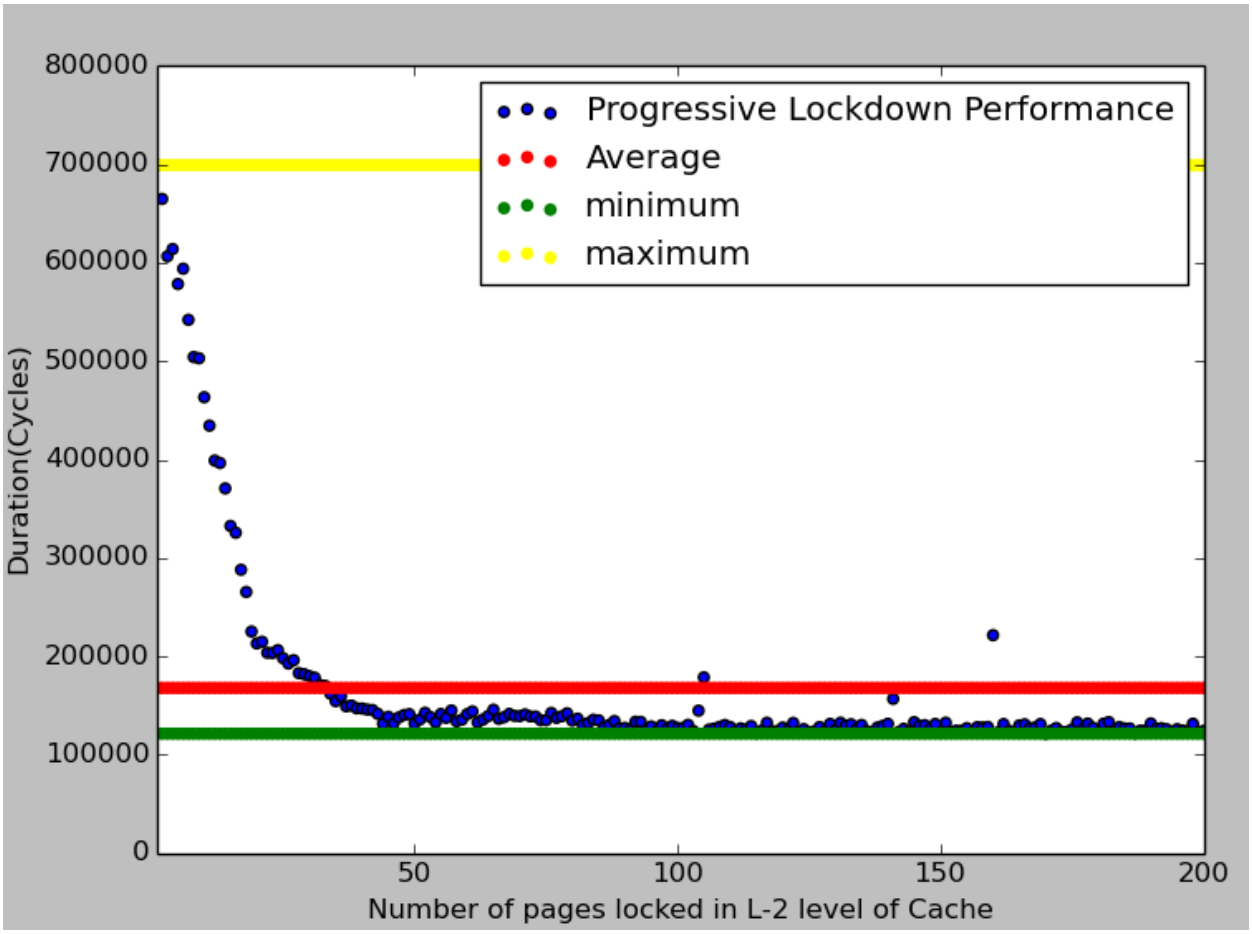


Figure 4.6: progressive colored lockdown performance for second case

Through this and several other similar tests, it was found that a higher number of aggregated samples for memory profiling improved colored lockdown performance, on average. We attribute this to a better selection of top "hot" pages. As can be observed from comparing Figure 4.5 and Figure 4.6 (see to the left of the '50' marking on the x-axes in both scatter plots), in the second case it was found that the curve settled much faster to the minimum value, with fewer number of pages locked owing to a better selection of "hot" pages.

4.2 Process of Compiling and Running a Custom Benchmark

A custom benchmark named 'susan' from [16] was tested with the GUI. The process of compiling it with the L-2 Profiler was similar to what is described in 4.1.1 with specific changes such as updating targets and dependencies, as well as ensuring that other libraries

used by the program were included in the compilation. Arguments were added when running the executable with the L-2 profiler.

4.2.1 Results from a Run of Custom Benchmark

The memory profile of stack and heap for the benchmark are shown in Figure 4.7 and Figure 4.8, respectively. A zoomed out view of the progressive colored lockdown performance is shown in Figure 4.9 and a zoomed in view of the progressive colored lockdown performance is shown in Figure 4.10.

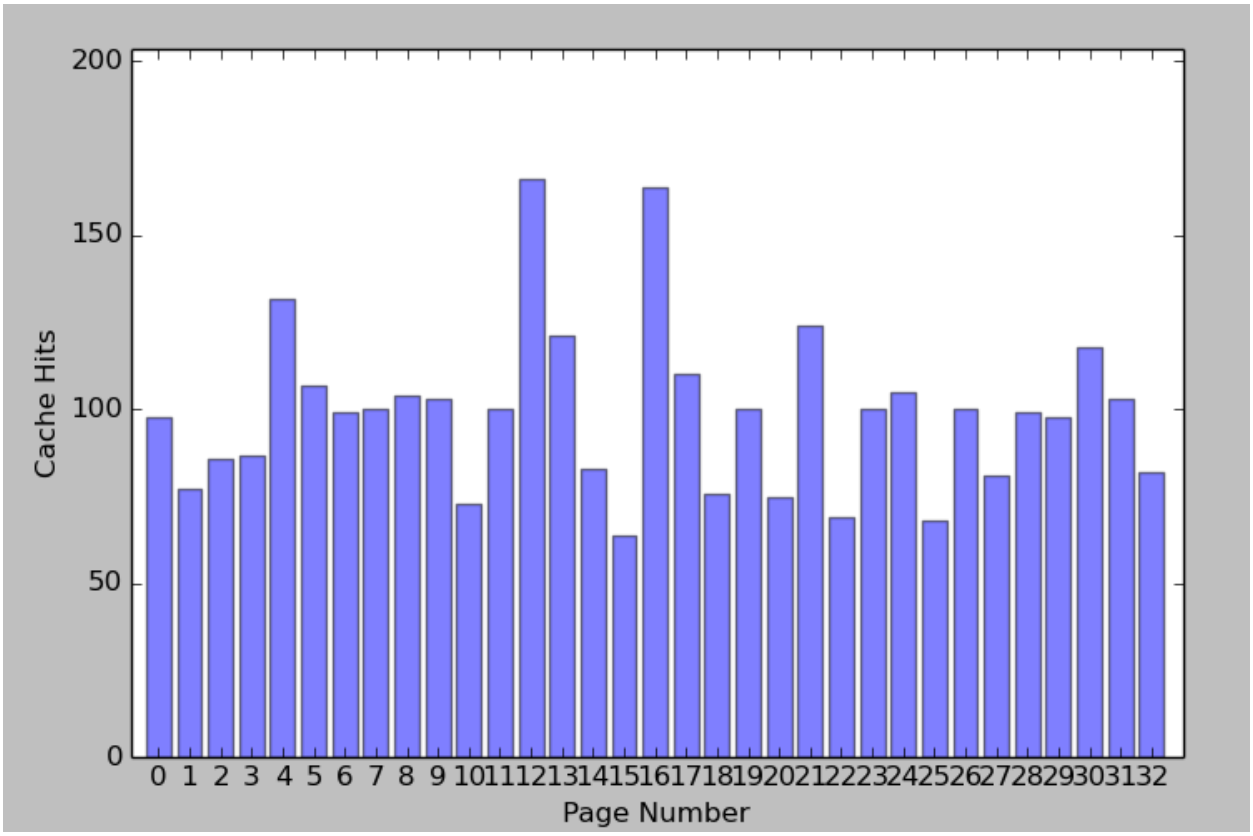


Figure 4.7: memory profile of stack for custom benchmark

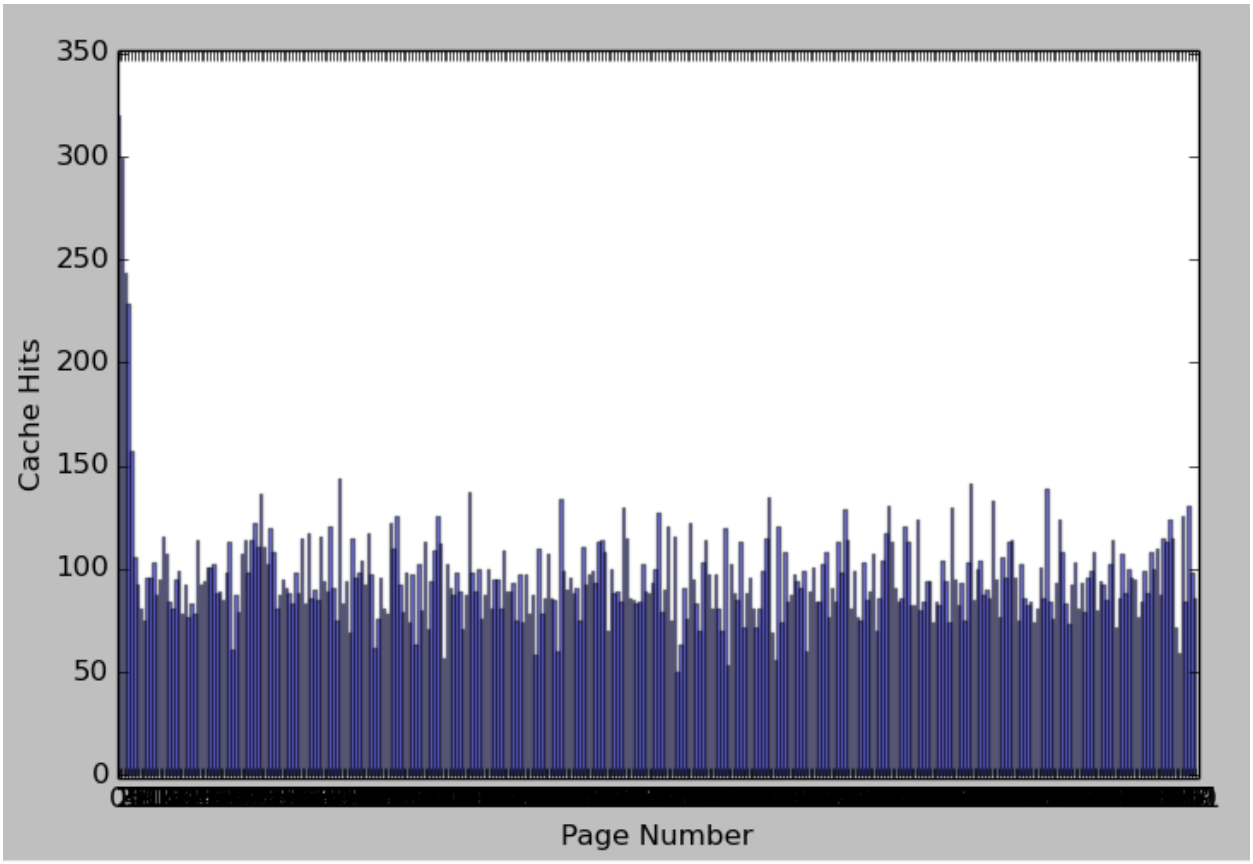


Figure 4.8: memory profile of heap for custom benchmark

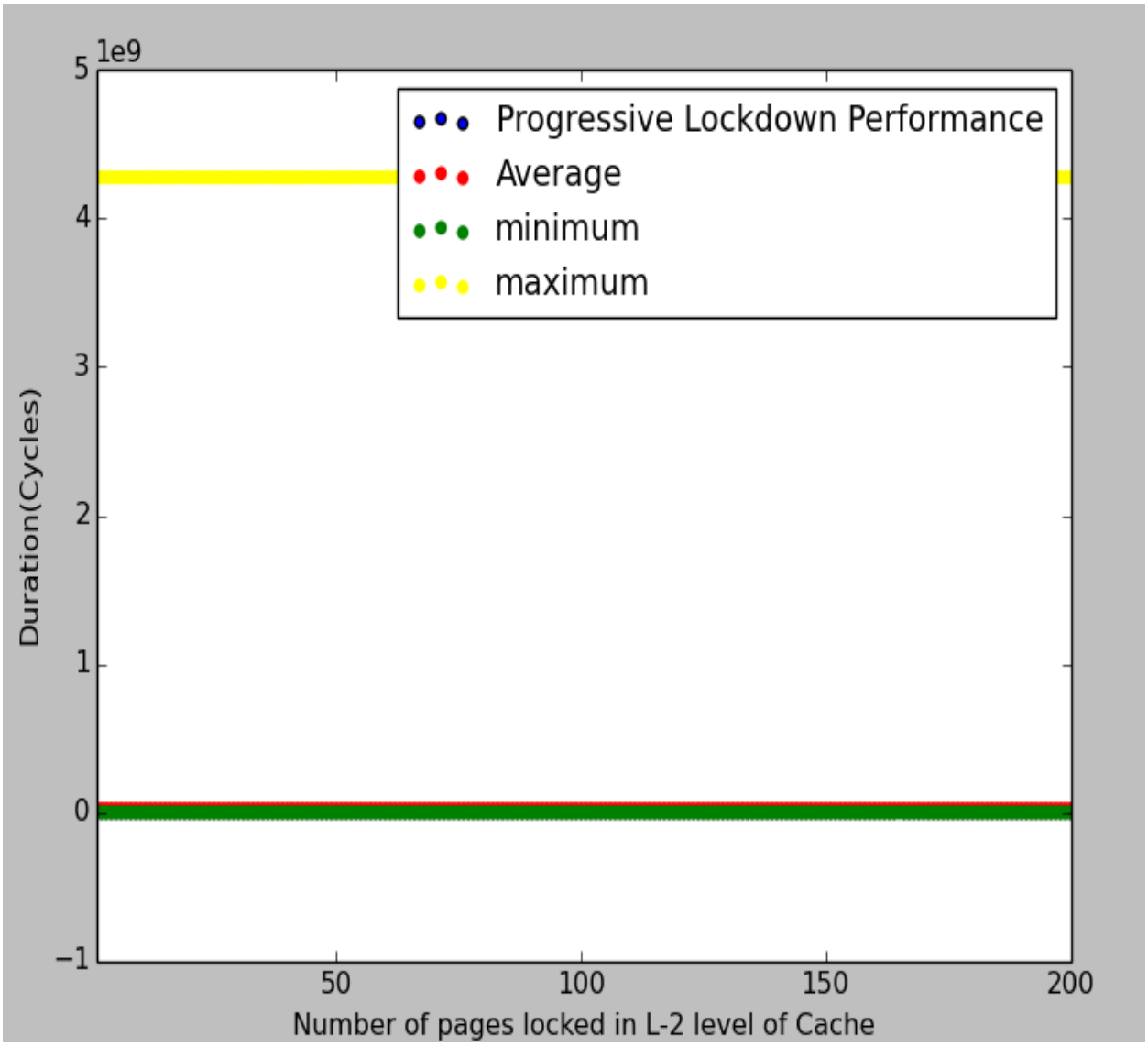


Figure 4.9: zoomed out view of progressive colored lockdown performance for custom benchmark

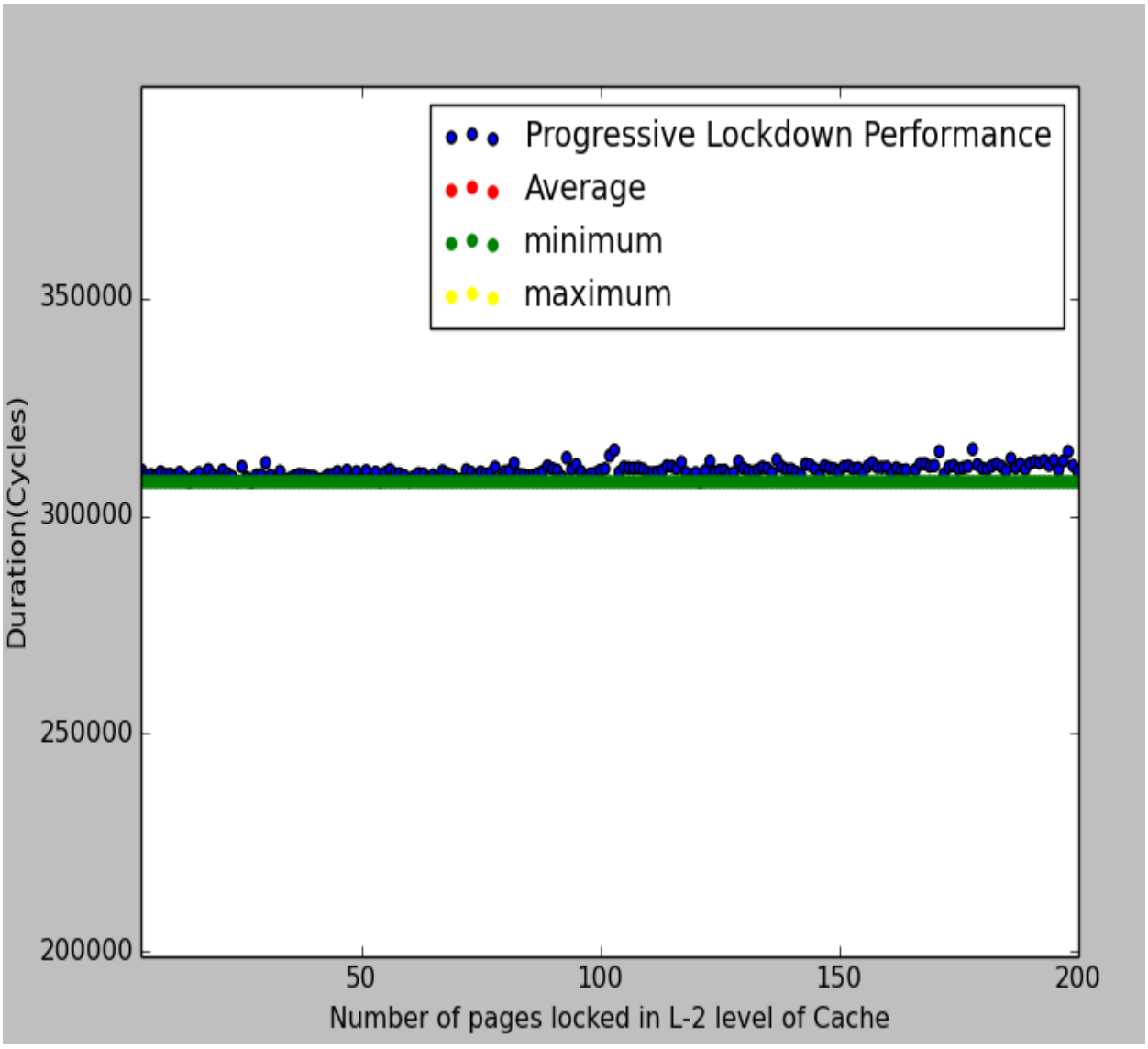


Figure 4.10: zoomed in view of progressive colored lockdown performance for custom benchmark

It can be observed from Fig 4.10 that most points from progressive colored lockdown performance can be approximated to lie in a straight line close to the minimum value in performance. This is because on profiling the pages of the custom benchmark it was found that the different pages did not have a significant variation in their access patterns. All of the 200 most accessed pages had number of accesses in the range of 100 to 350.

CHAPTER 5

SUPPORT FOR SYSTEMS WITH LOCKDOWN BY WAY

5.1 Basis

To inform the reader about the difference between 'lockdown by way' and 'lockdown by master', let us consider a cache with 4 possible ways–1, 2, 3 and 4 in a system with 4 cores–1, 2, 3 and 4. Consider the 4-bit bitmap `bbbb`, where `b` could either be 0 or 1. In 'lockdown by way', a value of 0 for any of the bits can be used to indicate whether a specified way is open for allocation. In this example, the least significant bit corresponds to way 1, and most significant bit corresponds to way 4. A bitmap 0000, indicates that all 4 ways are open for allocation by all the cores. A bitmap 0100 indicates that way 3 is locked and no allocation can be performed on way number 3 by any of the cores. 'Lockdown by master' for this system can be explained using the same bitmap (since there are 4 cores in this example). A value of 0000 could indicate that all four cores can perform allocation of data on cache (to all the open ways) while for instance, a value of 0110 indicates that core 2 and 3 are not allowed to perform allocation. It is to be noted that the number of cores on the system and the number of cache ways available could differ.

In several Cortex A-9 processors, there is no mechanism to perform 'lockdown by master', which is the method to perform lockdown of a piece of data in cache by specifying which cores of the processors are allowed to lock a memory block in any given cache way. On these processors, only cache 'lockdown by way' can be performed (a subset of lockdown by master), that is lockdown can be performed by specifying the way numbers on which lockdown can be performed. However when a way number is specified for allocation through 'lockdown by way' in the absence of 'lockdown by master', it is available to all cores for allocation on the system. This can lead to unpredictability in the final state of the cache

way if allocation of a memory block was expected to be performed on that cache way by a particular processor, but instead allocation of a different memory block was performed by a different processor.

5.2 Process of Synchronizing Cores during Prefetch and Lock

For those systems that do not have support for cache 'lockdown by master' but have support for cache 'lockdown by way' we propose a first conservative synchronization mechanism which leverages a spinlock to stall unimportant cores and a second optimized mechanism which performs stalling based on an atomic variable. These two mechanisms are presented in 5.2.1 and 5.2.2, respectively.

5.2.1 First Approach

The embedded board is a Symmetric Multiprocessing (SMP) system. The different cores of the board share memory. We leverage existing functions in `asm/smp.h` as well as atomic variables and a spinlock to achieve desired synchronization between cores. During the 'prefetch and lock' phase of colored lockdown in framework [1], while a core is trying to prefetch a page or a number of pages, there exists an unlocked cache way. To reduce unpredictability in the final state of this cache way, we propose a synchronization mechanism which works as follows:

1. The core performing prefetch and lock (current core) is identified, as well as other online and present cores (other cores). A CPU mask containing the IDs of the other cores is made.

2. The current core acquires a spin lock.

3. The other cores are sent to a function (using the CPU mask) where they try to acquire the spin lock and keep waiting. The function is designed so any core running it relinquishes control of the spin lock immediately after it receives it.

4. A barrier is placed to ensure that program execution does not proceed until all other cores are stalled.

5. Once all other cores are stalled, the current core performs prefetch of the memory blocks and locks them in a given cache way.

6. Finally, once the current core is done performing prefetch and lock, it releases the spin lock.

The pseudo code for this mechanism is detailed in Figure 5.1.

Figure 5.1: pseudo code for first synchronization mechanism

```
1  spinlock_t prefetch_lock ; //spinlock
2  atomic_t barrier = ATOMIC_INIT(0); //act as barrier
3
4  int stallcores()
5  {
6          atomic_sub(1,&barrier) ;
7          unsigned long flags;
8          preempt_disable();
9          spin_lock_irqsave(&prefetch_lock,flags);//wait
10         spin_unlock_irqrestore(&prefetch_lock,flags); //release
11         preempt_enable();
12 }
13
14
15 int prefetch_and_lock(void)
16 {
17         spin_lock_init(&prefetch_lock);
18         unsigned long flags;
19         struct cpumask other_cores_mask ; //mask for cpus
```

```
20          int cur_core = get_cpu(); //get id of current cpu
21
22          cpumask_empty(&other_cores_mask);
23          int i;
24      for_each_possible_cpu(i)
25          {
26          if(cpu_online(i) && cpu_present(i) && (i!=cur_core))
27              {
28
29                  cpumask_set_cpu(i,&other_cores_mask) ;
30                  //find other cpus and prepare mask
31                  atomic_add(1,&barrier1);
32              }
33          }
34          spin_lock_irqsave(&prefetch_lock,flags); //acquire lock
35          smp_call_function_many(&other_cores_mask, //contd
36          (smp_call_func_t) stallcores,0,0); //other cores to wait
37          while(atomic_read(&barrier) != 0) ; //barrier
38          // Perform Prefetch and Lock here
39          spin_unlock_irqrestore(&prefetch_lock,flags); //release
40          put_cpu();
41          return 0;
42  }
```

Figure 5.1 (contd): pseudo code for first synchronization mechanism

However once implemented, this caused some unexpected behavior on the kernel and caused the process to be very slow. So we present an optimization using atomic variables.

5.2.2 Optimized Synchronization Mechanism

In the optimized synchronization mechanism, we instead use two barriers controlled by the same atomic variable. Instead of counting all the other online and present cores, it counts the total of all cores on the system. The atomic counter is inititalized to zero and counts up to all the cores present. It serves as a barrier in the prefetch and lock function by testing if all other cores have been stalled, that is it's value has reached 1. The stalling is also performed by the same atomic variable spinning in a while loop until it's value reaches zero. Once the current core has finished performing prefetch and lock, it decrements the atomic variable setting it to zero. The pseudo code for the optimized synchronization mechanism is illustrated in Figure 5.2.

Figure 5.2: pseudo code for optimized synchronization mechanism

```
1  atomic_t barrier = ATOMIC_INIT(0); //act as barrier
2
3  int stallcores()
4  {
5      preempt_disable();
6      atomic_sub(1,&barrier) ;
7      /* wait for prefetch and lock to complete */
8      while(atomic_read(&barrier)!=0);
9      preempt_enable();
10 }
11
12
13 int prefetch_and_lock(void)
14 {
15     spinlock_t prefetch_lock ; //spinlock
16         spin_lock_init(&prefetch_lock);
17         unsigned long flags;
```

```
18          struct cpumask other_cores_mask ; //mask for cpus
19          int cur_core = get_cpu(); //get id of current cpu
20          cpumask_empty(&other_cores_mask);
21          int i;
22      for_each_possible_cpu(i)
23          {
24          if(cpu_online(i) && cpu_present(i))
25              {
26                  cpumask_set_cpu(i,&other_cores_mask) ;
27                  atomic_add(1,&barrier1);
28
29              }
30          }
31          /* Remove current core from cpu mask */
32          cpumask_clear_cpu(cur_core,&other_cores_mask) ;
33          spin_lock_irqsave(&prefetch_lock,flags); //acquire lock
34          smp_call_function_many(&other_cores_mask, //contd
35          (smp_call_func_t) stallcores,0,0); //other cores to wait
36          while(atomic_read(&barrier) != 1) ; //barrier
37          // Perform Prefetch and Lock here
38          spin_unlock_irqrestore(&prefetch_lock,flags); //release
39          atomic_sub(1,&barrier);
40          put_cpu();
41          return 0;
42  }
```

Figure 5.2 (contd): pseudo code for optimized synchronization mechanism

5.3 Colored Lockdown Performance before and after Synchronizing Cores during Prefetch and Lock

To compare between the case before implementing the optimized synchronization (original cache module) and the case after implementing the optimized synchronization mechanism in the cache module, we use two different metrics. The first metric compares the time taken to perform 'prefetch and lock' before and after implementing synchronization. To measure this we measure times from kernel through `Printk-times`, required for completion of the particular function in both cases. We measure the timings for ten runs of the function in both cases and the average values for each case, the results from this are presented in Table 5.1.

Case	Time taken(microseconds)
Before synchronization	31
After synchronization	48

Table 5.1: average time from 10 runs of prefetch and lock

For the second metric, the synthetic benchmark illustrated in Figure 4.1 is used to compare the average progressive colored lockdown performance in the two cases. Batch profiling for the memory was performed the same number of times in both cases, one in both. Three runs, each comprising of batch profiling with one sample and colored lockdown of the top 200 pages, were made. The red scatter plot corresponds to the average progressive colored lockdown (from the three runs) performance before implementing the optimized synchronization mechanism (original cache module) and the green scatter plot corresponds to the average progressive colored lockdown (from three runs) performance after implementing the optimized synchronization mechanism in the cache module.

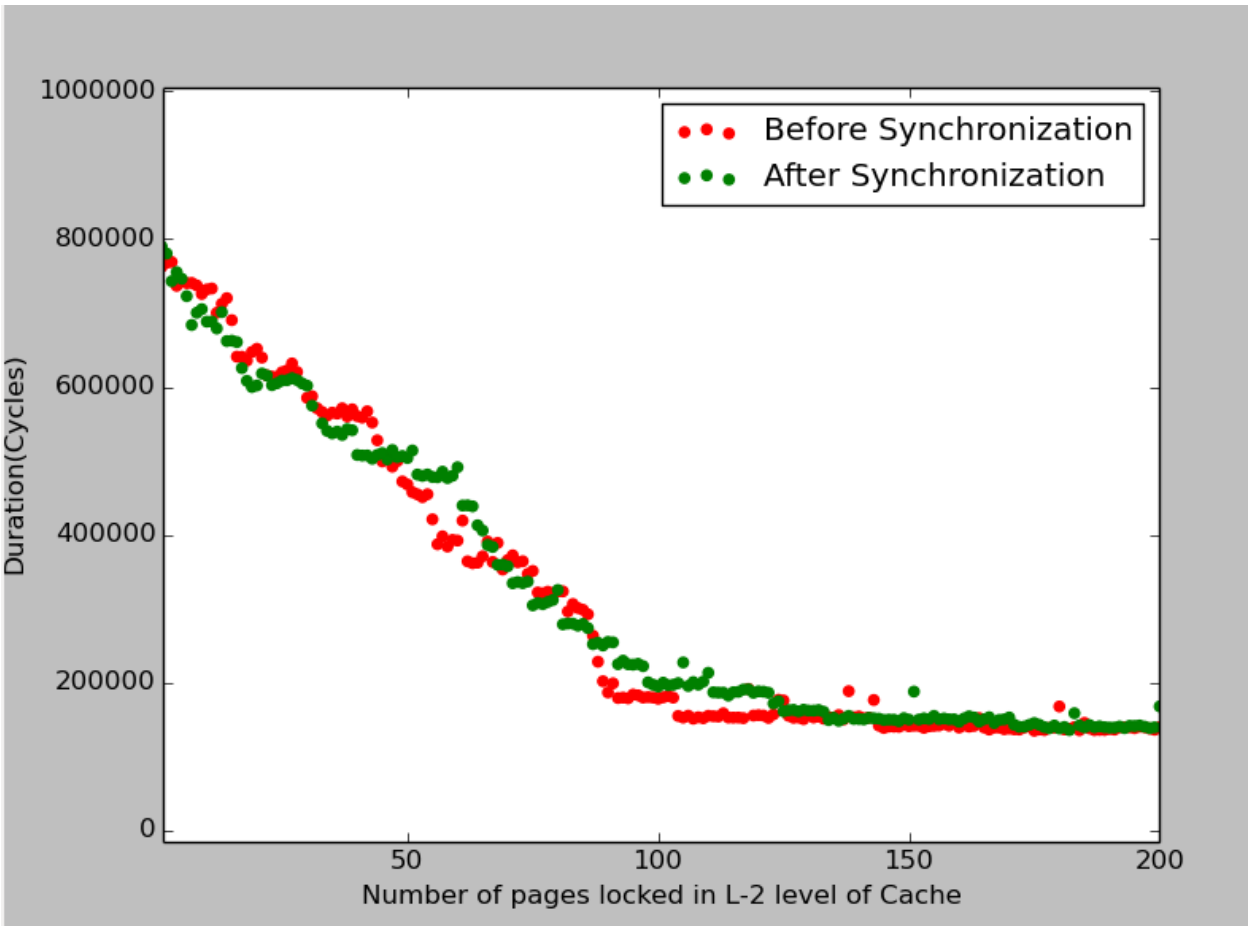


Figure 5.3 : Average Colored Lockdown Performance before and after synchronization

From the first metric, we can infer the synchronization mechanism does not cause a significant performance overhead while performing colored lockdown. And from the second metric we can observe that there was no statistically significant difference in the average colored lockdown performance between the before and after cases. This means that the synchronization mechanism did not affect the results from colored lockdown on a program. In sum, even though there was no improvement in the performance of colored lockdown from the synchronization mechanism, the optimized synchronization mechanism can be implemented on existing Cortex A-9 processors, which have no option to perform 'lockdown by master' but an option to perform 'lockdown by way' without a significant performance overhead, to ensure a predictable final state of a given cache way.

CHAPTER 6

DESIGN AND DEVELOPMENT OF THE GRAPHICAL USER INTERFACE

A Graphical User Interface(GUI) is a user interface that provides a mapping of the user's interaction with graphical content to outcomes. The graphical content is displayed on a screen, and is manipulated by interaction devices such as keyboards or mouses.

The most common structural elements that can be observed in modern GUIs are windows,icons and widgets. And, the most common interaction elements that can be observed in modern GUIs are cursors,pointers and insertion points.

6.1 Design

In designing the Graphical User Interface, principles detailed in [5] were followed to ensure a good design for the GUI.

6.1.1 Determining Users' Skill Levels

The users for this tool are primarily expected to be computer science students who can be considered as "knowledgeable intermittent users". The users are expected to have one or more years of experience in systems programming and to have used similar tools. To ensure that the tool serves the users' needs, it employs context-dependent help in the form of detailed button texts and labels to guide the users' actions.

6.1.2 Identifying the Tasks

The tasks that the GUI performs are :

1. Automating the compiling of benchmarks, memory profiling and performing colored lockdown, initiated through a simple user action.

2. Display snapshot of memory.

3. Present data visualization for the results from memory profiling and performance measurements after performing colored lockdown of the top 200 "hot" (most accessed) pages, sequentially, beginning from a ranked list of most accessed pages and associated memory regions in tabular form, then moving on to cache hits of pages in different memory regions presented as bar graphs, and finally moving onto a scatter plot of 'number of pages locked in L-2 cache' vs 'cycles' to visualize performance.

The widgets designed on the screen are likely to be used frequently to accomplish these tasks, the design choices for widgets were made so as to reduce user actions and increase productivity.

6.1.3 Choosing an Interaction Style

The interaction style chosen was direct manipulation for it's ease of use. Out of the six types of interaction tasks [6], four were employed: select, position, orient and path.

6.1.4 Preventing Errors

To reduce errors, most actions rely on a button press or choosing from a list of possible items, instead of having to type specific commands. Easy transitions to the next and previous screens were designed to make navigation intuitive.

6.1.5 Integrating Automation while Preserving Human Control

A user may want to make changes to his/her benchmark, and visualize the results. This would be a routine task, so the design includes a button to automate the compilation of the benchmark, perform batch profiling and gather performance data from colored lockdown with the modified benchmark. We preserve human control by allowing the user to make

changes to this task, a helper shell script `make_helper.sh` allows the user to change parameters as required, for instance the change in arguments in execution of a program or changing the number of samples for batch profiling is performed. Once modified, this is the script that is invoked on pressing the button.

With these design principles, a flowchart(Figure 1) showing the basic features and transitions between the different frames of the GUI, was designed to guide development.

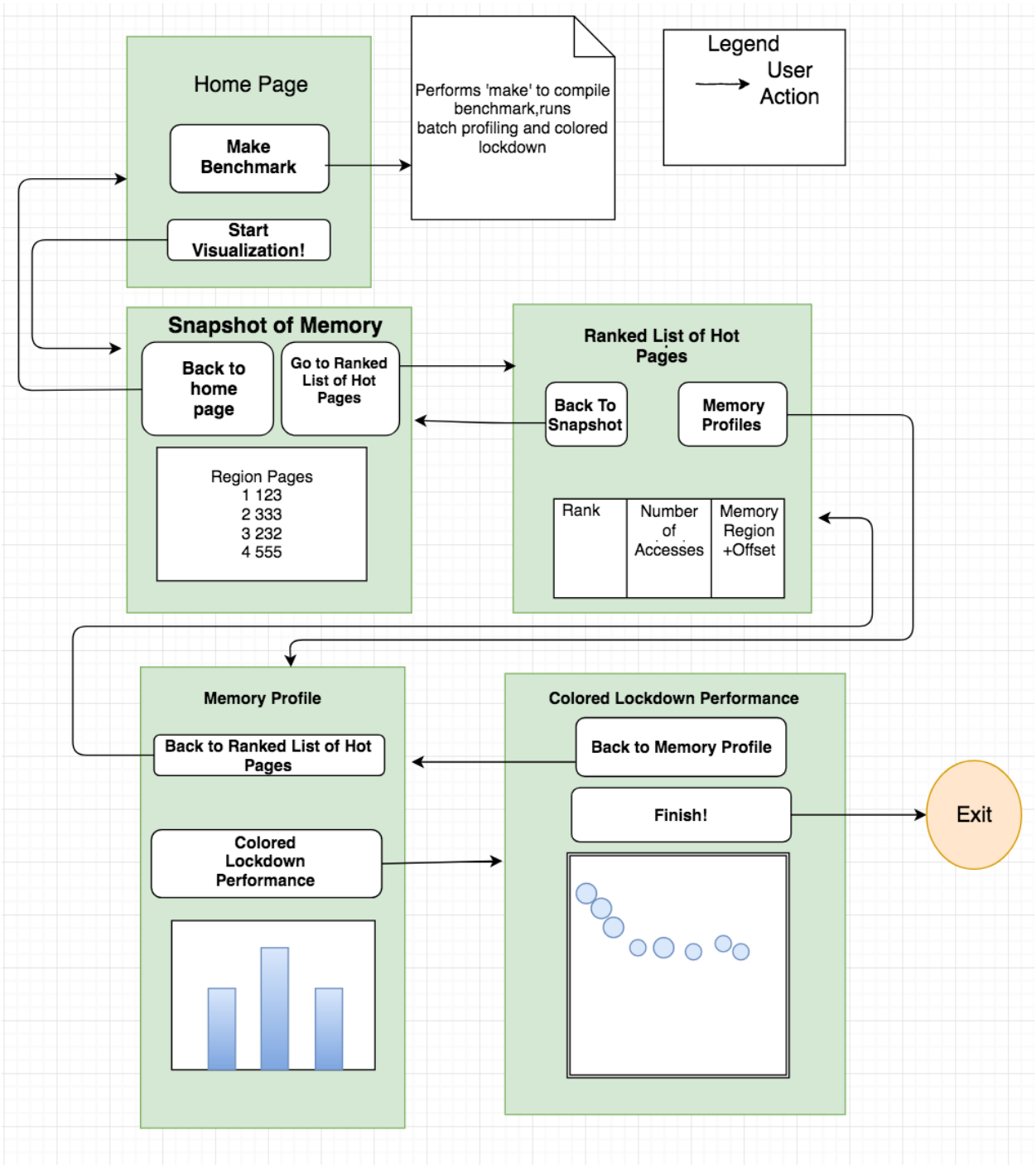


Figure 6.1: Flowchart of the Graphical User Interface showing the basic design

6.2 Development

This sub-chapter serves to highlight the choices made during development and the features of the GUI post-development.

6.2.1 Programming Language and Modules Used

Among several possible options, Python was chosen for the development of the GUI. Python is a scripting language, which works through an interpreter. Python code need not be compiled, this offers the advantage of testing and running code quickly [17]. Thus, shortening development time. The presence of high level native data types such as dictionaries and lists also factored in choosing Python for development as a large amount of data had to be read from files of different formats and organized.

The most important modules to develop the GUI were `Tkinter` and `Matplotlib`[10]. `Tkinter` is a standard GUI package for Python [18]. It has methods for the creation of windows, buttons, scroll bars, menus, progress bars, etc. `Matplotlib`[10] allows us to embed 2-D plots on our GUI.

6.2.2 Model-View-Controller

An Object Oriented approach was employed for the development of the GUI. The Model-View-Controller (MVC) design pattern was used for development of the GUI. The different frames of the GUI were constructed as classes, each with their own functions and members, such as widgets and information about their coordinate placements. These are the "views" that the user sees. The parent class that holds the different views is the "model". The "controller" mediates between the view and the model. The user sees the view and his/her actions cause the controller to manipulate the model and then the view is updated, based on the model.

6.2.3 Responsiveness and Multi-Threading

To ensure that the GUI remains responsive, multi-threading was employed. The GUI works in a loop, waiting for asynchronous user action. Computationally heavy tasks, such as updating a graph were run in a separate thread to ensure that the GUI does not freeze.

6.2.4 Structural Elements and Features of the GUI

The most important structural elements and features of the GUI were:

1. Buttons-Buttons were embedded in all pages (see Figure 6.2,6.3,6.4,6.5,6.6) to perform a transition between neighboring screens. The button labelled `Make Benchmark!` was used for compiling the benchmark and running the L-2 profiler (see Figure 6.2) and the button labelled `See the cache hits on selected memory region` to update the graph (see Figure 6.5).

2. Menu-A menu was embedded with two drop downs (see Figure 6.2, 6.3, 6.4, 6.5, 6.6), an option to exit was provided under the `File` drop-down of the menu and option to transition to any of the screens was provided under the `View` drop-down.

3. Treeview-A treeview was used (see Figure 6.4) to tabulate and display the ranked list of pages in memory regions, and respective associated cache hits. Treeview allows the user to highlight specific rows.

4. Dropdown Menu-For selecting a particular memory region from a list of possible memory regions a drop-down was used (see Figure 6.5).

5. Progress Bars-For time consuming tasks such as preparing benchmarks and updating graph, progress bars (see Figure 6.2,6.5) set in 'indeterminate' mode were used to assure the user that a task was being carried out.

6. Text Widget- A text widget was used to display the snapshot of memory (see Figure 6.2).

7. Embedded Plot and Toolbar for plot-Using the `Matplotlib` module [10], plots were embedded on two screens (see Figure 6.5, 6.6). A toolbar (from this module) with options to zoom in, save the plot, navigate to different parts of the plot and shuttle between different views, was embedded to allow the user to study the graphs in more detail. Bar graphs were used for the memory profile, as bar graphs are an easy way to visualize frequencies (see Figure 6.5), and a scatter plot was used for colored lockdown performance (see Figure 6.6) to allow the user to observe the trend in performance as well as view an individual measurement for a specific case.

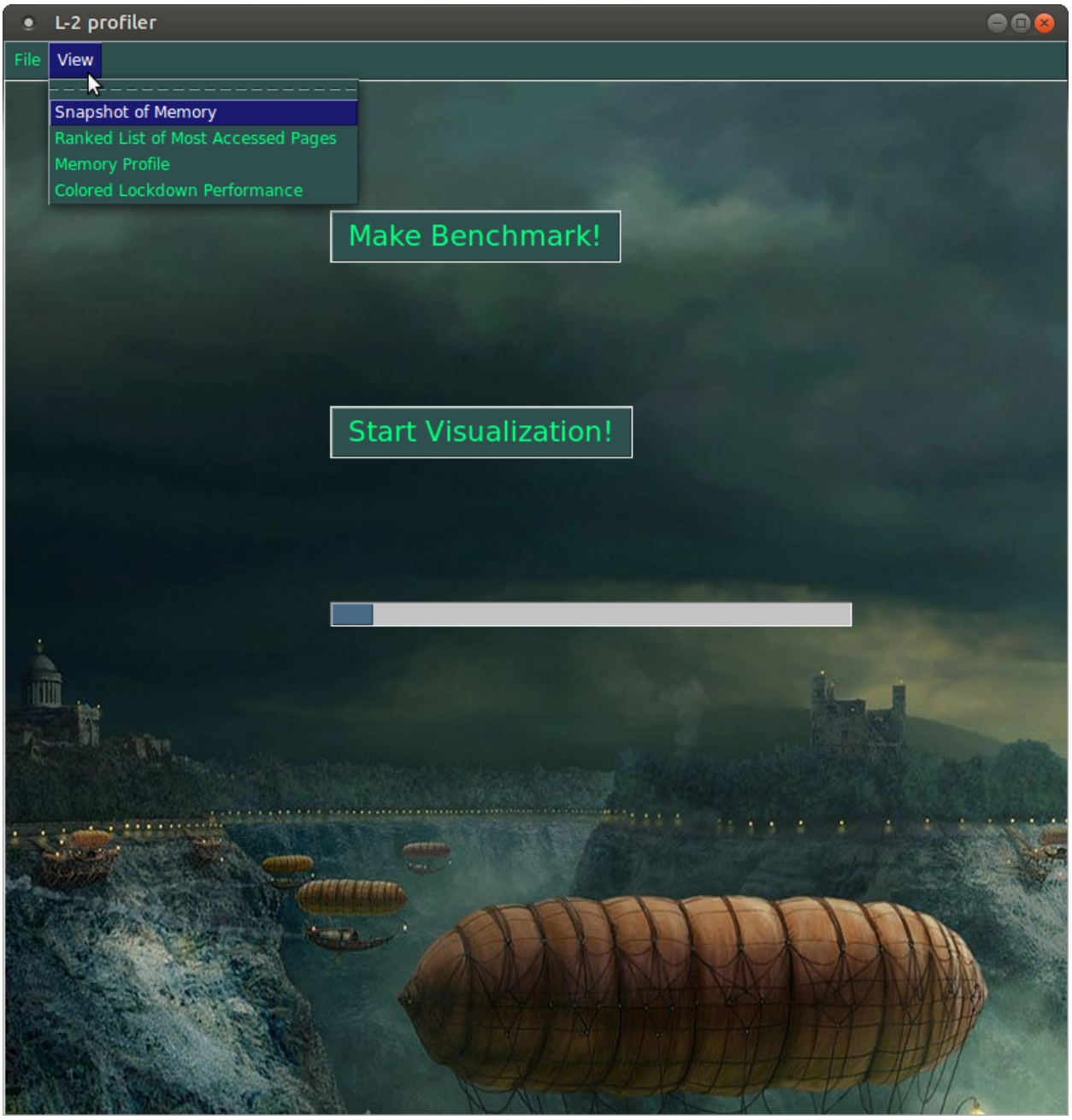


Figure 6.2: home page of the GUI

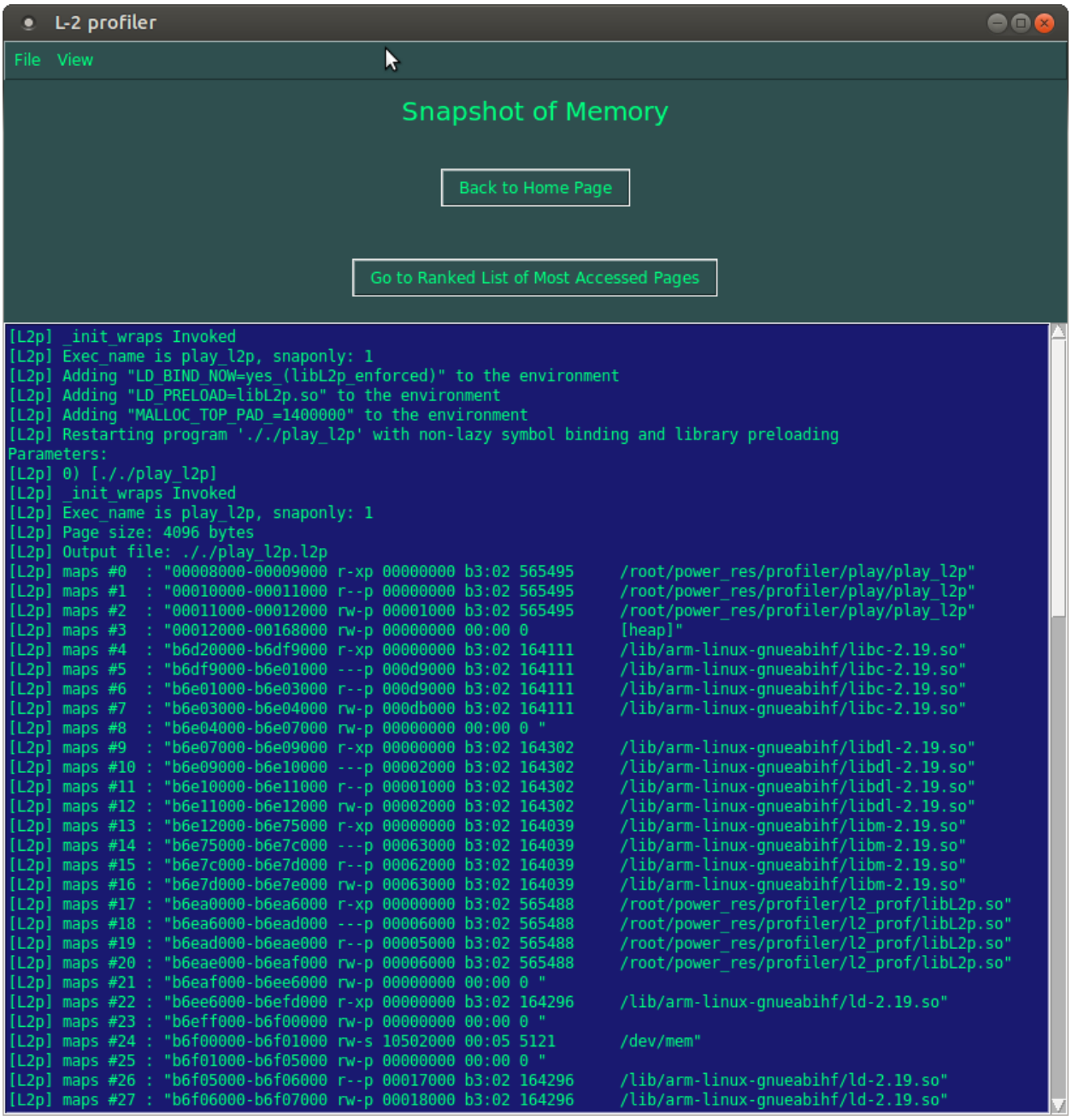
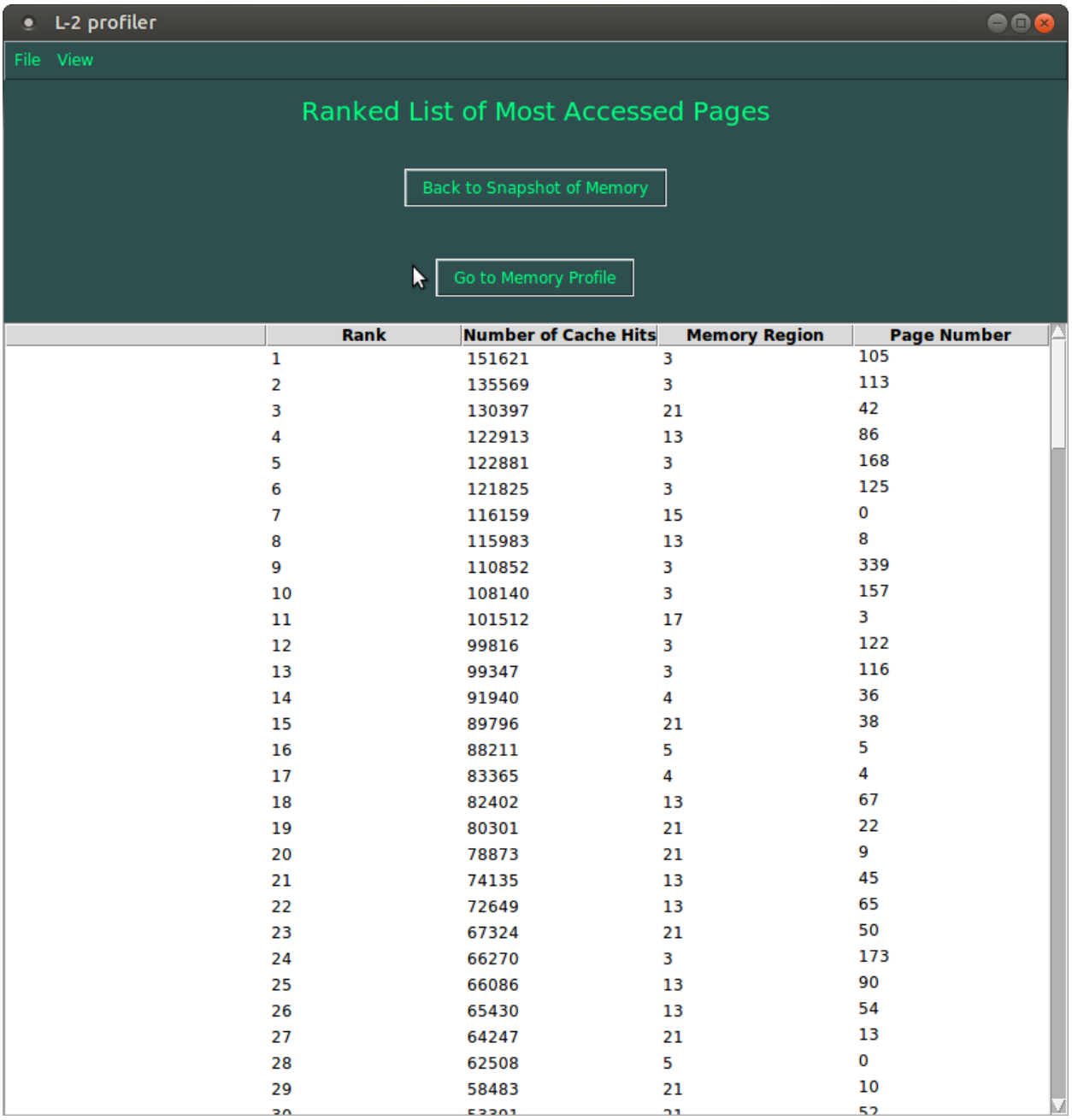


Figure 6.3: second frame of the GUI



	Rank	Number of Cache Hits	Memory Region	Page Number
	1	151621	3	105
	2	135569	3	113
	3	130397	21	42
	4	122913	13	86
	5	122881	3	168
	6	121825	3	125
	7	116159	15	0
	8	115983	13	8
	9	110852	3	339
	10	108140	3	157
	11	101512	17	3
	12	99816	3	122
	13	99347	3	116
	14	91940	4	36
	15	89796	21	38
	16	88211	5	5
	17	83365	4	4
	18	82402	13	67
	19	80301	21	22
	20	78873	21	9
	21	74135	13	45
	22	72649	13	65
	23	67324	21	50
	24	66270	3	173
	25	66086	13	90
	26	65430	13	54
	27	64247	21	13
	28	62508	5	0
	29	58483	21	10
	30	53301	21	52

Figure 6.4: third frame of the GUI

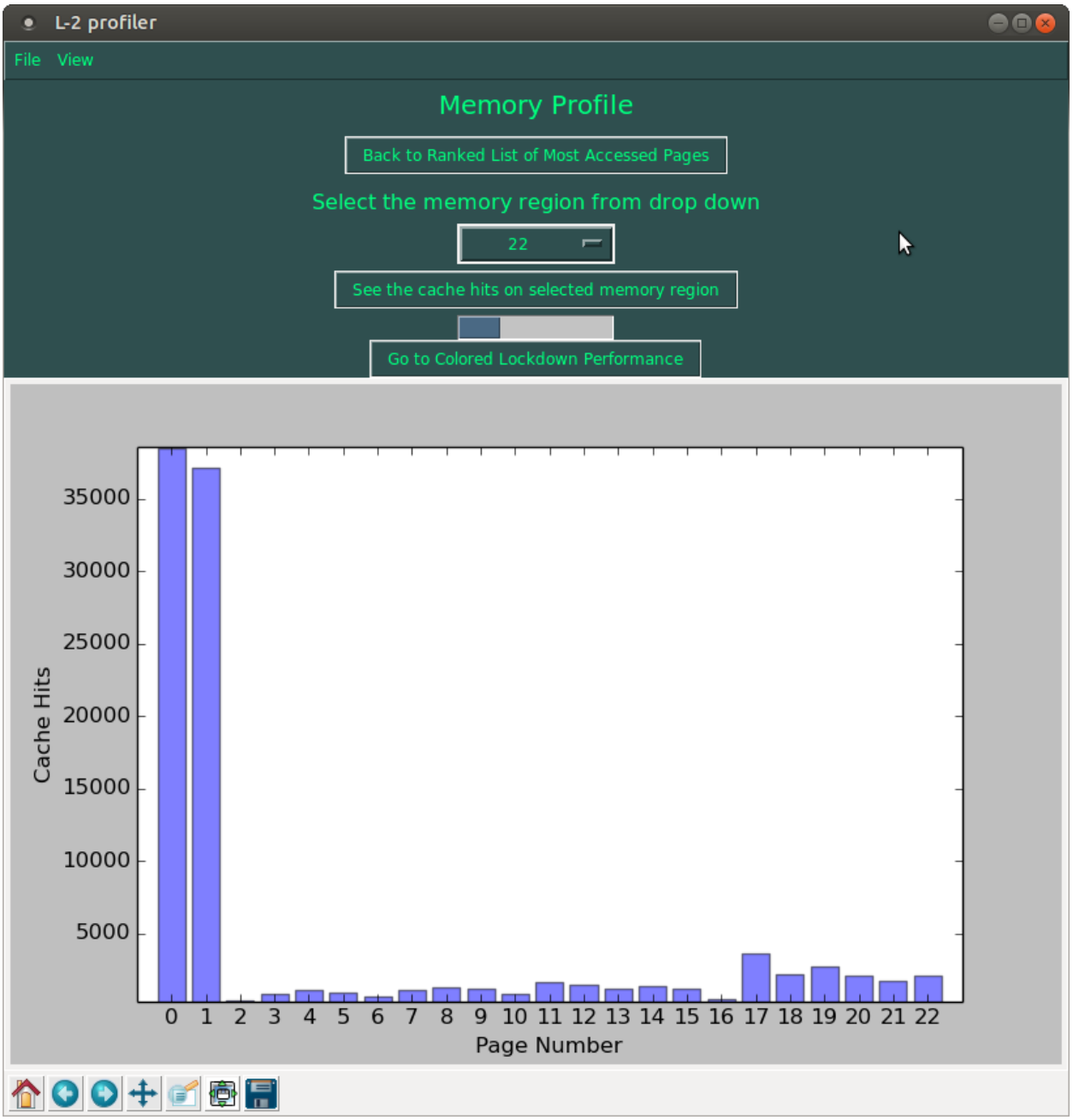


Figure 6.5: fourth frame of the GUI

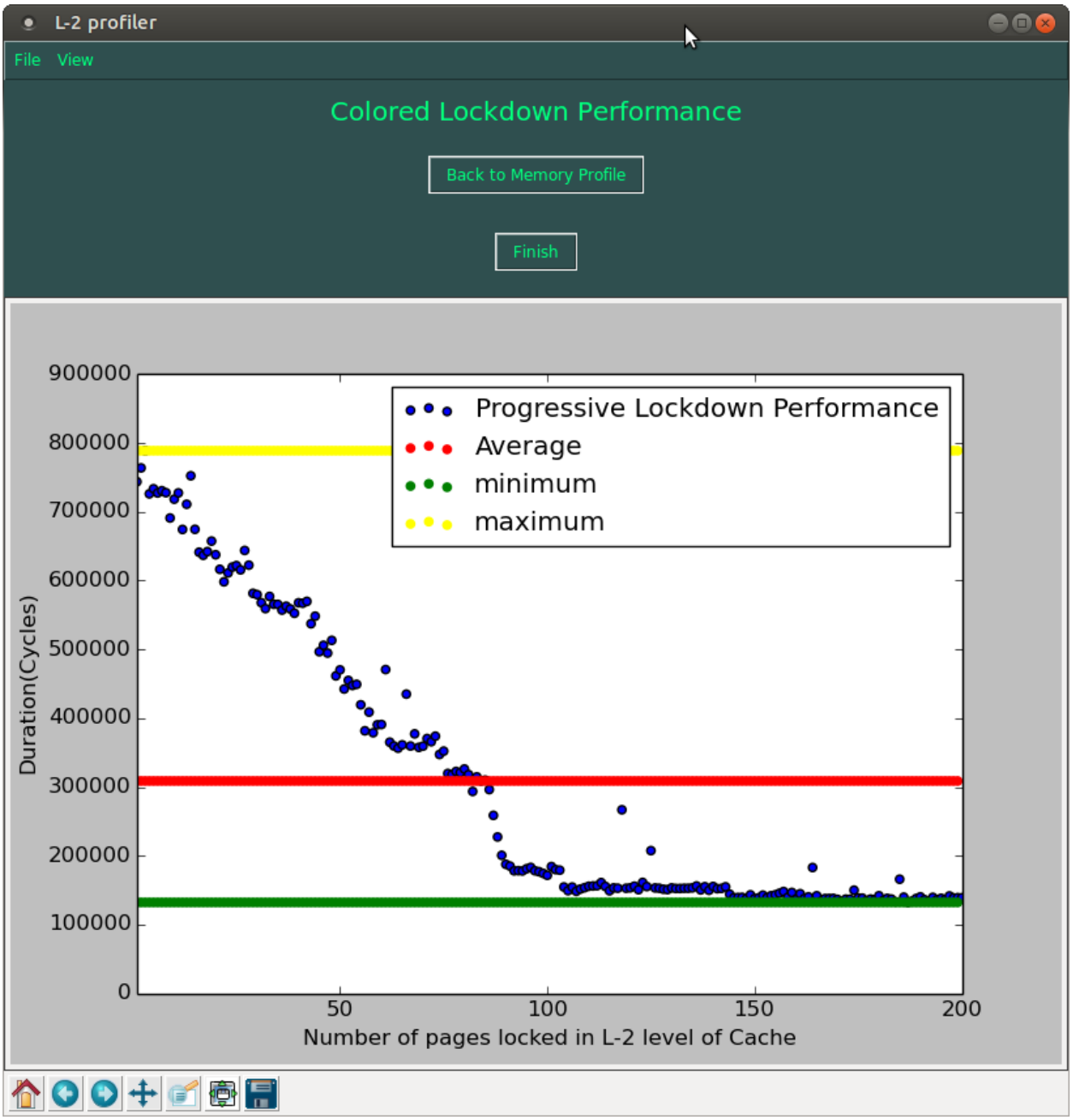


Figure 6.6: fifth frame of the GUI

CHAPTER 7

USER EXPERIENCE STUDY AND EVALUATION OF GUI

Designers, and researchers often seek to measure the usability of a system to evaluate a system. Several researchers have tried to define the term "usability". According to the International Standards Organization, usability is defined as "the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use" [4].

7.1 User Experience Study

Any system is designed, with the end users in mind. It is important to allow end users to test the system and evaluate it based on their feedback. Evaluation of the system can be done, based on metrics that are of importance to the researcher. For our user experience study, we recruited five participants who we believe are representative of the end users of the tool. 80% of the usability issues can be detected with four or five subjects [7]. We evaluate the usability of the tool using the System Usability Scale (SUS) because it correlates well with other subjective usability measures [8]. By SUS we indicate system usability scale. SUS has been used by numerous researchers and in industry to evaluate usability of systems. In addition to this we are interested in the performance, usefulness and ease of use of our GUI. For this purpose, we use our own custom metrics, to give us an insight into some specific factors we think are important.

7.1.1 Methodology

Participants in the user experience study were students attending University of Illinois at Urbana-Champaign (UIUC) between the ages of 19 and 30 with two or more years of programming experience.

A description was given to participants about the framework [1] that the GUI aims to visualize and a synthetic benchmark was already created for the participants to test the GUI on and study the framework. The description provided to the participants was as follows:

"This is a user experience study for a front end tool for a framework for cache management on multi-core systems. The framework first identifies different memory regions and associated pages during the execution of a program, then performs a profiling of the pages of these memory regions in cache i.e. storing them in the L-2 level of cache and measuring the cache hits, finally by a process called 'colored lockdown' it caches some of the pages with the most cache hits to the L-2 level of cache. Your first task is to use this GUI with a given benchmark. Your second task is to modify the benchmark and use this GUI to compile and test the modified benchmark. Please complete the survey form provided to you once you have finished these two tasks."

Participants were allowed to ask questions about the framework and any other general technical questions (not related to the tool) to fill in for any potential knowledge gaps. It is assumed that end users of this tool would already be familiar with the framework and underlying concepts, and use it to visualize results from benchmarks they create. Each participant was provided with an online form to be filled post-completion of their interaction with the tool. The online form consisted of 16 questions. The first 10 questions were adopted from the SUS survey [8]. The last 6 questions were aimed at providing us information about the usefulness of the tool as an educational tool specific to our use case, ease of use for the expected audience as well as seeking to understand universal ease of use, and finally performance of the tool based on responsiveness of the GUI and quality of user interaction.

The 16 questions in the survey were as follows:

Rate the first 10 items, on a scale of 1 to 5, where 1 is 'strongly disagree' and 5 is 'strongly agree'. Rate items 11 to 16, on a scale of 1 to 10, where 1 is 'very poor' and 10 is 'perfect'.

1. I think that I would like to use this GUI again.
2. I found this GUI unnecessarily complex.
3. I thought this GUI was easy to use.
4. I think that I would need assistance to be able to use this GUI.
5. I found the various functions in this GUI were well integrated.
6. I thought there was too much inconsistency in this GUI.
7. I would imagine that most people would learn to use this GUI very quickly.
8. I found this GUI to be very cumbersome/awkward to use.
9. I felt very confident using this GUI.
10. I needed to learn a lot of things before I could get going with this GUI.
11. The usefulness of this tool in studying the described cache management strategy on multicore systems, with a prepared synthetic benchmark.
12. The usefulness of this tool in studying the described cache management strategy on multicore systems, after modifying the synthetic benchmark.
13. The design of the tool considering it is used by a person from a non-computer science background to learn the concepts.
14. The design of the tool considering it is used by a person from a computer science background to learn the concepts.
15. The responsiveness of the tool.
16. The tool does what it is asked to do, through my interactions with it.

7.2 Evaluation

The responses from the survey forms for each participant, were evaluated in the following ways:

- Responses from items 1 to 10, were evaluated based on the SUS metric [8].
- Responses from items 11 and 12 were averaged to provide us with an absolute measure for our 'usefulness' metric. This metric can be considered as a direct measure of usefulness for the end use case, with 1 being very poor and 10 being perfect.
- Responses from items 13 and 14 were averaged to provide us with an absolute measure for our 'ease of use' metric. This metric can be considered as a direct measure of how easy it is to use factoring the end user as well as universal ease of use, with 1 being very poor and 10 being perfect.
- Responses from items 15 and 16 were averaged to provide us with an absolute measure for our 'performance' metric. This metric can be considered as a direct measure of performance of the GUI based on it's responsiveness and the quality of interaction, with 1 being very poor and 10 being perfect.

These metrics from all individual participants were averaged for analysis. The individual System Usability Scale (SUS) scores from the participants are shown in Figure 7.1. The average of the SUS scores from our user experience study was 81. A SUS score of 66 indicates an average usability while a score of 77 and above can be considered to be in the top 75th percentile [9]. Thus, the usability of our tool can considered to be in the top 75th percentile of usability.

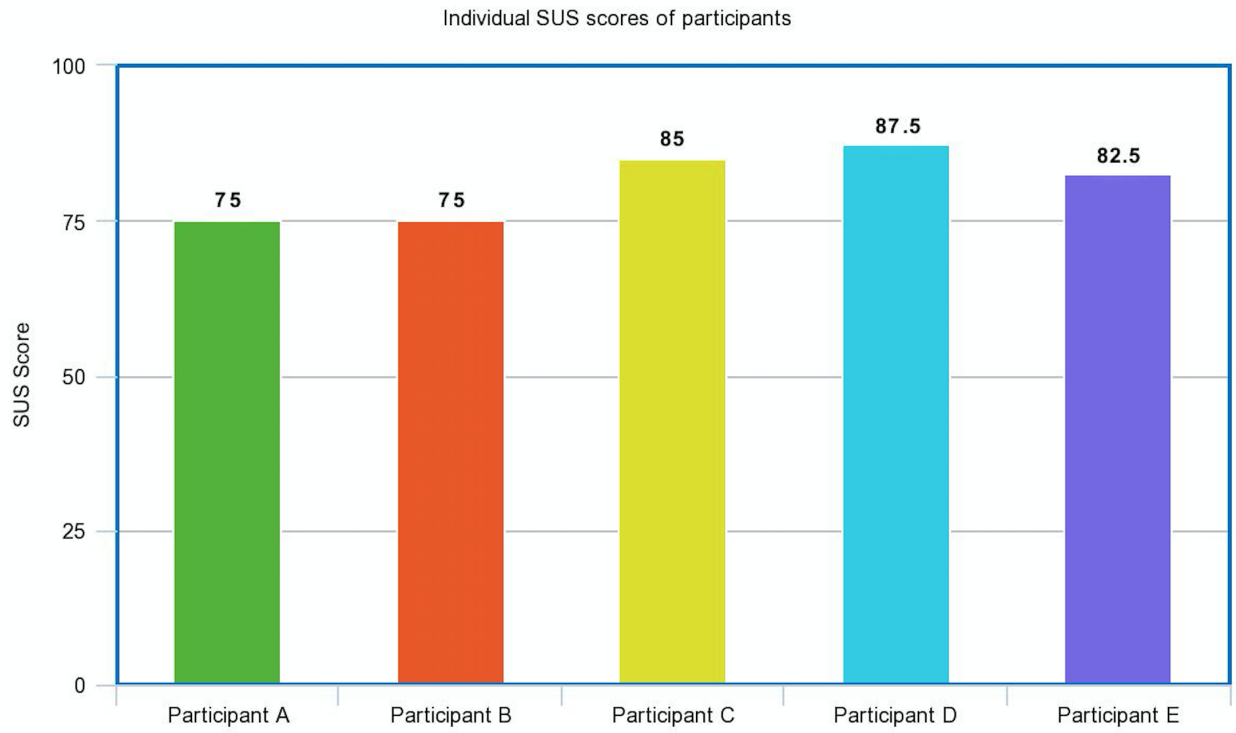


Figure 7.1: Individual SUS scores of the participants

Our custom metric for 'usefulness' received an average score (average of all participants) of 9.3. 'Ease of use' received an 8.9 (average of all participants). Finally, 'performance' received an 8.4 (average of all participants). Hence, we believe that our tool can be used in an educational setting for it's intended purpose.

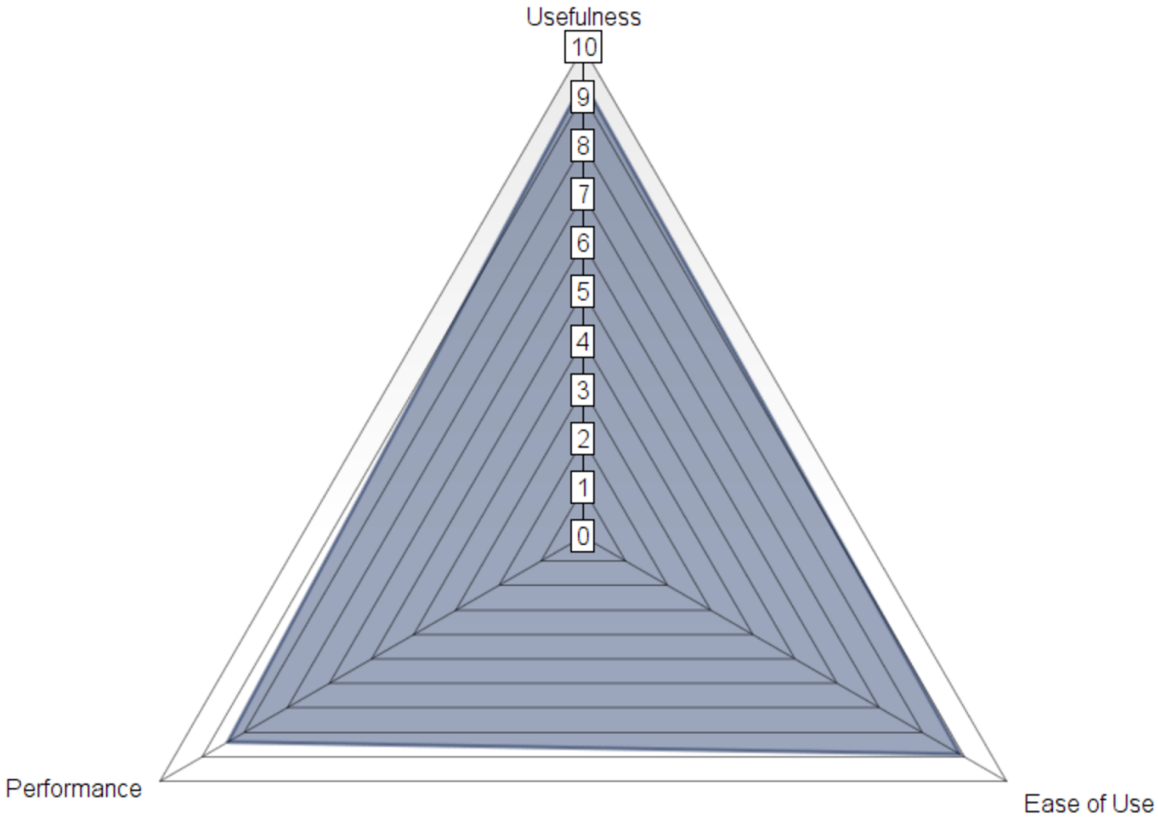


Figure 7.2: radar chart showing the average of results of all participants for our custom metrics for usefulness,performance and ease of use.

CHAPTER 8

RELATED WORK

This chapter serves to highlight some similar GUI tools. The GUI tool perfometer mentioned in [24] is a graphical tool that allows users to monitor events in program execution and facilitates understanding the correlation between the structure of source code and the underlying architecture, to leverage the architecture to it's full potential. The tool also uses plots to visualize performance and allows the user to change parameters as needed. DEEP mentioned in [24] provides a graphical user interface for program structure browsing, performing profiling analysis, and helps visualize profiling results in relation to source code. Both of the mentioned tools work with the PAPI library and have a different end use-case.

CHAPTER 9

CONCLUSION

Thus, we present a highly usable front-end GUI tool which can be used with synthetic and custom benchmarks to study the framework [1], in an educational setting. Further, we propose a mechanism to synchronize cores on Symmetric Multiprocessing (SMP) systems in which there is an absence of the option to perform cache 'lockdown by master'.

REFERENCES

[1] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, "Real-time cache management framework for multi-core architectures," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*. IEEE, 2013, pp. 45–54.

[2] P. J. Denning, "The locality principle," *Communications of the ACM*, vol. 48, no. 7, pp. 19–24, 2005.

[3] B. Jacob, S. Ng, and D. Wang, *Memory Systems: Cache, DRAM, Disk*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[4] W. Iso, "9241-11. ergonomic requirements for office work with visual display terminals (vdts)," *The international organization for standardization*, vol. 45, 1998.

[5] B. Shneiderman, *Designing the user interface: strategies for effective human-computer interaction*. Pearson Education India, 2010.

[6] J. D. Foley, V. L. Wallace, and P. Chan, "The human factors of computer graphics interaction techniques," *IEEE computer Graphics and Applications*, vol. 4, no. 11, pp. 13–48, 1984.

[7] R. A. Virzi, "Refining the test phase of usability evaluation: How many subjects is enough?" *Human factors*, vol. 34, no. 4, pp. 457–468, 1992.

[8] J. Brooke et al., "Sus-a quick and dirty usability scale," *Usability evaluation in industry*, vol. 189, no. 194, pp. 4–7, 1996.

[9] W. Albert and T. Tullis, *Measuring the user experience: collecting, analyzing, and presenting usability metrics*. Newnes, 2013.

[10] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing In Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[11] "Odroid-x product page with specifications," http://www.hardkernel.com/main/products/prdt_info.php?g_code=G133999328931, accessed: 2017-06-30.

[12] "Secured shell protocol," https://www.ssh.com/ssh/protocol/, accessed: 2017-06-30.

[13] "Minicom page from ubuntu help," https://help.ubuntu.com/community/Minicom, accessed: 2017-06-30.

[14] "Make manual," https://www.gnu.org/software/make/manual/make.html, accessed: 2017-07-05.

[15] "Gcc online documentation," https://gcc.gnu.org/onlinedocs/gcc/, accessed: 2017-07-05.

[16] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on.* IEEE, 2001, pp. 3–14.

[17] P. K. Reed, "A comparison of programming languages for graphical user interface programming," 2002.

[18] "Tkinter documentation," https://docs.python.org/2/library/tkinter.html, accessed: 2017-07-05.

[19] J. A. Stankovic and K. Ramamritham, "What is predictability for real-time systems?" *Real-Time Systems*, vol. 2, no. 4, pp. 247–254, 1990.

[20] D. J. Sorin, M. D. Hill, and D. A. Wood, "A primer on memory consistency and cache coherence," *Synthesis Lectures on Computer Architecture*, vol. 6, no. 3, pp. 1–212, 2011.

[21] "Blog - linux kernel address space," https://hungys.xyz/linux-kernel-process-address-space/, accessed: 2017-07-11.

[22] "Memory management in linux," http://www.makelinux.net/ldd3/chp-15-sect-1, accessed: 2017-07-11.

[23] D. Bovet and M. Cesati, *Understanding The Linux Kernel.* Oreilly & Associates Inc, 2005.

[24] K. S. London, J. Dongarra, S. Moore, P. Mucci, K. Seymour, and T. Spencer, "End-user tools for application performance analysis using hardware counters." in *ISCA PDCS*, 2001, pp. 460–465.

[25] "Access time wikipedia page," https://en.wikipedia.org/wiki/Access_time, accessed: 2017-07-11.

[26] "Usb-uart module kit page," http://www.hardkernel.com/main/products/prdt_info.php?g_code=G134111883934, accessed: 2017-07-11.

APPENDIX A: IRB APPROVAL FORM FOR RECRUITING HUMAN PARTICIPANTS

The User experience study is aimed at evaluating the tool and not human participants. The following page shows approval from the Office for the Protection of Research Subjects, UIUC for recruiting human participants.

UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN

Office of the Vice Chancellor for Research

Office for the Protection of Research Subjects
805 West Pennsylvania Avenue
Urbana, IL 61801



June 30, 2017

Suraj Venkat
Computer Science
201 N Goodwin Avenue
Urbana IL 61801

RE: *Development and Evaluation of Graphical User Interface and Benchmark Creation for Cache Management on Multi-Core Systems*
IRB Protocol Number: 17671

Dear Mr. Venkat:

Thank you for submitting the completed University of Illinois at Urbana-Champaign IRB (IRB) Application form for your project entitled *Development and Evaluation of Graphical User Interface and Benchmark Creation for Cache Management on Multi-Core Systems*. Your project was assigned IRB Protocol Number 17671 and reviewed. You will recruit volunteers to compile code then test out your visualization tool (an on screen Graphical User Interface) and then rate the usefulness of the tool. Their feedback isn't really 'about' them but rather the entire focus is on evaluating the visualization tool.

It has been determined that this project as described does not meet the definition of human subjects research as defined in 45CFR46(d)(f) or at 21CFR56.102(c)(e) and does not require IRB approval.

This determination only applies to the research study as submitted. Please note that modifications to your project need to be submitted to the IRB for review and status determination or approval before the modifications are initiated.

We appreciate your commitment to university policies and regulations regarding human research. If you have any questions about the IRB process, or if you need assistance at any time, please feel free to contact me at the OPRS office, or visit our website at http://www.oprs.research.illinois.edu.

Sincerely,

Ronald A Banks

Ronald Banks, MS, CIP
Human Subjects Research Coordinator, Office for the Protection of Research Subjects

U of Illinois at Urbana-Champaign • IORG0000014 • FWA #00008584
telephone (217) 333-2670 • *fax* (217) 333-0405 • *email* IRB@illinois.edu