CS810: Advanced Computer Architecture

Assignment 1

15 August 2023

Shashank P (200010048), Tabish Khalid Halim (200020049)

The objective of this assignment was to use the **Intel Pin** tool to analyze the instruction mix of **five benchmarks** from **SPEC CPU 2017**. We have targeted the following instructions:

1. Integer Arithmetic
2. Floating-Point Arithmetic
3. Load Instructions
4. Store Instructions
5. Unconditional Branches
6. Conditional Branches

Then we used **Linux Perf Tool** to measure and plot run-time metrics for every benchmark with an interval of **60 ms**. The metrics measured are shown below

1. Instructions Per Cycle (IPC)
2. Branch Prediction Accuracy
3. Number of Accesses to L1, L2 and L3 caches
4. Hit rates of L1, L2 and L3 caches
5. Number of memory requests
6. Power Consumed by Processor

1 SPEC CPU 2017 Benchmarks

The SPEC (Standard Performance Evaluation Corporation) CPU 2017 benchmarks are industry-standardized, CPU-intensive software for measuring and comparing compute-intensive performance, stressing a processor, memory subsystem, and compiler. The benchmarks were designed to provide a wide range of coverage in hardware using real user applications. Details of **5** benchmarks used are shown below

1.1 GCC_R

This benchmark is based on GCC version **4.5.0**. It compiles a given program with a lot of optimizations enabled. Since these benchmarks are to stress the CPU, all the inputs are large files instead of many small files. It has a tendency to access a **lot of memory**

1.2 MCF_R

This benchmark implements an **algorithm to schedule single-depot vehicles** for public transportation. The program is written in C. The algorithm used is the network simplex algorithm which is a specialized version of the **simplex algorithm** for network flow problems. It uses concepts such as graph cycles and spanning trees. It is heavy on **Integer Arithmetic Operations**.

1.3 NAMD_R

This benchmark is derived from another benchmark for parallel simulation of large bio-molecular systems. Most of the logic involves **calculating inter-atomic interactions**. It is designed to scale over $200,000$ cores. The input to this simulation contains 92224 atoms.

1.4 LBM_R

This program implements the **Lattice Boltzmann Method**. This program simulates in-compressible **fluids** in a **3D model**. The code extensively uses macros.

1.5 XALANCBMK

The benchmark is written in C++. It is a modified version of **XALAN C++**, used to convert **XML,XLST document to HTML,CSS**. It takes in an **input file of around 100MB** and prints the converted **HTML** into the terminal.

2 Part A: Intel Pin Tool

2.1 CPU Specifications

CPU details are as follows:

```
AddressWidth=64
Architecture=9
Availability=3
Caption=Intel64 Family 6 Model 140 Stepping 1
ConfigManagerErrorCode=
ConfigManagerUserConfig=
CpuStatus=1
CreationClassName=Win32_Processor
CurrentClockSpeed=2419
CurrentVoltage=7
DataWidth=64
Description=Intel64 Family 6 Model 140 Stepping 1
DeviceID=CPU0
ErrorCleared=
ErrorDescription=
ExtClock=100
Family=205
InstallDate=
L2CacheSize=5120
L2CacheSpeed=
LastErrorCode=
Level=6
LoadPercentage=25
Manufacturer=GenuineIntel
MaxClockSpeed=2419
Name=11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz
OtherFamilyDescription=
PNPDeviceID=
PowerManagementCapabilities=
PowerManagementSupported=FALSE
ProcessorId=BFEBFBFF000806C1
ProcessorType=3
Revision=
Role=CPU
SocketDesignation=U3E1
Status=OK
StatusInfo=3
Stepping=
SystemCreationClassName=Win32_ComputerSystem
SystemName=LAPTOP-OMB0A1T9
UniqueId=
UpgradeMethod=1
Version=
VoltageCaps=
```

2.2 Commands and Metrics

We have made a bash file to auto-run the commands for the benchmark as follow:

```
sudo bash commands.sh
```

```
    #!/bin/bash

# Hardcoded "name, command" pairs
commands=(
    "gcc ./502.gcc_r/cpugcc_r_base.mytest-m64 ./502.gcc_r/scilab.c -O3 -finline-limit=0 -fif-conversion -fif-conversion2 -o ./502.gcc_r/scilab.opts-O3_-finline-limit_0_-fif-conversion_-fif-conversion2.s"
    "mcf ./505.mcf_r/mcf_r_base.mytest-m64 ./505.mcf_r/inp.in"
    "namd ./508.namd_r/namd_r_base.mytest-m64 --input ./508.namd_r/apoa1.input --iterations 5"
    "lbm ./519.lbm_r/lbm_r_base.mytest-m64 300 ./519.lbm_r/lbm.in 0 0 ./519.lbm_r/100_100_130_cf_a.of"
    "xalancbmk ./523.xalancbmk_r/cpuxalan_r_base.mytest-m64 -v ./523.xalancbmk_r/allbooks.xml ./523.xalancbmk_r/xalanc.xsl"
)

count=1
# Loop through the hardcoded commands
for cmd in "${commands[@]}"; do
    # Extract name and command using awk
    name=$(echo "$cmd" | awk '{print $1}')
    command=$(echo "$cmd" | awk '{$1=""; print}' | xargs)

    # Print the name and command
    echo "Running $name..."
    echo "Command: $command"

    ./pin-3.28-98749-g6643ecee5-gcc-linux/pin -t ./pin-3.28-98749-g6643ecee5-gcc-linux/source/tools/SimpleExamples/obj-intel64/opcodemix.so -- $command
    ./pin-3.28-98749-g6643ecee5-gcc-linux/pin -t ./pin-3.28-98749-g6643ecee5-gcc-linux/source/tools/SimpleExamples/obj-intel64/ldstmix.so -- $command
    ./pin-3.28-98749-g6643ecee5-gcc-linux/pin -t ./pin-3.28-98749-g6643ecee5-gcc-linux/source/tools/ManualExamples/obj-intel64/inscount1.so -- $command

    python3 getDynamic.py opcodemix.out inscount.out ldstmix.out $count
    ((count+=1))
done
```

The Intel Pin Tools used for analysis were opcodemix, inscount & ldstmix. Opcodemix allowed for data abstraction of all dynamic instructions in a benchmark. Inscount allowed to count the total number of instructions executed and the Ldstmix allowed abstraction of data related to memonry read and write. To complete the objective, we have used a python script that abstracts information from the .so files generated by Intel Pin tools.

2.3 Plots

Symbol	Representation
Int	Arithmetic operations based instructions
Float	Float operations based instructions
Cond_branch	Conditional Branch instructions
Uncond_branch	Unconditional Branch instructions
Load	Load based instructions
Store	Store based instructions

The following are the opnames that were used to classify the instructions :

```
1    cond_jumpinstr = ["JA","JAE","JB","JBE","JBE","JC","JCXZ","JE","JECXZ","JG
    ","JGE","JLE","JNAE","JNB","JNBE","JNC","JNE","JNG","JNGE","JNL","JNLE","
    JNO","JNP","JNS","JNZ","JO","JP","JPE","JPO","JS","JZ"]
2    jumpinstr = ["JMP"]
```

2.3.1 gcc

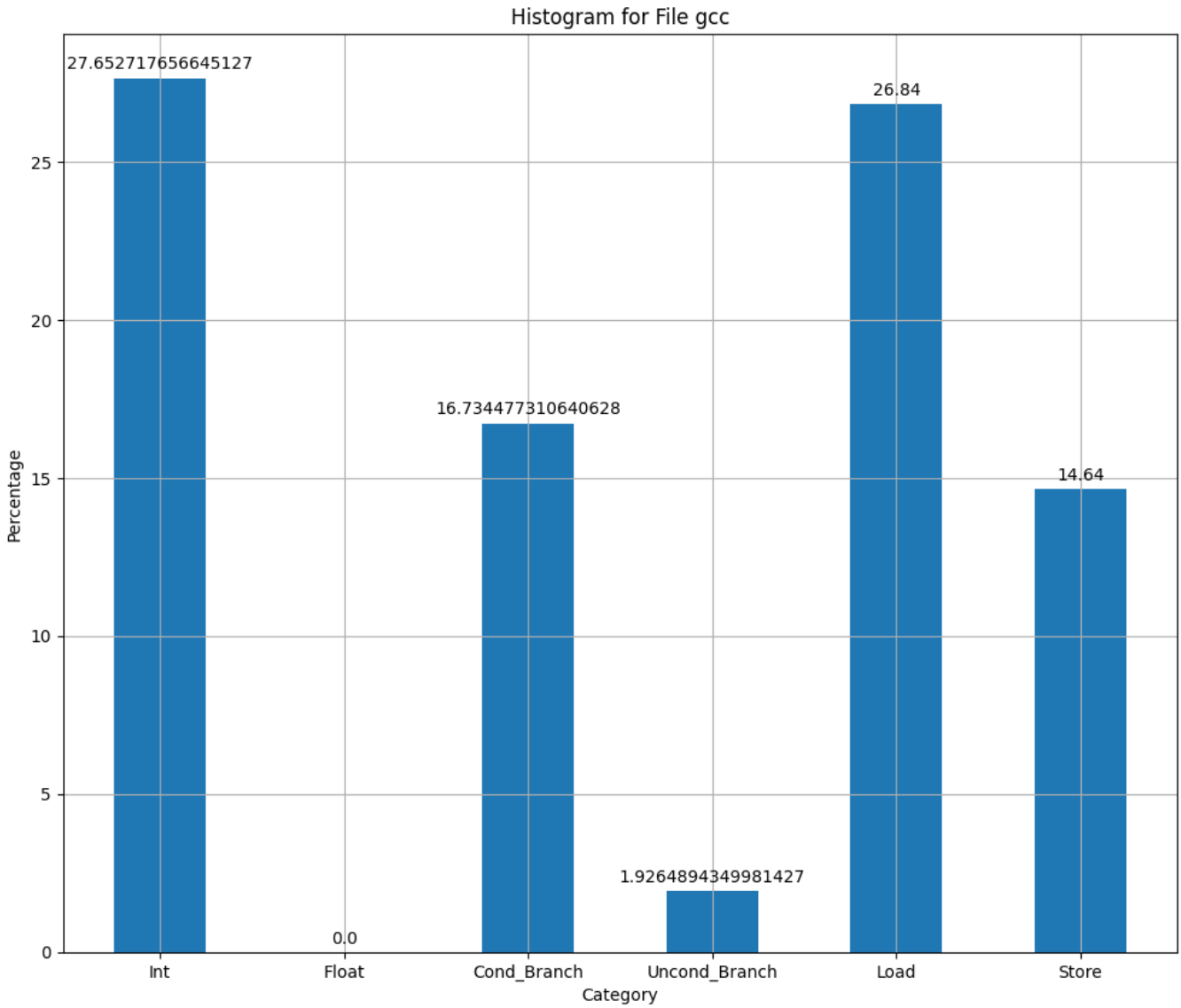

Figure 1: Plot for gcc benchmark

2.3.2 mcf

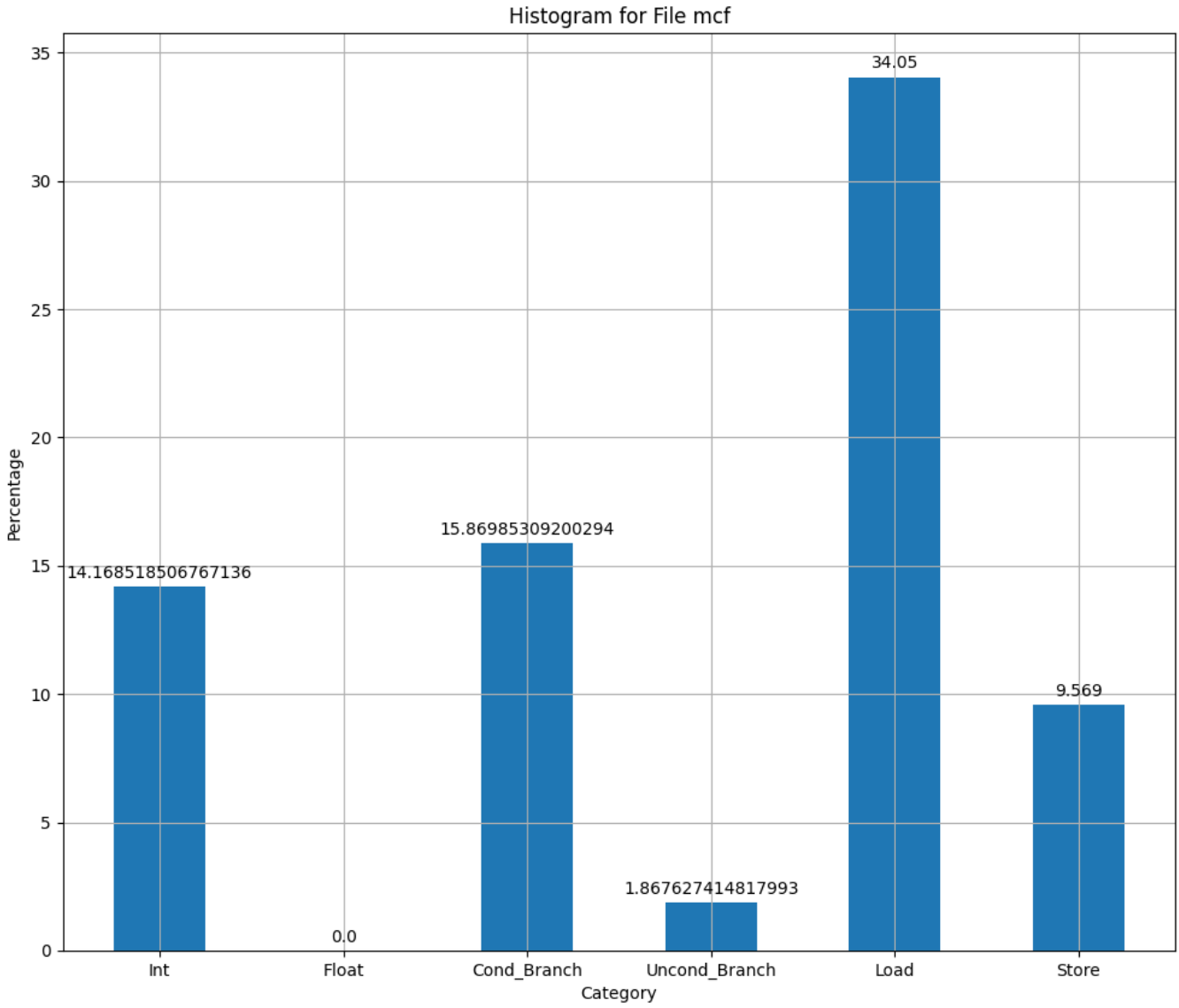

Figure 2: Plot for mcf benchmark

2.3.3 namd

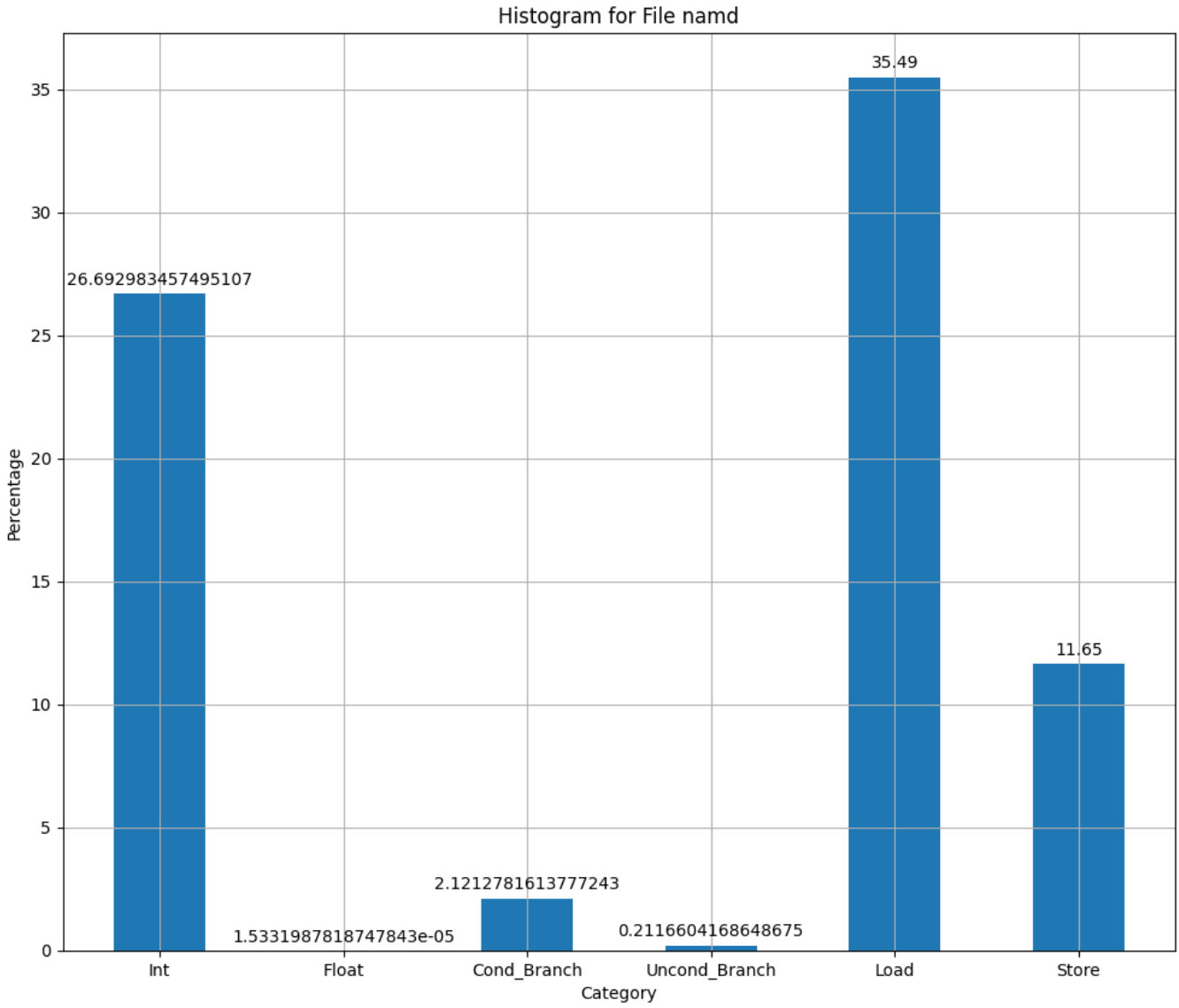

Figure 3: Plot for namd benchmark

2.3.4 LBM

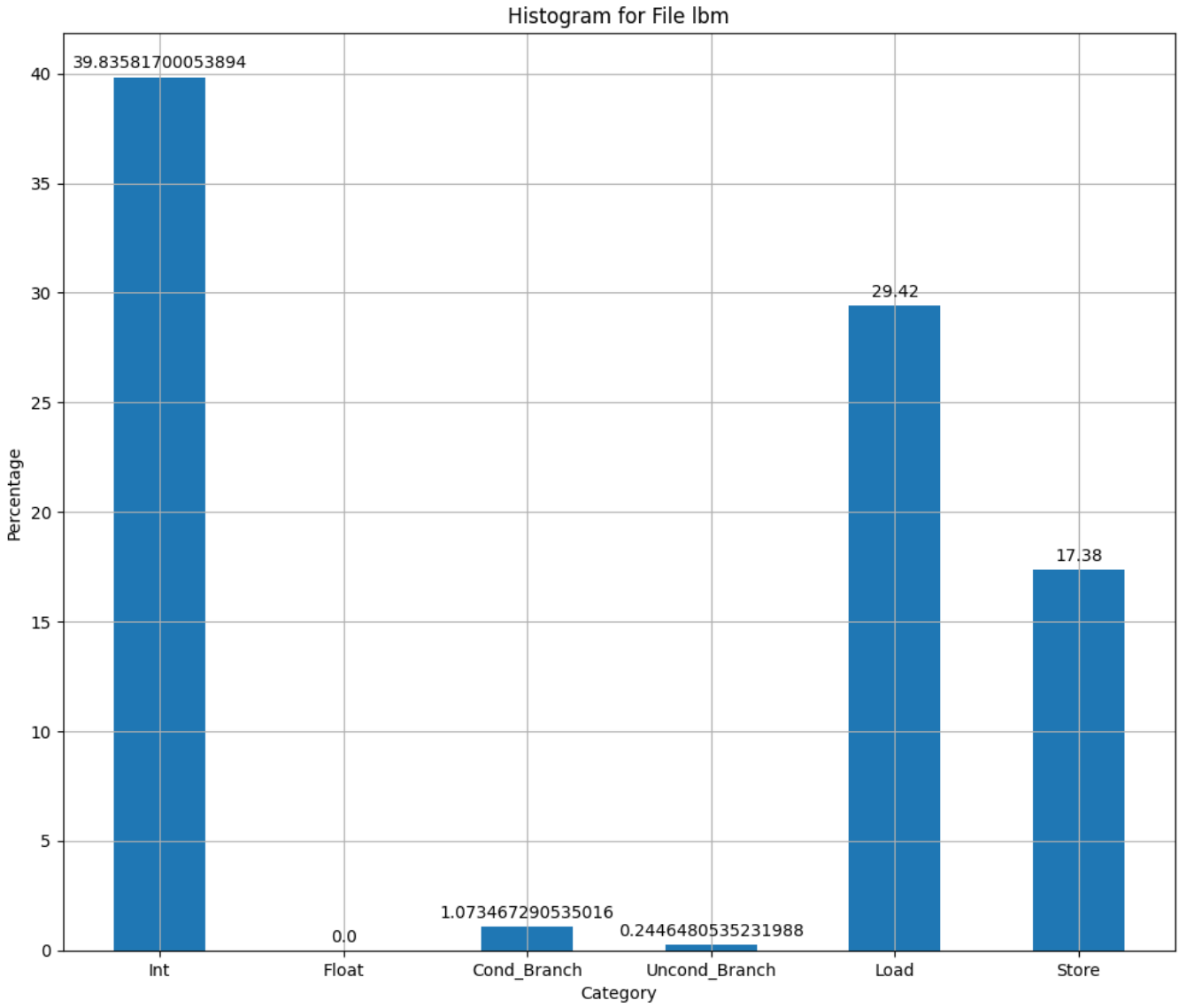

Figure 4: Plot for lbm benchmark

2.3.5 xalancbmk

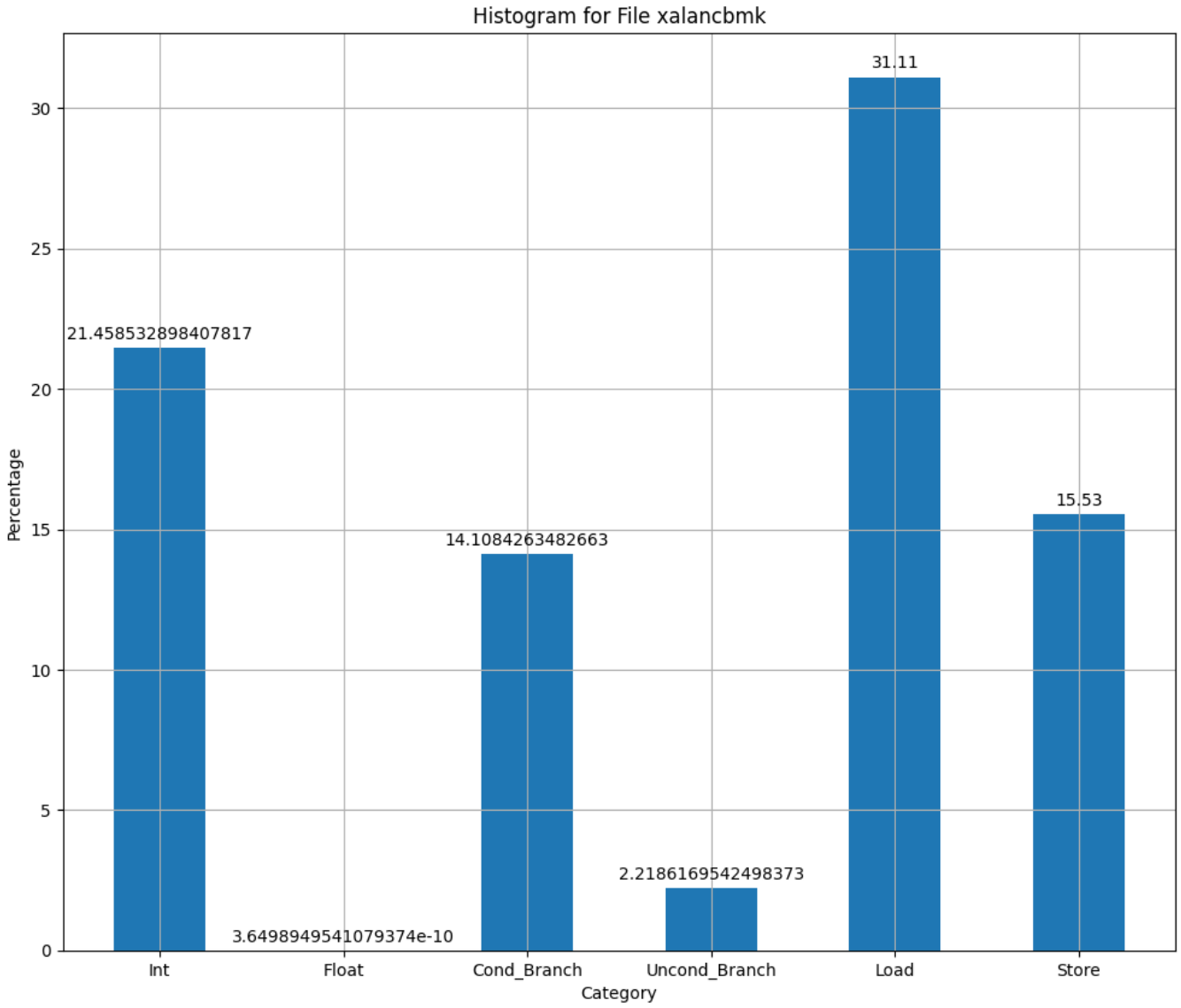

Figure 5: Plot for xalancbmk benchmark

2.4 Observations

- Load-based instructions constitute a substantial portion of the ISA instructions compared to other types of instructions. This might be because load-based instructions often involve fetching data from memory, which is a common operation in various applications and benchmarks.
- Float-based instructions are relatively sparse, forming the smallest portion of the instruction set. Many benchmarks and applications may not heavily rely on floating-point calculations, leading to a lower count of float-based instructions.
- The number of instructions linked to conditional branches consistently surpasses that of unconditional branches. Programs frequently involve decision-making and branching based on conditions, resulting in a higher occurrence of conditional branch instructions.
- Among the benchmarks, xalancbmk and namd stand out due to their significant utilization of floating-point instructions. Applications like xalancbmk (XSLT processor) and namd (molecular dynamics simulator) often involve complex mathematical computations using floating-point arithmetic.
-

Benchmark	Observation
gcc	CPU intensive & frequent access of memory
mcf	CPU intensive with huge arithmetic operations
lbm	CPU intensive & large number of memory accesses
namd	CPU intensive & frequent access of memory along with floating operations
xalancbmk	CPU intensive along with floating operations

3 Part B: Linux Perf Tool

The Linux **perf** tool is used to profile the execution of a program using performance counters. It provides a mapping between existing hardware counters and reads them into an output file during execution. Each type of counter is called an **event**. For example, the metric **IPC** can be calculated using the events **Instructions** and **Cycles**.

Note: There maybe some counters supported by **Linux Perf**, but **not in the CPU**

3.1 CPU Specifications

The CPU details on which the benchmarks were run with perf are shown below

Model: Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz (6 Phy. 12 Logical Cores)
Caches: L1 (6 nos.): 192 KB, L2 (6 nos.): 1.5 MB, L3 (1 nos.): 12MB

3.2 Commands and Metrics

Before running the perf tool on benchmarks we observed that keeping the measurement interval too small like **10 ms**, didn't give the counters enough time to fetch and gave **<not counted>** as output. Therefore through trial and error an interval of **60ms** was set. The following bash file was used to automate the run and all the events used and shown

```
#!/bin/bash

# Hardcoded "name, command" pairs
commands=(
    "gcc ./502.gcc_r/cpugcc_r_base.mytest-m64 ./502.gcc_r/scilab.c -O3 -finline-limit=0 -fif-conversion -fif-conversion2 -o ./502.gcc_r/scilab.opts-O3_-finline-limit_0_-fif-conversion_-fif-conversion2.s"
    "mcf ./505.mcf_r/mcf_r_base.mytest-m64 ./505.mcf_r/inp.in"
    "namd ./508.namd_r/namd_r_base.mytest-m64 --input ./508.namd_r/apoa1.input --iterations 5"
    "lbm ./519.lbm_r/lbm_r_base.mytest-m64 300 ./519.lbm_r/lbm.in 0 0 ./519.lbm_r/100_100_130_cf_a.of"
    "xalancbmk ./523.xalancbmk_r/cpuxalan_r_base.mytest-m64 -v ./523.xalancbmk_r/allbooks.xml ./523.xalancbmk_r/xalanc.xsl"
)

# Loop through the hardcoded commands
for cmd in "${commands[@]}"; do
    # Extract name and command using awk
    name=$(echo "$cmd" | awk '{print $1}')
    command=$(echo "$cmd" | awk '{$1=""; print}' | xargs)

    # Print the name and command
    echo "Running $name..."
    echo "Command: $command"
    perf stat -I $1 -e mem-loads,mem-stores,mem_inst_retired.any,branch-instructions,branch-misses,instructions,cycles,L1-dcache-loads,L1-dcache-load-misses,L2-loads,L2-load-misses,L2-stores,L2-store-misses,LLC-loads,LLC-load-misses,LLC-stores,LLC-store-misses $command 2>$name-math.count
    perf stat -I $1 -e power/energy-cores/ $command 2>$name-phy.count
done
```

This will generate **count files** which is then **parsed using python** and appropriate equations and used to get required metrics from counts

Note: The L1 store hits and misses were **not supported** on my hardware. Therefore the hit rate is based on only **load instructions**

3.3 Plots

The plots are shown below with appropriate headings and captions. The x and y axis is mentioned in the plots. To reduce unwanted noise a window average was taken with **8%** windows size

3.3.1 Branch Prediction

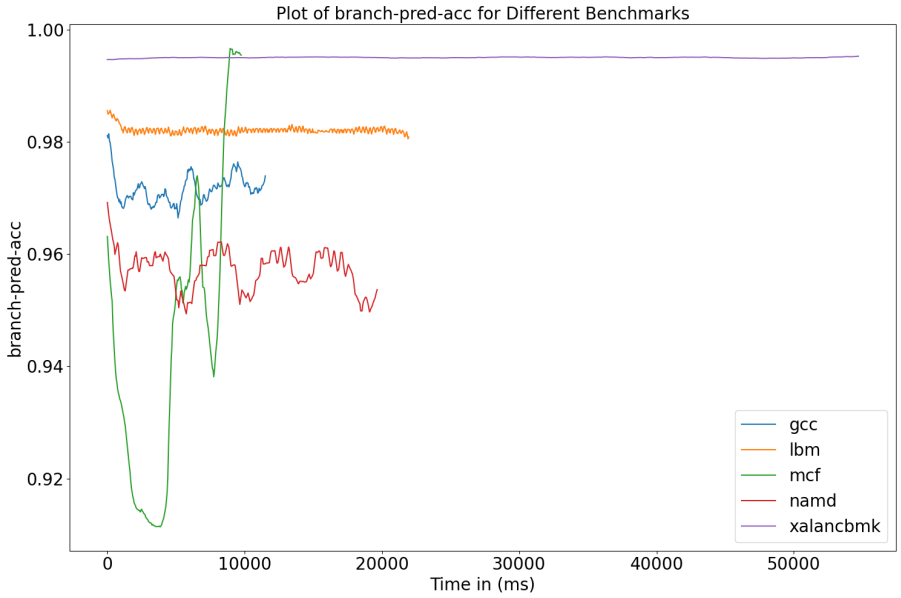

Figure 6: Plot of Branch Prediction Acc. Fraction vs Time

3.3.2 Instructions per Cycle

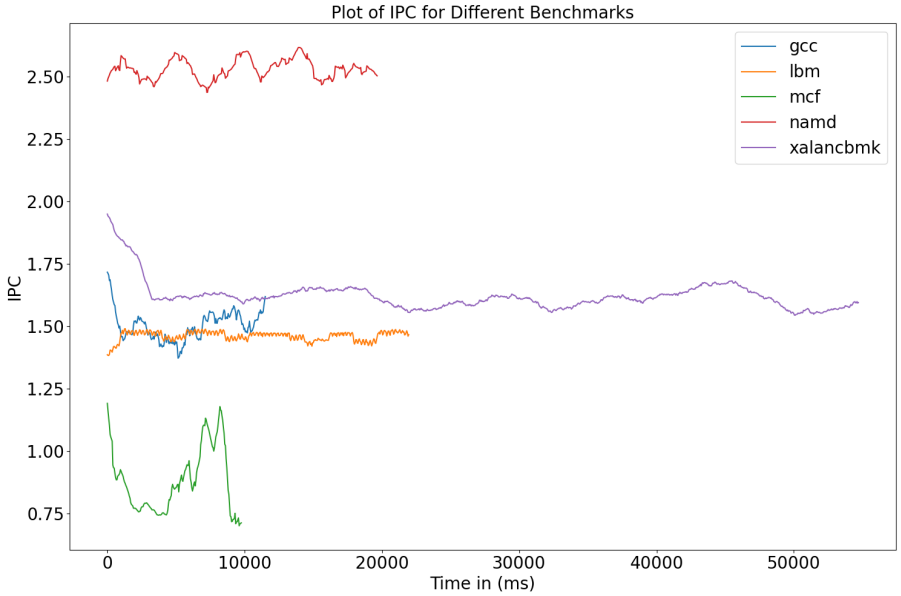

Figure 7: Plot of IPC vs Time

3.3.3 L1 Cache Accesses

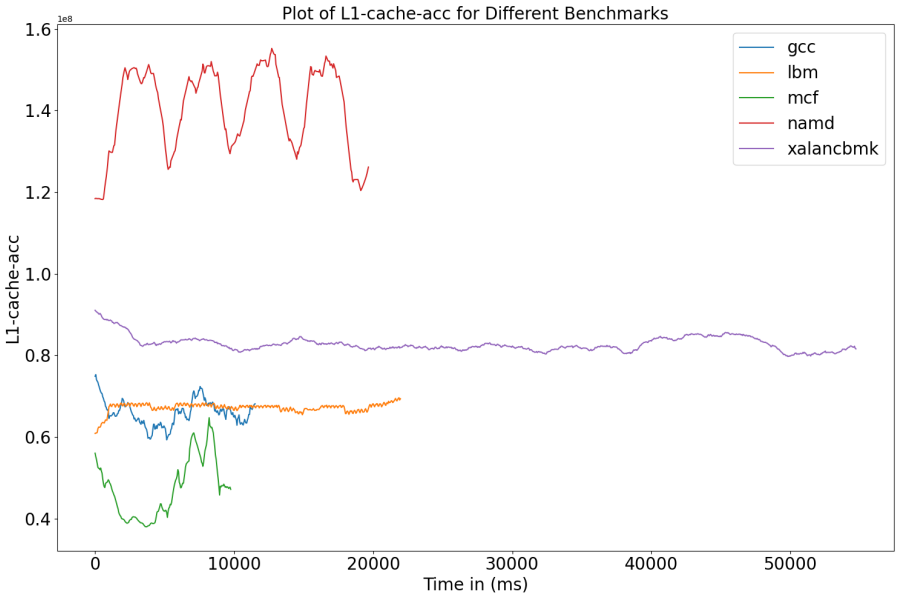

Figure 8: Plot of L1 Cache Accesses vs Time

3.3.4 L2 Cache Accesses

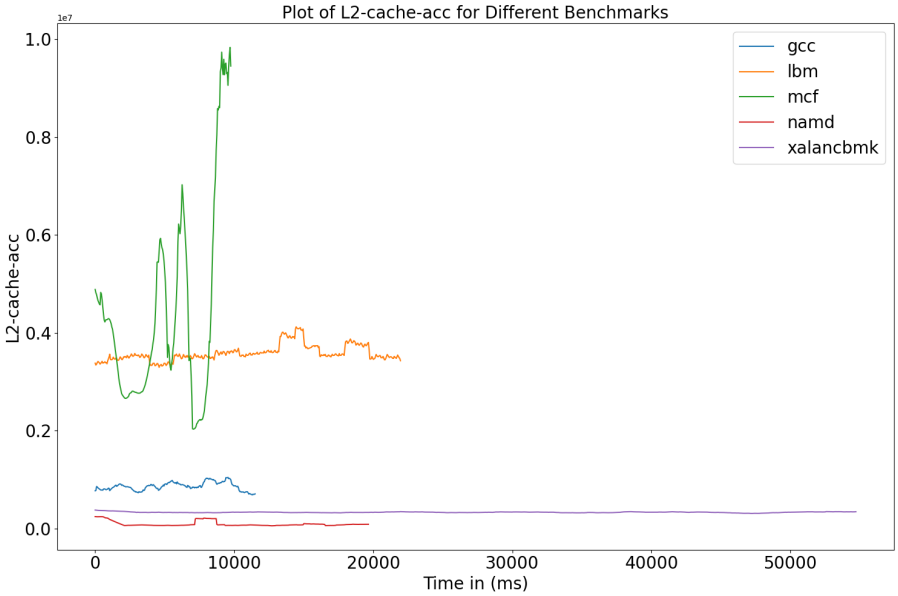

Figure 9: Plot of L2 Cache Accesses vs Time

3.3.5 L3 Cache Accesses

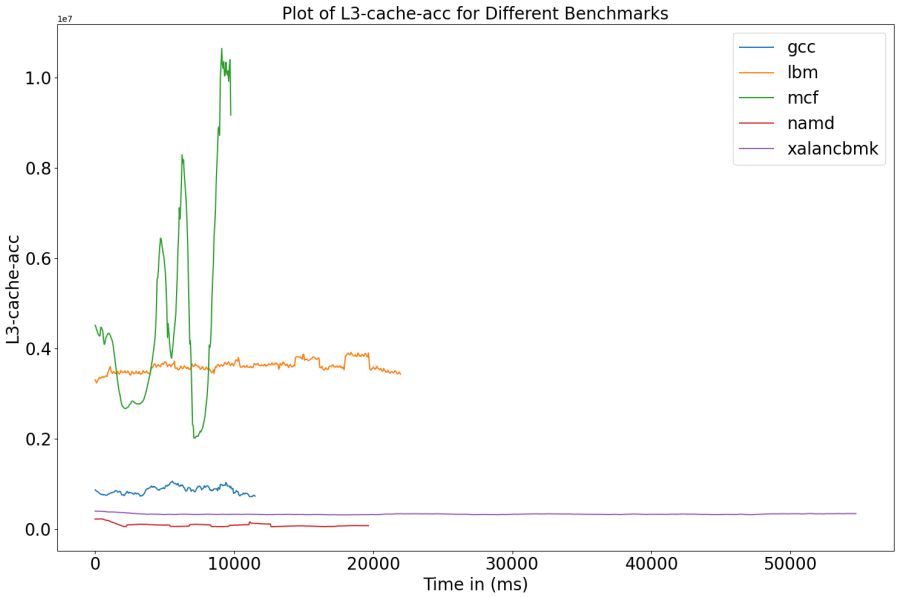

Figure 10: Plot of L3 Cache Accesses vs Time

3.3.6 L1 Hit Rate

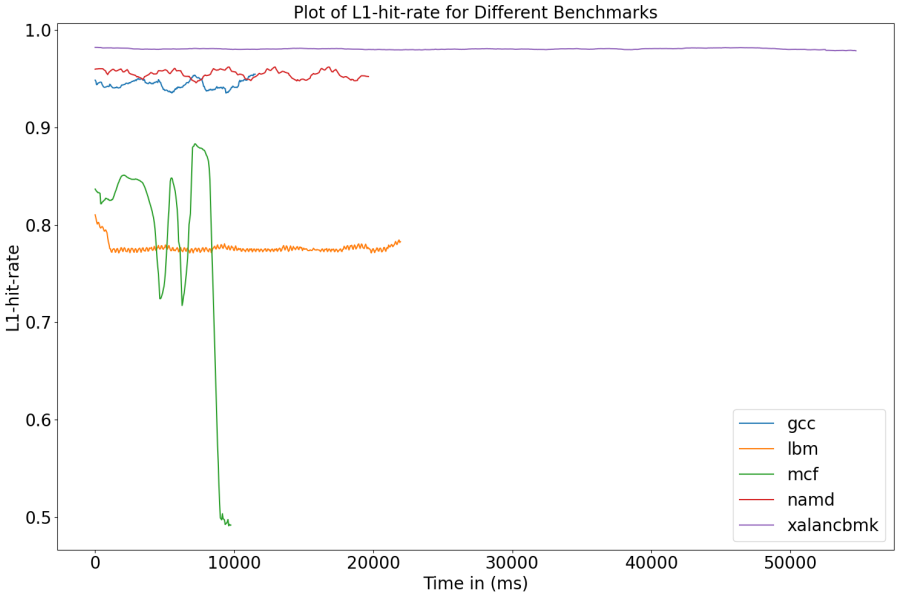

Figure 11: Plot of L1 Hit Rate Fraction vs Time

3.3.7 L2 Hit Rate

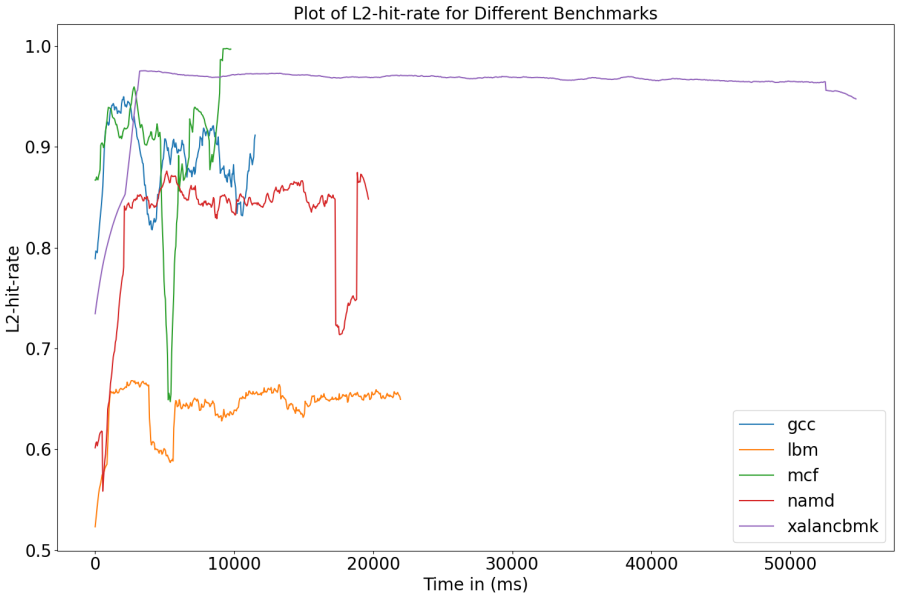

Figure 12: Plot of L2 Hit Rate Fraction vs Time

3.3.8 L3 Hit Rate

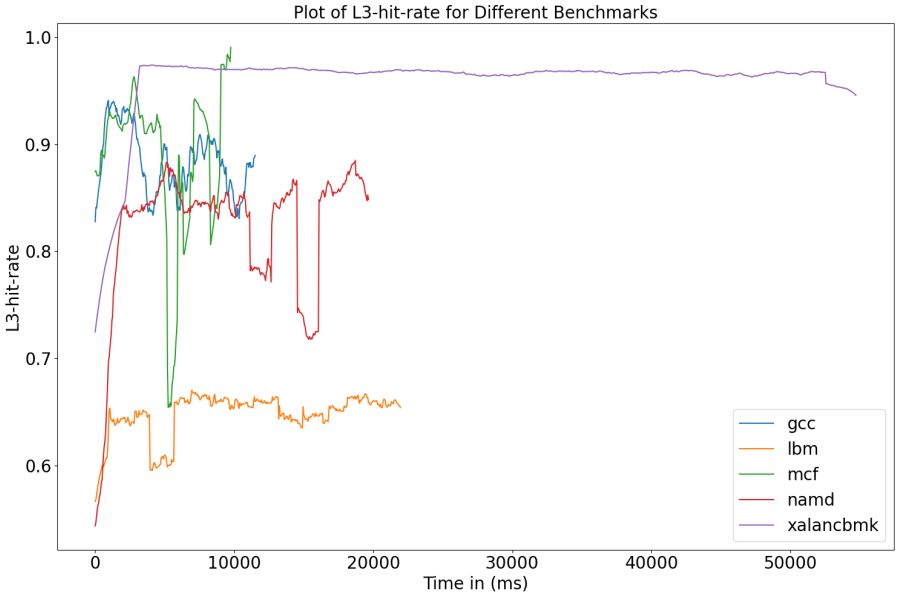

Figure 13: Plot of L3 Hit Rate Fraction vs Time

3.3.9 Memory Access Count

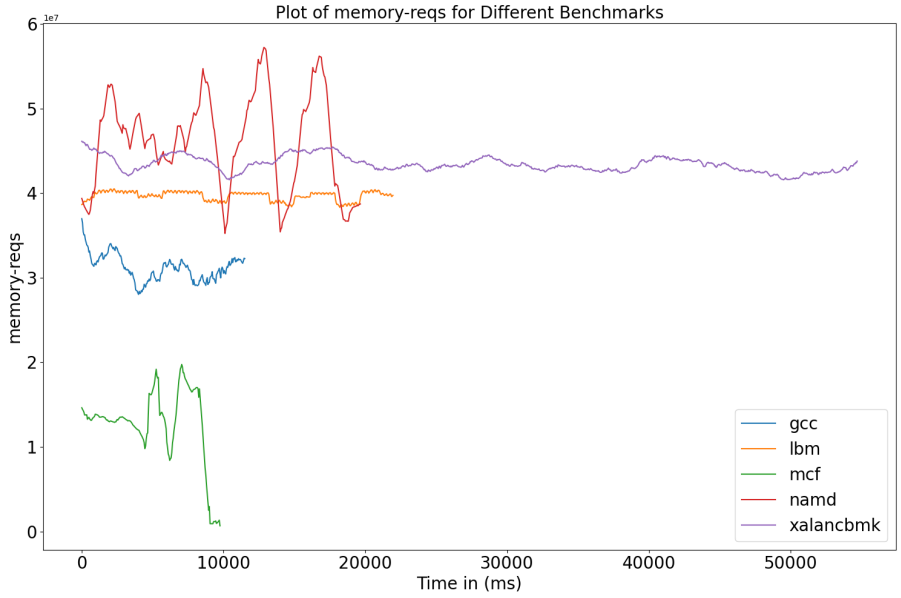

Figure 14: Plot of Memory Accesses vs Time

3.3.10 Power Consumed

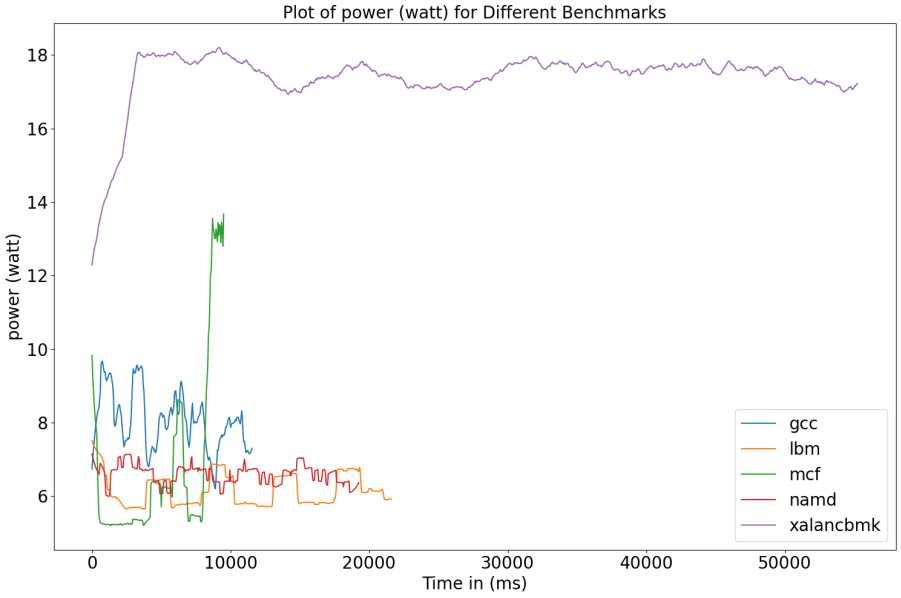

Figure 15: Plot of Power Consumed (watt) vs Time

3.4 Observations

After looking at the plots the following observations can be made,

- For **MCF** benchmark the initial branch prediction and cache hit is low, but suddenly it shoots up. This might be because the benchmark sets up a graph-like network using pointers that do not have regularity. It then applies the simplex algorithm which is based on linear algebra and therefore more predictable and spacially local
- If we look at the benchmark **NAMD**, most of the plots have a kind of periodic oscillation of fixed interval. If we look at the parameters of the benchmark we have '*–iterations 5*'. Therefore it might have some sort of work repeated **5 times**. For example, one iteration involves first calculating inter-atomic interactions and then the next step to update
- Among the benchmarks **XALANCBMK** benchmark has the least variations. This might be because it has only one functionality as a whole which is repeated multiple times unlike the other 4. This might also be the reason for the very high cache hit. This benchmark also uses the most power from the CPU

4 Conclusion

Intel Pin Tool allows us to read, modify and insert instructions at run-time programmatically. It is a very powerful tool to analyze the execution of a program on a given CPU. Linux Perf Tool allows us to measure CPU performance metrics in a very minute detail. It supports hundreds of hardware counters. Together they can help us to better optimize our software or hardware for increased performance.