CS810: Advanced Computer Architecture

Assignment 2

22 August 2023

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

The objective of this assignment was to use the **Intel Pin** tool to perform a data dependency characterization of the different SPEC CPU2017 workloads. We have targeted the following dependencies:

RAW: Read After Write

WAR: Write After Read

WAW: Write After Write

SL: Load After Store

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.

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

Table 1: Benchmarks and their nature

2 Approach

2.1 Algorithm

To implement the **Dependency Detector** algorithm, we have first used the following data structures:

Frequency-Map: This stores a map of **Dependency Distance** ($j - i$) to the number of instances of this distance (For all RAW, WAR, WAW and SL)

History-Map: This stores a map of all registers and memory locations to the last instruction that it was read or written to.

At each step, we compare the current instruction operands with **History-Map** and update the **Frequency-Map** accordingly. Then for all registers or memory locations that are read from or written to, the **History-Map** is updated. All these take place after every instruction dynamically.

2.2 Sampling

On checking the total number of instructions per benchmark, it was found that, on average, the number of dynamic instructions was of the order of 10^{11} (100 billion)! It would be infeasible to run the custom handler after every instruction. Therefore the following sampling algorithm was used:

1. Sample 10^6 (1 Million) instructions starting from the current instruction
2. Do nothing for 10^9 (1 Billion) instructions
3. Clear the **History-Map** that was defined in the previous subsection
4. Repeat from Step 1

3 Commands

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

```
bash copy.sh
```

```
    # Copy and Build
version=pin-3.28-98749-g6643ecee5-gcc-linux
current=$(pwd)
cp ./dpdcount.cpp ~/$version/source/tools/ManualExamples/dpdcount.cpp
cp ./makefile.rules ~/$version/source/tools/ManualExamples/makefile.rules
cd ~/$version/source/tools/ManualExamples
make

home=$(eval echo ~)
```

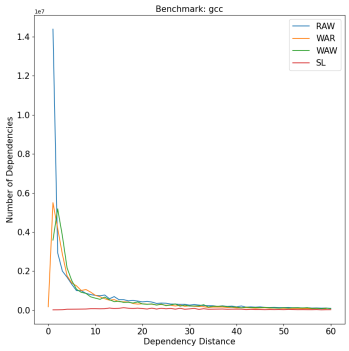
```
tool="$home/pin-3.28-98749-g6643ecee5-gcc-linux/source/tools/ManualExamples/
    obj-intel64/dpdcount.so"

pin="$home/pin-3.28-98749-g6643ecee5-gcc-linux/pin"
resource="$home/cs810_resources/CPU2017_benchmarks/linux_executables"


# Go to resources
eval cd $resource
pwd

# Initializing all the commands
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 ""
    echo "--------------------------------"
    echo "$pin -t $tool -- $command"
    echo "--------------------------------"
    echo ""

    $pin -t $tool -- $command 2> $current/err/$name.err

    # Copy the output to current directory
    mv dpdcount.out $current/data/$name.count
    ((count+=1))
done
```

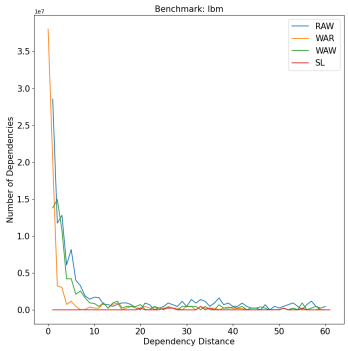
This will generate **dependency count files**, which is then copied and **parsed using python** and graphs are plotted with various **x** and **y** axis metrics.

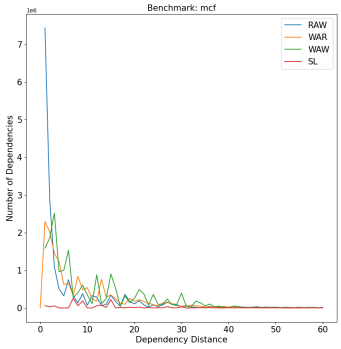
4 Plots

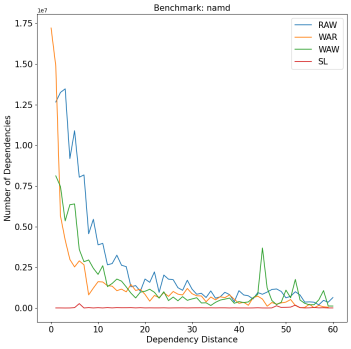
The plots are shown below with appropriate headings. The x and y axes and legends are mentioned in the plots.

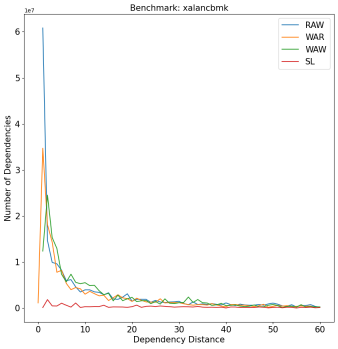
4.1 Dependency Distance vs Number of Dependencies Per Benchmark



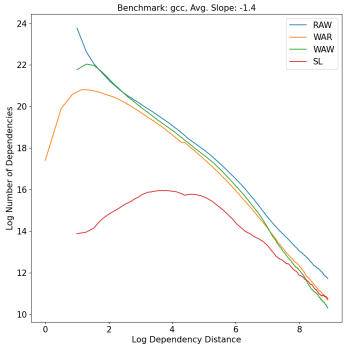


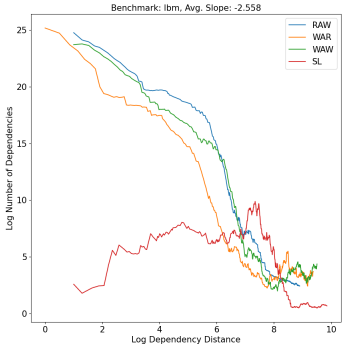


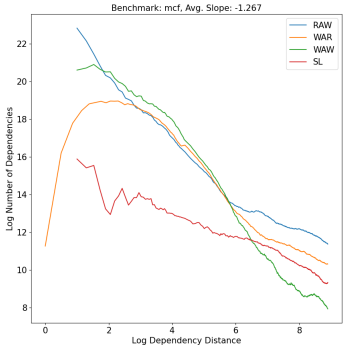


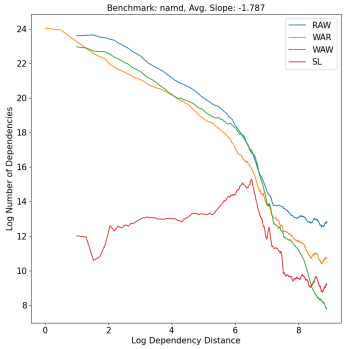


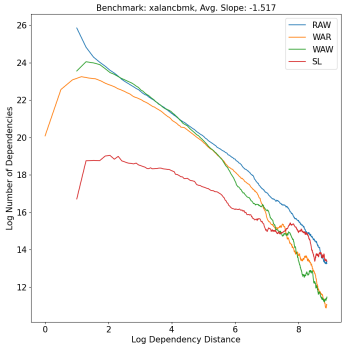
4.2 Log (Dependency Distance) vs Log (Number of Dependencies) per Benchmark



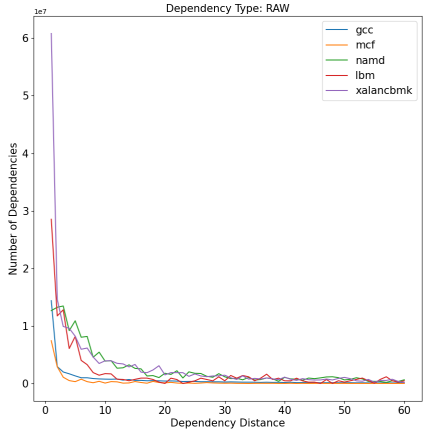


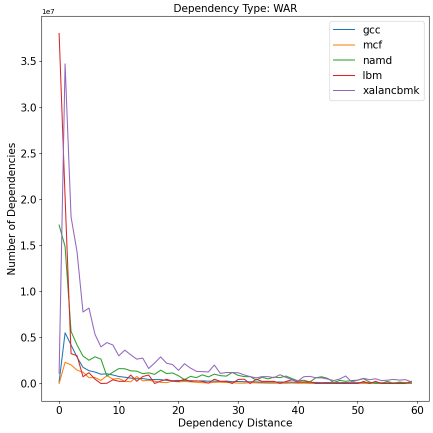


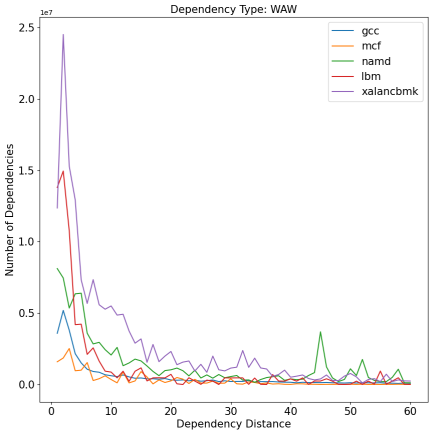


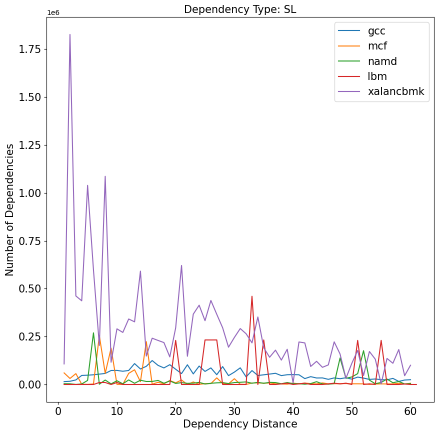


4.3 Dependency Distance vs Number of Dependencies per Dependency Type

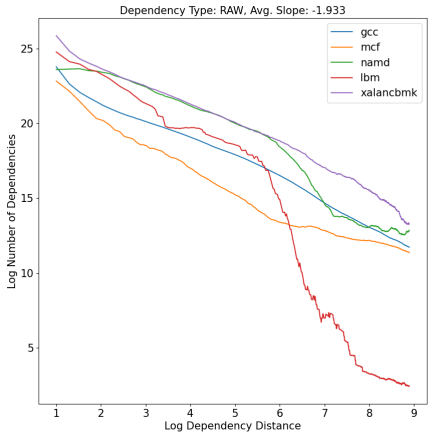


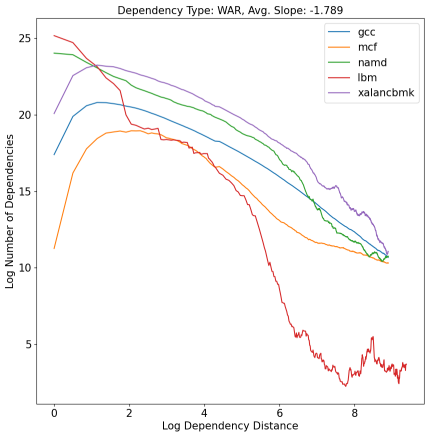


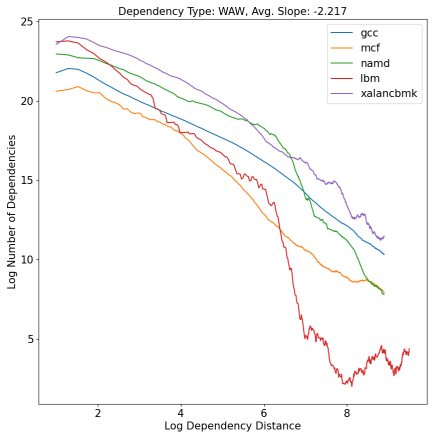


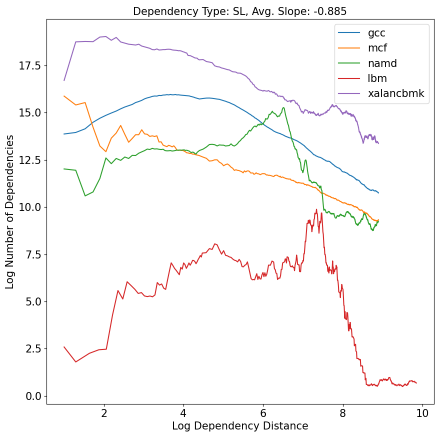


4.4 Log(Data Dependency Distance) vs Log(Number of Dependencies) per Dependency Type









5 Observations and Inferences

5.1 Per Benchmark Plots

The following observations are for **Plots 4.1**

- From the un-scaled data, we found that the average dependency distance is 31.08 instructions (Ignoring outliers with a distance greater than 500). From this, we can infer that most of the reads and writes have **temporal locality** (only short-range dependency)

- We can observe that **gcc, mcf, xalancbmk** have a significantly high **RAW** dependency while the other two have a high **WAR** dependency. This might be because the former benchmarks are conversion types (one file to another), while the other two are simulation types where previous data is read, updated and immediately written back

- The dependency frequency for Store-Load is insignificant compared to register dependencies, as there are more register-based instructions than memory-based ones. They can be analyzed in **log** scaled plots

5.2 Per Benchmark Log Plots

The following observations are for **Plots 4.2**. All the logs are taken **base 2**

- From the **log** scaled graphs, we can almost make out the linear nature of **RAW, WAR and WAW** log dependencies. Using linear curve fit, the average slope (p) was obtained.

$$log(freq) = c - p\ log(dist)$$
$$freq \propto \frac{1}{(dist)^p}$$

  The higher the **p** value, the faster the drop in frequency with distance

- **RAW, WAR and WAW** dependencies follow a consistent variation in the graph, which load-store dependency does not. This might be because the amount of addressable space is very big compared to number of registers. But it does follow a downward trend

- **gcc, mcf, xalancbmk** benchmarks have kind of peak frequency at $log(dist) = 2$ for WAR. The other two do not have a maximum. This may be because the last two have a large number of inline (or next line) same register updates.

- **lbm and namd** seem to have Store-Load dependencies that increase up to $log(dist) = 7$ then decrease, which differs from the other three benchmarks.

5.3 Per Dependency Plots

The following observations are for **Plots 4.3**

- From the graphs depicted above, we can infer that the lesser the distance of dependency the more is the corresponding frequency of that dependency. Thus, we can conclude that the dependency distance is inversely proportional to the number of dependencies.

- The frequency of dependencies is superseding with respect to all other benchmarks in terms of WAW, WAR, RAW and store-load dependencies. This can be due to the fact that the xalancbmk involves large number of frequent read & write operations which may be because of the tremendous amount of floating point operations involved.

- The lbm benchmark has higher WAR dependency than xalancbmk. This can be because the high usage of register assignment followed by arithmetic operations in the algorithmic implementation of the lbm benchmark.
- The namd and mcf benchmarks have near-constant frequency of dependencies as we vary the dependency distance. This may be because of the simplistic but repetitive nature of their algorithms.
- The WAW, WAR & RAW follow this trend accurately, but the store-load dependencies do not. This may be because memory operations involve more complex interactions with the memory hierarchy, which can lead to different behaviour in terms of dependencies.

5.4 Per Dependency Log Plots

The following observations are for **Plots 4.4**. All the logs are taken **base 2**

- From the graphs depicted above, we can infer that the logarithmic dependencies vs the frequency of occurrence has linear dependency ($y = -mx + c$). Thus, we can conclude that the logarithmic dependency distance is linearly dependent on logarithm of number of dependencies.
- The WAW, WAR & RAW follow this trend accurately, but the store-load dependencies do not. This may be because memory operations involve more complex interactions with the memory hierarchy, which can lead to different behaviour in terms of dependencies.
- In the WAR dependency graph, across the benchmarks, we can see at the start there is a maximum threshold that each of the benchmarks reaches beyond which the y-axis values start approaching 0 gradually. From this, we can infer that once the data is loaded inside the registers, followed by arithmetic operation algorithms of the benchmark, the same register values may be copied to other registers and used for the later part of the computation. The LBM benchmark doesn't follow the given trend; this can be because the LBM benchmark follows frequent access of memory.

6 Conclusion

This assignment focused mainly on writing our own tool to get dependency counts. We got to learn various APIs, flags and arguments that are used to insert custom instruction handlers in an existing binary. The dependencies obtained were **mostly** local and only a few dependencies that are far apart. We also found that RAW, WAR and WAW dependencies are consistent, while SL dependencies show few inconsistencies due to the nature of the program.