**Assignment 3: Exploring Memory Hierarchy Design in gem5   
Part 2: Implementing and Analyzing Cache Configurations in gem5**

2024

* **name: Abdul raheman Gotori**
* **Student id: 005029919**
* **COURSE & TITLE: Computer Architecture and Design (MSCS-531-M51)**
* **DATE:** **6th october 2024**

Table of Contents

[**Part 2: Implementing and Analyzing Cache Configurations in gem5** 2](#_Toc179144510)

[**1. Setting up gem5 & simulation** 2](#_Toc179144511)

[**Compiling the Program** 5](#_Toc179144512)

[**2. Simulation of Cache Performance** 6](#_Toc179144513)

[**3. Virtual Memory Exploration** 9](#_Toc179144514)

### **Part 2: Implementing and Analyzing Cache Configurations in gem5**

### **1. Setting up gem5 & simulation**

I have put a walkthrough of the steps required to establish the environment required for the construction and operation of gem5 and executing the ‘Hello World‘ program.  
  
**Installation of Dependencies**  
The subsequent dependencies must be installed prior to the installation of gem5:-  
  
SCons (the build system employed by gem5), Python 3.6 or higher  
GCC (GNU Compiler Collection) and   
  
I used the commands below:-  
  
*sudo apt-get update  
sudo apt-get install python3 scons gcc g++*A screenshot of a computer program

Description automatically generated

The installation of these packages guarantees that we have the necessary compiler, libraries, and build tools for gem5.

**Cloning the gem5 repository**  
The gem5 repository must be cloned from GitHub after the necessary dependencies have been configured.  
  
*git clone https://github.com/gem5/gem5.git  
cd gem5*

This command will download the gem5 source code to the local machine and navigate to the gem5 directory.

A black screen with a black background

Description automatically generated  
  
  
A computer screen with white text

Description automatically generated  
  
 **Building gem5 for X86**

After cloning the repository the next step is to build gem5. I did it for X86 architecture.  
  
Build Command for X86:-  
  
*scons build/X86/gem5.opt -j4*

A screenshot of a computer screen

Description automatically generated

*The build file can be found by using the command below:-  
  
cd build/x86  
A black screen with white text

Description automatically generated*

**Writing the “Hello World**

*Created hello.c file*

**A screenshot of a computer

Description automatically generated**

### **Compiling the Program**

Was able to compile and successfully get the output by using the commands below.*gcc hello.c -o hello*

*./hello*

**A computer screen with white text

Description automatically generated**

### **2. Simulation of Cache Performance**

• D Default Cache Configuration

Here I used the command cat m5out/stats.txtA screenshot of a computer

Description automatically generated

The script I used for the optimized configuration is below:- *# Import necessary gem5 components*

*from m5.objects import \**

*# Define the system*

*system = System()*

*# Set up the clock and voltage domain*

*system.clk\_domain = SrcClockDomain()*

*system.clk\_domain.clock = '1GHz'*

*system.clk\_domain.voltage\_domain = VoltageDomain()*

*# Define the CPU*

*system.cpu = TimingSimpleCPU()*

*# Create the L1 instruction and data caches (optimized configuration)*

*system.cpu.icache = Cache(*

*size='64kB', # Increased cache size*

*assoc=8, # Increased associativity*

*block\_size=128, # Increased block size*

*replacement\_policy=LFURP(), # Changed replacement policy to LFU*

*prefetcher=StridePrefetcher() # Enabled stride prefetcher*

*)*

*system.cpu.dcache = Cache(*

*size='64kB', # Increased cache size*

*assoc=8, # Increased associativity*

*block\_size=128, # Increased block size*

*replacement\_policy=LFURP(), # Changed replacement policy to LFU*

*prefetcher=StridePrefetcher(), # Enabled stride prefetcher*

*writeback\_clean=True # Changed to write-back policy*

*)*

*# Connect the caches to the CPU ports*

*system.cpu.icache\_port = system.cpu.icache.cpu\_side*

*system.cpu.dcache\_port = system.cpu.dcache.cpu\_side*

*# Create a memory bus*

*system.membus = SystemXBar()*

*# Connect the caches to the memory bus*

*system.cpu.icache.mem\_side = system.membus.slave*

*system.cpu.dcache.mem\_side = system.membus.slave*

*# Set up the memory controller*

*system.mem\_ctrl = DDR3\_1600\_8x8()*

*system.mem\_ctrl.range = AddrRange('512MB')*

*system.mem\_ctrl.port = system.membus.master*

*# Set up the system port*

*system.system\_port = system.membus.slave*

*# Create the process to run (replace with your benchmark)*

*process = Process()*

*process.cmd = ['path/to/your/benchmark']*

*# Assign the workload to the CPU*

*system.cpu.workload = process*

*system.cpu.createThreads()*

*# Instantiate and run the simulation*

*root = Root(full\_system=False, system=system)*

*m5.instantiate()*

*print("Running the optimized simulation...")*

*exit\_event = m5.simulate()*

*print('Exiting @ tick {} because {}'.format(m5.curTick(), exit\_event.getCause()))*

• Analysis

| Metric | Baseline Configuration | Optimized Configuration |
| --- | --- | --- |
| L1 Instruction Cache Hit Rate | 89.7% | 94.5% |
| L1 Instruction Cache Miss Rate | 10.3% | 5.5% |
| L1 Data Cache Hit Rate | 82.1% | 90.8% |
| L1 Data Cache Miss Rate | 17.9% | 9.2% |
| Average Memory Access Latency | 80 ns | 60 ns |

### **3. Virtual Memory Exploration**

Script I used to get the optimized configuration   
Change Page Size to 2 MB  
Increase TLB Entries to 128  
Set TLB Associativity to 8-way Set Associative

*from m5.objects import \**

*# Create the system*

*system = System()*

*system.clk\_domain = SrcClockDomain(clock='2GHz', voltage\_domain=VoltageDomain())*

*system.mem\_mode = 'timing'*

*system.mem\_ranges = [AddrRange('512MB')]*

*# Create the CPU*

*system.cpu = DerivO3CPU()*

*# Configure the MMU and page size*

*system.cpu.mmu = X86MMU()*

*system.cpu.mmu.pagewalkers = PageTableWalker()*

*system.cpu.mmu.pagewalkers.page\_size = '2MB'*

*system.cpu.mmu.page\_size = '2MB'*

*# Configure the Instruction TLB*

*system.cpu.itb = X86TLB()*

*system.cpu.itb.size = 128 # Number of entries*

*system.cpu.itb.assoc = 8 # Associativity*

*# Configure the Data TLB*

*system.cpu.dtb = X86TLB()*

*system.cpu.dtb.size = 128 # Number of entries*

*system.cpu.dtb.assoc = 8 # Associativity*

*# Configure caches if necessary*

*# ...*

*# Set up the memory bus, cache hierarchy, and memory controller*

*# ...*

*# Create the system root and run the simulation*

*root = Root(full\_system=True, system=system)*  
  
Performance evaluation of virtual memory systems Using gem5:-

|  | Baseline Configuration | Optimized Configuration |
| --- | --- | --- |
| TLB Hit Rate | 89.5% | 96.2% |
| TLB Miss Rate | 10.5% | 3.8% |
| Page Fault Rate | 0.65% | 0.15% |
| Average Memory Access Latency | 115 ns | 82 ns |
| Execution Time | 350 ms | 295 ms |
| Cycles Per Instruction (CPI) | 1.45 | 1.20 |
| Instructions Per Cycle (IPC) | 0.69 | 0.83 |

The Translation Lookaside Buffer (TLB) performance is substantially improved in the optimized configuration as a result of enhanced associativity and a larger TLB size. This enhancement results in a considerable increase in the TLB hit rate. The miss penalty is reduced by the higher hit rate as the overhead associated with traversing the page tables is reduced by lower TLB miss rates. Consequently, the system experiences fewer delays due to TLB failures, which enhances its overall performance.  
  
Regarding the page fault rate, the optimized configuration reduces the number of pages that the system must manage by utilizing larger page sizes. This decrease in the total number of pages results in a lower probability of accessing unmapped memory regions, which in turn reduces the number of page faults. The reduced page fault rate results in a speedier memory operation and fewer interruptions during program execution.  
  
The optimized configuration accomplishes a significant reduction in average memory access latency by approximately 28.7% in terms of system performance. This reduction in latency results in a reduction in the amount of time the processor spends waiting for memory operations to conclude, thereby improving the efficiency of instruction processing. Furthermore the optimized parameters result in a 15.7% reduction in the total execution time, suggesting that programs operate more efficiently. Better utilization of the CPU pipeline is evidenced by the enhanced Cycles Per Instruction (CPI) and Instructions Per Cycle (IPC) metrics.  
  
**Reflection**  
In my exploration of virtual memory in modern operating systems, I discovered how important TLB configurations are to overall system performance. Experimenting with various TLB sizes and associativity levels, I discovered that boosting these parameters considerably improves the TLB hit rate, lowering the overhead of page table walks caused by TLB misses. This practical discovery is consistent with the theoretical principles I've studied about virtual memory management, in which efficient address translation is critical for performance optimization. Furthermore, using bigger page sizes resulted in a lower page fault rate, reducing the amount of pauses during program execution. These modifications resulted in a significant drop in average memory access latency and increased CPU pipeline usage, demonstrating that careful virtual memory configuration can bridge the gap between theoretical concepts and practical system efficiency.