**Exploring Cache Architectures Using gem5 Simulation**

Chibuzo Ufomba

University of the Cumberlands

MSCS-531-B50: Computer Architecture and Design

Dr. Brandon Bass

July 20, 2025

**Exploring Cache Architectures Using gem5 Simulation**

This report documents successful experiments using the gem5 simulator to examine cache configuration and how different setups influence system performance. It also explores virtual memory, emphasizing how cache parameters and virtual memory settings affect system performance.

**Environment Setup on Ubuntu VM (UTM) for Apple Silicon M4 Mac**

This section provides a comprehensive guide to setting up the gem5 simulator on an Ubuntu virtual machine (VM) running on an Apple Silicon M4 Mac using UTM. It includes the software dependencies required to build gem5 and step-by-step instructions for installing these dependencies on gem5 with the Ubuntu VM.

**Software Dependencies**

The software dependencies needed to build gem5 on an Ubuntu VM (ARM64 architecture) running on an Apple Silicon M4 Mac via UTM include the following (gem5 Community, n.d.):

1. Git: For cloning the gem5 repository from GitHub.
2. Python 3.6+: Python is required by the gem5 libraries.
3. gcc 10+: The compiler for building gem5.
4. SCons 3.0+: The build system used by gem5.
5. Protobuf 2.1+ (Optional)

**Installation Steps**

The following steps outline how to set up the Ubuntu VM on UTM:

1. Download UTM from the official website—https://mac.getutm.app/ (UTM, n.d.).
2. Download Ubuntu ARM64 ISO from the official Ubuntu website.
3. Create Ubuntu VM using UTM.
4. Install required dependencies using the Ubuntu package manager.
5. Clone the gem5 source code from GitHub using this command:

git clone <https://github.com/gem5/gem5.git>

**Simulation of Cache Configuration**

**Default Cache Configuration**

The default simulation script used was the two\_level.py script located in the folder directory: configs/learning\_gem5/part1/two\_levels.py. The two\_level.py uses an X86TimingSimpleCPU CPU and DDR3\_1600\_8x8 memory. It uses classes from the cache.py file to simulate an x86 system with:

1. L1I Cache: 16KiB default size, 2-way associative, 64B block size, 2-cycle latency.
2. L1D Cache: 64KiB default size, 2-way associative, 64B block size, 2-cycle latency.
3. L2 Cache: 256KiB default size, 8-way associative, 64B block size, 20-cycle latency.

**Running the Simulation**

To run the default simulation, the following command was executed: ./build/X86/gem5.opt configs/learning\_gem5/part1/two\_level.py

The simulation completed successfully, demonstrating the proper configuration of the cache hierarchy and system components. Figure 1 shows the output of the successful simulation run.

**Figure 1**

*Screenshot showing a successful gem5 simulation output with default cache configuration*

A screenshot of a computer program

AI-generated content may be incorrect.

**Cache Performance Metrics**

The performance metrics for this simulation can be retrieved from the stats.txt folder in the m5out directory. This file contains the detailed simulation statistics for the “Hello World” workload on an x86 system with a two-level cache hierarchy (L1I, L1D, and L2 caches). This section will analyze the performance metrics (hit rate, miss rate, and average memory access latency).

The cache hit rate is calculated as the ratio of cache hits to total cache accesses, expressed as a percentage (Hennessy & Patterson, 2019).

|  |  |  |
| --- | --- | --- |
|  |  | (1) |

The performance metrics are shown below:

1. L1 Instruction Cache:

* Number of Cache Hits (system.cpu.icache.overallHits::total) = 8044
* Total Memory Accesses (system.cpu.icache.overallAccesses::total) = 8279
* Hit Rate = 97.16%
* Miss Rate (system.cpu.icache.overallMissRate::total) = 2.84%
* Average Miss Latency (system.cpu.icache.overallAvgMissLatency::total) = 101080.85 ns

1. L1 Data Cache:

* Number of Cache Hits (system.cpu.dcache.overallHits::total) = 1953
* Total Memory Accesses (system.cpu.dcache.overallAccesses::total) = 2087
* Hit Rate = 93.58%
* Miss Rate (system.cpu.dcache.overallMissRate::total) = 6.42%
* Average Miss Latency (system.cpu.dcache.overallAvgMissLatency::total) = 108119.40 ns

1. L2 Cache:

* Number of Cache Hits (system.l2cache.overallHits::total) = 6
* Total Memory Accesses (system.l2cache.overallAccesses::total) = 369
* Hit Rate = 1.63%
* Miss Rate (system.l2cache.overallMissRate::total) = 98.37%
* Average Miss Latency (system.l2cache.overallAvgMissLatency::total) = 99914.60 ns

**Optimizing Cache Parameters**

The simulation results showed that the L2 cache has a very low hit rate and a very high miss rate. To address this, the following modifications were made.

1. Increase of L2 cache size to 512Kib as shown in Figure 2.
2. Increase of L2 cache size to 1MiB as shown in Figure 3.
3. Increase of L2 associativity to 16 as shown below in Figure 4

**Figure 2**

*Screenshot showing L2 cache size as 512Kib in two\_level.py*

A computer screen shot of a black background

AI-generated content may be incorrect.

**Figure 3**

*Screenshot showing L2 cache size as 1Mib in two\_level.py*

A screen shot of a computer

AI-generated content may be incorrect.

**Figure 4**

*Screenshot showing L2 cache associativity as 16 in two\_level.py*

**A computer screen shot of a black screen

AI-generated content may be incorrect.**

**Table 1**

*Table showing L2 cache performance metrics with varying L2 configuration parameters*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Parameter | Value | Hit Rate | Miss Rate | Average Miss Latency |
| Size (Baseline) | 256KiB | 1.63% | 98.37% | 99914.60 ns |
| Associativity (baseline) | 8 | 1.63% | 98.37% | 99914.60 ns |
| Size | 512KiB | 1.63% | 98.37% | 99914.60 ns |
| Size | 1MiB | 1.63% | 98.37% | 99914.60 ns |
| Associativity | 16 | 1.63% | 98.37% | 99914.60 ns |

The performance metrics of the L2 cache remained unchanged. Because of this, I decided to adjust the L1 cache configurations to see if there would be any changes in the results. The following changes were made to the L1 cache:

1. Increase of L1 cache size to 256KiB as shown in Figure 5
2. Increase of L1 cache associativity to 16 as shown in Figure 6

**Figure 5**

*Screenshot showing L1 cache size as 256KiB in two\_level.py*

A black background with white text

AI-generated content may be incorrect.

**Figure 6**

*Screenshot showing L1 cache associativity as 16 in two\_level.py*

A screen shot of a computer

AI-generated content may be incorrect.

**Table 2**

*Table showing L1 instruction cache performance metrics with varying L1 instruction configuration parameters*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Parameter | Value | Hit Rate | Miss Rate | Average Miss Latency |
| Size (Baseline) | 16KiB | 97.16% | 2.84% | 101080.85 ns |
| Associativity (baseline) | 8 | 97.16% | 2.84% | 101080.85 ns |
| Size | 256KiB | 97.22% | 2.78% | 103234.78 ns |
| Associativity | 16 | 97.22% | 2.78% | 103234.78 ns |

**Analysis of Performance Metrics**

The gem5 simulation produced different behaviors for the L1 instruction cache and L2 cache when running the Hello World benchmark. For the L1 instruction cache, increasing the size from 16KiB to 256KiB produced a modest improvement in hit rate from 97.16% to 97.22%, corresponding to a decrease in miss rate from 2.84% to 2.78%. However, this improvement came at the cost of increased miss latency, which rose from 101,080.85 ns to 103,234.78 ns. Increasing the associativity to 16 did not improve any of the performance metrics. These results suggest that the baseline 16KiB cache is enough for the Hello World program. The program’s instructions fit in the cache easily. The minimal hit rate improvement from the cache size increase indicates that the program is not suffering from a lack of space. The increase in access time with a larger cache size demonstrates the fundamental trade-off in cache design, where larger caches require longer access times due to increased complexity.

The L2 cache showed no improvements when the configuration parameters were changed. The hit rate remained at 1.63%, the miss rate remained at 98.37%, and the average miss latency remained unchanged at approximately 99,915 ns across all configurations. The Hello World program is not a memory-intensive program, as the L1 cache can handle most of the memory requests. The L2 cache receives the memory requests that the L1 cache was unable to respond to, as shown by the 369 total L2 cache accesses. The 369 total accesses represent first-time accesses to memory locations that have not been cached. This explains why increasing the L2 cache size from 256KiB to 512KiB and 1MiB produced no performance gains.

The L2 cache could benefit from advanced prefetching techniques to help reduce the miss rate; however, a more advanced workload would be needed to show the benefits of advanced prefetching techniques. Victim caching also presents another optimization technique that could improve the L2 cache performance.

**Virtual Memory Exploration**

This section describes a gem5 simulation setup aimed at assessing virtual memory system performance on an x86 architecture with configurable Translation Lookaside Buffers (TLBs) and memory management units. The simulation uses a timing-accurate CPU model with a two-level cache hierarchy and DDR3 memory to offer realistic insights into virtual memory operations.

**Target Architecture**

 **ISA**: x86-64

 **Simulation Mode**: Syscall Emulation (SE mode)

 **CPU Model**: X86TimingSimpleCPU

 **System Clock**: 1 GHz

 **Memory Model**: Timing-accurate memory accesses

**Creation of C Program**

For this step, I created a C program called stress\_tlb.c to stress the virtual memory system in gem5. This program is designed to perform varied memory accesses to trigger virtual memory behavior, such as TLB misses and page faults, which can be observed in gem5's simulation output.

**Figure 7**

*Screenshot showing stress\_tlb.c*

A screen shot of a computer screen

AI-generated content may be incorrect.

The C program was compiled with a cross-compiler because I was running gem5 with an x86 simulation. The command used:

x86\_64-linux-gnu-gcc -o stress\_tlb stress\_tlb.c -static

**Running the Simulation**

To simulate the virtual memory experiment, the following command was executed: ./build/X86/gem5.opt configs/learning\_gem5/part1/virtual\_memory\_setup.py

The simulation completed successfully as shown below:

**Figure 8**

*Screenshot showing successful simulation*

A screenshot of a computer screen

AI-generated content may be incorrect.

**Performance Metrics**

Multiple simulations were conducted to show the varying effects of performance with different configuration values. In the first experiment, I used different page sizes with the baseline TLB. The page sizes used were the baseline (4KiB), 2MiB, and 100MiB.

**Table 3**

*Screenshot showing performance metrics with varying page sizes*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Page Size | Simulation Time | CPI | Data TLB Miss Rate | Instruction TLB Miss Rate | Combined TLB Miss Rate |
| 4KiB | 0.0036s | 7.45 | 6.23% | 0.0083% | 1.99% |
| 2MiB | 0.0006s | 4.30 | 0.16% | 0.0417% | 0.06% |
| 100MiB | 0.0006s | 4.30 | 0.15% | 0.0419% | 0.06% |

For the next experiment, I used different TLB sizes with 4KiB page size. The following table shows the results

**Table 4**

*Screenshot showing performance metrics with varying TLB sizes*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ITLB Size | DTLB Size | L2TLB Size | Simulation Time | CPI | Data TLB Miss Rate | Instruction TLB Miss Rate | Combined TLB Miss Rate |
| 32 | 32 | 512 | 0.004s | 7.46 | 6.980% | 0.014% | 2.068% |
| 64 | 64 | 1024 | 0.004s | 7.46 | 6.979% | 0.013% | 2.066% |
| 128 | 128 | 2048 | 0.004s | 7.46 | 6.975% | 0.013% | 2.065% |

**Performance Metrics Analysis**

The performance data reveals that page size configuration has a dramatically greater impact on system performance than TLB sizing. Moving from 4KiB to 2MiB pages delivers an 83% reduction in simulation time (0.0036s to 0.0006s) and a 42% improvement in CPI (7.45 to 4.30), primarily due to a 97% reduction in data TLB miss rate from 6.23% to 0.16%. The 4KiB configuration suffers from severe TLB thrashing, while larger pages (2MiB and 100MiB) achieve nearly identical performance, indicating that 2MiB pages provide optimal benefits for this workload.

In contrast, doubling TLB sizes from 32 to 64 entries and expanding the L2TLB from 512 to 1024 entries produces negligible performance improvements, with simulation time remaining constant at 0.004s and only marginal reductions in miss rates. This demonstrates that when page sizes are suboptimal, simply increasing TLB capacity cannot compensate for the fundamental issue of excessive page table pressure. The analysis clearly shows that **page size optimization should be the primary focus** for memory-intensive workloads, as it addresses the root cause of TLB misses rather than merely expanding TLB capacity to handle an inefficient page granularity.

**Hands-On Discussion**

Virtual memory serves as a fundamental abstraction in modern operating systems, enabling process isolation, memory protection, and efficient resource management by translating virtual addresses to physical addresses through page tables (Hornyack et al., 2013). However, this translation process introduces performance overhead that is mitigated by the Translation Lookaside Buffer (TLB), which caches recent address mappings. The experimental data validates core virtual memory theory: the dramatic 83% performance improvement when moving from 4KiB to 2MiB pages demonstrates how larger pages exploit spatial locality more effectively, allowing the TLB to cover a larger working set with the same number of entries. With small pages, the system spent too much time performing expensive memory lookups (6.23% miss rate); however, larger pages alleviated this issue by reducing the total number of translations required.

The minimal impact of doubling TLB sizes indicates that having more TLB entries doesn't help significantly if the page size is incorrect for the workload. When small pages don't match how a program accesses memory, adding more TLB capacity can't fix the core problem of too many page table lookups. This explains why modern systems offer large page options (like 2MiB pages) instead of just building bigger TLBs.

**Troubleshooting Issues**

Throughout this assignment, I faced various technical challenges that demanded careful debugging. During the virtual memory simulation section, I was unable to set the page size explicitly in the MMU. This is a standard issue when performing Syscall emulation. I was able to resolve this by setting the page size in the workload. Another issue I encountered was forgetting to include the crucial m5.simulate() function call in my configuration script. This caused my simulation not to run, wasting my time.

Another issue I faced was related to cross-compilation for the x86 target architecture. Initially, I compiled my test workloads for my host system rather than the x86-64 target that gem5 was simulating, resulting in execution errors. I had to install the correct cross-compilation tools to ensure all the binaries and shared libraries were compatible with gem5's system call emulation mode.

**Conclusion**

This assignment demonstrated the role of cache architecture and virtual memory configuration in system performance through systematic gem5 simulations. The cache hierarchy experiments revealed that the Hello World benchmark achieved excellent L1 instruction cache performance (97.16% hit rate). The L2 cache showed minimal utilization with only 1.63% hit rate, highlighting that cache performance benefits are highly dependent on workload characteristics and memory access patterns.

The virtual memory analysis showed that choosing the right page size is much more important than having bigger TLBs. Using 2MiB pages instead of 4KiB pages made the simulation run 83% faster and improved performance by 42%, proving that larger pages help programs access memory more efficiently.

**References**

gem5 Community. (n.d.). gem5 Computer Architecture Simulator. <https://www.gem5.org/>

Hennessy, J. L., & Patterson, D. A. (2019). Computer Architecture: A Quantitative Approach (6th ed.). Morgan Kaufmann.

Hornyack, P., Ceze, L., Gribble, S., Ports, D., & Levy, H. (2013). A study of virtual memory usage and implications for large memory. University of Washington Technical Report. Retrieved from <https://homes.cs.washington.edu/~luisceze/publications/vmstudy-uwtr2013.pdf>

UTM. (n.d.). UTM: Virtual machines for Mac. Retrieved July 16, 2025, from https://mac.getutm.app/