Computer Architectural And Design

Assignment 7: Exploring Memory Hierarchy Design in Gem5

Nguyen The Duy Khanh

005019181

Table of Contents

1. Understand memory hierarchy

- Memory hierarchy design is one of the most required things in optimizing the available memories in the computer to increase performance in exchange for complexity and energy efficiency. Understanding these trade-offs is crucial for optimizing system performance while managing costs and power consumption.

a. Memory technologies

- Nowadays, memory technologies span a variety of types, and each has distinct characteristics that could impact the memory hierarchy and overall system performance. Static Random Access Memory (SRAM) [1] is part of the memory hierarchy design. It is fast and efficient and does not require a capacitor[2], which will not leak and lose information as rapidly as DRAM. Because of its design, it allows more rapid access times and consumes less power when the memory is not accessed. In exchange, SRAM requires a larger physical size due to increased transistors, higher production costs, and higher complexity. However, SRAM introduces complexity involving L1, L2, and L3 caches (IBM uses DRAM memory for L3 [3]), with a different design for each cache. Although they use SRAM, they do not all use the *same* SRAM design. SRAM for L2 and L3 are optimized for *size* (to increase the capacity given limited manufacturable chip size or reduce the cost of a given capacity). At the same time, SRAM for L1 is more likely to be optimized for speed.
- SRAM, at a lower pace, consumes less power than DRAM because SRAM only requires a small, steady current, while DRAM requires bursts of power every few milliseconds to refresh. The refresh current is several orders greater than the low SRAM standby current. Thus, SRAM is used in most portable and battery-operated equipment. At higher frequencies, SRAM can consume as much energy as DRAM.

- There are many other sides to DRAM and SRAM, such as Flash Memory (slower but cheaper than DRAM and is mainly used for large storage capacities) or MRAM (slower than SRAM but faster than DRAM, but it is non-volatile, which is a better aspect than SRAM).
- Therefore, each memory technology has its own use case and trade-off under these criteria: speed, cost, volatility, and density. High-speed, low-latency technologies (e.g., SRAM) are essential for Cache, while DRAM is the primary system memory, balancing speed and capacity. Non-volatile options like Flash and emerging technologies like MRAM and PCM provide essential persistent storage, influencing overall system performance and responsiveness.

b. Advanced Cache Optimization

- Cache optimization is another critical aspect of modern computer architecture, which aims to minimize cache misses, increase cache hit rate, and increase the system's throughput. We will discuss some of the techniques which are used to fulfill the aforementioned goal:
    - **Prefetching[4]:** Instead of waiting for the instructions computed and stored in the cache, we will calculate beforehand and store it beforehand, which is often used in pipelines to reduce stalls. Even though it increases the hit rate and reduces the latency, it also has the possibility of polluting the cache data since it will evict other data or instructions.
    - **Victim caches [5]:** When the cache sizes meet a specific policy [6](e.g., least recently used), we would need to evict some instructions or data. Therefore, instead of evicting completely, victim caches will maintain a second cache and ensure some of the evicted instructions or data will be stored in the second

cache. It helps increase the hit rate and reduce the cache misses by allowing the evicted one to remain accessible for a short time. It is used mostly in environments with reuse patterns, especially in workloads with high locality. However, it is not usable in highly constrained memory environments.

- **Cache partitioning [7]:** Instead of having a single memory system for all different threads and processes, we will reduce a hotspot cache and cache thrashing by dividing caching resources into different access patterns. It also helps ensure that critical data for high-priority processes will remain in the cache, which will improve throughput.

c. Virtual memory and virtual machine

- Virtual memory [8] gives each process a given physical memory. The process is unaware of details of its physical memory but only aware of how big the chunk is, and the chunk begins at address 0). Additionally, each process is unaware of other chunks of virtual memory belonging to other processes. Even if they did know, it's prevented by the kernel from accessing the memory, which increases the built-in memory protection and allows the processes to share memory between RAM and disk memory more efficiently.
- Each time a process wants to read or write to memory, its requests must be translated from a VM address to a physical memory address via MMU (memory management unit). However, when the RAM is full of memory, the OS must decide which page to evict to make room for the new page. That's when MMU will use page swapping (moving pages to secondary storage from main memory based on specific policies). Additionally, to improve the performance and avoid access to the page table, MMU

also stores the recently mapped address translation in the Translation Look Inside Buffer (TLB).

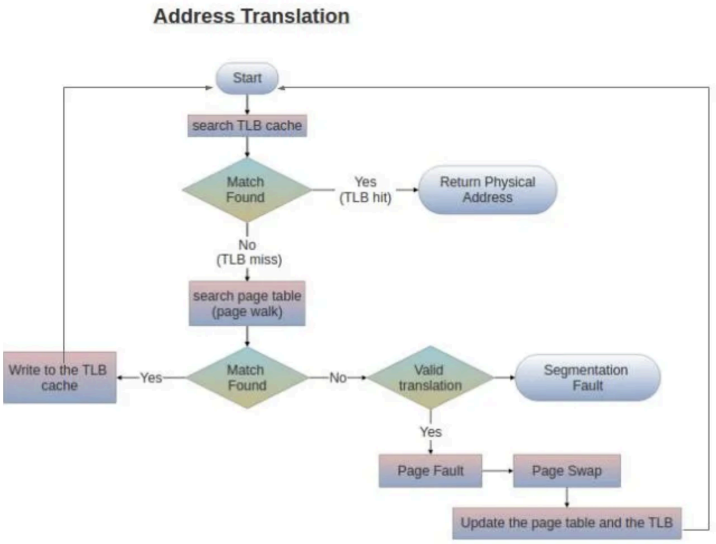

Figure 1: Memory translation in MMU with TLB

- Virtual memories are instances of an operating system (e.g, Virtual Box) that run on a hypervisor, which manages the underlying physical hardware. Each VM has its own virtual memory, which is mapped to the physical memory of the host machine. VMs add another layer to the memory hierarchy. The hypervisor manages the memory for multiple VMs, often using techniques similar to those in traditional virtual memory (e.g, page tables) to provide each VM with its own virtual address space.

d. Cross-Cutting Issues

- There are certain trade-offs when coming to designing memory hierarchies:
    - **Cost and Performance:** high-performance technologies (e.g., SRAM) are more expensive than cheaper technologies (e.g., DRAM, Flash). Therefore, designers sometimes have to balance the need for speed with budget constraints, often leading to compromises in cache sizes or speeds.

- **Power Consumption:** Faster memory technologies often consume more power (e.g., SRAM requires more transistors, which draws more power than DRAM). Therefore, designers must carefully select memory technologies to meet performance goals without exceeding power budgets.
- **Complexity:** With the goal of improving performance, re-using the current technology is not an option. Therefore, increasing complexity (e.g multiple levels of cache) which is more challenging when coming to design or debugging. (e.g., SRAM introduces complexity involving L1, L2, and L3 caches).
- **Workload characteristics:** Different workloads exhibit varying access patterns (e.g., sequential vs. random access). A memory hierarchy optimized for one type of workload may underperform with another. Therefore, it makes it hard to understand, predict workloads, and design a one-size-fits-all memory hierarchy.

- The more trends and technologies emerge, the more the current challenges grow. An example would be Non-Volatile Memory, where technologies such as Phase Change Memory (PCM), Resistive RAM (ReRAM), and MRAM provide non-volatility with performance characteristics closer to DRAM.

2. Implementing and Analyzing Cache Configuration In Gem5

a. Simulation of Cache

- As a **base for comparing performance** between different cache configurations, we will use a Cache with a size of 1KB

```python
# Create a simple cache (4-way set associative)
system.cpu.icache = Cache(
    size='1kB',
    assoc=4,  # 4-way set associative
    tag_latency=2,
    data_latency=2,
    response_latency=1,
    mshrs=16,
    tgts_per_mshr=4
)       You, 1 second ago • Uncommitted changes
system.cpu.dcache = Cache(
    size='1kB',
    assoc=4,  # 4-way set associative
    tag_latency=2,
    data_latency=2,
    response_latency=1,
    mshrs=16,
    tgts_per_mshr=4
)
```

Figure 2: Cache's configuration in Python

- When running with the following command with different cache sizes,

```
gem5 on  stable [!?] via  v3.12.5 on ☁  kel.nguyen@vietnamtechsociety.org(us-east1)
❯ ./build/X86/gem5.opt ./configs/assignment/simple_cache.py
```

the simulation process will run successfully with the following statistics:

```
Global frequency set at 1000000000000 ticks per second
warn: No dot file generated. Please install pydot to generate the dot file and pdf.
src/mem/dram_interface.cc:692: warn: DRAM device capacity (8192 Mbytes) does not match the
ssigned (512 Mbytes)
src/base/statistics.hh:279: warn: One of the stats is a legacy stat. Legacy stat is a stat
elong to any statistics::Group. Legacy stat is deprecated.
system.remote_gdb: Listening for connections on port 7000
Beginning simulation!
src/sim/simulate.cc:199: info: Entering event queue @ 0.  Starting simulation...
Hello world!
Exiting @ tick 53828000 because exiting with last active thread context
```

Figure 3: Simulation with ISA x86 with Cache in Python

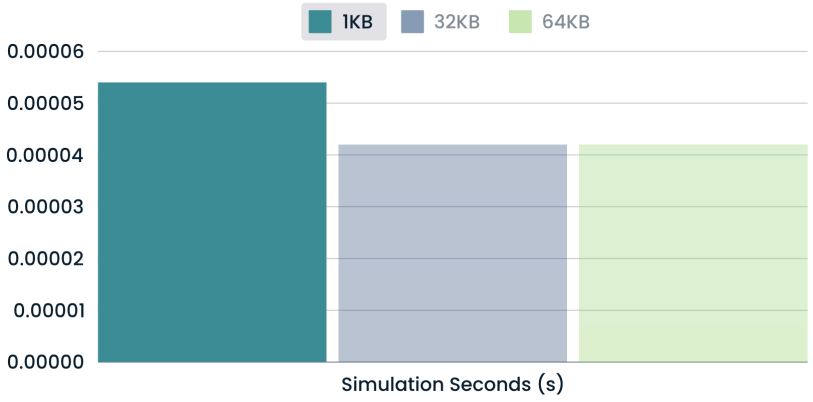

Figure 4: Simulation second(seconds) for different cache sizes

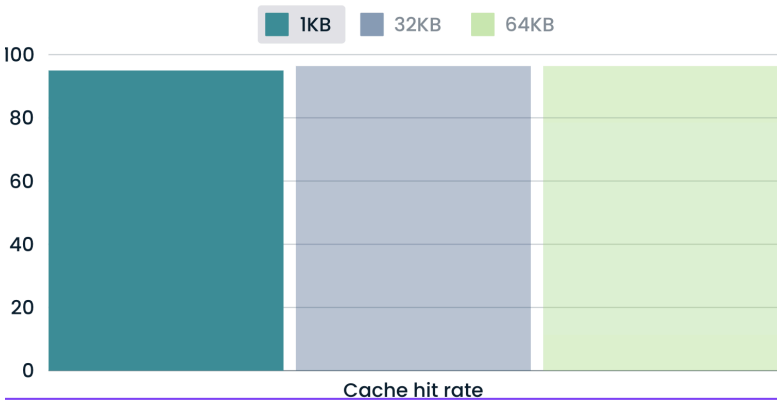

Figure 4: Cache hit rate (%) for different cache sizes

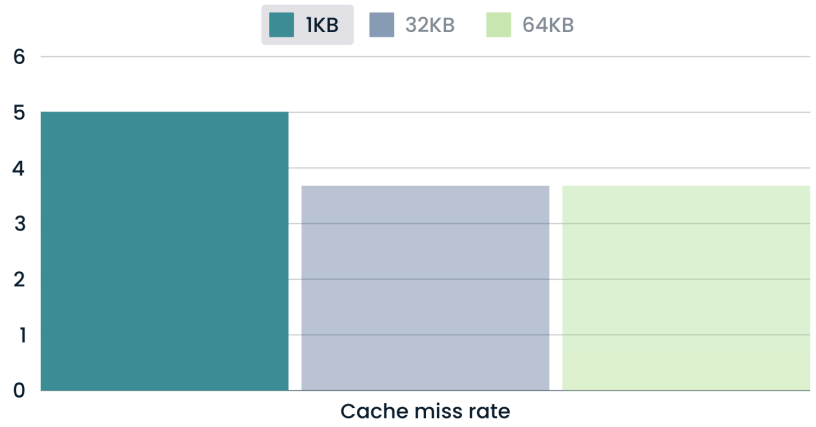

Figure 5: Cache miss rate (%) for different cache sizes

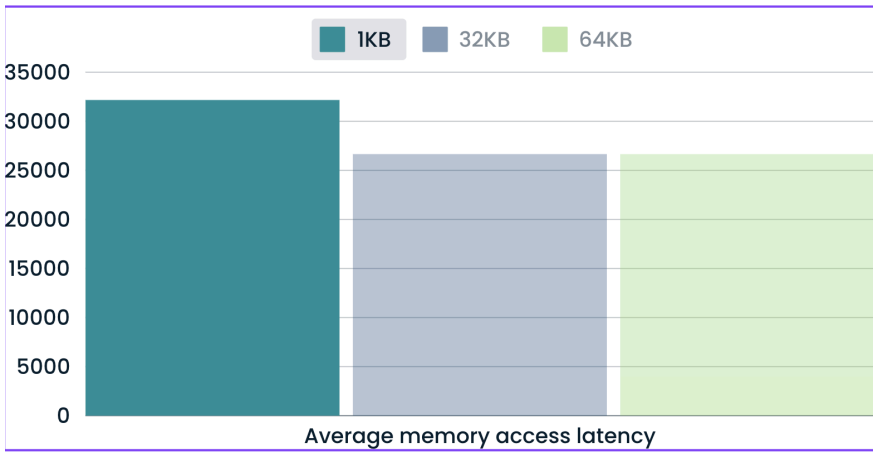

Figure 6: Average memory access latency (tick/count) for different cache sizes

Even though we can increase more cache size, however, the cache hit-and-miss rate won't change since the instructions set are the same and the calculated data.

- When changing the associative via Cache, the result will be

```
system.cpu.icache = Cache(
    size='32kB',
    assoc=1,  # 4-way set associative
    tag_latency=2,
    data_latency=2,
    response_latency=1,
    mshrs=16,
    tgts_per_mshr=4
)
system.cpu.dcache = Cache(
    (function) tag_latency: Any
    tag_latency=2,
    data_latency=2,
    response_latency=1,
    mshrs=16,
    tgts_per_mshr=4
```

Figure 7: Simulation second(seconds) for different cache associatives

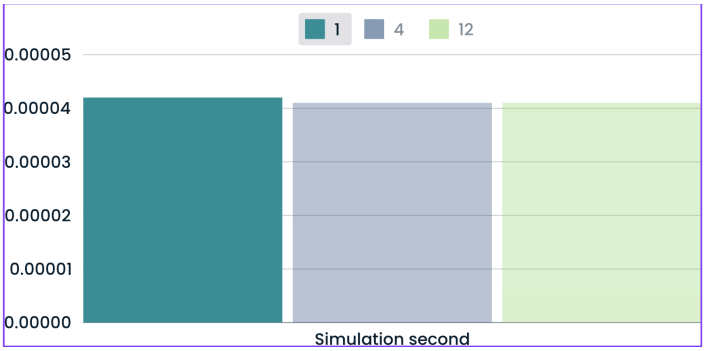

Figure 8: Simulation second(seconds) for different associatives

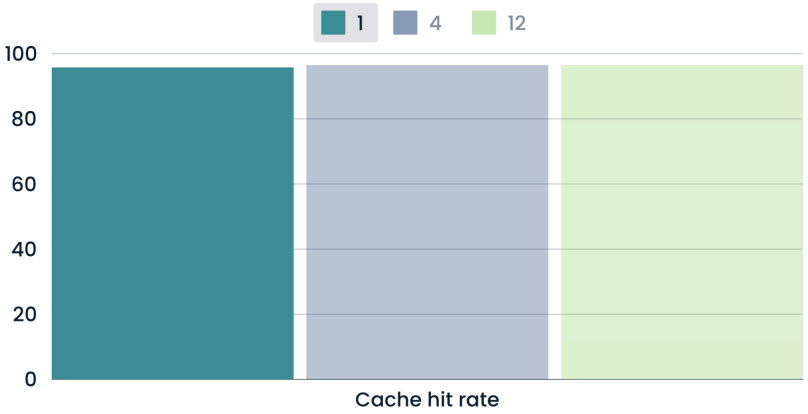

Figure 9: Cache hit rate for different associatives

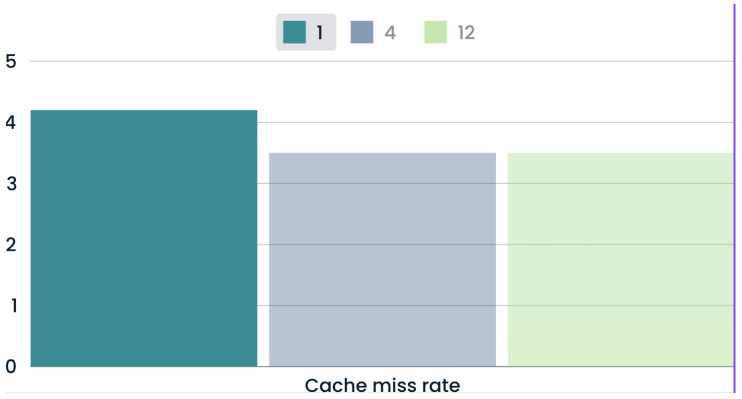

Figure 10: Cache miss rate for different associatives

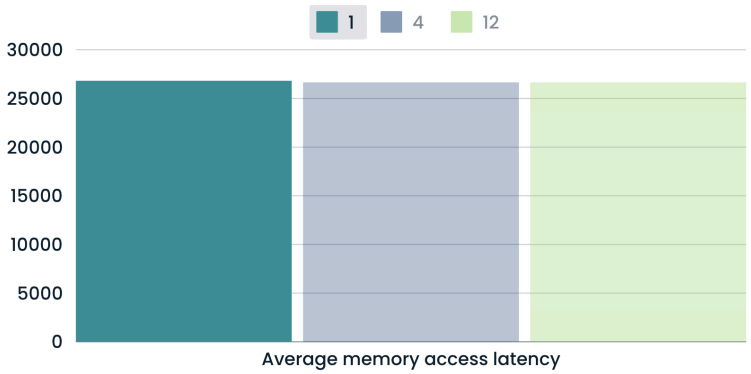

Figure 11: Average memory access latency for different associatives

- When using Cache which inherits from BaseCache, the block size is based on the WriteAllocator system cache line size.

```
Hoa Nguyen, 22 months ago | 4 authors (Nikos Nikoleris and others)
class WriteAllocator(SimObject):
    type = "WriteAllocator"
    cxx_header = "mem/cache/cache.hh"
    cxx_class = "gem5::WriteAllocator"

    # Control the limits for when the cache introduces extra delays to
    # allow whole-line write coalescing, and eventually switches to a
    # write-no-allocate policy.
    coalesce_limit = Param.Unsigned(
        2, "Consecutive lines written before delaying for coalescing"
    )
    no_allocate_limit = Param.Unsigned(
        12, "Consecutive lines written before skipping allocation"
    )

    delay_threshold = Param.Unsigned(
        8,
        "Number of delay quanta imposed on an "
        "MSHR with write requests to allow for "
        "write coalescing",
    )

    block_size = Param.Int(Parent.cache_line_size, "block size in bytes")
      Nikos Nikoleris, 8 years ago • mem: Add write coalescing and write-no-
```

Figure 11: Write allocator for Cache

Therefore, to configure the block size with a simple cache, we would need to change the system cache line size. By default, the cache line size for the system is 64 bytes.

When **changing the block size** via **cache_line_size**,

```
# Create a simple cache
system.cache_line_size = 128      You, 5 second
system.cache = SimpleCache(size='32kB')  # B
```

Figure 12: Change cache's block size in Python

the simulation process will run successfully with the following statistics

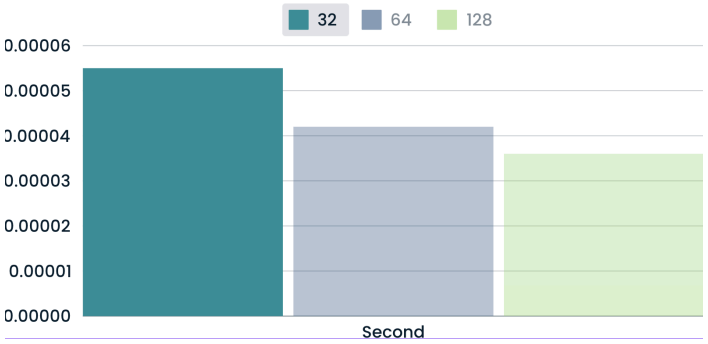

Figure 13: Simulation second (seconds) for different block sizes

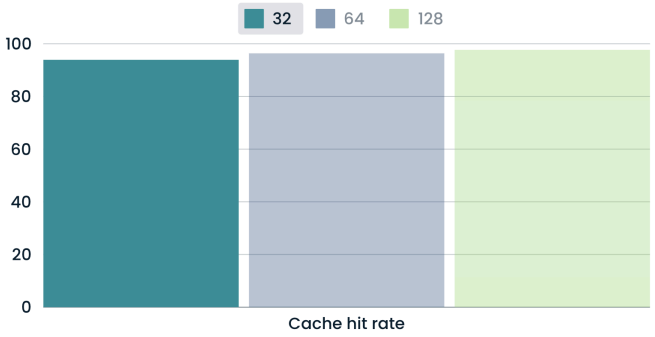

Figure 14: Cache hit rate for different block sizes

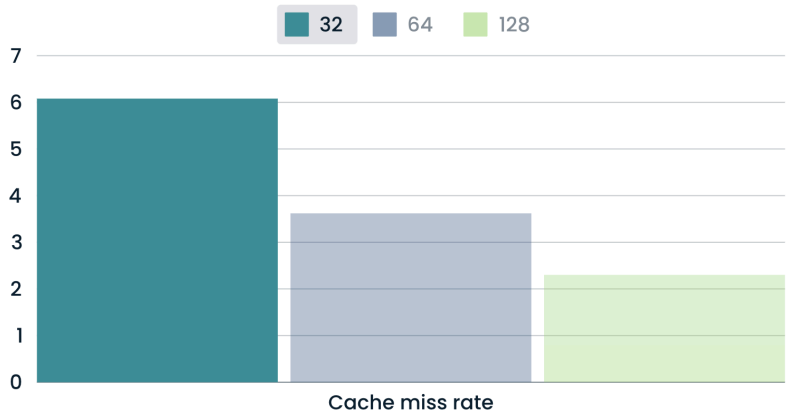

Figure 15: Cache miss rate for different block sizes

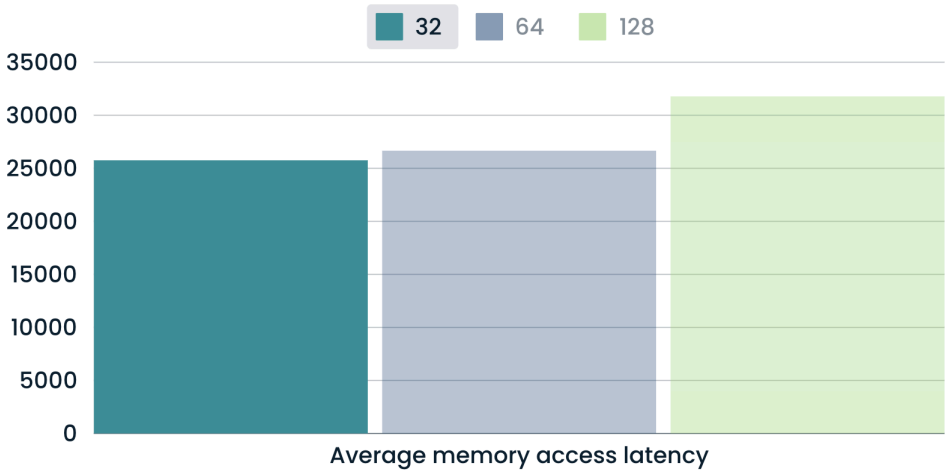

Figure 16: Average memory access latency (tick/count) for different block sizes

b. Simulation of Virtual Memory

- By default, Virtual Memory is enabled on the x86 platform within gem5 with a page size of 4kB and a Translation Lookaside Buffer of 64 entries, which is a crucial feature for simulating modern operating systems that rely on virtual memory to manage memory addresses. This can be confirmed via the config.json, which is the simulated configuration in the target x86ISA environment when running the following commands:
    - **Step 1:** Build the target ISA x86 with SCons:

```
gem5 on  stable [!?] via  v3.12.5 on ☁  kel.nguyen@vietnamtechsociety.org(us-east1)
 8% ❯ python3 -m SCons build/X86/gem5.opt -j 4
```

    - **Step 2:** Compile the script and run the artifacts in the target ISA X86

```
gem5 on  stable [!?] via  v3.12.5 on ☁  kel.nguyen@vietnamtechsociety.org(us-east1)
❯ ./build/X86/gem5.opt ./configs/assignment/simple_cache.py
```

    - **Step 3:** Checking the the result of m5out/config.json

```
"mmu": {
    "type": "X86MMU",
    "cxx_class": "gem5::X86ISA::MMU",
    "name": "mmu",
    "path": "system.cpu.mmu",
    "dtb": {
        "type": "X86TLB",
        "cxx_class": "gem5::X86ISA::TLB",
        "name": "dtb",
        "path": "system.cpu.mmu.dtb",
        "entry_type": "data",
        "eventq_index": 0,
        "next_level": null,
        "size": 128,
        "system": "system",
        "walker": {
```


Figure 17: Simulation's configuration

- As a **base for comparing performance** between different virtual memory configurations, we will TLB with 64 entries and a page size of 4kB.

```
...
class X86TLB(BaseTLB):
    type = "X86TLB"
    cxx_class = "gem5::X86ISA::TLB"
    cxx_header = "arch/x86/tlb.hh"

    size = Param.Unsigned(64, "TLB size")
    system = Param.System(Parent.any, "system object")
    walker = Param.X86PagetableWalker(
        X86PagetableWalker(), "page table walker"
    )
```

```
namespace X86ISA
{
    const Addr PageShift = 12;
    const Addr PageBytes = 1ULL << PageShift;
} // namespace X86ISA
} // namespace gem5
```

Figure 18: Changing page size and TLB size in Python

- When changing the page size, the simulation process will run successfully with the following statistics

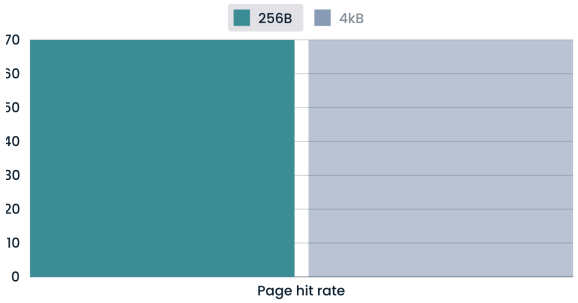

Figure 19: Page hit rate (%) for different page sizes

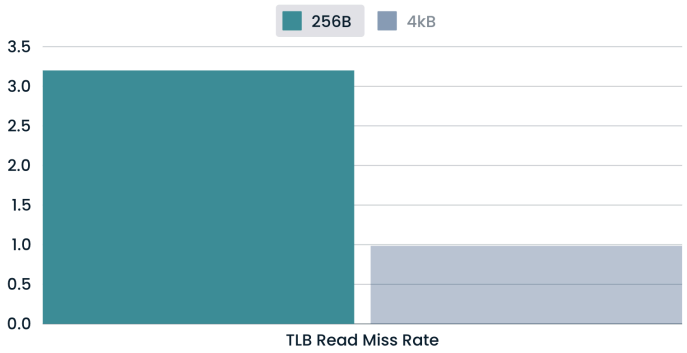

Figure 20: TLB Read Miss Rate (%) for different page sizes

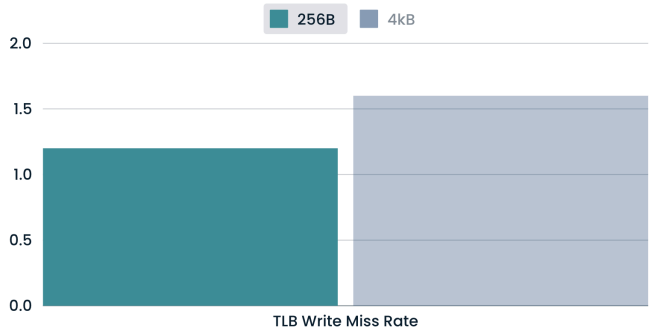

Figure 20: TLB Write Miss Rate (%) for different page sizes

- When changing the TLB Size, the simulation process will run successfully with the following statistics

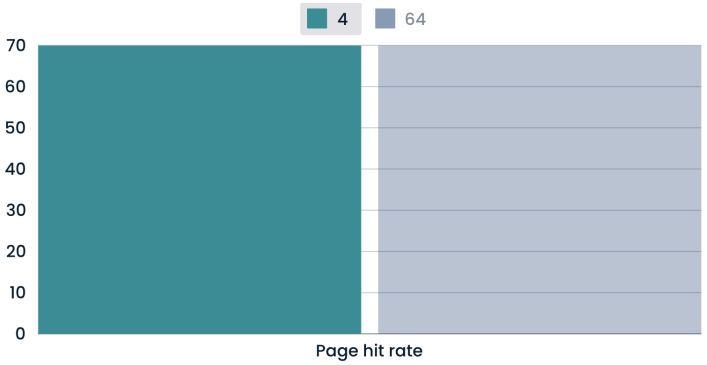

Figure 21: Page hit rate (%) for different TLB sizes

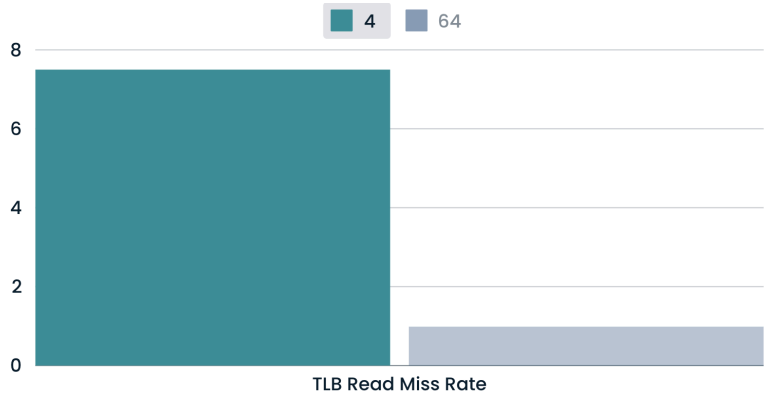

Figure 22: TLB Read Miss Rate (%) for different TLB sizes

Figure 23: TLB Write Miss Rate (%) for different TLB sizes

c. Simulation output

- In the case of cache, increasing cache size will store more instruction sets which will less data/instructions evicted and increase more hit rate as expected (increase simulation seconds from .052s to 0.041 and reduce cache miss rate from 5 to 3.7 %) [9]. Since the instruction sets and the data being cached within the same application stay constant. Therefore, it does not increase more after a certain limit cache size.
- The same can be said for block size, which increases locality and increases the chance to fetch useful data that will be used soon after the initial request increases when there

is a larger block size. (increase simulation seconds from 0.052s to 0.037s and reduce cache miss rate from 6.1 to 2.3%)

- In the case of virtual memory, increasing the page size will Larger page sizes mean that each TLB entry maps a larger block of memory. This can enhance TLB hit rates due to spatial locality (decrease TLB Read Miss Rate from 3.2% to 1.0% and increase TLB Write Miss Rate from 1.2% to 1.6%); when one address is accessed, it's likely that other addresses within the same page will also be accessed soon after.
- The same can be said for TLB Size. A larger TLB can store more translations, increasing the probability that the required virtual address translation is already in the TLB when memory access occurs (decrease TLB Read Miss Rate from 7.2% to 0.95% and decrease TLB Write Miss Rate from 1.2% to 0.57%)

3. References

- [1] Anastazija Spasojevic (2024). What is SRAM? Phoenix Nap. https://phoenixnap.com/glossary/sram-static-random-access-memory
- [2] Talon Homer (2021). What is the difference between static RAM and dynamic RAM? How Stuff Works. https://computer.howstuffworks.com/question452.htm
- [3] Wikipidea (2026) Power 4. https://en.wikipedia.org/wiki/POWER
- [4] Jim Jeffers, James Reinders (2013). Lack of prefetch instructions. Science Direct. https://www.sciencedirect.com/topics/computer-science/prefetch-instruction#:~:text=Prefetch%20instructions%20are%20non%2Dblocking,have%20no%20other%20side%20effects.
- [5] David Rodenas Phd(2024). A simple but powerful caching algorithm. Medium. https://drpicox.medium.com/a-simple-but-powerful-caching-algorithm-eb55c04c5d8f

- [6] Janardan (2024). Cache Eviction Policies. GeeksForGeeks. https://www.geeksforgeeks.org/cache-eviction-policies-system-design/#
- [7] L3 Cache Partitioning. Arm Developer. https://developer.arm.com/documentation/100453/0300/functional-description/l3-cache/l3-cache-partitioning
- [8] Sravanthi Sinha (2017). All about virtual memory. Medium. https://medium.com/@SravanthiSinha/all-about-the-virtual-memory-1c8a3cf306b7
- [9] Greg Young (2021). Why does increasing the capacity of cache memory tend to increase its hit rate? Quora. https://www.quora.com/Why-does-increasing-the-capacity-of-cache-memory-tend-to-increase-its-hit-rate