Part 1 understanding Memory Hierarchy

Memory Technology

Modern computing systems rely on a hierarchy of memory technologies to balance performance and cost. This hierarchy ranges from the fastest but most expensive static random-access memory (SRAM) to the slower, cheaper dynamic random-access memory (DRAM), and beyond.

- **SRAM** is typically used for **cache memory**. It offers low latency and high-speed access, making it suitable for storing frequently accessed data. However, its high cost and power consumption limit its use to smaller, critical sections of the memory hierarchy, such as Level 1 (L1) cache (Hennessy & Patterson, 2017).
- **DRAM**, on the other hand, is used for **main memory (RAM)** due to its lower cost and higher density, despite its higher latency compared to SRAM. The trade-off is that while DRAM offers larger capacity, it is slower in terms of access time, impacting overall performance during data-intensive operations (Stallings, 2015).
- **Emerging technologies**, like **3D stacked memory** (e.g., HBM - High Bandwidth Memory) and **non-volatile memory** (e.g., NAND flash, phase-change memory), aim to improve this balance by providing faster access and higher densities. These advancements aim to bridge the gap between fast but expensive memory and slower, cheaper options (Borkar & Chien, 2011).

The placement of these technologies within the hierarchy influences overall system performance. For example, SRAM-based caches reduce the average memory access time,

while DRAM allows systems to maintain larger memory pools. Choosing the right memory technology for each level of the hierarchy can make a significant difference in reducing latency, maximizing throughput, and optimizing cost-performance ratios.

Advance Cache Optimization

Caches are crucial in reducing the time processors spend accessing data from main memory. Advanced techniques such as **prefetching**, **victim caches**, and **cache partitioning** further enhance cache performance.

- **Prefetching**: This technique anticipates future memory accesses based on access patterns and brings data into the cache before it is explicitly needed. By doing so, it hides memory latency and ensures that required data is already available when the processor requests it. However, prefetching introduces a trade-off between accuracy and efficiency—too many incorrect prefetches can lead to cache pollution and wasted bandwidth (Jouppi, 1990).
- **Victim Caches**: These small caches store data evicted from the main cache (usually the L1 cache) before it is completely discarded. By holding onto recently evicted cache lines, victim caches give the system a second chance to access data without needing to go to main memory, thereby reducing conflict misses (Hennessy & Patterson, 2017).
- **Cache Partitioning**: This technique divides the cache into segments assigned to different cores or processes, ensuring that critical processes maintain enough

cache space to perform efficiently. This prevents one process from monopolizing cache space at the expense of others, a situation known as cache contention. Dynamic cache partitioning can allocate cache space based on the real-time needs of each core, improving overall system throughput (Kim, 2010).

These techniques work together to minimize **cache misses**, which significantly improves system performance by reducing the time processors spend waiting for data (Stallings, 2015).

Virtual Memory and Virtual Machines

Virtual memory is a key component in modern systems, providing a layer of abstraction between physical and virtual memory spaces. This allows systems to run larger applications and manage memory more efficiently by using page tables to map virtual addresses to physical memory (Hennessy & Patterson, 2017).

Page replacement algorithms like least recently used (LRU) ensure that frequently used data stays in faster memory while less-used data is swapped out to secondary storage, balancing memory usage across multiple applications (Stallings, 2015).

Virtual machines (VMs) add another layer of complexity by introducing additional levels of memory translation, but they enable efficient resource management and process isolation. Techniques like memory deduplication reduce the memory footprint by sharing common data between VMs, improving overall system performance (Barham et al., 2003).

Cross-Cutting Issues

Designing an efficient memory hierarchy involves navigating a series of trade-offs, including:

- **Cost**: SRAM is fast but expensive, while DRAM is cheaper but slower. Designing a system with a mix of these technologies requires balancing cost constraints with performance requirements (Hennessy & Patterson, 2017)
- **Power Consumption**: SRAM and high-speed caches consume more power compared to DRAM. Energy-efficient designs often require power-saving techniques, such as dynamic voltage and frequency scaling (DVFS), to adjust power levels based on workload demands. Cache power consumption also scales with the number of cache levels and size, so advanced techniques like adaptive cache sizing can be employed to reduce power in lower-demand scenarios (Borkar & Chien, 2011).
- **Complexity**: Adding advanced techniques like cache partitioning, prefetching, or multiple levels of caching increases system complexity. Managing this complexity requires sophisticated hardware and software support to ensure that the benefits of reduced latency and increased throughput outweigh the overhead costs of these techniques (Stallings, 2015).
- **Workload Sensitivity**: Different workloads, such as compute-bound tasks or data-heavy applications, benefit from different memory configurations. Memory hierarchy designs must consider these variations to maximize performance (Kim, 2010).

Finally, emerging technologies like **persistent memory** and **in-memory computing** are reshaping memory hierarchies by reducing latency and improving bandwidth, paving the way for faster, more efficient systems in the future (Borkar & Chien, 2011).

Part 2: Implementing and Analyzing Cache Configurations in gem5

After setting up the gem5 environment and build the X86 I followed the documentation for adding cache to the configuration script.

I have created a cache.py script

```
import m5
from m5.objects import Cache

# Add the common scripts to our path
m5.util.addToPath("../../")

from common import SimpleOpts

# Some specific options for caches
# For all options see src/mem/cache/BaseCache.py


class L1Cache(Cache):
    """Simple L1 Cache with default values"""

    assoc = 2
    tag_latency = 2
    data_latency = 2
    response_latency = 2
    mshrs = 4
    tgts_per_mshr = 20

    def __init__(self, options=None):
        super().__init__()
        pass

    def connectBus(self, bus):
        """Connect this cache to a memory-side bus"""
        self.mem_side = bus.cpu_side_ports

    def connectCPU(self, cpu):
        """Connect this cache's port to a CPU-side port
        This must be defined in a subclass"""
        raise NotImplementedError
```

```python
class L1ICache(L1Cache):
    """Simple L1 instruction cache with default values"""

    # Set the default size
    size = "16kB"

    SimpleOpts.add_option(
        "--l1i_size", help=f"L1 instruction cache size. Default: {size}"
    )

    def __init__(self, opts=None):
        super().__init__(opts)
        if not opts or not opts.l1i_size:
            return
        self.size = opts.l1i_size

    def connectCPU(self, cpu):
        """Connect this cache's port to a CPU icache port"""
        self.cpu_side = cpu.icache_port


class L1DCache(L1Cache):
    """Simple L1 data cache with default values"""

    # Set the default size
    size = "64kB"

    SimpleOpts.add_option(
        "--l1d_size", help=f"L1 data cache size. Default: {size}"
    )

    def __init__(self, opts=None):
        super().__init__(opts)
        if not opts or not opts.l1d_size:
            return
        self.size = opts.l1d_size

    def connectCPU(self, cpu):
        """Connect this cache's port to a CPU dcache port"""
        self.cpu_side = cpu.dcache_port

```

```python
class L2Cache(Cache):
    """Simple L2 Cache with default values"""

    # Default parameters
    size = "256kB"
    assoc = 8
    tag_latency = 20
    data_latency = 20
    response_latency = 20
    mshrs = 20
    tgts_per_mshr = 12

    SimpleOpts.add_option("--l2_size", help=f"L2 cache size. Default: {size}")

    def __init__(self, opts=None):
        super().__init__()
        if not opts or not opts.l2_size:
            return
        self.size = opts.l2_size

    def connectCPUSideBus(self, bus):
        self.cpu_side = bus.mem_side_ports

    def connectMemSideBus(self, bus):
        self.mem_side = bus.cpu_side_ports

```

```
brendan-yeong@brendan-yeong-VirtualBox:~/MSCS-01/gem5$ build/X86/gem5.opt configs/learning_gem5/part1/two
_level.py --l2_size='1MB' --l1d_size='128kB' --l1i_size='16kB'
gem5 Simulator System.  https://www.gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 version 24.0.0.1
gem5 compiled Sep  6 2024 13:43:30
gem5 started Oct  4 2024 16:01:19
gem5 executing on brendan-yeong-VirtualBox, pid 12742
command line: build/X86/gem5.opt configs/learning_gem5/part1/two_level.py --l2_size=1MB --l1d_size=128kB
--l1i_size=16kB

Global frequency set at 1000000000000 ticks per second
src/mem/dram_interface.cc:690: warn: DRAM device capacity (8192 Mbytes) does not match the address range
assigned (512 Mbytes)
src/base/statistics.hh:279: warn: One of the stats is a legacy stat. Legacy stat is a stat that does not
belong 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 58125000 because exiting with last active thread context
```

After running the newly file with configurations this is the out with “Hello world!”

```
system.cpu.dcache.demandHits::cpu.data          1956                       # number of demand (read+write) hits (Count)
system.cpu.dcache.demandHits::total             1956                       # number of demand (read+write) hits (Count)
system.cpu.dcache.overallHits::cpu.data         1956                       # number of overall hits (Count)
system.cpu.dcache.overallHits::total            1956                       # number of overall hits (Count)
system.cpu.dcache.demandMisses::cpu.data         134                       # number of demand (read+write) misses (Count)
system.cpu.dcache.demandMisses::total            134                       # number of demand (read+write) misses (Count)
system.cpu.dcache.overallMisses::cpu.data        134                        # number of overall misses (Count)
system.cpu.dcache.overallMisses::total           134                        # number of overall misses (Count)
system.cpu.dcache.demandMissLatency::cpu.data    14260000                       # number of demand (read+write) miss ticks (Tick)
system.cpu.dcache.demandMissLatency::total    14260000                      # number of demand (read+write) miss ticks (Tick)
system.cpu.dcache.overallMissLatency::cpu.data    14260000                       # number of overall miss ticks (Tick)
system.cpu.dcache.overallMissLatency::total    14260000                      # number of overall miss ticks (Tick)
system.cpu.dcache.demandAccesses::cpu.data        2090                       # number of demand (read+write) accesses (Count)
system.cpu.dcache.demandAccesses::total          2090                       # number of demand (read+write) accesses (Count)
system.cpu.dcache.overallAccesses::cpu.data        2090                       # number of overall (read+write) accesses (Count)
system.cpu.dcache.overallAccesses::total         2090                       # number of overall (read+write) accesses (Count)
system.cpu.dcache.demandMissRate::cpu.data    0.064115                       # miss rate for demand accesses (Ratio)
system.cpu.dcache.demandMissRate::total    0.064115                      # miss rate for demand accesses (Ratio)
system.cpu.dcache.overallMissRate::cpu.data    0.064115                       # miss rate for overall accesses (Ratio)
system.cpu.dcache.overallMissRate::total    0.064115                      # miss rate for overall accesses (Ratio)
system.cpu.dcache.demandAvgMissLatency::cpu.data 106417.910448                       # average overall miss latency in ticks ((Tick/Count))
system.cpu.dcache.demandAvgMissLatency::total 106417.910448                      # average overall miss latency in ticks ((Tick/Count))
system.cpu.dcache.overallAvgMissLatency::cpu.data 106417.910448                       # average overall miss latency ((Tick/Count))
system.cpu.dcache.overallAvgMissLatency::total 106417.910448                      # average overall miss latency ((Tick/Count))
system.cpu.dcache.blockedCycles::no_mshrs            0                       # number of cycles access was blocked (Cycle)
system.cpu.dcache.blockedCycles::no_targets            0                       # number of cycles access was blocked (Cycle)
system.cpu.dcache.blockedCauses::no_mshrs            0                       # number of times access was blocked (Count)
system.cpu.dcache.blockedCauses::no_targets            0                       # number of times access was blocked (Count)
system.cpu.dcache.avgBlocked::no_mshrs          nan                       # average number of cycles each access was blocked ((Cycle/Count))
system.cpu.dcache.avgBlocked::no_targets          nan                       # average number of cycles each access was blocked ((Cycle/Count))
system.cpu.dcache.demandMshrMisses::cpu.data          134                       # number of demand (read+write) MSHR misses (Count)
```

Here we can see the results:

Overall hits for the dcache: 1956

Overall dcache average miss latency rate: 0.061

```
(Tick)
system.cpu.interrupts.clk_domain.clock          16000                       # Clock period in ticks (Tick)
system.cpu.mmu.dtb.rdAccesses                    1138                       # TLB accesses on read requests (Count)
system.cpu.mmu.dtb.wrAccesses                     954                       # TLB accesses on write requests (Count)
system.cpu.mmu.dtb.rdMisses                        11                       # TLB misses on read requests (Count)
system.cpu.mmu.dtb.wrMisses                         9                       # TLB misses on write requests (Count)
system.cpu.mmu.dtb.walker.power_state.pwrStateResidencyTicks::UNDEFINED     58125000                       # Cumulative time (i
(Tick)
system.cpu.mmu.itb.rdAccesses                       0                       # TLB accesses on read requests (Count)
system.cpu.mmu.itb.wrAccesses                    8185                       # TLB accesses on write requests (Count)
system.cpu.mmu.itb.rdMisses                         0                       # TLB misses on read requests (Count)
system.cpu.mmu.itb.wrMisses                        37                       # TLB misses on write requests (Count)
system.cpu.mmu.itb.walker.power_state.pwrStateResidencyTicks::UNDEFINED     58125000                       # Cumulative time (i
(Tick)
```

Here we can see the TLB statistics.

References

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., & Warfield, A. (2003). Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, *37*(5), 164–177. https://doi.org/10.1145/1165389.945462

Borkar, S., & Chien, A. A. (2011). The future of microprocessors. *Communications of the ACM*, *54*(5), 67–77. https://doi.org/10.1145/1941487.1941507

Hennessy, J. L., & Patterson, D. A. (2019). *Computer Architecture: A quantitative approach*. Morgan Kaufmann Publishers.

Jouppi, N. P. (1990). Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *ACM SIGARCH Computer Architecture News*, *18*(2SI), 364–373. https://doi.org/10.1145/325096.325162

Stallings, W. (2016). *Computer Organization and Architecture: Designing for performance*. Pearson-Prentice Hall.