# Computer Architectural And Design

**Assignment 6:** Exploring Thread-Level Parallelism (TLP) in Shared-Memory Multiprocessors

**Nguyen The Duy Khanh**

005019181

## Table of Contents

## Understand Thread Level Parallelism (TLP)

### Thread-Level Parallelism Milestone

Thread-level parallelism (TLP) (Anita Johnson, 2023)  refers to the system's ability to execute multiple threads simultaneously, utilizing parallel processing capabilities to improve performance. Many hardware manufacturers and software programs have been developed to advance and use TLP to a higher degree to meet computational needs.

- **In the early pre-1990s**, the CPU only executed a specific workload within an application in a serialization order with one instruction processed at a time. To improve performance and meet the computational demand, mainframes and supercomputers (e.g., IBM's 360, Cray's systems) experimented with multi-tasking and multi-threading, but these were often confined to context switching and multiprogramming rather than true parallelism.
- **In the late 1990s and early 2000s**, multicore processors became a reality (e.g., Intel Pentium D with multicore processors, AMD Opteron with multicore designs, etc.). Multicore processors allow the simultaneous execution of multiple threads, making it possible to scale TLP without relying on the single-core's limitations. However, the challenges remain: how can various cores be utilized efficiently? This required both programming models and developer tools to manage parallelism to manage at the thread level.
- **In the early 2000s**, developers must manage thread creation, synchronization, and communication via explicit threading tools (e.g., OpenMP). However, as thread became more complex, higher-level programming models (e.g., Intel's Threading Building Blocks (TBB) and OpenCL) focused on task parallelism (Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, Michael Upton, 2002). In these models, the programmer specifies tasks or units of work, and

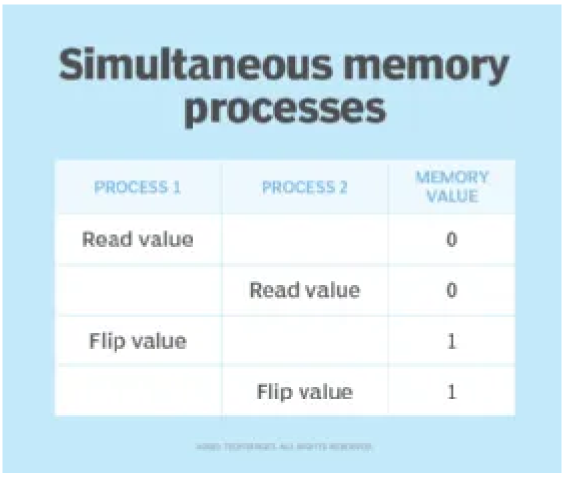
the system dynamically maps them to threads and cores. Additionally, programming languages such as Java, Python, and C++ introduced built-in concurrency models (e.g., Python Thread), which makes TLP more accessible and has continued until now.

- Besides software development, hardware is critical in enabling TLP in modern processors. In the 2000s, chip manufacturers began to include even more cores on a single die. This was driven by the limitations of clock speeds (which could not continue to increase) and the need for more parallel execution units. For example, Intel introduced quad-core processors (early 2000s) and later chips with up to 18 or more cores per processor in the 2010s, including the Xeon line. Besides the increase in core count, GPU (e.g., NVIDIA's CUDA) became a powerful alternative, especially in heterogeneous computing, compared to a traditional CPU core. GPUs are designed with hundreds or thousands of simple cores, making them highly suited for highly parallel workloads

Many challenges remain there even though there are so many advancements coming from software tools and hardware resources (Abde Mannaf Ghadiali, 2023):

- **Memory Hierarchy Bottlenecks**: Memory access becomes a significant bottleneck as cores increase. Cache coherence and memory consistency models are critical to ensuring efficient parallelism, but managing shared memory across many cores remains a considerable challenge.
- **Energy Efficiency**: As the number of cores increases, power consumption becomes a limiting factor. New architectures focus on optimizing energy usage while still achieving high TLP.

- **Concurrency bugs and race conditions:** Data corruption will happen when multiple threads access the shared resources simultaneously uncontrolled. This only appears when race conditions arise when two threads perform operations simultaneously for the same resources or deadlock when threads wait for each other to release resources. Even though there are mechanisms to prevent it (e.g., lock, atomicity transaction, mutexes) or detect it (e.g., ThreadSanitizer and Helgrind**)**, the cost of synchronization overhead still exists, and it can be complex when it involves an intensive workload application.



**Simultaneous memory processes**

| PROCESS 1 | PROCESS 2 | MEMORY VALUE |
|---|---|---|
| Read value | | 0 |
| | Read value | 0 |
| Flip value | | 1 |
| | Flip value | 1 |

**Figure 1:** Data Corruption in a thread-level environment without synchronization

- **Scalability and Amdahl's Law:** states that the speedup of a program using parallelism is limited by the serial portion of the program. Therefore, to counter the issue, algorithms need to be designed or restructured to reduce the serial portion, or the application's owner needs to break the task into independent tasks that are scheduled and executed independently to improve parallel efficiency.
- **Heterogeneous architectures:** (Abdezarak Ben Abdallah, 2017) with the CPU as the host and the GPU as the device. GPUs fundamentally differ from Central Processing Units (CPUs) regarding architecture and processing goals. While CPUs prioritize low-latency processing for single or limited-thread applications, GPUs are optimized

for high throughput, enabling them to handle thousands of threads simultaneously. Moving data between CPU and GPU memory also incurs significant latency and bandwidth constraints. To avoid that problem, many systems have used unified memory (used in CUDA and OpenCL), allowing GPUs and CPUs to access the same memory space

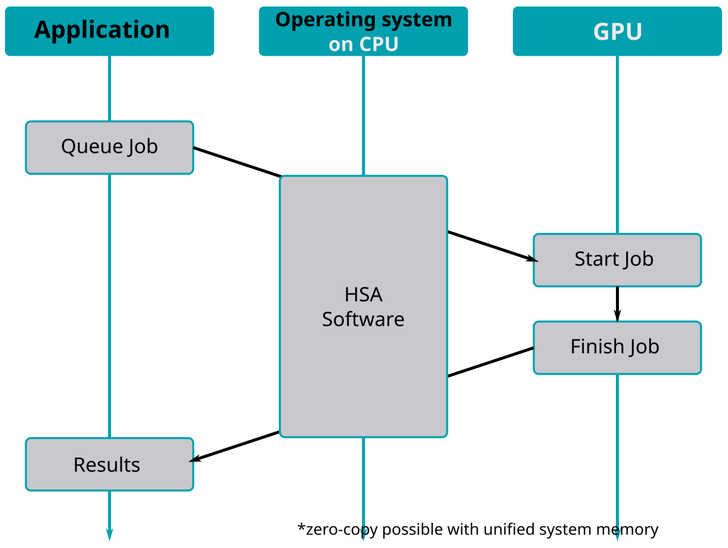


**Figure 2:** Heterogeneous architectures

Researchers are exploring various novel techniques and approaches to overcome the challenges of Thread-Level Parallelism (TLP) in contemporary systems. Some key developments include:

- **New Programming Models and Languages:** New extensive representation, such as Multi-Level Intermediate Representation, enables seamless integration across different platforms and compilers, supporting scalable, high-performance parallelism. MLIR facilitates better utilization of heterogeneous architectures (e.g., GPUs and TPUs) by lowering abstractions progressively and optimizing code automatically

- **Hardware Enhancements:** New cache coherence protocols (e.g., WARDen) (Peng Zhang, 2010) address multi-core systems' false sharing and true sharing issues. Traditional cache coherence mechanisms can be inefficient, particularly when managing benign data races or handling unnecessary synchronization
- **Compiler Optimizations:** intelligently distribute work and reduce the complexity of manual parallel coding**.** For example, TensorFlow now uses advanced compilers to optimize tensor computations and parallel tasks efficiently across different hardware architectures. These compilers are also experimenting with super optimization technique**s (**Socrates Wong, 2020), using reinforcement learning and constraint-solving methods to automatically select the most efficient parallel implementation for a given task
- **Runtime Systems:** New runtime systems are being developed to dynamically manage resources in TLP systems based on the availability of different hardware resources and manage synchronization across threads. For example, TensorFlow Runtime allows for better distribution of workloads, minimizing bottlenecks in heterogeneous systems

## Analyzing Core Concepts

To determine how parallel tasks are organized and executed in a TLP environment, the parallelism model will be the key factory and serve as a foundation framework. There are two primary models for expressing and managing parallelism in TLP systems:

- **Shared Memory:** (Pranav Bapat, 2024) All threads have access to a common memory space in the shared memory model. This allows for direct communication and synchronization between threads. Shared memory is commonly used in systems with multiple cores, or processors interconnected to a shared memory pool. The main challenge would involve avoiding data conditions and ensuring data consistency, which can be managed using a lock mechanism (e.g., semaphore, lock, etc.). However, this increases the cost of synchronization overhead and the complexity of managing within multi-core environments.

- **Message Passing:** (GeeksForGeeks, 2024) In the message-passing model, threads (often called processes in this context) communicate with one another by sending and receiving messages over explicit channels. There is no shared memory; each thread has private memory space.
- **Hybrid models (a combination of shared memory and message passing):** In heterogeneous computing systems, this is common where GPUs (which use message passing) interact with CPUs (which use shared memory).

In a TLP system, proper synchronization and communication between threads are critical to ensure correctness, consistency, and efficient use of system resources. Without synchronization, concurrent threads may interfere with each other, leading to data races or inconsistent states.

- **Synchronization:** Developers often utilize the following mechanisms: locks and mutexes, which only allow one thread to use resources until it’s not needed anymore; semaphores, which signal between threads; and atomic operations - which would enable threads to update shared data without the need for explicit locking- to ensure correctness and consistency.
- **Communication:** threads send and receive messages to pass data between them (e.g., MPI for distributed systems, ZeroMQ for inter-process communication). However, it can occur in race conditions or deadlocks, leading to corrupted data and causing inconsistency.

Efficient load balancing and scheduling are essential for maximizing the utilization of available cores and minimizing idle times.

- **Load balancing:** Load balancing involves distributing the computational work evenly across all threads (or cores) to prevent some threads from overloading while others are idle. There are two types of load balancing: Static Load Balancing, where the work is divided upfront based on estimated costs or predefined rules, and Dynamic Load Balancing, where workloads are redistributed during execution and tasks are rescheduled based on current load and system performance.

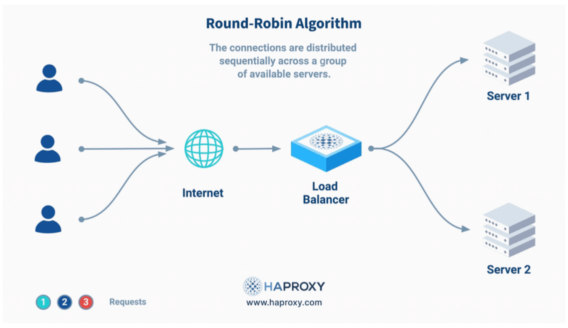


**Figure 3:** Load balancing

- **Scheduling:** Scheduling determines when each thread runs and on which core it executes. There are two types of scheduling: Preemptive Scheduling, where the operating system can interrupt a running thread to schedule another one, and Non-preemptive Scheduling, where the thread runs to completion unless it voluntarily yields control.

The effectiveness of TLP systems is measured using various performance metrics. Different metrics emphasize different aspects of performance, and there are often trade-offs between them.

- **Throughput:** refers to the amount of work completed over a given period. In TLP, this is typically the number of tasks or operations that can be processed in parallel.

Increasing throughput often means adding more threads or cores, which may come with overhead costs (e.g., synchronization, memory contention) that reduce individual thread performance.

- **Latency:** refers to the time to complete a single task or operation. In parallel systems, latency can be influenced by communication delays, synchronization, and load imbalances. Lower latency is often prioritized in real-time systems or applications with critical responsiveness. However, minimizing latency may limit throughput if parallelism is not fully exploited.
- **Scalability:** Scalability measures how well a TLP system can handle increased load (e.g., more threads, cores, or data). Not all systems scale linearly with the addition of resources. Scalability can be constrained by memory bandwidth, synchronization overhead, and diminishing returns from adding more cores.

## Synthesize Future Directions

As Thread-Level Parallelism (TLP) continues to evolve, several promising future directions and emerging trends could significantly shape the landscape of TLP research and its practical applications:

- **Hybrid Parallelism:** Combining SIMD and TLP could lead to more efficient use of hardware resources. For instance, multi-core CPUs with SIMD extensions can simultaneously exploit task- and data-level parallelism. This approach is already seen in AVX-512 instructions, where vectorization is combined with multi-core parallelism to handle intensive workloads**.**
- **Domain-Specific Accelerators**: Specialized hardware (e.g., GPUs, FPGAs, TPUs, etc) are already being used to accelerate certain computations. These accelerators are

designed with parallelism in mind and can greatly enhance the performance of specific TLP workloads, such as those in AI or high-performance computing (HPC).

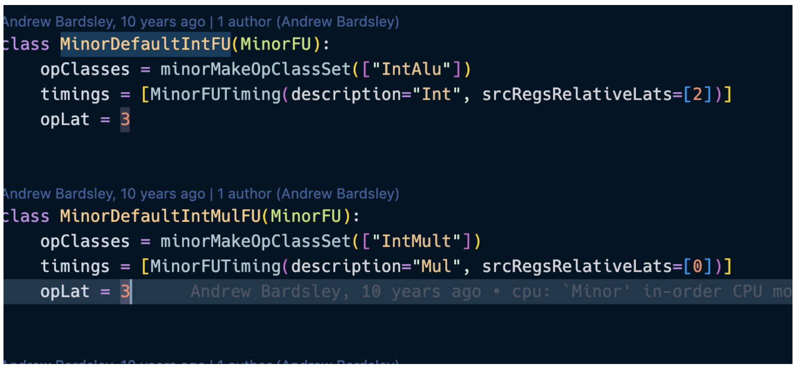
## Explored Shared-Memory Architectures with gem5

### MinorCPU

MinorCPU is an in-order CPU model with four fixed pipeline stages:

- **First Fetch:** fetches lines from memory
- **Second Fetch:** Decompose lines into macro-op instructions
- **Decode:** decompose macro-op instructions into micro-op instructions
- **Execute:** Execute those micro-op instructions

However, to execute those micro-op instructions, we would need a functional unit pool (FUPool) with the corresponding hardware resources (e.g., IntAlu, FloatMul).



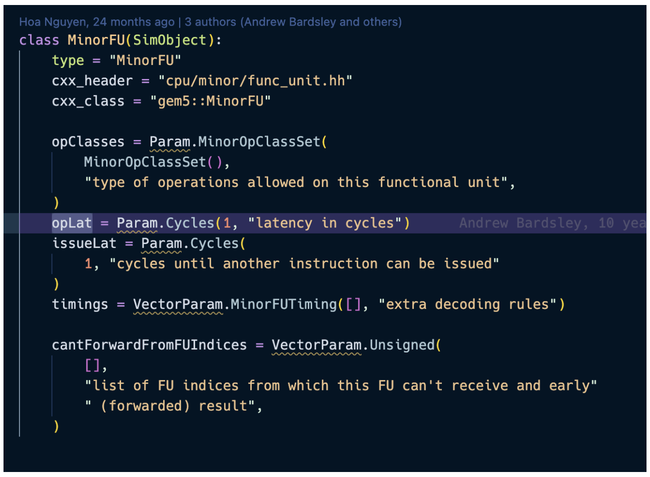
```
ndrew Bardsley, 10 years ago | 1 author (Andrew Bardsley)
lass MinorDefaultIntFU(MinorFU):
    opClasses = minorMakeOpClassSet(["IntAlu"])
    timings = [MinorFUTiming(description="Int", srcRegsRelativeLats=[2])]
    opLat = 3


ndrew Bardsley, 10 years ago | 1 author (Andrew Bardsley)
lass MinorDefaultIntMulFU(MinorFU):
    opClasses = minorMakeOpClassSet(["IntMult"])
    timings = [MinorFUTiming(description="Mul", srcRegsRelativeLats=[0])]
    opLat = 3      Andrew Bardsley, 10 years ago • cpu: `Minor' in-order CPU mo
```

**Figure 4:** Functional unit pool for each operation

Each functional unit pool has the following configuration:

- **Operation Latency (opLat):** how long the functional unit takes to complete an operation.
- **Issue Latency (issueLat):** the delay between issuing successive operations to the same unit

- **Operation Classes(opClasses): t**he type of operation to execute the micro-op instruction (e.g., IntAlu will execute integer arithmetic operations)

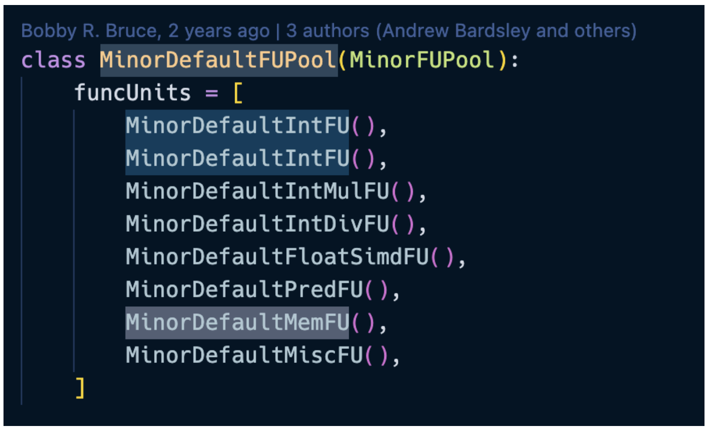


```
Hoa Nguyen, 24 months ago | 3 authors (Andrew Bardsley and others)
class MinorFU(SimObject):
    type = "MinorFU"
    cxx_header = "cpu/minor/func_unit.hh"
    cxx_class = "gem5::MinorFU"

    opClasses = Param.MinorOpClassSet(
        MinorOpClassSet(),
        "type of operations allowed on this functional unit",
    )
    opLat = Param.Cycles(1, "latency in cycles")      Andrew Bardsley, 10 yea
    issueLat = Param.Cycles(
        1, "cycles until another instruction can be issued"
    )
    timings = VectorParam.MinorFUTiming([], "extra decoding rules")

    cantForwardFromFUIndices = VectorParam.Unsigned(
        [],
        "list of FU indices from which this FU can't receive and early"
        " (forwarded) result",
    )
```

**Figure 5:** Functional unit pool's configuration

Currently, MinorCPU has seven different functional unit pools, and each serves a different purpose to execute micro-ops instructions:

- **MinorDefaultIntFU:** handles integer arithmetic operations
- **MinorDefaultIntMulFU:** handles integer multiplication operations
- **MinorDefaultIntDivFU:** handles integer division operations
- **MinorDefaultFloatSimdFU:** handles SIMD operations
- **MinorDefaultPredFu:** resolves control flow instructions to guide instruction fetch
- **MinorDefaultMemFU:** handles memory-related instructions
- **MinorDefaultMiscFU:** executes remaining instruction that is not covered by the specialized FUs

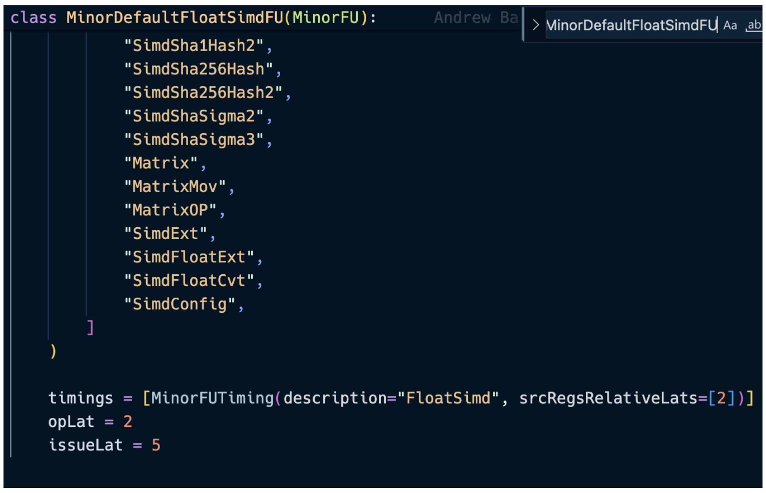


```
Bobby R. Bruce, 2 years ago | 3 authors (Andrew Bardsley and others)
class MinorDefaultFUPool(MinorFUPool):
    funcUnits = [
        MinorDefaultIntFU(),
        MinorDefaultIntFU(),
        MinorDefaultIntMulFU(),
        MinorDefaultIntDivFU(),
        MinorDefaultFloatSimdFU(),
        MinorDefaultPredFU(),
        MinorDefaultMemFU(),
        MinorDefaultMiscFU(),
    ]
```

**Figure 6:** Default functional unit pools for MinorCPU

### FloatSimd Design Space Exploration

When explorative testing with different combinations of operation latency and issue latency that sum up to 7, the simulation latency hasn't changed over time, and the same can be said for the number of instruction cycles. However, if we raise the number to 3 figures, there will be a small change.



```
class MinorDefaultFloatSimdFU(MinorFU):          Andrew Ba
                "SimdSha1Hash2",
                "SimdSha256Hash",
                "SimdSha256Hash2",
                "SimdShaSigma2",
                "SimdShaSigma3",
                "Matrix",
                "MatrixMov",
                "MatrixOP",
                "SimdExt",
                "SimdFloatExt",
                "SimdFloatCvt",
                "SimdConfig",
            ]
        )

    timings = [MinorFUTiming(description="FloatSimd", srcRegsRelativeLats=[2])]
    opLat = 2
    issueLat = 5
```

**Figure 7:** Changing opLat and issueLat with MinorDefaultFloatSimdFU

Additionally, when the issue latency increases, the idle cycle increases from 22516 for issue latency 0 to 22521 for issue latency 6. It is expected that when the FU cannot accept a new instruction due to a high issue lat, it forces the pipeline to wait before dispatching the next instruction.

This delays subsequent stages of the pipeline, increasing the total time the pipeline remains active.

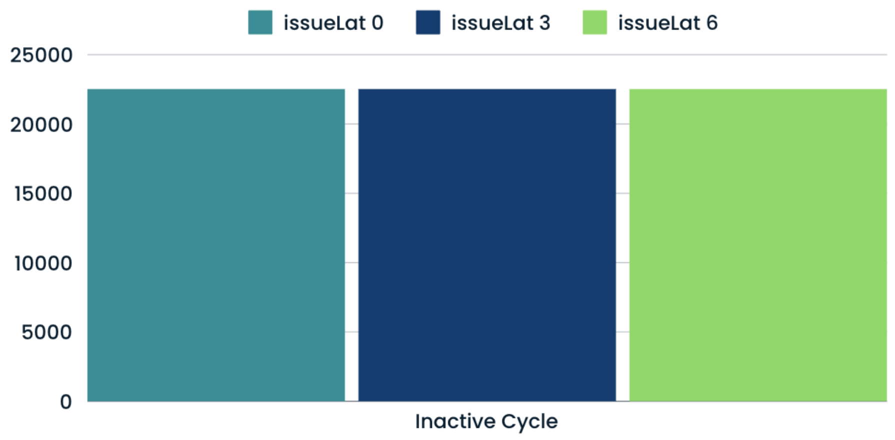


**Figure 8:** Inactive cycle's comparison for different issue latency

The optimal balance between opLat and issueLat depends on several factors, including the workload's dependency characteristics, the hardware's pipeline structure, and the number of available threads

- **High opLat / Low issueLat:** relies heavily on thread-level parallelism to hide latency. Multiple threads must be active and ready to issue instructions to ensure the hardware is always busy.
- **Low opLat / High issueLat:** emphasizes faster execution per thread but requires fewer concurrent threads to achieve peak utilization.
- **Balanced opLat and issueLat:** By keeping opLat manageable and issueLat short enough to maintain throughput, hardware can balance execution efficiency with parallel task dispatch.

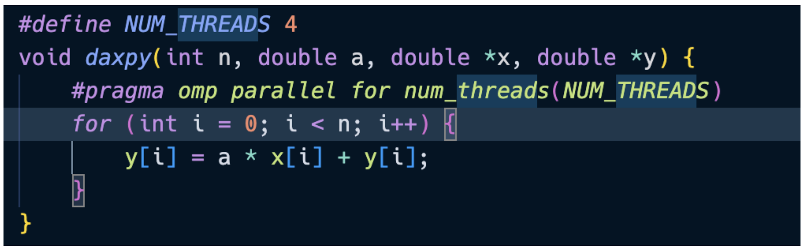
Maximizing parallel speedup involves achieving a balance such that:

- Thread-level parallelism compensates for high operation latency (when opLat > issueLat).

- Instructions are issued efficiently without overburdening the hardware with dependencies or resource contention

## Multi-Threaded Daxpy Kernel Simulation

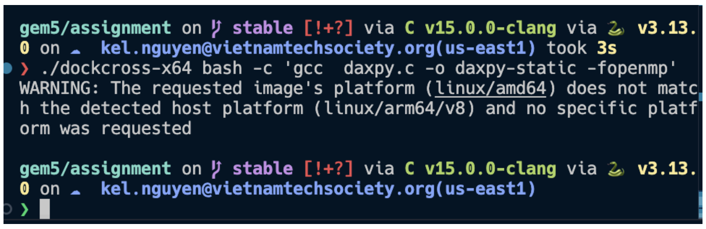
To implement daxpy in a multi-thread environment, the following annotation can be used for parallelism with OpenMP:



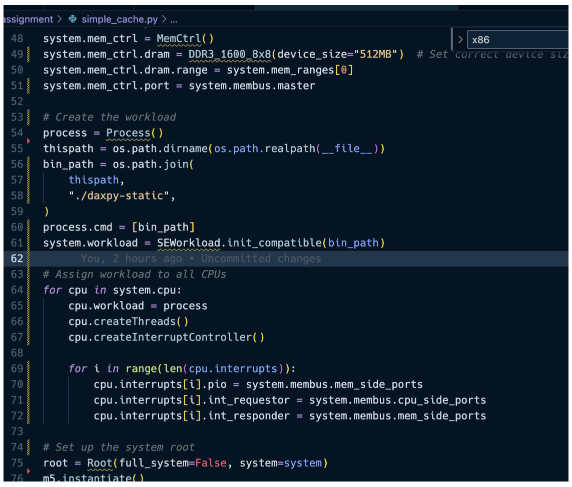
```c
#define NUM_THREADS 4
void daxpy(int n, double a, double *x, double *y) {
    #pragma omp parallel for num_threads(NUM_THREADS)
    for (int i = 0; i < n; i++) {
        y[i] = a * x[i] + y[i];
    }
}
```

**Figure 9:** Daxpy Computations with multi-thread / multi-core environment.

Then, we would use the following command with -fopenmp configuration to enable OpenMP within C x64 Artifacts for x86 ISA:



```
gem5/assignment on  stable [!+?] via C v15.0.0-clang via 🐍 v3.13.
0 on ☁  kel.nguyen@vietnamtechsociety.org(us-east1) took 3s
❯ ./dockcross-x64 bash -c 'gcc  daxpy.c -o daxpy-static -fopenmp'
WARNING: The requested image's platform (linux/amd64) does not matc
h the detected host platform (linux/arm64/v8) and no specific platf
orm was requested

gem5/assignment on  stable [!+?] via C v15.0.0-clang via 🐍 v3.13.
0 on ☁  kel.nguyen@vietnamtechsociety.org(us-east1)
❯ 
```



ssignment > simple_cache.py > ...

```python
system.mem_ctrl = MemCtrl()
system.mem_ctrl.dram = DDR3_1600_8x8(device_size="512MB")  # Set correct device siz
system.mem_ctrl.dram.range = system.mem_ranges[0]
system.mem_ctrl.port = system.membus.master

# Create the workload
process = Process()
thispath = os.path.dirname(os.path.realpath(__file__))
bin_path = os.path.join(
    thispath,
    "./daxpy-static",
)
process.cmd = [bin_path]
system.workload = SEWorkload.init_compatible(bin_path)
          You, 2 hours ago • Uncommitted changes
# Assign workload to all CPUs
for cpu in system.cpu:
    cpu.workload = process
    cpu.createThreads()
    cpu.createInterruptController()

    for i in range(len(cpu.interrupts)):
        cpu.interrupts[i].pio = system.membus.mem_side_ports
        cpu.interrupts[i].int_requestor = system.membus.cpu_side_ports
        cpu.interrupts[i].int_responder = system.membus.mem_side_ports

# Set up the system root
root = Root(full_system=False, system=system)
m5.instantiate()
```

**Figure 10:** Daxpy's simulation within Gem5

As a result of the multi-core and multi-thread environment, the system has utilized all the core processor's resources to the fullest, resulting in the following metrics:

- Simulation time:

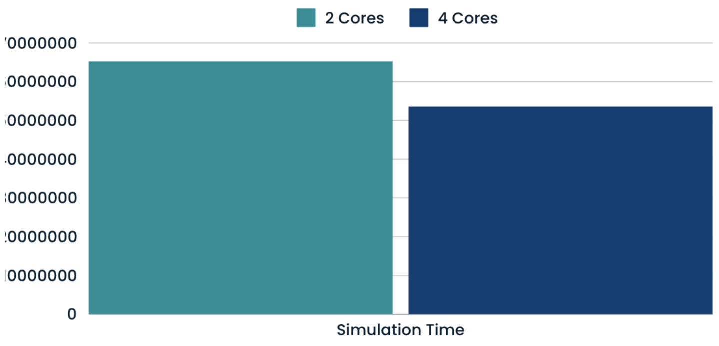


**Figure 11:** Simulation time comparison for different cores with Minor CPU

- Instruction Per Cycle:

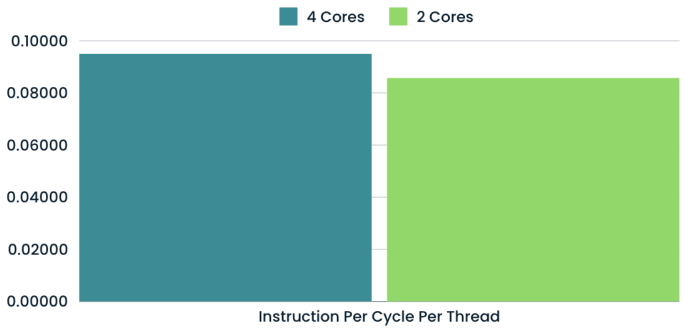


**Figure 12:** Instruction Per Cycle for different cores with Minor CPU

- Cycle Per Instruction

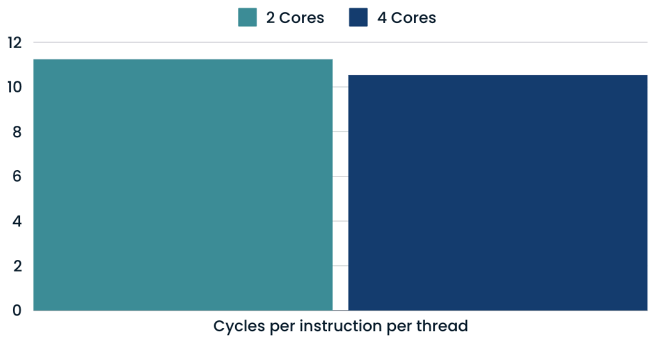


**Figure 13:** Cycles per instruction for different cores with Minor CPU

However, when enabled with an out-of-order pipeline CPU (e.g O3CPU), the simulation result has been improved further:

- Simulation time:

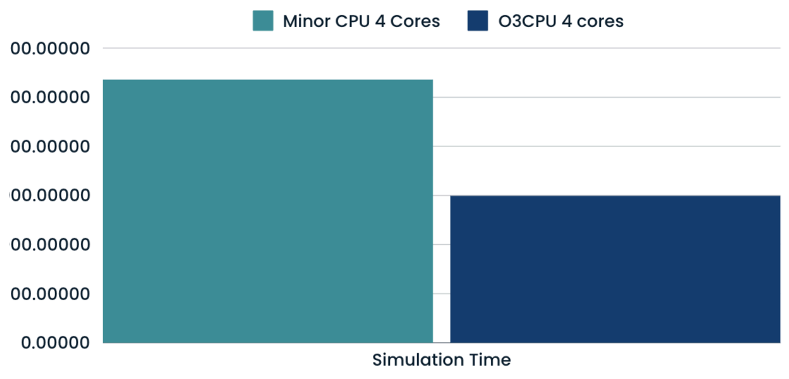


**Figure 14:** Simulation time comparison with different CPUs (O3CPU vs Minor CPU)

- Instruction Per Cycle (per thread)

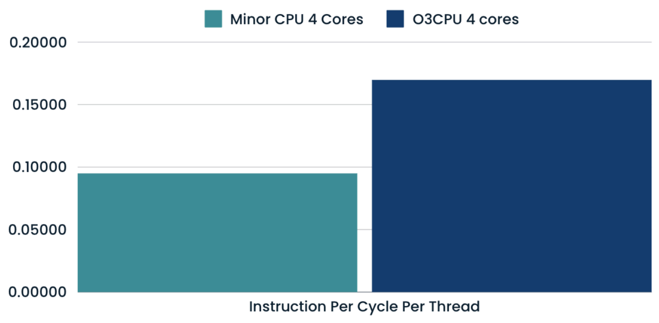


**Figure 15:** Instruction Per Cycle  comparison with different CPUs (O3CPU vs Minor CPU)

- Cycle Per Instruction (per thread)

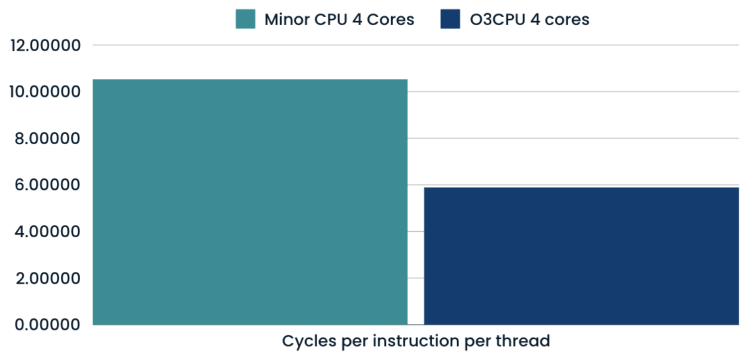


**Figure 16:** Cycle Per Instruction comparison with different CPUs (O3CPU vs Minor CPU)

## References

Hennessy, J. L., & Patterson, D. A. (2017). *Computer Architecture: A Quantitative Approach* (6th ed.). Morgan Kaufmann.

Anita Johnson (2023). *What is thread-level parallelism in computer architecture?* Architecture.

Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, Michael Upton (2002). *Hyper-Threading Technology Architecture and Microarchitecture.* Intel Technical Journal.

Abde Mannaf Ghadiali (2023). Computer System Architecture Part 2 - Thread Level Parallelism. Medium.

Abdezarak Ben Abdallah (2017). *Heterogeneous Computing: An Emerging Paradigm of Embedded System Design*. Science Direct.

Peng Zhang (2010). *Advanced Industrial Control Technology*. Science Direct.

Socrates Wong (2020). *Super Optimization.* Cornell University.

Pranav Bapat (2024). *What is shared memory?* GeeksForGeeks

GeeksForGeeks (2024). *Message passing in the distributed system*.

Khanh Nguyen. https://github.com/khanhntd/MSCS531_Assignment6