Computer Architectural And Design

Assignment 4: Exploring Instruction-Level Parallelism (ILP) in Modern Processors

Nguyen The Duy Khanh

005019181

Table of Contents

Understand Instruction Level Parallelism (ILP)

The evolution of ILP has been a cornerstone of modern computer architecture, driven by innovations to maximize the number of instructions executed per clock cycle:

- In 1965, IBM's System/360 Model 91 introduced the first pipelined processor, allowing multiple stages of instruction execution to overlap it.
- In late 1960, Tomasulo's algorithm was introduced to handle out-of-order execution in the IBM System/360 Model 91. It allows instructions to execute as soon as their operands are available rather than in program order, mitigating pipeline stalls.
- In the early 1990s, Intel, IBM, and other companies began implementing superscalar architecture**s** in general-purpose CPUs (e.g., Intel Pentium (1993) and IBM PowerPC)
- In 1992, IBM's POWER processors formalized the use of reorder buffers (ROBs) to maintain precise exceptions in out-of-order execution, allowing for speculative execution while preserving program order for committed instructions.
- In 1996, the Alpha 21264 (1996) by Digital Equipment Corporation introduced two-level adaptive branch prediction, providing high accuracy and reducing mispredictions' performance cost.
- In 2002, Intel introduced Hyper-Threading Technology with its Pentium 4 processor. SMT allowed multiple threads to issue instructions in parallel on the same core, increasing ILP by making better use of execution units that might otherwise remain idle.
- At the point at which further ILP yields minimal performance benefits, the processor design started to shift toward multicore architectures in 2003 (e.g., AMD and Intel began focusing on Thread-Level Parallelism (TLP) through multicore designs)

At the end of 2002, multiple chip manufacturers faced the wall of ILP. However, ILP still serves as an essential foundation for optimizing CPU performance. Before we come to an ILP's challenges, we need to understand what ILP is and how it is used in processors.

ILP is a potential overlap used in the execution of instructions by using pipelines to improve the system's performance. Instruction-level parallelism measures how many operations are executed simultaneously. Every modern high-performance CPU has the following three main components for ILP:

- **Pipelining** is based on the idea that a single instruction can often take quite a while to execute. Still, it only uses a specific processor region at any given time. (e.g., when reading from memory, some instructions are being fetched; when figuring out what the instruction does, some are being decoded, etc.)
- **Superscalar:** duplicating certain CPU features to support multiple instructions in each pipeline stage (e.g., by copying the decode stage into four different decoders, the CPU can decode the four instructions simultaneously).
- **Out-of-order execution (OOE)** gives processors the freedom to execute instructions in a different order than they're laid out in memory, as long as it guarantees the correctness of results.

Pipelining and superscalar execution alone aren't enough to extract maximum performance from CPUs since the CPU is still limited by how the instructions are laid out in memory. Only when all of these together (including OOE), plus several others, can the CPU retire up to 4 or even more instructions per clock cycle (IPC). The CPU isn't just working on one instruction at a time but many times more than that (e.g., the Apple M1 reorder buffer has 600 entries, potentially hundreds of micro-operations can be in flight at any given time).

To determine the effectiveness of a single ILP feature, we can use the following metrics:

- **Latency (ticks):** signifies the total time the simulated environment has run for. The

higher the latency, the higher the time it takes to calculate the corresponding algorithms / execute the instruction sets.

- **Simulator Instruction Rate(number of instructions per second):** the total of instructions the CPU core needs to execute per second. The higher the rate is, the higher the power consumption per second.
- **Energy Consumptions (nJ):** signifies the total consumption that a given application uses.

To optimize these metrics further, the CPU would need more complex features (e.g., deeper pipelines, wider issue widths, etc). Even though these features improve ILP, they add significant complexity to the processor. Additionally, increasing IPC with additional execution units and speculative execution techniques consumes more power. At a certain point, increasing IPC yields smaller performance gains due to dependencies in code that limit parallel execution.

Even with all the benefits that ILP can bring to an architecture, it imposes several constraints:

- **Data Dependencies:** occur when two instructions depend on each other (e.g., read-after-write, write-after-write, write-after-read). These instructions serialize instruction execution, limiting parallelism. To resolve it, the architecture can use Instruction Scheduling (by re-ordering, grouping, and executing the instructions that do not depend on each other) and Register Renaming (if there are no dependant on each other, use a different physical register instead)
- **Control Hazards:** occur when branching instructions (e.g., If there is a conditional branching instruction and leads to different results based on the condition, we will not able to execute the following instruction but have to depend on the condition

beforehand). To resolve it, the architecture can use branch prediction (guess the outcome of conditional branches and execute subsequent instructions ahead of time).

- **Resource Limitation:** occurs when the number of functional units, register files, and memory bandwidth is limited. Techniques such as Superscalar architectures (duplicate certain CPU features to support multiple instructions in each pipeline stage simultaneously) are standard solutions to this challenge.

When designing a processor with ILP, it imposes several challenges:

- **Complexity with minimum returns:** ILP features (e.g., out-of-order execution, registering renaming**)** have matured enough. Therefore, exploiting the features and increasing the complexity of ILP features even more only returns a minimum (John L. Hennessy, David A. Patterson, 2011).
- **Thermal constraints:** ILP techniques like speculative execution and complex Out-of-order pipeline designs significantly increase power consumption, as they require extensive hardware resources (e.g., large reorder buffers, branch prediction tables) (Mark Oskin, 1999). Additionally, increasing power densities create thermal hotspots, making it harder to cool processors effectively. Therefore, modern processors face strict power budgets, especially in mobile and embedded systems, making it difficult to justify additional ILP optimizations.

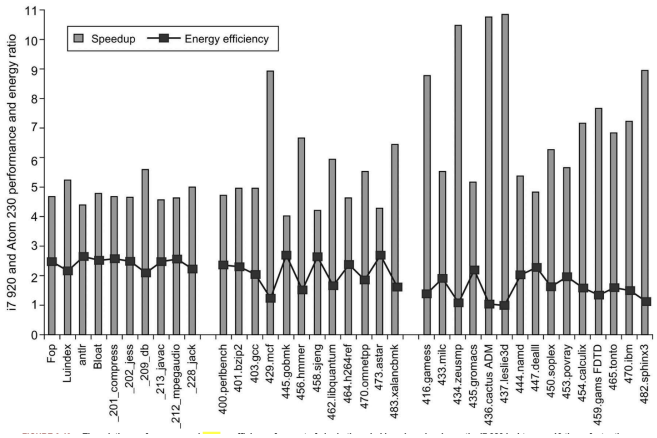


Figure 1: Energy efficiency for different i7 processors

To encounter the challenges, there are several solutions that a microprocessor can use:

- **Heterogeneous system architectures (HSA):** Instead of pushing for greater ILP in a single, complex core, researchers increasingly focus on heterogeneous architectures that use cores with varying capabilities (e.g., NVIDIA Cuda). This allows high-performance cores to handle intensive tasks while more energy-efficient cores handle more straightforward tasks, improving overall performance and energy efficiency. The rationale behind HSA is to ease the burden on programmers when offloading calculations to the GPU. Initially driven solely by AMD and called the FSA, the idea was extended to encompass processing units other than GPUs. Modern GPUs are very well suited to perform single instruction, multi-data (SIMD), and single instruction, multi-threads (SIMT), while modern CPUs are still being optimized for branching.

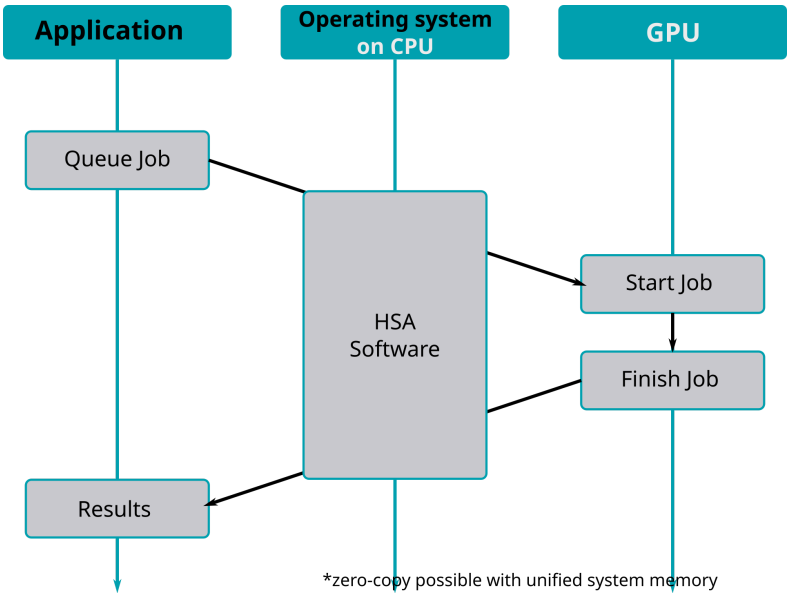


Figure 2: Heterogeneous system architectures workloads

- **Task-Level Parallelism (TLP)**: Thread-level parallelism (e.g., multicore processors) provides an alternative to ILP by focusing on running multiple threads or processes concurrently (e.g, the ILP structures from POWER 4 to Power 8 ISA went from 5 issues to 8, from 8 functional units to 16, whereas the SMT support when from non-existent to 8 threads/processor)

	Power4	Power5	Power6	Power7	Power8
Introduced	2001	2004	2007	2010	2014
Initial clock rate (GHz)	1.3	1.9	4.7	3.6	3.3 GHz
Transistor count (M)	174	276	790	1200	4200
Issues per clock	5	5	7	6	8
Functional units per core	8	8	9	12	16
SMT threads per core	0	2	2	4	8
Cores/chip	2	2	2	8	12
SMT threads per core	0	2	2	4	8
Total on-chip cache (MiB)	1.5	2	4.1	32.3	103.0

Figure 3: Comparison between different processors amongst SMT and ILP

- **Reduce power constraints:** Algorithms such as power gating or Dynamic Voltage and Frequency (DVFS) can lower the power consumption within the CPU. When there is no active usage/thread from the CPU or adjust the power/frequency on-demand, the CPU can mitigate power constraints associated with ILP hardware, allowing ILP techniques to be used more effectively within power limits.

Future research in ILP is increasingly focused on leveraging new architectural paradigms, specialized hardware, and intelligent optimizations to overcome the diminishing returns of traditional ILP techniques. In my opinion, the most promising trend for ILP would be: **Machine learning-based optimizations** could improve prediction accuracy, scheduling efficiency, and resource management to mitigate dependencies and minimize pipeline stalls. For example, neural branch predictors and reinforcement learning-based schedulers have

already shown promise in boosting ILP by making more accurate predictions and smarter scheduling decisions. Additionally, it can also used to accurately predict power consumption with minimal runtime overhead, which can be particularly advantageous for applications where traditional simulation models are impractical due to high computational demands based on certain workloads, which reduces power consumption in CPU architectures, critical for both mobile and data center applications where energy efficiency is paramount (Sami, Ajay, Hussam, 2022)

Practical Exploration of ILP Techniques

Basic Pipeline Simulation

To simulate a simple pipeline process, we will use the Hello World program and compile it with the corresponding artifacts. Since the default environment is Mac x64 ISA; we would need to use dockcross as a hypervisor container to build the artifacts in the corresponding environment (in this case, it is x86 ISA)

```c
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

Figure 4: Hello World Program

```
gem5 on  stable [!?] via  v3.12.5 on ☁  kel.nguyen@vietnamtechsociety.org(us-east1)
⊗ ❯ docker run --rm dockcross/linux-x86 ./dockercross-x86 chmod +x ./dockcross-x86 bash -c 'gcc hello.c -o hello64 -static'
```

Figure 5: Compile artifacts across x86 ISA

After building the artifacts, we will need to simulate the environment with the artifacts and turn on the trace to determine the underlying instruction set process. However, by default, TimingSimpleCPU() is an in-order CPU, so there won't be any pipeline.

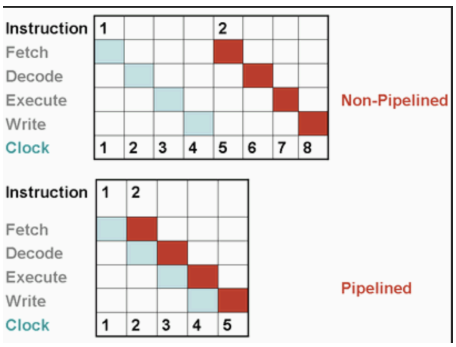


Figure 6: Non-Pipelined vs Pipelined

When generated with TimingSimpleCPU(), we won't have any pipeline to view. Instead of using TimingSimpleCPU(), we will use Out Of Order CPU (O3CPU)

```
# Create CPU and Memory Bus
system.cpu = O3CPU()
system.membus = SystemXBar()
```

Figure 7: Out of Order CPU (O3CPU) for simulation

Therefore, when running the following command to enable trace, which will track the end-to-end request workflow and the interaction between components, the pipeline will show each instruction stage: fetch, decode, rename, execute, and commit.

```
gem5 on  stable [!?] via  v3.12.5 on  kel.nguyen@vietnamtechsociety.org(us-east1)
❯ ./build/x86/gem5.opt --debug-flags=O3PipeView --debug-file=trace.out ./assignment/simple_cache.py --cpu-type=detailed --caches -c ./hello64-static
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 Oct 17 2024 00:46:04
gem5 started Oct 18 2024 23:33:20
gem5 executing on Kels-MacBook-Pro.local, pid 38993
command line: ./build/x86/gem5.opt --debug-flags=O3PipeView --debug-file=trace.out ./assignment/simple_cache.py --cpu-type=detailed --caches -c ./hello64-stat
ic

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 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 depre
cated.
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 19877000 because exiting with last active thread context
```

```
gem5 on  stable [!?] via  v3.12.5 on  kel.nguyen@vietnamtechsociety.org(us-east1)
❯ ./util/o3-pipeview.py -c 500 -o pipeview.out --color m5out/trace.out

Processing trace...  done!
```

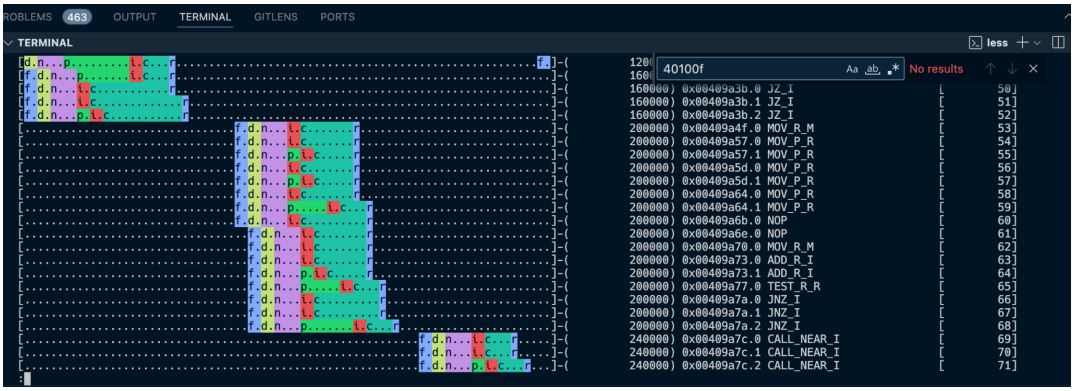


Figure 8: Trace and pipeline visualization with Gem5

In addition to the pipeline, we can also track the instruction set for a given artifacts

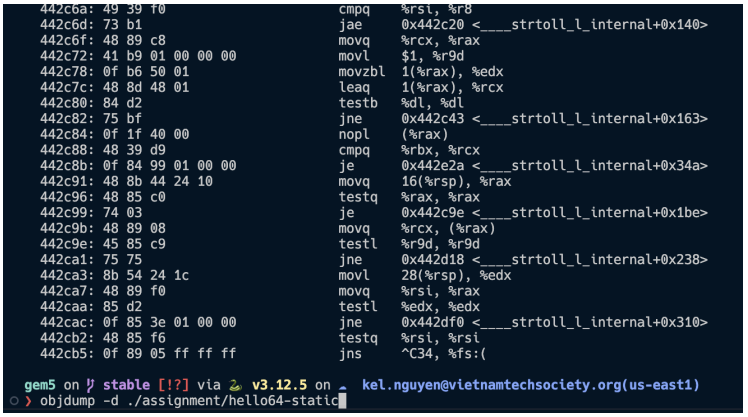


Figure 9: Hello World's Instruction Sets

Branch prediction

Branch prediction is a technique to enhance ILP by allowing the processor to guess the outcome of conditional branches and execute subsequent instructions ahead of time. It can help the processors to utilize all the internal components (e.g when the control unit works is done fetching the instruction, it directly jumps to another set of instructions, and at the same time, the ALU will decode the instructions), reducing wasteful cycles and improving throughput, which can also happen during the branch is stalling. However, there is a risk of wasting resources and time if misspeculation is used. One of the ways is to maintain a branch

history table [3] - a table that keeps track of recent branches' outcomes and makes predictions based on that history.

```
# Create CPU and Memory Bus
system.cpu = O3CPU()
system.membus = SystemXBar()

system.cpu.branchPred = LocalBP()
```

Figure 10: Local branch prediction

When simulating with and without branch prediction, the difference can be clearly seen:

- **Latency:** decrease from 41984000 to 19391000 ticks - 53.81%

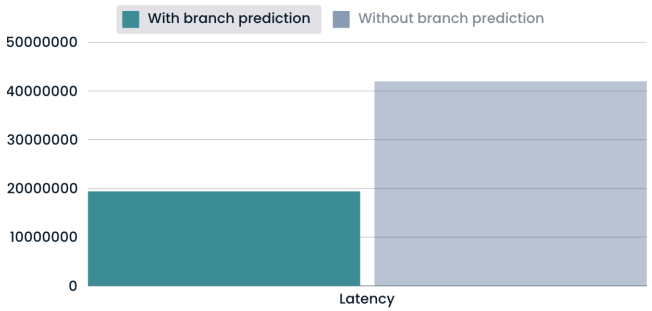


Figure 11: Branch prediction's improvement for latency (ticks)

- **Simulator Instruction rates:** decrease from 200835 to 80245 instructions/seconds - 60.04%

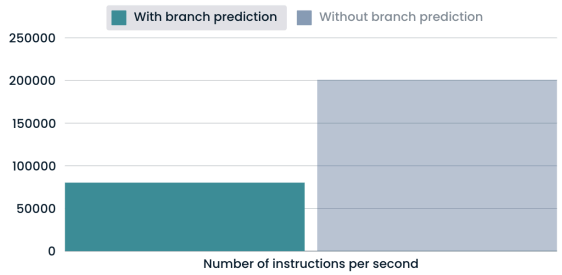


Figure 12: Branch prediction's improvement for the number of instructions (instructions/second)

Superscalar processor

Superscalar is a technique to enhance ILP by duplicating certain CPU features to support multiple instructions in each pipeline stage (e.g. by duplicating the decode stage into four different decoders, the CPU can decode the four instructions simultaneously). To configure the super scalar parameters, we can change the CPU configurations directly:

```
# Create CPU and Memory Bus
system.cpu = O3CPU()
system.membus = SystemXBar()

# Modify superscalar parameters
system.cpu.fetchWidth = 12    # Fetch up to 12 instructions per cycle
system.cpu.issueWidth = 12    # Issue up to 12 instructions per cycle
system.cpu.dispatchWidth = 12 # Dispatch up to 12 instructions per cycle
system.cpu.commitWidth = 12   # Commit up to 12 instructions per cycle
```

Therefore, the difference between enabling super scalar and without enabling it can be clearly seen in the metrics:

- **With integer operations:** decrease from 59863500 to 31176000 ticks for latency and 164870 to 75001 instructions per second

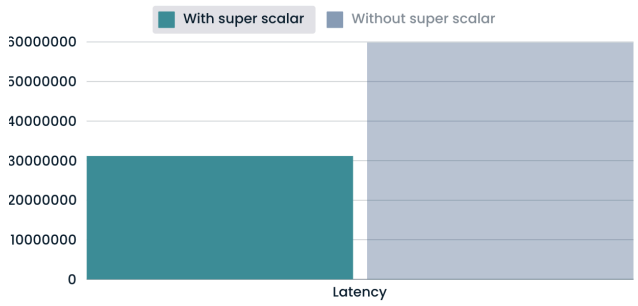


Figure 13: Super scalar's improvement of integer operations for latency(ticks)

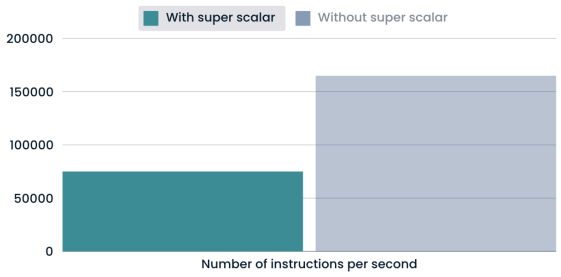


Figure 14: Super scalar's improvement of integer operations for the number of instructions (instructions/second)

- **With memory operations:** decrease from 63973000 to 30711500 ticks for latency and 315410 to 23487 instructions per second

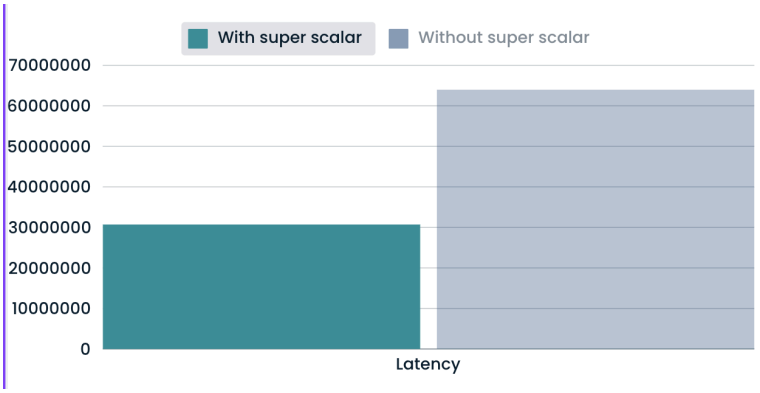


Figure 15: Super scalar's improvement of memory operations for latency(ticks)

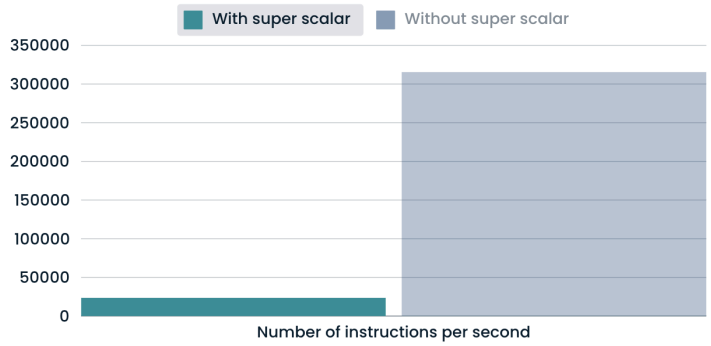


Figure 16: Super scalar's improvement of memory operations for latency(ticks)

Superscalar shines the most when coming to memory operations. It can be easily explained since memory operations often benefit from data locality[5], meaning that if a program accesses one memory location, it will likely access nearby locations soon after. This spatial locality allows caches to prefetch data effectively, improving performance for memory-bound workloads.

Multithreading

Multithreading can significantly improve uniprocessor throughput by allowing

multiple threads to share the execution resources of a single core and avoid single-point-of-failure threads due to stalling or long-latency instructions. ILP executes instructions from a single thread by parallelism the instructions; however, for multithreading, it enables the CPU to use multiple threads to execute. To enable multithreading, we can configure the CPU directly with as follows:

```
system.cpu = O3CPU(numThreads=2)
system.multi_thread = True
```

Figure 17: Enable multi-thread in O3 CPU

As Gem5 only allows to run multi-thread when and only when the number of processes are equal to the number of threads, we have to add an additional process as a CPU workload, and it needs to have a different PID (process identifier) to run successfully.

```
thispath = os.path.dirname(os.path.realpath(__file__))
binpath = os.path.join(thispath, "../", "tests/test-progs/hello/bin/x86/linux/hello")
# Set up processes for SMT
process_1 = Process(pid=102, cmd = binpath)
process_2 = Process(pid=101, cmd = binpath)

# Assign processes to CPU workload
system.cpu.workload = [process_1, process_2]
system.cpu.createThreads()
```

Figure 18: Simulated two separate processes in a multi-thread environment

Therefore, the difference between allowing SMT and without SMT can be clearly seen in the metrics:

- **Latency:** decrease from 41984000 to 19391000 ticks - 8.14%

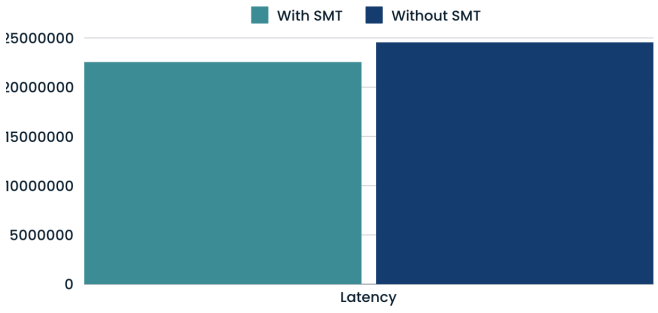


Figure 19: SMT's improvement for latency(ticks)

However, the more processes the simulation has while the resources stay the same, the more the simulation with SMT will behave the same in the serialization environment due to the following reasons:

- In an SMT system, multiple threads share the same core resources, including the pipeline stages, execution units, and caches.
- If two or more threads frequently require the same execution units (e.g., ALUs or load/store units), they'll often need to wait for the other thread to finish, leading to stalls
- Shared caches (L1, L2, etc.) also limit performance in SMT because multiple threads compete for cache lines

Simulation Challenges

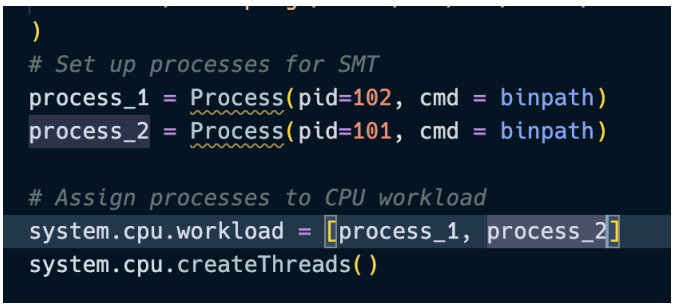
When running the simulation in SE mode and the number of workloads is less than the number of threads, Gem5 will throw an error

```
if (!FullSystem && params().workload.size() != numThreads) {
    fatal("Number of processes (cpu.workload) (%i) assigned to the CPU "
          "does not equal number of threads (%i).\n",
          params().workload.size(), numThreads);
}
```

Figure 20: Assertion for ensuring the number of threads equal to the number of processes

To avoid the aforementioned issue, we must ensure the number of threads corresponds to the number of processes in the simulated environment.

Another challenge we encountered was that when simulated in the SMT environment, we would need to enable the multi-thread option and provide different process identifiers (PID) for other processes since each process should have a unique PID.



```
)
# Set up processes for SMT
process_1 = Process(pid=102, cmd = binpath)
process_2 = Process(pid=101, cmd = binpath)

# Assign processes to CPU workload
system.cpu.workload = [process_1, process_2]
system.cpu.createThreads()
```

Figure 21: Configure PID for each process

References

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

Ajay Krishna Ananda Kumar , Sami Al-Salamin, Hussam Amrouch (2022). *Machine Learning Based Microarchitecture*. Research Gate.

Mark Oskin (1999). *Exploiting ILP In Page-Based Intelligent Memory*. Research Gate.

Bilal Ali Ahmad (2020). *Spectre and Meldown attack detection using machine learning and hardware performance counters.* Research Gate.

Akank Shasadvelkar (2021). *Superscalar Architectural*. GeeksForGeeks.

Ankith (2023). *Why out of order pipelines?* Medium.

Phillip Lane (2023). *Computer Architecture - What is Instruction Level Parallelism?* Reddit.

Reddit (2022) *Why aren't linked list more popular?*

Ankita Saini (2023). *Multithreading C#.* GeeksForGeeks.

Khanh Nguyen. https://github.com/khanhntd/MSCS531-ILP