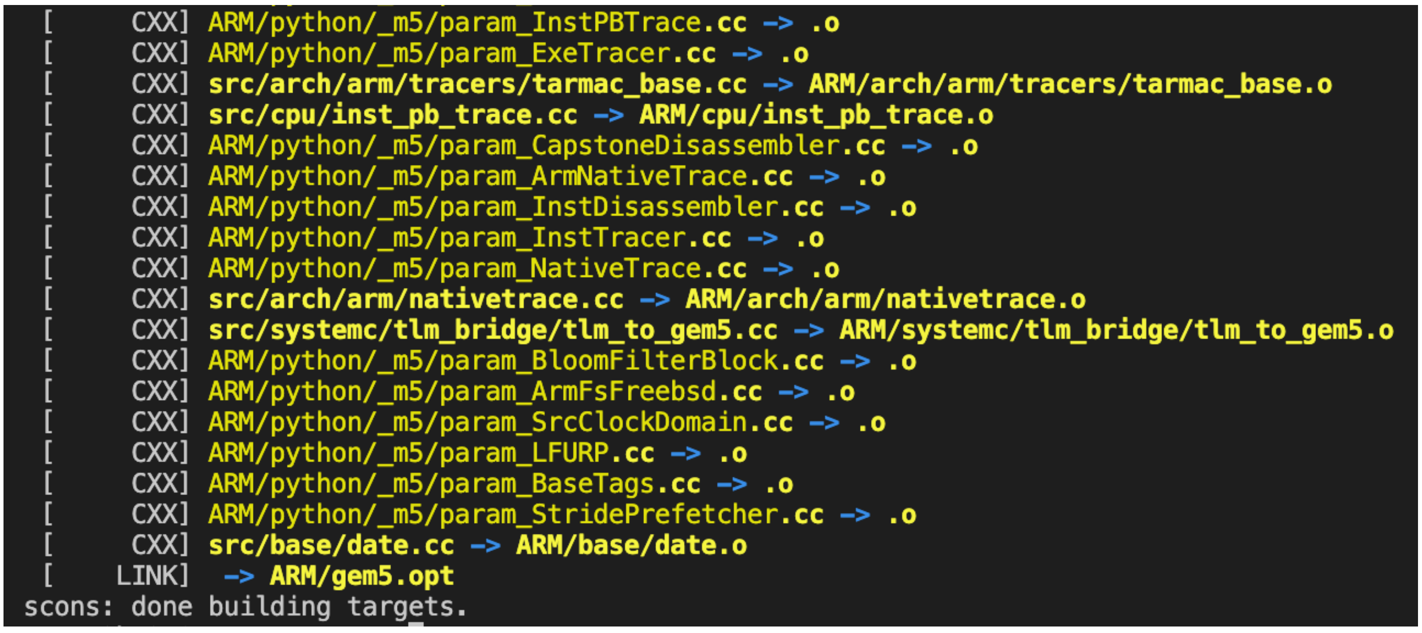
Project 1 Report

2100012289, Peiyu Liu, School of EECS

Overview: This project focuses on setting up the required environment for conducting coding experiments. Specifically, we utilize **gem5** to simulate fully functional computer systems with varying performance levels. The first step in this project involves creating a virtual machine using **Docker**, where gem5 will be installed. During the installation process, we clone the gem5 code repository from GitHub and compile it. Afterward, we configure personalized computer systems to run on gem5 and execute specific benchmark programs to measure the performance of different systems.

Setup: Install Toolchain and Compile gem5

In this part, we complete the configuration of Docker and the installation of gem5. Following the steps provided in the documentation, we pull the Docker image, clone the gem5 repository, and create a Docker container where gem5 is compiled. On a 2021 MacBook Pro with an M1 Pro chip, the compilation takes approximately 3 hours. Due to the lengthy runtime of the benchmark program during the subsequent experiments (part 3) in local environment, the final experiments are conducted on the server provided by the course. The partial output of the successful compilation is shown in the screenshot below.

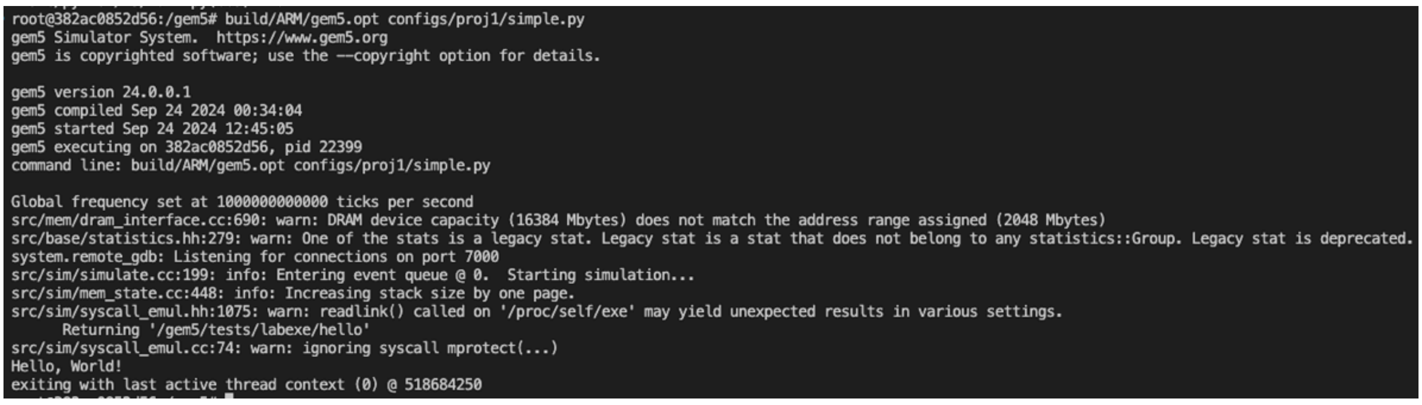


```
 [     CXX] ARM/python/_m5/param_InstPBTrace.cc -> .o
 [     CXX] ARM/python/_m5/param_ExeTracer.cc -> .o
 [     CXX] src/arch/arm/tracers/tarmac_base.cc -> ARM/arch/arm/tracers/tarmac_base.o
 [     CXX] src/cpu/inst_pb_trace.cc -> ARM/cpu/inst_pb_trace.o
 [     CXX] ARM/python/_m5/param_CapstoneDisassembler.cc -> .o
 [     CXX] ARM/python/_m5/param_ArmNativeTrace.cc -> .o
 [     CXX] ARM/python/_m5/param_InstDisassembler.cc -> .o
 [     CXX] ARM/python/_m5/param_InstTracer.cc -> .o
 [     CXX] ARM/python/_m5/param_NativeTrace.cc -> .o
 [     CXX] src/arch/arm/nativetrace.cc -> ARM/arch/arm/nativetrace.o
 [     CXX] src/systemc/tlm_bridge/tlm_to_gem5.cc -> ARM/systemc/tlm_bridge/tlm_to_gem5.o
 [     CXX] ARM/python/_m5/param_BloomFilterBlock.cc -> .o
 [     CXX] ARM/python/_m5/param_ArmFsFreebsd.cc -> .o
 [     CXX] ARM/python/_m5/param_SrcClockDomain.cc -> .o
 [     CXX] ARM/python/_m5/param_LFURP.cc -> .o
 [     CXX] ARM/python/_m5/param_BaseTags.cc -> .o
 [     CXX] ARM/python/_m5/param_StridePrefetcher.cc -> .o
 [     CXX] src/base/date.cc -> ARM/base/date.o
 [    LINK]  -> ARM/gem5.opt
scons: done building targets.
```

Part 1: Create and Run a Simple Configuration Script

In this section, we create and simulate the execution of a customized computer system on gem5. This step is achieved by creating a configuration file `simple.py` (configs/proj1/simple.py). In `simple.py`, we specify various parameters of the computer system, including those for the CPU, physical memory (DRAM), and parameters related to process initialization. In this section, we designate the test program `hello` (tests/labexe/hello) in `simple.py` as the program to be loaded and executed. After completing these steps, the simulation is started with the following command, and the screenshot of the output is shown in the image below.

```
build/ARM/gem5.opt configs/proj1/simple.py
```



```
root@382ac0852d56:/gem5# build/ARM/gem5.opt configs/proj1/simple.py
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 24 2024 00:34:04
gem5 started Sep 24 2024 12:45:05
gem5 executing on 382ac0852d56, pid 22399
command line: build/ARM/gem5.opt configs/proj1/simple.py

Global frequency set at 1000000000000 ticks per second
src/mem/dram_interface.cc:690: warn: DRAM device capacity (16384 Mbytes) does not match the address range assigned (2048 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
src/sim/simulate.cc:199: info: Entering event queue @ 0.  Starting simulation...
src/sim/mem_state.cc:448: info: Increasing stack size by one page.
src/sim/syscall_emul.hh:1075: warn: readlink() called on '/proc/self/exe' may yield unexpected results in various settings.
      Returning '/gem5/tests/labexe/hello'
src/sim/syscall_emul.cc:74: warn: ignoring syscall mprotect(...)
Hello, World!
exiting with last active thread context (0) @ 518684250
```

We can see from the result that the program has been executed successfully. For this project, warnings can be ignored without error.

Part 3: Add Options in the Configuration Script

In this section, we modify `simple.py` to accept parameters passed from the command line. This functionality is primarily implemented using Python's `argparse` library, allowing us to specify system parameters more conveniently without needing to modify the configuration file each time. After making these modifications, we run more complex benchmark programs (e.g., `binary_search`, `shell_sort`, etc.) in addition to `hello`. The system’s performance is measured by calculating **IPC（instructions per cycle）**, with the necessary data found in the output file `stat.txt` during gem5 execution. For convenience, we automate the execution of all benchmark programs via a shell script according to the documentation instructions （noted that in the document, the shell script does **not** designate clock frequency as 2GHz） and save various statistical data. The results are presented and explained below.

Parameter Setting

According to the document, we set a fixed clock frequency as 2GHz for all executables and set various types of CPUs and physical memories. All parameters are presented in **Table 1**.

CPU	TimingSimpleCPU	MinorCPU	O3CPU
Mem (DRAM)	DDR3_1600_8x8	DDR4_2400_8x8	DDR5_8400_4x8
Clock Freq.	2GHz		

Table 1. Parameter set for computer systems

In subsequent experiments, for any specific program, we will run the program with all parameter combinations and calculate the IPC to obtain a relatively accurate metric that can measure CPU performance.

Instruction Counts

In order to ensure that each benchmark program can complete within a certain time, we set the instruction limit for each thread to 500,000,000 by specifying `system.cpu.max_insts_any_thread` in `simple.py`. After checking the experimental results, we found that all threads reached this limit, and the number of executed instructions was within the range of 500,000,000 - 500,000,003. Therefore, we can assume that the number of executed instructions for all parameter sets and all executed programs is 500,000,000.

Clock Cycles

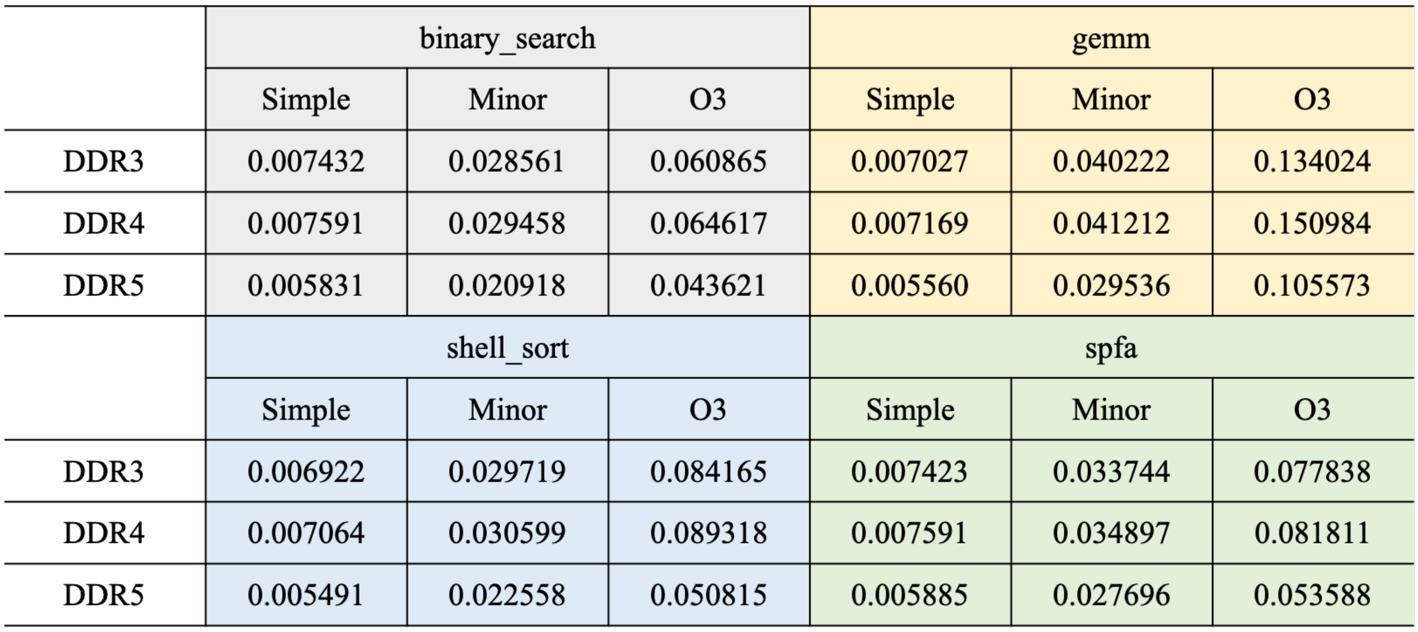
The counts of clock cycles are shown in **Table 2**. Generally speaking, the more advanced the CPU, the fewer clock cycles it will consume. In most cases, the number of clock cycles used by the O3CPU is **an order of magnitude** lower than that of the other two CPUs, while the number of clock cycles used by the SimpleCPU is about **4-5 times** that of the MinorCPU. The former is because the O3CPU can more efficiently utilize processor resources, improving the parallel execution capability of instructions. The latter may be due to the fact that the SimpleCPU spends a significant amount of time waiting when accessing memory, whereas the MinorCPU implements pipeline, allowing for higher throughput in executing instructions. It can also be noted that physical memory has a certain impact on the speed at which the CPU executes instructions. Compared to DDR3, DDR4 can provide a slight improvement in CPU performance. However, DDR5 may actually lead to a decrease in CPU performance under certain conditions.

	binary_search			gemm		
	Simple	Minor	O3	Simple	Minor	O3
DDR3	67276041388	17506498448	8214889619	71150143497	12431057073	3730661950
DDR4	65865145214	16973053202	7737867712	69741514934	12132479784	3311602372
DDR5	85742533718	23902880026	11462441964	89933284403	16928670588	4736044287
	shell_sort			**spfa**		
	Simple	Minor	O3	Simple	Minor	O3
DDR3	72234717212	16824207354	5940745724	67358062477	14817613609	6457054580
DDR4	70783137222	16340187972	5598006738	65869244866	14328061065	6143492746
DDR5	91064765808	22164821843	9839557378	84957273572	18052827310	9379075510

Table 2. Number of clock cycles for each thread

Calculation of IPC

The IPCs are shown in **Table 3**. We can calculate IPC with data collected by gem5 during simulation.



	binary_search			gemm		
	Simple	Minor	O3	Simple	Minor	O3
DDR3	0.007432	0.028561	0.060865	0.007027	0.040222	0.134024
DDR4	0.007591	0.029458	0.064617	0.007169	0.041212	0.150984
DDR5	0.005831	0.020918	0.043621	0.005560	0.029536	0.105573
	shell_sort			spfa		
	Simple	Minor	O3	Simple	Minor	O3
DDR3	0.006922	0.029719	0.084165	0.007423	0.033744	0.077838
DDR4	0.007064	0.030599	0.089318	0.007591	0.034897	0.081811
DDR5	0.005491	0.022558	0.050815	0.005885	0.027696	0.053588

Table 3. IPC for each thread

Figure 1 visualizes data in the **Table 3**. As we mentioned above, the combination of DDR4 and O3CPU leads to the best performance in most cases, while using DDR5 or other CPUs will cause noticeable decrease in performance.

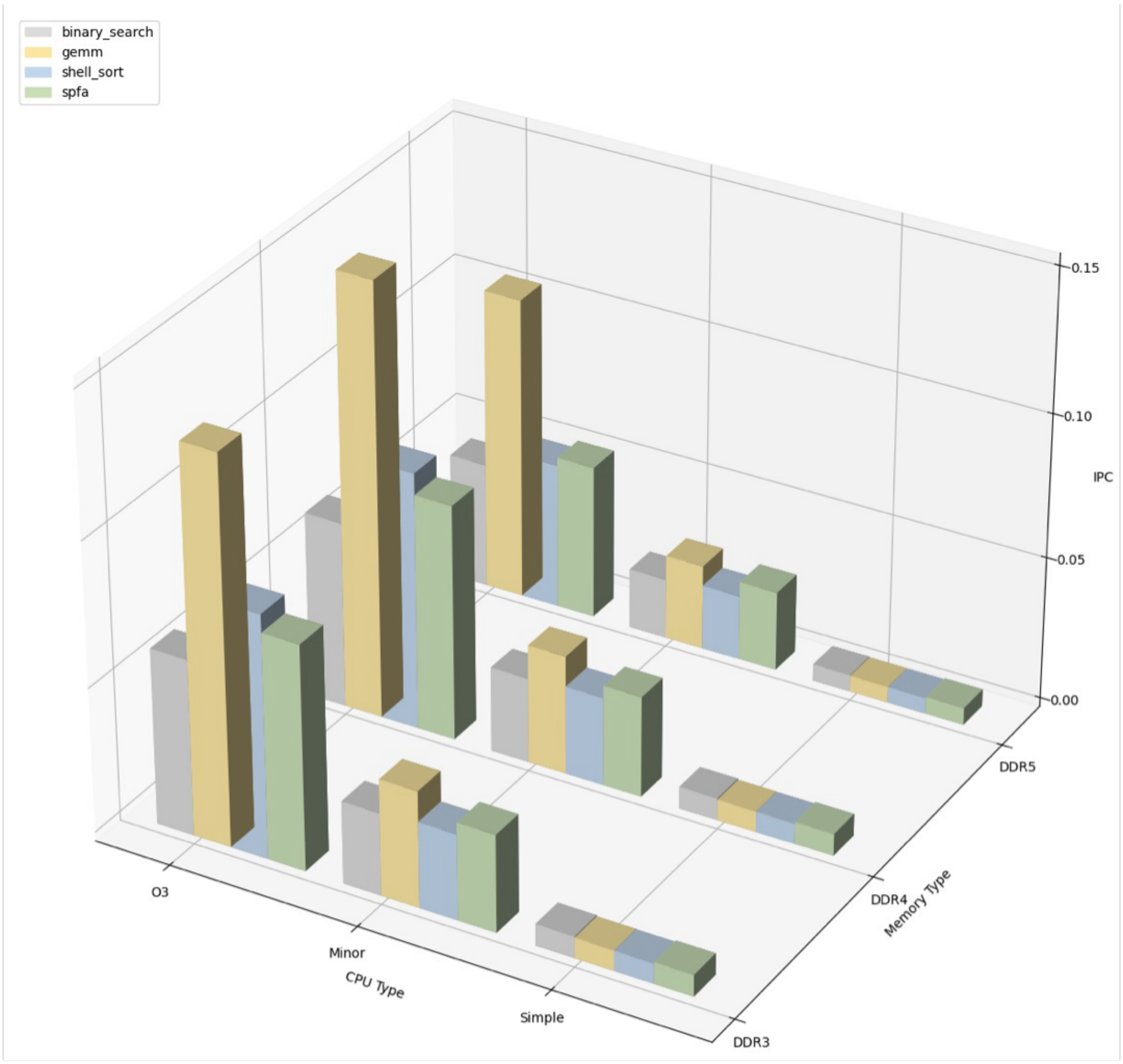


Figure 1. IPC for each thread

Conclusion

The implementation method of the CPU itself, as well as the "degree of harmony" in its interaction with memory, can significantly impact CPU performance. In certain cases, high-performance memory may not necessarily improve CPU performance. On the contrary, it may even cause a noticeable decline in CPU performance. Additionally, the complexity of the program itself can also affect CPU performance. In this project, compared to other algorithms, the gemm algorithm exhibits the highest IPC in most scenarios, which may be related to the program's memory access patterns, branch logic, and potential parallelism. In summary, the factors influencing CPU performance are quite complex. While improving the CPU's processing capabilities, attention should also be given to the interaction between the CPU and other parts of the computer system (such as physical memory).