Abstract

Our project compares the performance and efficiency of running multiple multithreaded programs across three different architectures: single-core without harts, single-core with harts, and cache-coherent multi-core processors.

Because our project requires simulated workloads for varying core and threading setups, we want to use gem5. Specifically, we plan to take advantage of the hart customization capabilities (to evaluate software vs hardware multithreading) and ability to define number of processors (to evaluate single core vs multi core processors) in gem5, and assign workloads accordingly. Oscar also allows us to choose the number of cores when running workloads, which we may incorporate into our project.

Through our evaluation, we hope to assess the effectiveness of the different parallel processing approaches. When considering multi-core processors versus multi-thread single core processors, we hope to understand how the architectural nuance impacts performance metrics such as energy consumption, throughput, execution time, idle time, etc. With this analysis, we hope to learn how to make design decisions regarding core and threading customizations, specifically as it pertains to multithreaded applications (such as MapReduce).

Hypotheses

1. Multicore processors should exhibit the highest throughput, followed by single-core processors with harts, while single-core processors without harts should have the lowest throughput.
    - Harts enhances parallelism at the hardware level, optimizing resource allocation and scheduling for more efficient core resource utilization.
    - Multiple cores allow for the concurrent execution of tasks, with each core operating independently, thereby maximizing parallel processing capabilities.

2. Multicore processors should exhibit the highest energy consumption, followed by single-core processors without harts, with single-core processors with harts using the least amount of energy.
    - This higher energy consumption in multi-core processors is primarily due to the simultaneous operation of multiple physical cores and additional hardware resources, such as the shared memory hierarchy, which require more power to sustain.
    - While supporting harts in a processor may not significantly differ in power consumption compared to single-core processors without harts, single-core processors with harts could achieve energy savings by reducing execution time through parallel thread execution.

3. Single-core processors without harts should have the lowest overall cache overhead followed by the single-core processors with harts, with cache-coherent multi-core processors having the highest cache overhead.
    - Supporting harts requires parallel execution of different harts which might lead to worse data locality, resulting in a higher cache miss rate.
    - Multi-core processors would have a higher read miss penalty and write miss penalty due to overhead of managing its cache-coherence hierarchy across different cores

Methodology/Set-Up

To assess all three hypotheses, we aimed to simulate an environment with variable CPU configurations, variable number of hardware threads, and consistent cache hierarchies. Recognizing that our tested workloads would be multithreaded (e.g. MapReduce), we aimed to use O3CPUs in our environment so that the advantage of SMT and multi-core architectures would be more pronounced. While in-order processors might still benefit from multithreading capabilities, out-of-order processors especially utilize resources and parallelize efficiently given multithreaded programs. We planned to set up simulation environments using either 1 or 4 O3CPUs to test single versus multi-core processors, and with either 1 or 2 hardware threads via the numThreads CPU parameter in gem5 or the gem5 SMT option either disabled or enabled. For caches, we were more concerned with consistent cache setups across all processor and threading configurations than with specific cache designs. Thus, we chose to use the gem5 default cache structure with 64kb d-cache and 32kb i-cache private to each core and a 512mb system memory. All of these environment configurations seemed possible with the system emulation (SE) example configuration script in gem5, so we planned to run all of our experiments in SE mode and use generated gem5 stats for our evaluation. Finally, we would compile our MapReduce go program to either ARM or x86 compatible binaries to emulate in gem5.

However, due to many limitations and issues within gem5, our ideal environment described above was unrealistic. In fact, almost every facet of the pre-planned environment from compiling binaries, setting up SMT, to simulating multicores, had varying problems and needed to be changed. Most importantly, our entire project hinged on testing multithreaded programs which are not supported in SE mode. For simplicity, gem5 SE mode does not include a thread scheduler and therefore cannot properly handle multithreading. Instead, the workaround gem5 suggests and follows in its examples is to manually assign one thread per core so any multithreaded program can only run on multiple cores and eliminates the possibility of evaluating single-processor architectures. Additionally, while backend support for SMT in SE mode seems to exist, gem5 can only simulate SMT if the user assigns a separate workload per hardware thread in the environment config script. Even then, our experiments with SMT enabled, multiple numThreads, and matching number of workload binaries never succeeded.

Given these setbacks, we determined that our project required full system (FS) mode gem5 emulation. With a thread scheduler, we could at least evaluate multithreaded workloads on single and multi-core schedulers in FS gem5. However, gem5 itself notes that SMT is not supported in FS mode. Thus, we also used a PC with 4-core Intel i5

CPUs where each core had 2 hardware threads and supported SMT. The PC ran on Linux 5.15.0-47-generic kernel and Ubuntu 22.04.1 LTS OS. This PC with SMT support made it possible for us to run workloads on single and multiple cores with and without hyperthreading enabled. Given the hours-long wait for Linux to boot up each time in FS mode, we utilized a gem5 checkpoint script that allowed us to restore a previously booted Linux state and directly start running our workloads.

Facing challenges and acknowledging the time-consuming nature of running FS mode, we also tested our workloads on the high-performance systems offered by Oscar. In contrast to simulated environments like Gem5, Oscar's infrastructure offered real-world architectural insights and supplementary data such as power consumption. Nevertheless, it's important to note that, as verified with CCV staff, SMT is disabled on Oscar computers. Consequently, our comparisons were limited to single-core versus multiple-core performance on the Oscar platform.

Below, we describe the exact binaries and environment setups including additional run scripts and scaffolding for gem5, Oscar, and Linux PC required to replicate our project.

Binaries

Given the difficulty we had successfully cross-compiling multithreaded programs for gem5, we decided to use provided gem5 resources compatible with FS mode as well as two of our own multithreaded binaries:

Figure: All the test binaries with name, source, arguments usage, multithreading usage, and description

Binary Name	Binary Source	Arguments	Multithreaded?	Description
x86-gapbs-bfs	gem5-resources	-g 13 -n 5	Yes	From the GAP benchmark suite for graph processing research, the bfs benchmark runs breadth-first-search on a graph of 2^13 nodes for 5 iterations.
x86-gapbs-tc	gem5-resources	-g 13 -n 5	Yes	Like gapbs-bfs, but instead runs triangle counting on a graph of 2^13 nodes for 5 iterations.
x86-npb-lu-size-s	gem5-resources	None	Yes	From NASA's Advanced Supercomputing Parallel Benchmarks, npb-lu runs a Lower-Upper Gauss-Seidel solver, a large-scale computing application for linear equations.

X86-matrix-multiply-omp	gem5-resources	4 4	Yes, can specify number of threads to run with	Two matrix multiplications on a 300x300 then 100x100 matrices are computed serially for 4 iterations, using 4 threads.
wordcount	Self-Written Go Code Compiled to x86	resources/big.txt 32 little 10	Yes, via Go routines	A MapReduce based Go program that spawns a specified number of mapper threads to search the number of occurrences of a query word in a file
mapCompute	Self-Written C Code Compiled to x86	None	Yes, via pthreads	A MapReduce style C program that spawns 100 threads and gives each a workload of light arithmetic computation.

Gem5 FS

To set up the gem5 FS environment (from gem5 root directory):

1. Download prebuilt gem5 kernel (`vmlinux-5.4.49`) and disk image (`x86-ubuntu.img`) and save to appropriate directories (`dist/binaries` and `dist/disks`)
2. Mount all benchmarks in the disk image into a benchmark directory using gear-up instructions. You might need to change `util/gem5img.py:197` from `r"type=(?P<type>\d+)"` to `r"type=(?P<type>\d+),bootable"`.
3. Create `bootscripts` directory, copy `configs/boot/hack_back_ckpt.rcS` into it, and create any bootscripts for running benchmarks
   Example:
   ```
   #!/bin/sh
   cd /benchmarks
   chmod +x <benchmark-binary>
   ./<benchmark-binary>
   ```
4. Set up a FS bash run script following gear-up instructions with appropriate kernel and disk image paths, O3CPU for cpu-type, and gem5.fast compilation for efficiency
   Example:
   ```
   $CURR_DIR/build/x86/gem5.fast -d $RESULT_PATH
   $CURR_DIR/configs/deprecated/example/fs.py --kernel
   $KERNEL_PATH --cpu-type $CPU_TYPE --disk $DISK_PATH
   --caches --script=$SCRIPT_PATH
   ```

5. Run the checkpoint bootscript first as the `$SCRIPT_PATH`, then add `-r 1` to the bash command for running all benchmark bootscripts afterwards. We used the `configs/boot/hack_back_ckpt.rcS` bootscript specifically provided for this purpose.
6. To switch number of cores, add `num-cpus=<number>` and rerun bootscripts (checkpoint first then benchmarks)

Using this setup, we ran all benchmarks on uniprocessor and four-processor architectures with O3CPUs specified. We saved all `stats.txt` files produced per run, as well as terminal outputs inherent to each binary.

Oscar

To run workload and collect data on Oscar:

1. Login through `ssh.ccv.brown.edu`
2. Run the `sinfo` command and select a set of desired nodes
    a. We used `node2417-2420`, Intel x86 64-core CPUs with Linux, as they appear to be performant, readily available, and well-suited for our testing purposes
3. Download all program binary to Oscar file system
4. Create a `sbatch` script for each benchmark, e.g.

```
#!/bin/bash
#SBATCH -N 1
#SBATCH -n 1
#SBATCH -w <node id>
#SBATCH --mem=64g
#SBATCH -t 10:00
#SBATCH --constraint=intel
time <path/to/program> <program args>
```

5. Run `sbatch <script>` and note the job id from message
6. Wait until an output file `slurm-<job id>.out` is created
7. Obtain the benchmark output and runtime data
8. Run `sacct -j <job id> -l` command to get more useful data on the job

Linux PC

To replicate our results, obtain hardware with the following specs (or as closely identical as possible):

- Architecture: x86_64

- CPU(s): 4
- Vendor ID: GenuineIntel
- Model name: Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz
- Thread(s) per core: 2
- Core(s) per socket: 2
- Socket(s): 1
- Operating System: Ubuntu 22.04.1 LTS
- Kernel: Linux 5.15.0-47-generic
- Hardware Vendor: HP
- Hardware Model: HP ProBook 440 G3

To run workload and collect data on PC:

1. Get all our benchmarks from our GitHub repository (https://github.com/nishchayp/1952y_gem5/tree/main) under benchmarks
2. Turn on SMT on/off the PC using the command echo on/off | sudo tee /sys/devices/system/cpu/smt/control
3. We then use the time command to obtain the timing of all the runs
4. And we use time taskset --cpu-list command to specify the CPUs on which to run the workload
5. We provide scripts in our GitHub repository under scripts. The scripts handle turning on/off SMT for you and running a given benchmark multiple times so that you can take median of all the run times

Evaluation/Results

To test each hypothesis, we ran all benchmarks across gem5, Oscar, and Linux hardware. All three tools allowed us to specify the number of cores, and we tested the performance of all benchmarks on a 1-core O3CPU versus 4-core O3CPUs. Linux hardware allowed us to also specify whether SMT should be enabled, and we tested benchmark performance on 1-core O3CPU with and without SMT enabled as well as 4-core O3CPUs with and without SMT enabled. With gem5 simulation, we were able to control variables like cache configurations, cpu type, and component latencies to name a few. The only adjusted configuration variables were the benchmarks themselves and number of CPU cores. Oscar and Linux tests similarly only modify a number of CPUs and SMT options, but are potentially more susceptible to side jobs and other running processes affecting runtimes. All results were generated via the gem5 simulation statistics (`stats.txt`), `slurm` and `sacct` output on Oscar, or the time command. We used a variety of benchmarks to test different computations and memory access algorithms and a variety of tools to corroborate throughput (in simulation and in real-time), assess specific cache performance (only possible with gem5 stats), and accurately measure power (best done with Oscar).

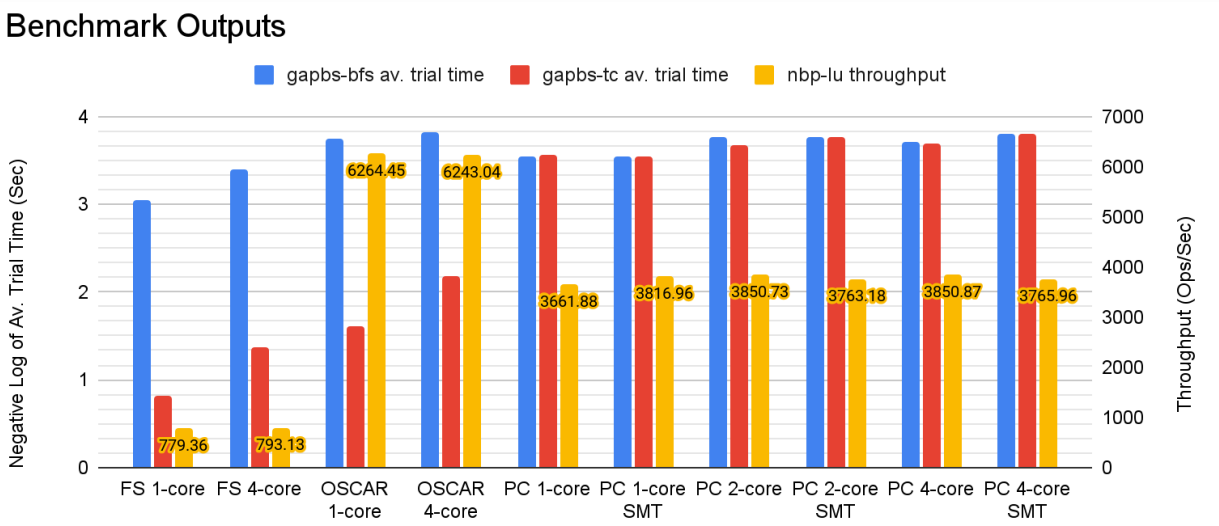
Figure: All tests ran for hypothesis testing with <tool><num-cpus> format. Only Linux PC supports SMT

	FS1	FS4	OSCAR1	OSCAR4	LINUX PC(1,2,4 cores) (SMT 2 threads/core)
gapbs-bfs	**X**	**X**	**X**	**X**	**X**
gapbs-tc	**X**	**X**	**X**	**X**	**X**
X86-npb-lu-size-s	**X**	**X**	**X**	**X**	**X**
X86-matrix-multiply-omp	**X**	**X**	**X**	**X**	**X**
map_compute	**X**	**X**	**X**	**X**	**X**
wordcount			**X**	**X**	**X**

Hypothesis 1: Throughput Evaluation

We evaluated gem5 simulation statistics in FS mode, Oscar, and PC outputs to measure throughput. Our results generally support our hypothesis, as groups with 4 cores consistently outperform single cores groups, and SMT groups often outperform groups without hardware threads.

Figure: GAP and NBP benchmarks results from all testing groups. (Note that the higher the negative logarithm of the trial time and the ops/sec throughput, the better the performance.)



There are pairs of experimental and control groups that do not conform to our hypothesis but are very closely aligned with each other. This similarity is likely attributable to the low number of test repetitions. Further tests, described below, confirmed this observation.

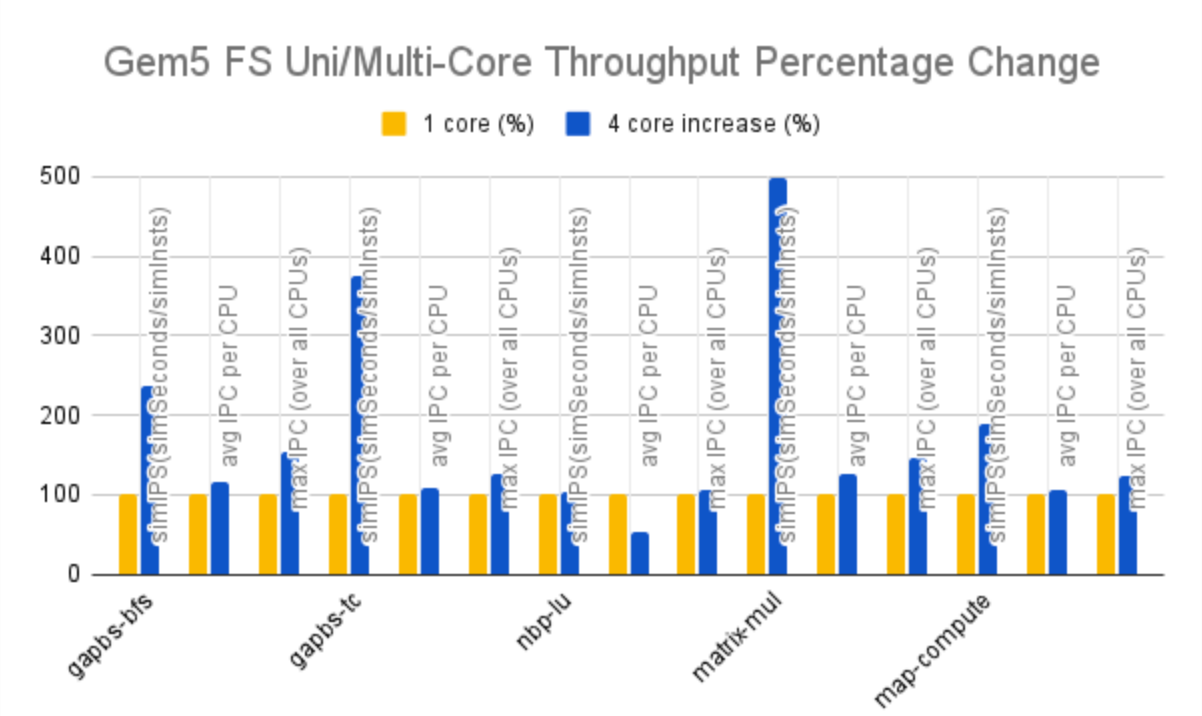
Gem5 FS: 1-core vs 4-core

Gem5 FS results supported our hypothesis that the optimized parallelism from multicore systems would outperform unicore ones.

For our gem5 results, we look specifically at the metrics of overall simulated instructions per second (IPS) and average/maximum instructions per cycle (IPC) across CPU/s. We chose these statistics because they were easily accessible via gem5 statistics, but also because IPS provides insights into the total system processing speed while maximum and average IPC provides insights into each CPU's instruction execution efficiency and the overall processor architecture efficiency. IPS was calculated by dividing gem5 simInsts by simSeconds. Average IPC was calculated by taking the uniprocessor's gem5 cpus.ipc for 1-core architecture, and averaging across four reported gem5 `cpus<number>.ipc` for 4-core architecture. Maximum IPC was

the same as average IPC for 1-core architecture, while maximum IPC was selected as the highest IPC among the four reported for 4-core.

Figure: IPS, average IPC, and maximum IPC measurements for binaries run on 1 vs. 4 cores. Percentage changes are calculated relative to 1-core performance (e.g. $\frac{4\ core\ bfs}{1\ core\ bfs} \times 100\%$*)*



Benchmark	Metric	1 core	4 core
gapbs-bfs	simIPS(simSeconds/simInsts)	907523668	2145410876
	avg IPC per CPU	0.458791	0.535943
	max IPC (over all CPUs)	0.458791	0.705442
gapbs-tc	simIPS(simSeconds/simInsts)	520615653	1946280912
	avg IPC per CPU	0.26062	0.284416
	max IPC (over all CPUs)	0.26062	0.330374
nbp-lu	simIPS(simSeconds/simInsts)	1070014721	1114401415
	avg IPC per CPU	0.537563	0.285929
	max IPC (over all CPUs)	0.537563	0.571845
matrix-mul	simIPS(simSeconds/simInsts)	27051701	135073867
	avg IPC per CPU	0.013538	0.017140

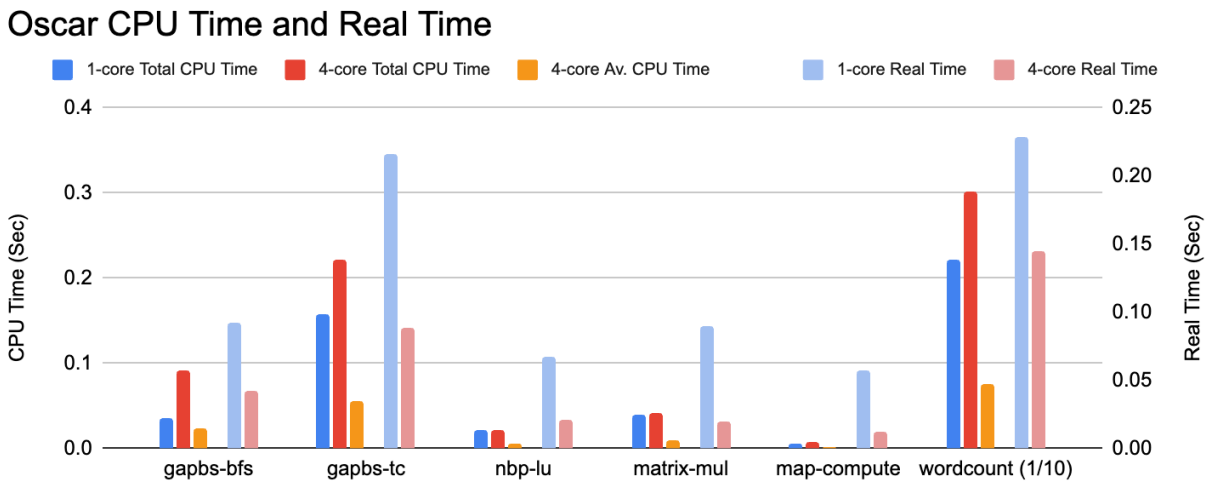
	max IPC (over all CPUs)	0.013538	0.019652
map-compute	**simIPS(simSeconds/simInsts)**	332389217	628238874
	avg IPC per CPU	0.16644	0.174732
	max IPC (over all CPUs)	0.16644	0.203537

Besides the decrease in average IPC on the npb-lu benchmark, 4-core architectures outperformed 1-core architectures across all benchmarks and all throughput metrics. This increase is especially obvious when looking at the increase in simIPS, where multi-core architectures at least approximately doubled and at most quintupled the number of instructions executed per second. This supports our initial hypothesis that the concurrency inherent to multicore architectures would result in higher throughput compared to single-core architectures since multiple cores independently and simultaneously executing instructions in a timestep inevitably outperform one core executing the same number of total instructions. We feel especially confident with this conclusion due to the sophistication of gem5's simulated statistics, and consistency in the results.

Oscar: 1-core vs 4-core

Next, we confirmed our hypothesis using real-world high-performance machines on Oscar. We measured both the wall-clock real time from time command and the total CPU time from the Slurm job record. Tests on the same binaries were run on the same idle CCV node with different numbers of cores allocated.

Figure: CPU time and real time running all binaries with 1 core or 4 cores allocated on Oscar



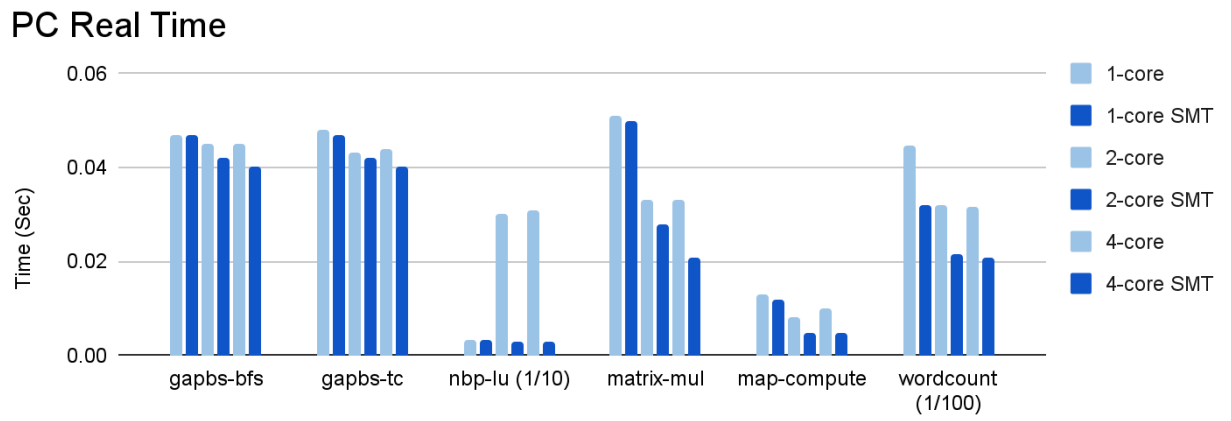
The real-time of running all binaries on a single core varies from 1.5 to 4 times longer compared to using 4 cores, depending on the degree of parallelization within the

program. The total CPU time on 4-core systems is usually longer, but when averaged across all 4 cores, is only 25%-60% of the 1-core total CPU time. Overall, utilizing 4 cores significantly increased the CPU efficiency compared to using a single core.

Linux PC: 1, 2, or 4 cores (each with/without SMT)

Lastly, we evaluated the effectiveness of SMT (2 harts per core). We ran all test programs with 6 different configurations: only the first core, only the first 2 cores, and all 4 cores, each with and without SMT enabled.

Figure: Real-time running all test binaries using 1 core, 1 core with SMT, 2 cores, 2 cores with SMT, 4 cores, and 4 cores with SMT on Linux PC



Enabling SMT always improved the real-time performance compared to the same number of cores without SMT. The effectiveness of hardware threads depends on the type of workload and OS' scheduling capabilities.

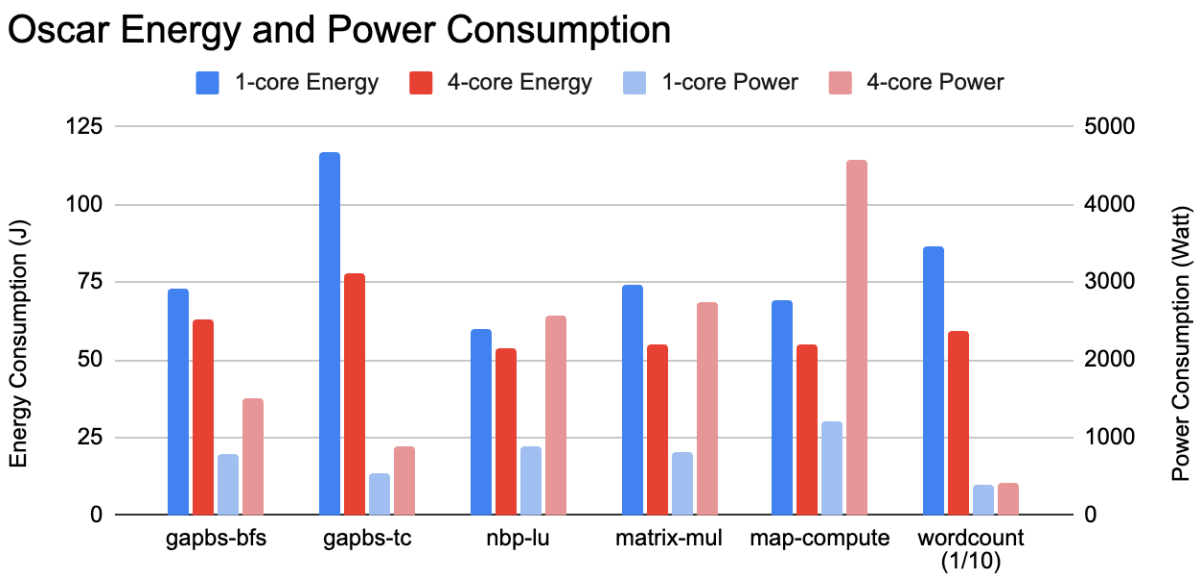
For example, in the map-compute program, each thread might be performing some simple tasks but have data dependencies and, thus, cannot use all functional units at the same time; SMT lets two threads run concurrently on a single core, sharing the functional units, nearly doubling the throughput. On the other hand, doubling the number of cores doesn't guarantee the same performance gain in this example, as using additional cores increases resource (especially memory) contention and scheduling overhead.

Note that the test results are subject to noises as the PC is natively running many other concurrent processes.

Hypothesis 2: Energy Consumption Evaluation

Although gem5 simulation outputs an exhaustive list of statistics, it's difficult to infer energy consumption directly from gem5 outputs. Instead, we explicitly specify Intel nodes to run our programs on Oscar, which allows us to collect real energy consumption data. This allows us to also analyze the power consumption rate by dividing consumed energy over the program runtime.

Figure: energy consumption and computed power consumption rate running all binaries with 1 core or 4 cores allocated on Oscar



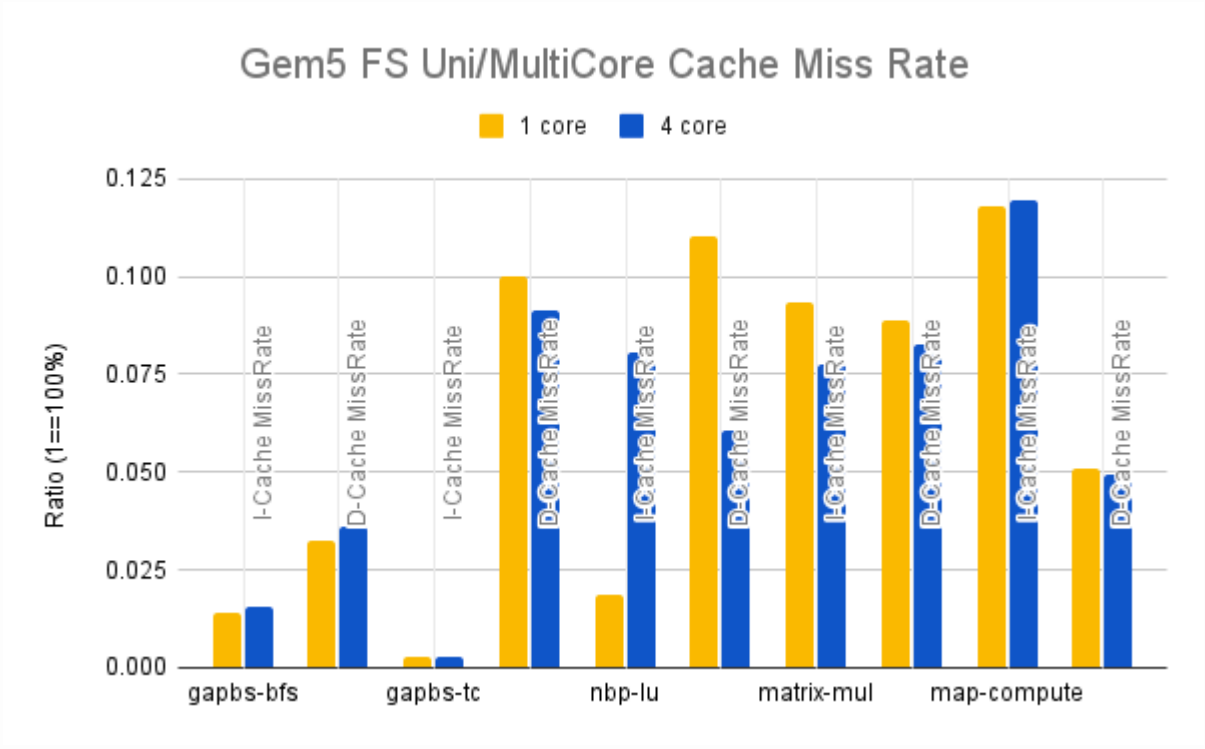
Surprisingly, the utilization of 4 cores consistently reduces total energy consumption by 10%-33% compared to using a single core. This result contradicts our hypothesis, which suggests that multicore systems consume more energy. While it is true that employing multiple cores increases power consumption, the significantly decreased runtime effectively lowers overall energy consumption.

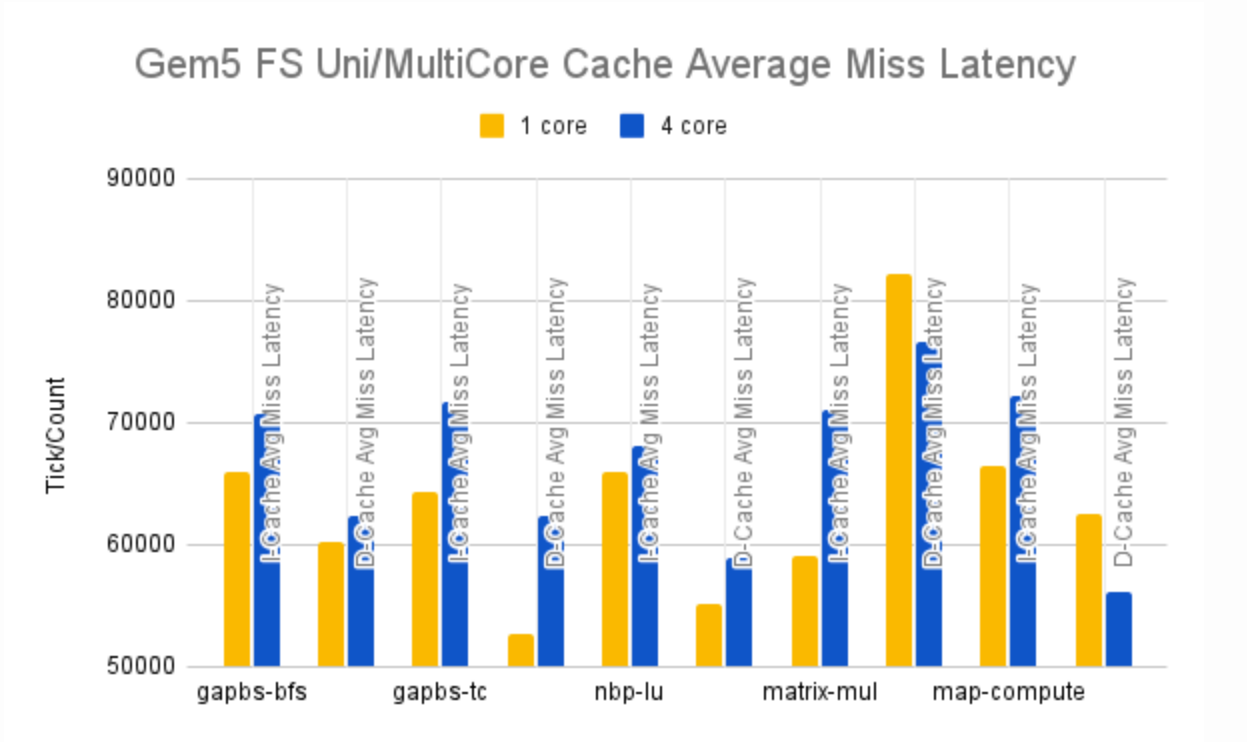
As Oscar doesn't support SMT, we weren't able to test the energy consumption impact on using hardware threads, unfortunately. However, given the results above, we would hypothesize that enabling SMT could also decrease total energy consumption, considering its capability to significantly shorten runtime with minimal additional hardware usage.

Hypothesis 3: Cache Efficiency Evaluation

To measure cache overhead and overall efficiency, the most applicable results came from gem5. Unfortunately, this limited our analysis of cache performance to single-core versus multicore architectures. While cache overhead is not directly measured, metrics such as average miss latency and overall miss rate for both i-caches and d-caches measure and indicate miss penalties due to cache complexity and cache effectiveness across cores.

Figures: Gem5 FS mode statistics for I-Cache and D-Cache Miss Rate and Average Miss Latency for 1 vs 4-core on all benchmarks





The average miss latencies for both i-caches and d-caches are higher with only two exceptions: matrix-mul d-caches and map-compute d-caches, possibly due to the low data contention nature of the two programs. As we theorized, multicore architectures have a cache-coherence maintenance overhead compared to single-core architectures likely due to the complexity of the snooping protocol by the memory bus. Given any cache miss, a uniprocessor architecture's miss protocols either only have to handle fetching from a lower level in the memory hierarchy (which is not included in our tests) or the system memory and running gem5 LRU eviction policy. For a multiprocessor architecture, although the memory bus cache-coherence maintenance overhead doesn't directly contribute to our cache miss latency measurement, it negatively affects the overall system performance and cache efficiency. This additional complexity expectedly increases miss latency when simulating on multicore architectures. It supports our hypothesis that single-core processors without harts should have lower cache overhead than cache-coherent multi-core processors.

However, the cache miss rate is not conclusively higher for multicore systems. Looking specifically at d-cache miss rates, a 4-core setup had lower miss rates for all benchmarks except gapbs-bfs. While we cannot examine the actual code for certain gem5 benchmark binaries, we assume that multiple cores' ability to leverage greater spatial locality reduces cache misses. For any multithreaded program dealing with contiguously stored data such as matrix-multiply, dividing sequentially-accessed data into slices or subsets, assigning threads a subsection to compute over, and scheduling

threads on different cores localizes data access to a greater degree compared to asking one core to iterate through the entire data structure (note SMT is disabled on FS mode, which means only one thread per core is simulated at a time). For i-cache, the 4-core systems have a higher miss rate in all tests except for matrix multiplication. This might be due to the fact that threads running on different cores concurrently might be accessing the same set of instructions in their own i-cache; however, as one core evicts its cache, the other cores' i-cache might be invalidated, causing an i-cache miss when they access the same code.

Thus, our results indicate more CPU cores increase cache overhead and latency per miss due to the additional complexity of maintaining coherent caches (especially when more complex cache hierarchies are chosen), but might benefit from an overall lower rate of cache misses due to greater per-core data locality, depending on the nature of the program.

Other
A full repository of benchmarks, run scripts, and results can be found here: https://github.com/nishchayp/1952y_gem5/tree/main

Summary

As intended, we were able to evaluate the performance tradeoffs between uniprocessor and multiprocessor systems, with or without SMT capabilities. We had hoped to test SMT in further detail via gem5 simulation, but were unfortunately limited by gem5's faulty SMT implementation in SE mode and incompatibility with SMT in FS mode. Nevertheless, we used all possible resources to provide evaluation, even if statistics were across tools and from varying environments, to evaluate our target tradeoffs.

While some of the observations we made during our analysis matched our expectations exactly, others reinforced our suspicions that the "perfect mix" of hardware and software optimizations for threading, depending on the underlying architecture, is still unclear. The magnitudes of improvement in throughput justify moving beyond uniprocessor architecture which is why mainstream computers are essentially all multi-core, but this improvement will still be limited by the software parallelism (Amdahl's Law), cache-coherent overhead, and other hardware resources that couldn't scale easily like the number of cores. On the other hand, the decrease in system runtime was far less consistent and more marginal when comparing systems with or without SMT/hyperthreading. At the end of the day, there could only be a limited number of per-core functional units to be shared among harts, and thus, the effect of SMT would be much less noticeable than multicore in high-performance supercomputers. We think that the interaction between software and hardware threads will continue to be a topic in computer architecture. As the need for parallelism and scalability increases, how to best acknowledge the underlying hardware from an OS perspective and optimize the hardware to accommodate multithreaded software programs is a present and future direction in the field.

If hardware and resources were unlimited, we would compare high-performance cache-coherent multicore systems with NUMA architecture distributed systems. Distributed systems are renowned for their ability to execute large, parallelized, and multithreaded programs, such as MapReduce and similar benchmarks utilized in our project. Moreover, they offer advantages in linear scalability: as additional nodes are added to the system, more memory capacity and bandwidth are incorporated without the overhead of maintaining coherent cache. Considering the fundamentally different approaches and designs of these two architectures, it would be highly insightful to explore their performance boundaries and understand which types of workloads are better suited to each architecture.