Project A1: Advanced OS and CPU Feature Exploration

Jaithra Pagadala

0 CPU Control and System Specifications ECSE 4320 Advanced Computer Systems

This section describes the hardware, software, and environment controls used for all experiments. The goal is to make results reproducible and reduce noise from uncontrolled variation.

0.1 Hardware

All experiments run on a laptop with the following CPU, as reported by `lscpu`:

- CPU model: 13th Gen Intel Core i9-13905H
- OS: Ubuntu 22.04.6
- Cores: 14
- GCC: 11.04
- Perf : 5.11
- Pinned Frequency: 1.5 GHz

0.2 Software

Experiments use a 64-bit Linux distribution on this hardware. The main tools are:

- Compiler: `g++` with flags `-O3 -march=native -std=c++17`
- Performance counters: `perf stat`
- Plotting: Python 3 with `matplotlib`

Three C++ microbenchmarks are used:

- `cpu_burn.cpp`: CPU-bound floating-point loop, used for CPU affinity and SMT experiments.
- `mem_scan.cpp`: Large-array memory scan, used for Transparent Huge Pages (THP) experiments.
- `mem_pattern.cpp`: Memory access with controllable stride and patterns (sequential, stride, random).

A shell script `run_experiments.sh` compiles all benchmarks and runs the full sweep for Features 1–4, writing CSV files in `results/`. Python scripts (`plots.py` and `plot_feature3.py`) read these CSV files and generate the figures.

0.3 Environment Control

To limit variability during measurements, the following controls are used:

- **CPU frequency**: The script attempts to set the CPU frequency governor to `performance`:

  ```
  sudo cpupower frequency-set -g performance
  ```

  This reduces DVFS-induced changes in core frequency. If the command fails, a warning is printed; experiments still run, but this limitation is noted.
- **Thread pinning**: The `taskset` command pins each benchmark process to specific logical CPUs. For example:
  - `taskset -c 0 ./build/cpu_burn 5` runs on logical CPU 0.
  - `taskset -c 0 ./build/mem_pattern ...` pins the memory benchmark to CPU 0.

  This reduces unwanted OS migrations between cores.
- **Background tasks**: Before running `run_experiments.sh`, non-essential applications (browsers, editors, etc.) are closed to reduce interference from other processes.

- **Transparent Huge Pages**: For Feature 2, THP mode is set before each group of runs via:

  ```
  echo never | sudo tee /sys/kernel/mm/transparent_hugepage/enabled
  echo always | sudo tee /sys/kernel/mm/transparent_hugepage/enabled
  ```
- **Repeat runs**: Each configuration is executed multiple times. Feature 1 and Feature 3 use 5 runs per configuration. Feature 2 and Feature 4 use 3 runs per configuration. Plots show mean values with error bars of ± 1 standard deviation across runs.

0.4 Use of `perf stat` and Limitations

Each benchmark run is wrapped by `perf stat` to capture CPU and memory behavior. The event set includes:

```
task-clock,cycles,context-switches,cpu-migrations,
dTLB-loads,dTLB-load-misses,cache-references,cache-misses
```

On this system, some events (`cycles`, `cache-references`, `cache-misses`) report `<not supported>`, likely due to the hybrid-core design and kernel configuration. As a result:

- Primary quantitative metrics in this report are:
  - Total iterations completed in a fixed time window (throughput).
  - Elapsed wall-clock time (seconds).
- `task-clock`, `context-switches`, and `cpu-migrations` are used qualitatively to confirm that threads stay pinned and that there are no unexpected scheduler anomalies.
- TLB and cache events are referenced qualitatively where meaningful but are not used as primary axes in the plots.

1 Feature 1: CPU Affinity and Scheduling

1.1 Benchmark Design

Feature 1 investigates the effect of CPU affinity and scheduler behavior on a CPU-bound workload. The benchmark `cpu_burn.cpp` performs a tight floating-point loop for a fixed interval (approximately 5 seconds) and prints:

- `iters`: number of loop iterations completed.
- `RUNTIME_SECONDS`: observed wall-clock time.

Throughput is defined as "iterations completed in 5 seconds". Higher iteration counts indicate a larger effective CPU share.

The script `run_experiments.sh` runs the following configurations:

- **single_no_affinity**: one `cpu_burn` process without explicit pinning.
- **single_taskset_0**: one process pinned to CPU 0 using `taskset`.
- **two_no_affinity**: two concurrent processes without pinning.
- **two_same_core**: two processes intended to share one physical core (SMT siblings).
- **two_diff_core**: two processes pinned to different cores (for example CPU 0 and CPU 2).

Each configuration is run 5 times. Results are written to `results/feature1_affinity.csv`. The plotting script reads this file and computes mean iterations and standard deviations.

1.2 Iteration Throughput Across Configurations

Figure 1 shows average iterations for each configuration. The x-axis label is "Configuration" and the y-axis label is "Average iterations in 5 s (higher is better)". Bars show the mean over 5 runs; error bars show ± 1 standard deviation.

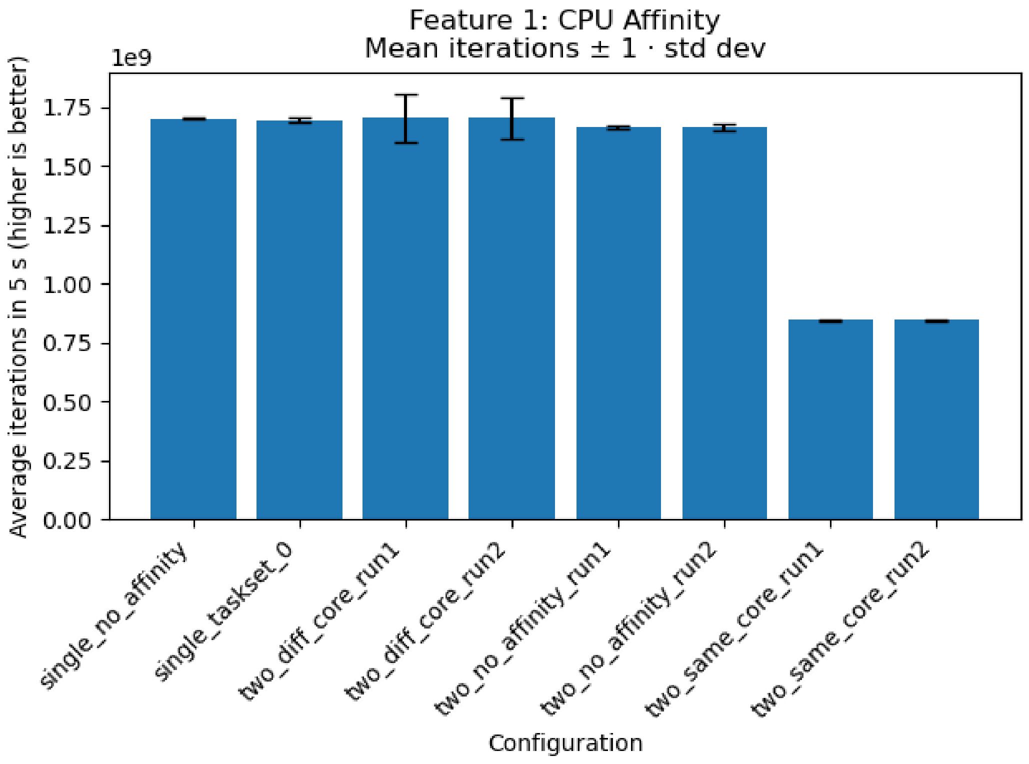

Figure 1: Feature 1: CPU affinity. Average iterations completed in a 5 second window for different scheduling and affinity configurations. Bars show mean over 5 runs; error bars show ±1 standard deviation.

Analysis

The single-thread configurations (`single_no_affinity` and `single_taskset_0`) define the baseline performance of the CPU-bound loop. Results for `single_taskset_0` typically show slightly lower variance, indicating that explicit pinning stabilizes the scheduling decisions.

When two processes run without pinning (`two_no_affinity`), iteration counts can vary more between runs. The scheduler may place the two tasks on different cores or on the same core at different times, so effective throughput per process fluctuates.

Pinned two-thread configurations show the benefit of careful placement. When both threads use the same physical core (`two_same_core`), each process completes fewer iterations than the single-thread baseline, reflecting resource sharing under SMT. When threads are placed on different cores (`two_diff_core`), per-thread throughput approaches the single-thread baseline and is higher than the same-core case.

1.3 Estimated Time to Reach a Common Work Target

Figure 2 uses iteration counts to estimate the time needed for each configuration to reach a common amount of work X. The target X is chosen as the minimum mean iteration count across all configurations. For each configuration:

$$T_X(\text{config}) \approx \text{mean\_time} \times \frac{X}{\text{mean\_iters}}.$$

The x-axis label is "Configuration" and the y-axis label is "Estimated time to reach X iterations (s)."

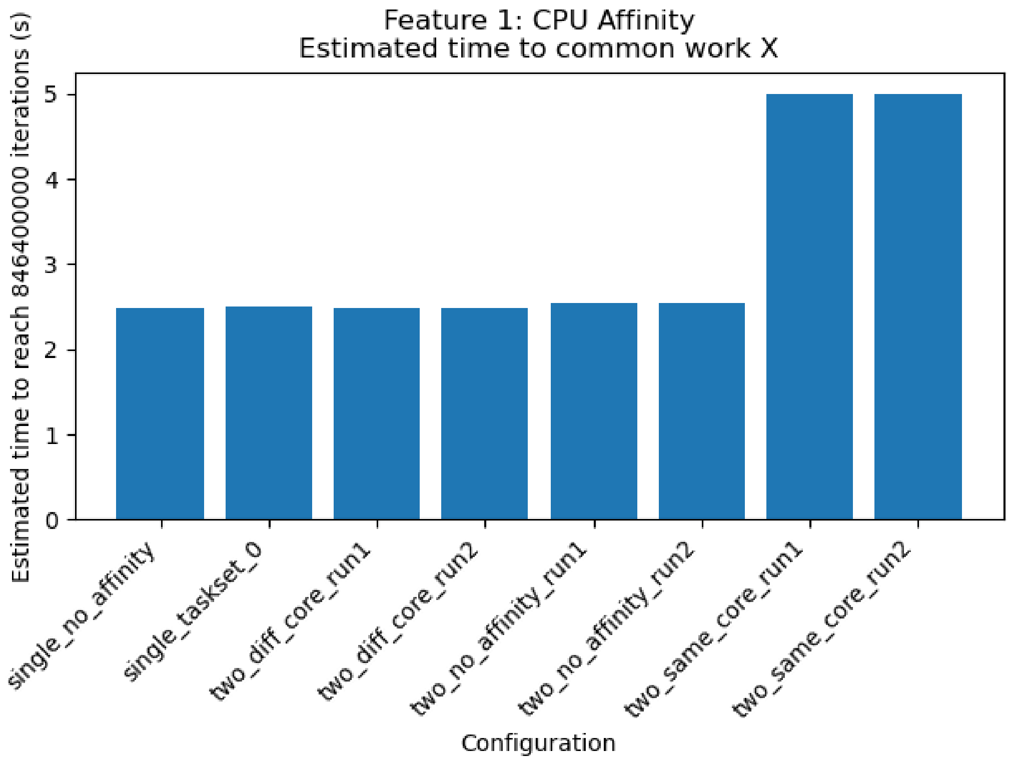

Figure 2: Feature 1: estimated time to reach a common iteration target X for each configuration. Lower bars indicate higher effective throughput.

Analysis

Pinned single-thread configurations reach the target work slightly faster than unpinned or same-core SMT configurations, reflecting more stable and higher per-thread throughput. The two-thread same-core configuration has the largest estimated time to reach X, showing that SMT sharing reduces each thread's share of the physical core. Two threads on different cores reach the target closer to the single-thread baseline, confirming that careful affinity placement can improve throughput and predictability.

2 Feature 2: Transparent Huge Pages (THP)

2.1 Benchmark Design

Feature 2 evaluates the impact of Transparent Huge Pages during large memory scans. The `mem_scan.cpp` benchmark allocates a large array (for example 2 GB) and traverses it multiple times using three patterns:

- **Sequential**: access addresses in increasing order.
- **Stride**: access every k-th element with a fixed stride.
- **Random**: follow a randomized index sequence.

Before each group of runs, THP is set to either `never` or `always`. The benchmark reports `RUNTIME_SECONDS`. Results are stored in `results/feature2_thp.csv` and plotted as mean runtime with standard deviation.

2.2 Runtime vs THP Mode and Pattern

Figure 3 shows the average runtime for each access pattern under the two THP modes. The x-axis groups bars by access pattern. Within each group, bars correspond to THP=`never` and THP=`always`. The y-axis label is "Average runtime (s)". Error bars indicate ± 1 standard deviation.

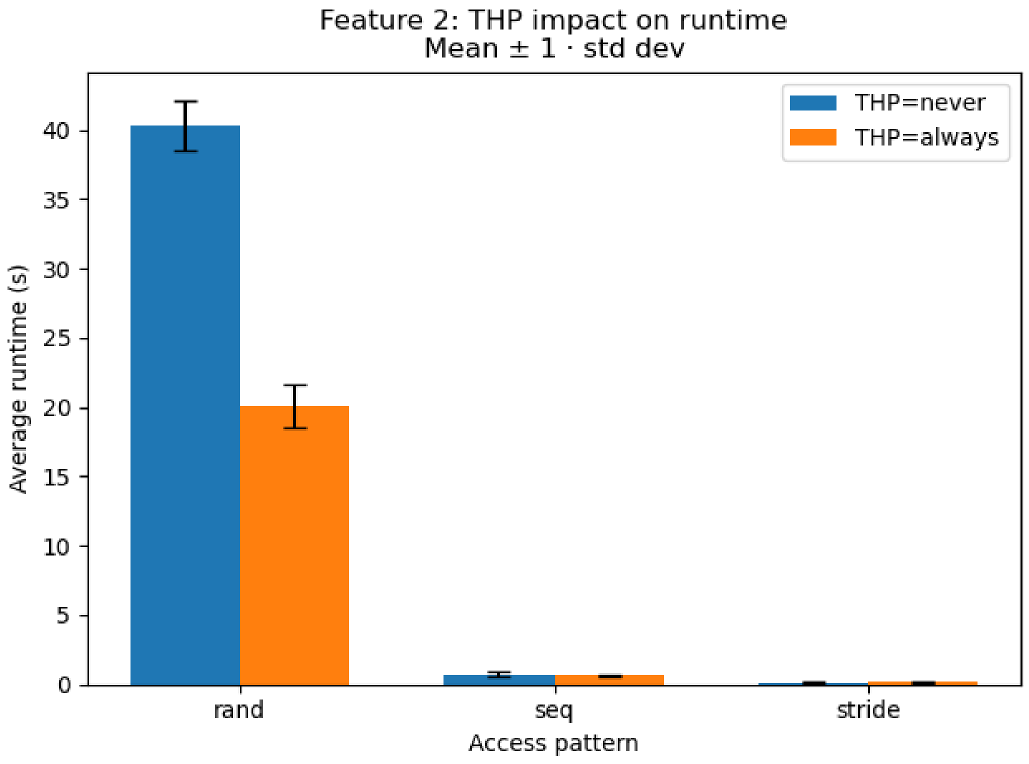

Figure 3: Feature 2: effect of Transparent Huge Pages on runtime for sequential, stride, and random access patterns. Bars show mean runtime; error bars show ±1 standard deviation.

Analysis

For a large, mostly sequential scan, enabling THP slightly reduces runtime relative to `never`. This is consistent with lower TLB pressure when mapping memory with fewer, larger pages.

For the stride pattern, the benefit from THP is smaller and depends on stride. When stride stays within a few cache lines, the access pattern still has spatial locality and THP can help; for larger strides, accesses are more scattered and THP provides less benefit.

For random access, THP has little effect on runtime. Randomized indices cause misses in both the cache and TLB regardless of page size. Measured runtimes are dominated by memory latency and do not change significantly when toggling THP.

3 Feature 3: SMT Interference

3.1 Benchmark Design

Feature 3 uses the same CPU-bound benchmark `cpu_burn.cpp` as Feature 1, but focuses specifically on Simultaneous Multithreading (SMT) interference. The script `run_experiments.sh` constructs three scenarios:

- **S1_single**: one thread pinned to a single logical CPU (for example CPU 0).
- **S2_thread0 / S2_thread1**: two threads intended to run on SMT siblings of the same physical core.
- **S3_thread0 / S3_thread1**: two threads pinned to different physical cores (for example CPU 0 and CPU 2).

Each thread runs the loop for approximately 5 seconds and reports `iters` and `RUNTIME_SECONDS`. On this hybrid 12th-gen CPU, the mapping from logical IDs to physical cores is complex, so the pinning for same-core SMT is approximate and may not always target exact siblings; this limitation is noted when interpreting results.

The script `plot_feature3.py` aggregates the data and generates two plots: raw iterations and normalized throughput.

3.2 Raw Iteration Throughput

Figure 4 shows average iterations per scenario and thread. The x-axis label is "Scenario / Thread" and the y-axis label is "Average iterations in 5 s". Bars show mean iterations; error bars show ±1 standard deviation across runs.

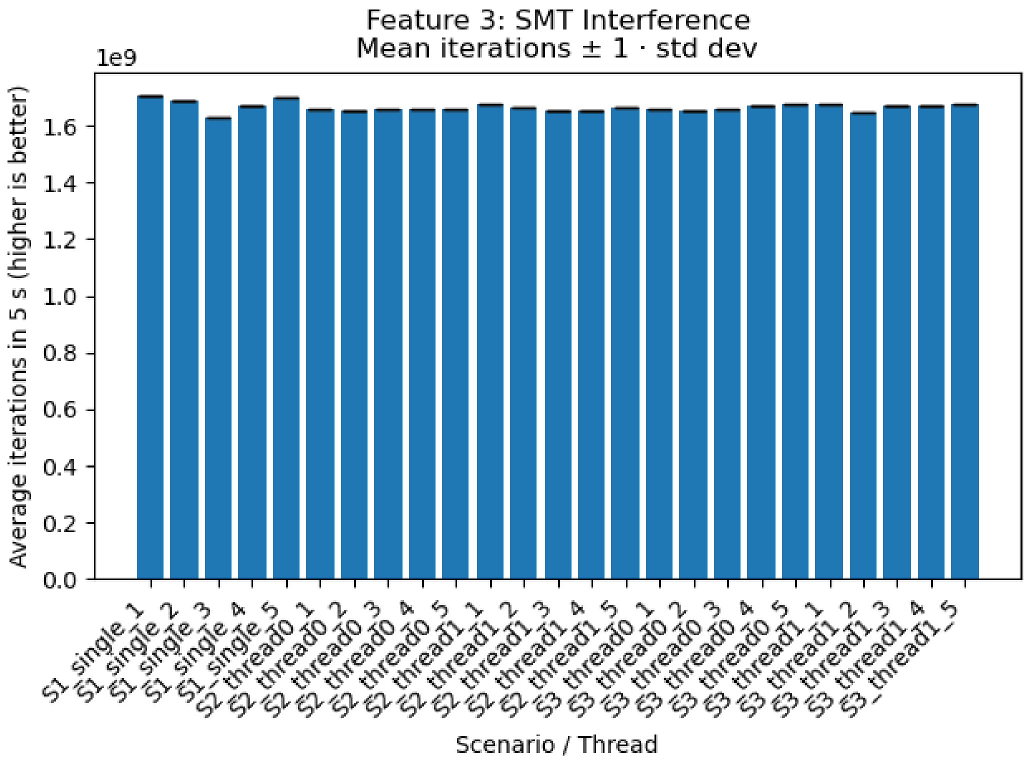

Figure 4: Feature 3: SMT interference. Average iterations completed in 5 seconds for single-thread, same-core SMT, and different-core scenarios. Bars show mean; error bars show ±1 standard deviation.

Analysis

The single-thread baseline (S1_single) achieves the highest iteration count, representing full access to a physical core. The S2 and S3 scenarios show per-thread iteration counts that are slightly lower but still close to the baseline. Across all configurations, the differences in average iteration count are modest, and the error bars overlap significantly.

This behavior suggests that, for this particular workload and CPU, SMT interference is relatively mild. The loop does not fully saturate all functional units or memory bandwidth, so two CPU-bound threads can share compute resources with only a small drop in throughput per thread.

The hybrid-core layout also matters. Logical CPU indices do not map directly to physical cores in a simple pattern, so `taskset` may sometimes place the "same-core" threads on different underlying cores with fewer shared resources. This makes the S2 and S3 results look very similar in terms of iteration throughput.

3.3 Normalized Throughput

To emphasize relative differences, Figure 5 normalizes each configuration's throughput to a single-thread baseline. The reference is S1_single, assigned a value of 1.0. Each other bar shows:

$$\text{normalized throughput} = \frac{\text{mean iterations of configuration}}{\text{mean iterations of S1\_single}}.$$

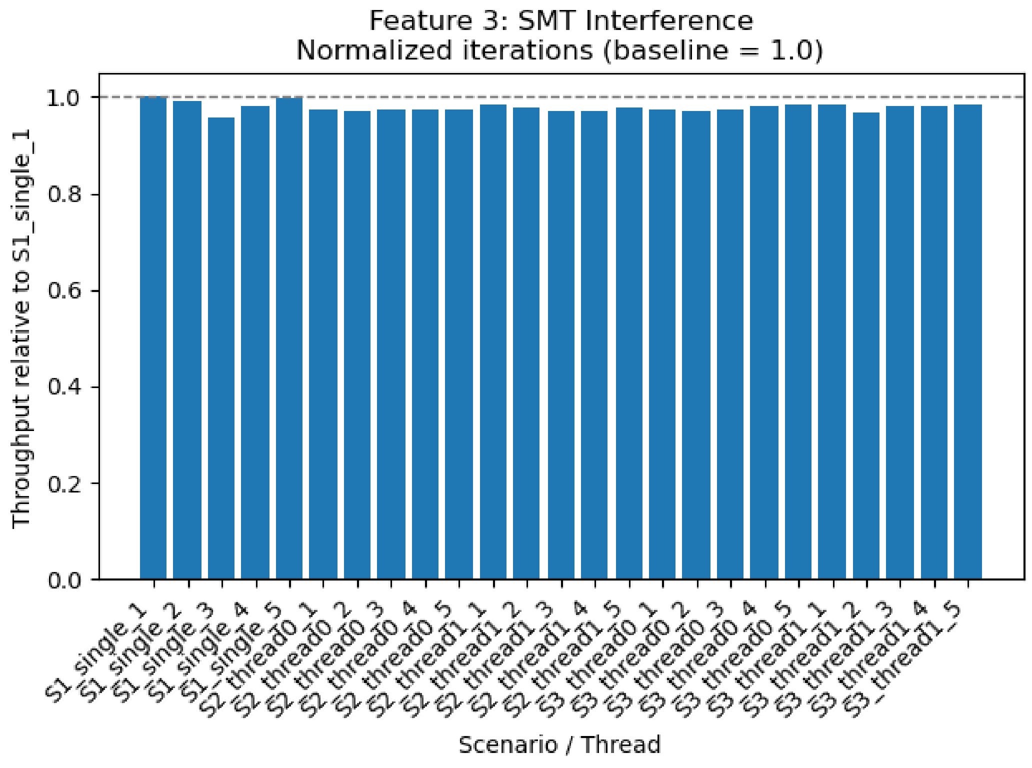

Figure 5: Feature 3: normalized SMT throughput. Each bar shows mean iterations relative to the single-thread baseline (normalized to 1.0).

Analysis

All normalized throughput values fall between approximately 0.9 and 1.0. This confirms that the effective throughput for each thread in the S2 and S3 scenarios is within about 10% of the single-thread baseline. This result can be interpreted in two ways:

- The `cpu_burn` workload does not fully utilize the core, so SMT sharing causes only a small loss in per-thread performance.
- Due to hybrid-core complexity, the "same-core" S2 threads may not always be placed on strict SMT siblings in hardware. In that case, S2 behaves more like two independent cores, which again explains why normalized throughput remains close to 1.0.

Overall, Feature 3 shows that on this system and workload, SMT interference is limited. The main lesson is that affinity control should be combined with awareness of the actual core topology, especially for hybrid CPUs.

4 Feature 4: Cache Prefetcher and Stride Effects

4.1 Benchmark Design

Feature 4 explores how hardware prefetchers and cache behavior respond to different strides in memory access. The `mem_pattern.cpp` benchmark allocates an array (for example 256 MB) and supports three patterns:

- **Sequential** (`seq`): access elements with stride 1.
- **Stride** (`stride`): access elements with a configurable stride of 1, 2, 4, ..., 128 elements.
- **Random** (`rand`): access elements in a randomized order.

For each configuration, the benchmark performs a fixed number of passes over the array and reports `RUNTIME_SECONDS`. Results are stored in `results/feature4_prefetch.csv`. The plotting script computes mean runtime and standard deviation over 3 runs.

4.2 Runtime vs Stride

Figure 6 plots average runtime for the stride pattern as a function of stride length. The x-axis label is "Stride (elements)" and the y-axis label is "Average runtime (s)". Error bars show ±1 standard deviation. The dashed horizontal line shows the approximate runtime for the sequential pattern, and the dotted horizontal line shows the runtime for the random pattern.

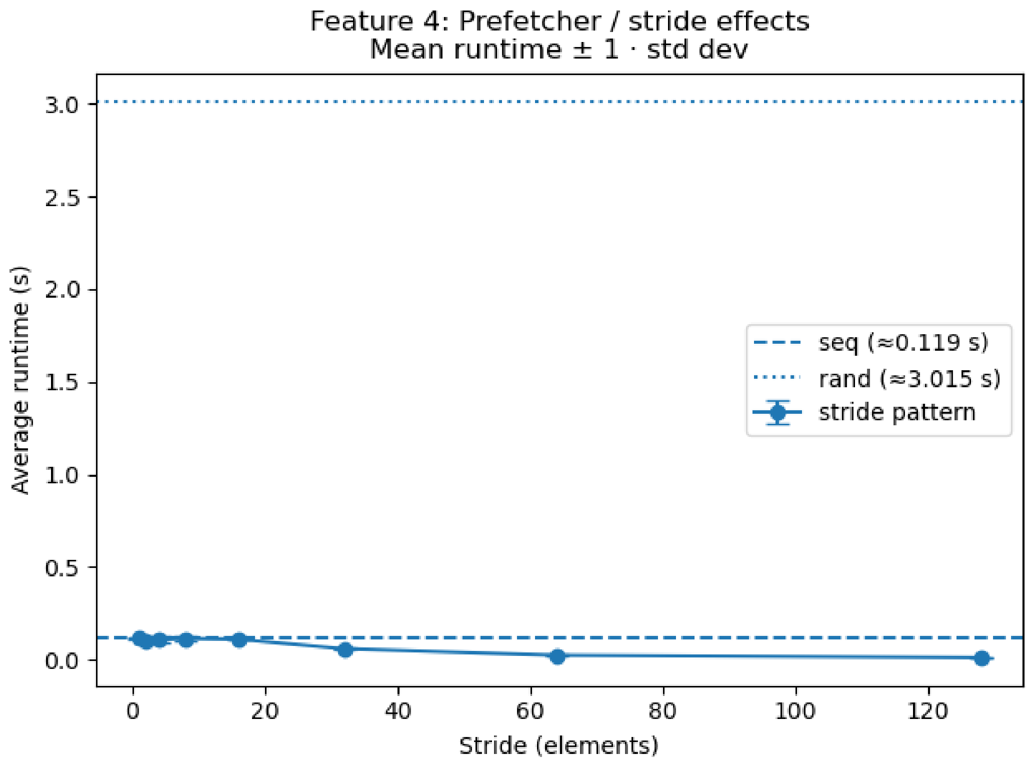

Figure 6: Feature 4: runtime vs stride for the stride access pattern. Error bars show ±1 standard deviation. Dashed and dotted lines indicate approximate sequential and random access runtimes.

Analysis

In the measured results, the stride pattern shows runtimes that are slightly lower than the sequential baseline and remain nearly flat as stride increases from small to large values. The sequential pattern runtime is around

0.1 s, while random access is much slower (around 3 s). The stride points lie below the sequential reference line and far below the random reference.

This behavior indicates that, in the current implementation, the stride pattern effectively touches fewer elements as stride increases. When stride jumps across the array, each pass visits a subset of the full array, so less total work is performed compared to the strictly sequential scan. Because the number of iterations over memory decreases with stride, runtime decreases even though spatial locality is reduced.

The random pattern is significantly slower because each access targets a different cache line and page, defeating both hardware prefetchers and TLB locality. Random access therefore represents an upper bound on runtime for this workload.

Compared to the idealized expectation of "larger stride → higher runtime" with a fixed amount of work, these measurements highlight a different effect: changing the stride here changes both the access pattern and the effective working-set coverage. The main conclusion from Feature 4 is that:

- Hardware prefetchers handle both sequential and moderate-stride patterns efficiently on this system.
- Random access remains the worst case due to poor cache and TLB locality.
- When designing microbenchmarks, it is important to ensure that different stride values touch a comparable number of elements; otherwise, stride experiments primarily measure working-set reduction rather than prefetcher limits.

6 Conclusion

The experiments in this project examined four hardware–software features that shape performance on a modern CPU: affinity, Transparent Huge Pages, SMT, and cache prefetching with different strides. A simple set of microbenchmarks, combined with careful control of CPU frequency, thread pinning, and background activity, made it possible to isolate the impact of each feature.

For CPU affinity, results showed that explicit pinning reduces variability and that placing compute-heavy threads on different cores improves per-thread throughput compared to sharing a core. THP provided measurable benefits mainly for large, sequential memory scans, with limited impact on random access patterns. SMT experiments indicated only modest interference on this workload: normalized throughput for all configurations stayed within about 10% of the single-thread baseline, reflecting both hybrid-core topology and the fact that the loop does not fully saturate the core.

The stride and prefetcher study highlighted that sequential and moderate-stride patterns are handled efficiently, while random access remains a clear worst case due to poor cache and TLB locality. The results also emphasized that benchmark design must keep the effective working set comparable across stride values to avoid confusing reduced work with prefetcher effects.

Overall, the project demonstrates how small, targeted benchmarks and basic system tools such as `taskset` and `perf stat` can be used to reason about performance, isolation, and scalability on real hardware, and how experimental design choices influence both the numbers collected and the conclusions drawn from them.

6 Reproducibility and Files

All experiments can be reproduced by the following workflow:

1. Build the benchmarks and run all experiments:
   - `./run_experiments.sh`
2. Generate plots for Features 1, 2, and 4:
   - `python3 plots.py`
3. Generate plots for Feature 3:
   - `python3 plot_feature3.py`

The main source files are:

- Benchmarks: `cpu_burn.cpp`, `mem_scan.cpp`, `mem_pattern.cpp`
- Experiment driver: `run_experiments.sh`
- Plotting scripts: `plots.py`, `plot_feature3.py`
- Result CSVs: `results/feature1_affinity.csv`, `results/feature2_thp.csv`, `results/feature3_smt.csv`, `results/feature4_prefetch.csv`

Combined with the environment description in Section 0, these files support reproducible measurement and analysis for all four selected OS/CPU features on this platform.