International Journal of Emerging Trends in Computer Science and Information Technology
ISSN: 3050-9246 | https://doi.org/10.63282/3050-9246.IJETCSIT-V7I2P125
Eureka Vision Publication | Volume 7, Issue 2, 187-190, 2026

Original Article

Performance Characterization of AI Workloads on CPU: A Methodological Framework

Rajalakshmi Srinivasaraghavan
Independent Researcher Leander, USA.

Received On: 12/03/2026 ***Revised On: 11/04/2026*** ***Accepted On: 19/04/2026*** ***Published On: 26/04/2026***

Abstract - This paper presents a systematic methodology for characterizing AI workload performance on CPU architectures through profiling, optimization, and analysis. We outline a complete framework encompassing workload selection, performance profiling using Linux perf tools, targeted optimization, multi-core scaling analysis, and system monitoring. This methodology provides a reusable framework for performance engineers across different architectures and deployment scenarios.

Keywords - CPU, Performance, AI, Optimization, Linux, Profiling.

1. Methodology Overview

Performance characterization of AI workloads on CPUs requires a systematic, repeatable approach that combines profiling tools, optimization techniques, and comprehensive monitoring. This paper documents a methodological framework applicable to diverse AI deployment scenarios.

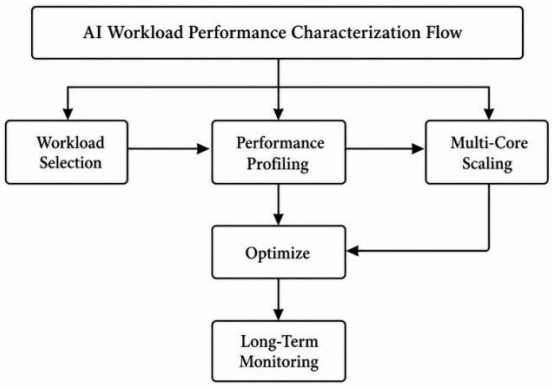

Figure 1. AI Workload Performance Characterization Flow

2. Workload Selection Profiling

Selecting appropriate AI workloads for CPU performance characterization requires consideration of several factors:

- Computational Characteristics - Mix of compute-bound and memory-bound operations - Representative of production deployment scenarios - Sufficient complexity to expose optimization opportunities
- Practical Relevance - Commonly deployed in real-world applications - Available in standard AI frameworks - Well-documented baseline performance
- Profiling Suitability - Execution time sufficient for meaningful profiling (>5 seconds) - Deterministic behavior for reproducible measurements - Clear hotspot functions for optimization targeting

3. Performance profiling with perf

Linux perf tools provide comprehensive performance analysis, capturing both high-level metrics and detailed micro-architectural events.

3.1. Profiling Methodology

```
# Step 1: High-level metrics
perf stat -e cycles,instructions,cache-misses <workload>

# Step 2: Function-level profiling
perf record -g --call-graph dwarf -F 999 <workload>

# Step 3: Analysis
perf report --stdio
perf annotate <hotspot_function>
```

3.2. Profiling Results Analysis

3.2.1. High-Level Metrics (perf stat)
Performance counter stats:

```
285,420,156,789  cycles                    #    3.498 GHz
198,745,321,456  instructions              #    0.70  insn per cycl
e
12,456,789,012  cache-references
3,789,456,123  cache-misses                #   30.42% of all cache
refs

  81.582 seconds time elapsed
  80.234 seconds user
```

1.298 seconds sys

Performance counter analysis provides insights into execution characteristics:

- Instruction Efficiency Metrics: Instructions Per Cycle (IPC) indicates how efficiently the CPU pipeline executes instructions. Low IPC suggests pipeline stalls, branch mispredictions, or resource contention.
- Memory Hierarchy Behavior: Cache reference and miss statistics reveal data locality patterns. High cache miss rates indicate memory bandwidth bottlenecks and opportunities for data layout optimization.
- Execution Time Distribution: The breakdown of time between user space, kernel space, and idle states reveals system overhead patterns and CPU utilization efficiency.

4. Multi-Core Scaling Analysis Methodology

4.1. Scaling Experiment Design

A systematic approach to analyzing multi-core performance:

Test Configuration Matrix: - **Core Count Variation**: Test with increasing core counts (1, 2, 4, 8, 16, etc.) - **Threading Modes**: Evaluate with and without simultaneous multithreading (SMT/Hyperthreading) - **Workload Consistency**: Maintain constant workload characteristics across tests - **Iteration Sufficiency**: Run sufficient iterations for statistical significance

4.1.1. Threading Configuration Strategies:

Single-Threaded Baseline

- Establish single-core performance baseline
- Measure single-thread efficiency
- Identify serial bottlenecks

SMT-Disabled Scaling

- Test scaling with one thread per physical core
- Measure parallel efficiency without SMT
- Identify resource contention patterns

SMT-Enabled Scaling

- Test with multiple threads per core
- Evaluate SMT benefit for the workload
- Assess thread-level parallelism utilization

4.2. Scaling Metrics and Analysis

4.2.1. Key Performance Indicators:

Absolute Performance

- Throughput: Operations per unit time
- Latency: Time per operation or batch
- Resource utilization: CPU, memory, cache

Scaling Efficiency

- Speedup: Performance relative to single-core baseline
- Parallel efficiency: Speedup divided by core count
- Scaling linearity: Deviation from ideal linear scaling

SMT Impact

- SMT benefit: Performance improvement with SMT enabled
- Thread utilization: Effective use of hardware threads
- Resource sharing efficiency: Impact of shared resources

SMT Benefit analysis:

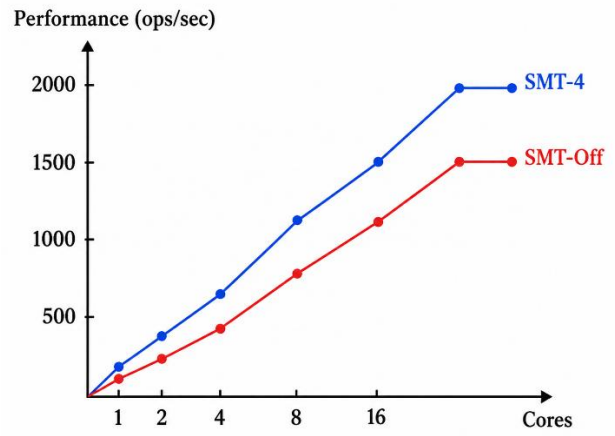

Figure 2. Performance Scaling With Core Count (SMT-4 Vs SMT-Off)

Typical SMT Benefit: 20-30%

Scaling Test Script:

```
for cores in 1 2 4 8 16; do
   export OMP_NUM_THREADS=$cores
   perf stat python inference.py --cores $cores
done
```

5. Long-Running System Characterization Methodology

5.1. Monitoring Setup

Table 1. Monitoring Stack

System Monitoring Tools
sar iostat vmstat perf (CPU) (I/O) (Memory) (HW)

Example:

```
# 2-hour  monitoring
sar -u -r 1 7200 > sar_output.txt &
iostat -x 1 7200 > iostat_output.txt &
vmstat 1 7200 > vmstat_output.txt &
python inference_continuous.py --duration 7200
```

5.2. Extended Monitoring Framework

Understanding system behavior over extended periods requires comprehensive monitoring:

5.2.1. Monitoring Tool Selection:

System Activity Reporter (sar)

- CPU utilization over time

- Memory usage patterns
- Network and I/O statistics

I/O Statistics (iostat)
- Disk I/O patterns
- Storage subsystem utilization
- I/O wait time analysis

Virtual Memory Statistics (vmstat)
- Memory allocation patterns
- Swap usage
- Context switching rates

Periodic Performance Sampling
- Regular perf snapshots
- Hardware counter trends
- Performance stability assessment

5.2.2. Monitoring Configuration:
- Sampling Frequency: Balance between granularity and overhead
- Duration: Run for sufficient time to observe steady-state behavior
- Data Collection: Automated logging with timestamps
- Baseline Comparison: Compare against short-term benchmark results

5.3. Resource Utilization Analysis

5.3.1. CPU Utilization Patterns

Analyze CPU usage distribution over time:
- **User Space Time**: Time spent in application code
- **System Time**: Kernel overhead and system calls
- **Idle Time**: Unused CPU capacity
- **I/O Wait**: Time blocked on I/O operations
- **Key Observations to Track:** - Consistency of utilization over time - Variations and anomalies - Core-to-core balance in multi-socket systems - Thermal throttling indicators

5.3.2. Memory Usage Characterization

Monitor memory consumption patterns:
- **Resident Set Size (RSS)**: Physical memory usage
- **Virtual Memory**: Total address space
- **Page Faults**: Major and minor page faults
- **Swap Activity**: Swapping to/from disk
- **Analysis Focus:** - Memory footprint stability - Memory leak detection - Swap usage patterns - Memory bandwidth utilization

5.3.3. Cache Behavior over Time

Track cache statistics throughout execution: - Cache miss rates across hierarchy levels - Temporal stability of cache behavior - Cache pollution or degradation indicators - Consistency with short-term profiling

5.3.4. Throughput Stability Analysis

Evaluate performance consistency: - Mean Performance: Average throughput over monitoring period - Variance: Standard deviation and coefficient of variation - Trend Analysis: Performance degradation or improvement over time - Outlier Detection: Identification of anomalous performance periods

5.3.5. Latency Distribution Characterization:

Analyze latency patterns: - Percentile Analysis: P50, P95, P99 latency values - Tail Latency: Maximum observed latencies - Latency Stability: Consistency of latency distribution - Anomaly Correlation: Relating latency spikes to system events

5.4. System-Level Insights and Patterns
- Thermal Behavior: - Monitor CPU temperature trends - Identify thermal throttling events - Evaluate cooling system adequacy - Assess sustained performance capability
- Power Consumption: - Track system power draw over time - Calculate energy efficiency metrics - Identify power optimization opportunities - Evaluate performance-per-watt
- NUMA Effects: - Analyze cross-socket memory access patterns - Measure local vs. remote memory latency - Evaluate NUMA-aware optimizations - Assess memory placement strategies
- Background System Activity: - Identify interference from OS tasks - Correlate performance variations with system events - Evaluate impact of background processes - Assess resource contention patterns
- Resource Headroom Analysis: - Evaluate unutilized capacity - Identify bottleneck resources - Assess scalability potential - Determine optimization priorities

6. Recommendations

Table 2. Performance Engineering Best Practices Checklist

Performance Engineering Checklist
☑ Profile before optimizing
☑ Establish baseline measurements
☑ Optimize iteratively
☑ Use architecture-specific features
☑ Monitor long-term stability
☑ Verify numerical correctness

- For Performance Engineers: - Always profile to identify actual bottlenecks - Apply optimizations incrementally with validation - Leverage architecture-specific features Monitor stability over extended periods
- For System Architects: - Ensure adequate memory bandwidth (2-3x headroom) - Design cache systems for AI workload characteristics - Consider NUMA effects in multi-socket systems - Plan for sustained high utilization without throttling
- For AI Practitioners: - Use quantization (INT8) for inference workloads - Optimize batch size for throughput-latency tradeoff - Select frameworks

with strong CPU optimization - Configure architecture-optimized libraries

7. Conclusion

This methodological framework provides a systematic, repeatable approach to characterizing and optimizing AI workload performance on CPU architectures. By following the structured processes outlined from workload selection through profiling, optimization, scaling analysis, and long-term monitoring performance engineers can achieve significant improvements while maintaining stability and efficiency.

The framework is architecture-agnostic and applicable across different CPU platforms, AI frameworks, and deployment scenarios. Success requires combining profiling tools, optimization techniques, and comprehensive monitoring with domain expertise and systematic analysis. This approach enables informed decision-making about resource allocation, optimization priorities, and deployment strategies for AI workloads on CPU systems.

References

[1] Brendan Gregg. “Systems Performance: Enterprise and the Cloud.” 2nd Edition, 2020.
[2] Linux perf Documentation. https://perf.wiki.kernel.org/
[3] PyTorch Performance Tuning Guide. https://pytorch.org/tutorials/recipes/recipes/tuning_guide.html
[4] TensorFlow Performance Guide. https://www.tensorflow.org/guide/performance
[5] Hennessy & Patterson. “Computer Architecture: A Quantitative Approach.” 6th Ed, 2017.