Welcome!

Optimization & Vectorization

D. Alexandridis, J. Bikker - Lecture 3: “Profiling”
spoiler alert!

today’s lecture is not a lecture on using vtune (or any other profiler for that matter...)

today:
- setting up good benchmarks
- terminology and metrics
- profiling behind the scene
Today's Agenda:

- Performance Analysis
- Terminology and Metrics
- Performance Analysis Approaches
Performance Analysis

Consistent approach

1. **Determine optimization requirements**
2. **Profile: determine hotspots**
3. **Analyze hotspots: determine scalability**
4. **Apply high-level optimizations to hotspots**
5. **Profile again**
6. **Parallelize / vectorize / use GPGPU**
7. **Profile again**
8. **Apply low level optimizations to hotspots**
9. **Repeat steps 7 and 8 until time runs out**
10. **Report**

Before we can actually optimize our application, we must know how to determine its performance!
Performance Analysis

How do we analyse performance?

Example on the right is called a ‘microbenchmark’.

Microbenchmarking is often used to test some simple hypothesis.

Q: What information do we get here?
- Duration to execute the code in … (milliseconds, cycles?)
- Compare your code to a different piece of code

```plaintext
int benchmark() {
    for (int i = 0; i < ITERATIONS; i++) {
        Timer . Start();
        // code to benchmark
        Timer . Stop();
    }
}
```
Performance Analysis

Performance analysis

- What do timers measure?
- Noise in measurements
- Testing environment
- How do we compare two versions of the same code to one another?
Performance Analysis

Timers

Usually, two types of timers for benchmarking.

1. System-wide high-resolution timer
2. Time Stamp Counter (TSC)
Performance Analysis

System-wide high-resolution counter

Number of ticks that have transpired since the epoch

- Unix: January 1st 1970, 00:00:00 UTC
- Windows: January 1st 1601, 00:00:00 UTC

```c
#include <time.h>

... clock_t t = clock();
// Code to benchmark

// CLOCKS_PER_SEC = 1000 for Windows

float timePassed = ((float)t) / CLOCKS_PER_SEC;
```

Q: what is the epoch for the Christian (our) calendar?

A: the day Jesus was born
Performance Analysis

Time Stamp Counter

Counts the number of reference cycles elapsed

- Hardware timer
- Does not account for DFS

```c
#include <x86intrin.h>

uint64_t start = __rdtsc();
// Code to benchmark
uint64_t clocks_elapsed = __rdtsc() - start;
```

1 GHz = \(10^9 = 1,000,000,000\) cycles per second!

64-bit values hold values up to
\(2^{64} = 18,446,744,073,709,551,616\)
Performance Analysis

Noise in measurements

Performance of consecutive runs usually vary

- Dynamic Frequency Scaling (DFS): increases the frequency of the CPU temporarily
- Filesystem cache
Performance Analysis

Testing environment

The environment in which you test your application might also greatly affect your performance analysis

- Cloud environments
- Resource-sharing with neighboring processes
- Testing device
Performance Analysis

How do we compare two versions of the same code to one another?

Usual approach:
1. Measure baseline performance of original application
2. Measure performance of modified application
3. Compare

Variance due to non-deterministic system behavior!
How do we compare two versions of the same code to one another?

1. Take N measurements per benchmark
2. Compare multiple metrics
   - Mean runtime
   - Entire interval
   - Probability densities
3. Conclusion
Performance Analysis

When do we have enough samples?

Depends on the required accuracy

Lower variance yields a more accurate comparison

Option: run at least n times, until the standard deviation lies within the desired range
Performance Analysis

Analysis of an application is not trivial!

- Run performance test in realistic scenario's
- Use the right tools for your analysis
- Use the right metrics to draw a conclusion
Today’s Agenda:

- Performance Analysis
- Terminology and Metrics
- Performance Analysis Approaches
Terminology and Metrics in performance Analysis

Until now we have only measured ‘time’ as a metric for performance

Time is a good indicator of performance, but it does not help us optimize our code

Use various metrics to understand the characteristics and optimization opportunities of your application!

Q: Why is time not (always) a good metric for software optimization?
Terminology and Metrics in performance Analysis

- Retired vs. executed instruction
- CPU utilization
- CPI and IPC
- Micro-operations
- Pipeline slot
- Core vs. reference cycles
- Cache misses
- Branch mispredictions

But these are not the only metrics for performance analysis...
### Terminology and Metrics

- **MUX Reliability**
- **OpenMP Analysis, Collection Time**
- **Page Walk**
- **OpenMP Potential Gain**
- **Lock Contention**
- **OpenMP Region Time**
- **Other**
- **Outgoing Bandwidth Bound**
- **Available Overhead Time**
- **Average Parallel Region Time**
- **Average Paused Time**
- **Average Pipeline Stalls**
- **Pre-Decycle Wrong**
- **Remote Cache Access Count**
- **Remote Dram Access Count**
- **Remote I/O Latency**
- **Retire Stalls**
- **Retiring**
- **Self Time and Total Time**
- **Serial CPU Time**
- **SMP Busy Wait Time**
- **Other**
- **Serial Time (outside parallel regions)**
- **SIMD Assists**
- **SIMD Compute-to-L1 Access Ratio**
- **SIMD Compute-to-L2 Access Ratio**
- **SIMD Instructions per Cycle**
- **Slow LEA Stalls**
- **SMC Machine Clear**
- **SMP FLops per Cycle**
- **SP GLOPS**
- **Spin Time**
- **Cache Clears**
- **Imbalance or Serial Spinning**
- **Clockt**
- **Lock Contention**
- **Other**
- **Spin and Overhead Time**
- **Atomic Tests**
- **Creation**
- **Scheduling**
- **Reduction**
- **Scheduling**
- **Tasking**
- **Split Stores**
- **Cycles**
- **Store Bound**
- **Store Latency**
- **Cycles**
- **Task Time**
- **Thread Concurrency**
- **Thread Overcommit**
- **Total Iteration Count**
- **[uops]**
- **VPU Utilization**
- **Wait Count**
- **Runtime**
- **Branch Div Factor**

### CPU Metrics

- **I/O Wait Time**
- **IPC**
- **L1 Bound**
- **4K Aligned**
- **DTLB Overhead**
- **FB Full**
- **Loads Blocked by Store Forwarding**
- **Lock Latency**
- **Split Loads**
- **L1 Hit Rate**
- **L1D Replacement Penalty**
- **L1D Replacements**
- **L1 Stall Cycles**
- **L2 Bound**
- **L2 Hit Bound**
- **L2 Hit Rate**
- **L2 Prefetcher Allocations**
- **L2 Input Requests**
- **L2 Miss Bound**
- **L2 Miss Count**
- **L2 Replacement Count**
- **L2 Replacements**
- **L3 Bound**
- **L3 Miss Count**
- **L3 Hit Rate**
- **L3 Latency**
- **L3 Cache**
- **SQ Full**
- **LLC Load Misses Serviced By Remote DRAM**
- **LLC Miss Count**
- **LLC Replacement Count**
- **LLC Replacements**
- **Local DRAM Access Count**
- **Local Persistent Communication (MPK)**
- **Logical Core Utilization**
- **Logical Core Utilization**
- **Loop Entry Count**
- **(Info) LSD Coverage**
- **Machine Cycles**
- **Max DRAM Single-Page Bandwidth**
- **Max DRAM System Bandwidth**
- **MCDRAM Bandwidth Bound**
- **MCDRAM Cache Bandwidth Bound**
- **MCDRAM Flat Bandwidth Bound**
- **Memory Bandwidth**
- **Memory Bound**
- **DRAM Bound**
- **NUMA: % of Remote Accesses**
- **Memory Efficiency**
- **Microarchitecture Usage**
- **Microcode Sequencer**
- **Mispredicts Restored**
- **MO Machine Clear Overhead**
- **MPI Imbalance**
- **MPI Rank on the Critical Path**
- **MS Entropy**

---

**Terminology and Metrics**

**Retired vs. Executed Instruction**

*Retired instruction* are instructions from which the results are committed after execution.

*Executed instructions* are instructions which are wrongly speculatively executed, and therefore never committed.
Terminology and Metrics

Core vs. Reference Cycles

CPUs employ a clock signal to pace their operations. The rate at which the CPU operates is the frequency. CPUs do not run at a steady rate.
Terminology and Metrics

Core vs. Reference Cycles

CPUs use a technique called ‘Dynamic Frequency Scaling’, temporarily boosting their clock speeds

Reference cycles: base frequency at which the CPU operates
Core cycles: actual frequency at which the CPU operates

Turbo Utilization = \frac{\text{Core Cycles}}{\text{Reference Cycles}}
Terminology and Metrics

CPU Utilization

The time that the CPU was not in ‘idle’ state

Low utilization usually indicates poor performance

CPU Utilization = \frac{\text{Reference Cycles Passed}}{\text{Time Stamp Counter}}
**Terminology and Metrics**

**CPI & IPC**

**IPC:** average number of retired instructions per cycle

**CPI:** average number of cycles to retire a single instruction

\[
IPC = \frac{\text{INST\_RETIR\_ANY}}{\text{CPU\_CLK\_UNHALTED\_THREAD}}
\]

\[
CPI = \frac{1}{PCI}
\]
Terminology and Metrics

Micro-operations (UOPs)

Instructions in x86 architecture are translated in micro-operations.

**ADD EAX, EBX**  // generates 1 micro-operation
**ADD EAX, [MEM1]**  // generates 2 micro-operations

Certain micro-operations can be fused in a single micro-operation

Read chapter 4.4 of Performance Analysis and Tuning for more in-depth information on micro-operations and fusions.

https://easyperf.net/blog/2018/02/23/MacroFusion-in-Intel-CPUs#micro-macro-fusion
Terminology and Metrics

Pipeline Slot

One pipeline slot can process one micro-operation

A 3-wide pipeline can handle 3 micro-operations each cycle.
Terminology and Metrics

Pipeline Slot

One pipeline slot can process one micro-operation.

A 3-wide pipeline can handle 3 micro-operations each cycle.

Q: What is the efficiency of the code displayed in the pipeline diagram below? (Note: the pipeline is 4-wide!)

A: We have 16 slots in total (4x4), 4 remain unutilized. Efficiency is 12 / 16 = 75%
Terminology and Metrics

Cache Miss

Cache misses will be discussed extensively in the next lectures.
Terminology and Metrics

Mispredicted Branch

Conditional code is executed speculatively.

Branch mispredictions involve a penalty of between 10 and 20 clock cycles.
Terminology and Metrics in performance Analysis

- Time is a good metric for performance, but provides little insight for optimization
- Other metrics provide more insight in optimization opportunities
Today's Agenda:

- Performance Analysis
- Terminology and Metrics
- Performance Analysis Approaches
Performance Analysis Approaches

We want to collect other metrics and information on our application than time, but how?
Performance Analysis Approaches

We will discuss several analysis approaches

- Code instrumentation
- Tracing
- Workload Characterization
- Sampling
Performance Analysis Approaches

Code Instrumentation

Insert extra code to collect runtime information

Mainly used on macro level, not micro (low) level

Advantages
- Same privileges as application
- Explore program flow
- Collect any information on any variable

Disadvantages
- Recompilation time
- Possibly adjusts program behavior
- Cannot access OS or CPU data (e.g., scheduling)

```c
int foo() {
    printf("foo called");
    // function body
}
```
Performance Analysis Approaches

**Binary Instrumentation**

Add instrumentation in the executable file on a source-code level

**Static:** performed before execution

**Dynamic:** performed during runtime

- Instruction count and function call counts
- Intercepting function calls and execution of any instruction
- Allow “record and replay” the program region
Performance Analysis Approaches

Tracing

Similar to code instrumentation; except we cannot access the source code

Tracing enables us to reconstruct the entire stack

However, there are some disadvantages:

• Compiling instrumented code takes more time
• Detailed traces are large
• Overhead of saving traces to disk
• Missing information may invalidate the entire trace
Performance Analysis Approaches

Workload Characterization

Describe the workload in means of quantitative parameters and functions

In the Top-Down Microarchitecture Analysis (TMA) methodology we characterize an application into 4 buckets:

1. Front End Bound
2. Back End Bound
3. Retiring
4. Bad Speculation

Read chapter 6.1 of the book for more on TMA
Performance Analysis Approaches

Performance Counter

Performance Monitoring Counters (PMCs) are used

Counting Mode vs. Sampling Mode
Performance Analysis Approaches

Performance Counter

Performance Monitoring Counters (PMCs) are used

Counting Mode vs. Sampling Mode

There are hundreds of counters
Performance Analysis Approaches

Multiplexing

We usually want to count more events than there are PMCs available (typically 4 per hardware thread)

Multiplexing is a method to enable counting of multiple events

Upscale to estimate actual values

\[
\text{event\_count} = \text{count} \times \frac{\text{time\_running}}{\text{time\_enabled}}
\]
Performance Analysis Approaches

Sampling

Interrupt execution multiple (thousands) of times per second and collect a ‘sample’

User-mode sampling embeds an agent library into the profiled application

Hardware event-based sampling (EBS) uses hardware PMCs to trigger interrupts
Performance Analysis Approaches

Hotspots with EBS

1. Initialize counter
2. Wait for overflow
3. Take sample, reset counter

The value of ‘N’ determines how frequently we take samples

Based on the samples, we can determine the ‘hottest’ places in our application

```
- Initialize counter
- Wait for overflow
- Take sample, reset counter

The value of ‘N’ determines how frequently we take samples

Based on the samples, we can determine the ‘hottest’ places in our application
```
Performance Analysis Approaches

Call Stacks

The ‘hottest’ function in an application might get called by multiple callers

The Control Flow Graph (CFG) helps us understand which part of the program calls this function most often

We can visualize the CFG as:

// Hot function
void foo() { /* code */ }

void func1() {
  /* code */
  foo(); // call foo
}

void func2() {
  /* code */
  foo(); // call foo
}

void func3() {
  /* code */
  foo(); // call foo
}
Performance Analysis Approaches

Call Stacks

Profiling tools can collect call stack information during sample collection

We can now see that func1 is calling foo most often
Performance Analysis Approaches

We have discussed several analysis approaches

Profiling tools do a better job than us in running and interpreting these analyses

Understanding how these methods work helps us
- Grasp the strengths and shortcomings of each method
- Select the appropriate analysis approach for our specific problem
Today's Agenda:

- Performance Analysis
- Terminology and Metrics
- Performance Analysis Approaches
Performance Analysis

Further reading

• From the book ‘Performance Analysis and Tuning on Modern CPUS’ by Denis Bakhvalov read:
  • Chapter 5.5 – roofline performance model
  • Chapter 6.1 – top-down microarchitectural analysis methodology
• List of all of Intel’s performance events
Performance Analysis

Remember

- Always profile your application before applying any optimizations
- Select the appropriate performance analysis approach
- Setup good benchmarks/tests: consider noise from the system and the environment
Profile your practical assignment

Deadline is this afternoon at 17.00, so you are almost done with assignment 1. Clone the original repository one more time and reconstruct your original testing-scenario (e.g., increase the number of flags, disable rendering, etc.).

Profile the original application and your optimized version with your testing-setup. Write down and compare the following metrics:

- Instructions per cycle (IPC)
- Memory reads, writes and cache-misses
- Number of total branches and branch-mispredictions
- CPU Utilization
- Hotspot analysis and call-stacks for top-3 hottest functions