Processor Specific Optimization

Part 4: Best Practices Performance Engineering

J. Eitzinger

PATC LRZ, 18.3.2019
Basics of Optimization

1. Define relevant test cases
2. Establish a sensible performance metric
3. Acquire a runtime profile (sequential)
4. Identify hot kernels (Hopefully there are any!)
5. Carry out optimization process for each kernel

Motivation:
- Understand observed performance
- Learn about code characteristics and machine capabilities
- Deliberately decide on optimizations

Iteratively
Best Practices Benchmarking

Preparation

- Reliable timing (Minimum time which can be measured?)
- Document code generation (Flags, Compiler Version)
- Get exclusive System
- System state (Clock, Turbo mode, Memory, Caches)
- Consider to automate runs with a skript (Shell, python, perl)

Doing

- Affinity control
- Check: Is the result reasonable?
- Is result deterministic and reproducible.
- Statistics: Mean, Best ??
- Basic variants: Thread count, affinity, working set size (Baseline!)
Best Practices Benchmarking cont.

Postprocessing

- Documentation
- Try to understand and explain the result
- Plan variations to gain more information
- Many things can be better understood if you plot them (gnuplot, xmgrace)
Focus on resource utilization

1. Instruction execution
   Primary resource of the processor.

2. Data transfer bandwidth
   Data transfers as a consequence of instruction execution.

What is the **limiting resource**?
Do you fully **utilize** available **resources**?
Overview

1. Reduce algorithmic work
2. Minimize processor work
3. Distribute work and data for optimal utilization of parallel resources
4. Avoid slow data paths
5. Use most effective execution units on chip
6. Avoid bottlenecks
Thinking in Bottlenecks

• A bottleneck is a performance limiting setting
• Microarchitectures expose numerous bottlenecks

Observation 1:
Most applications face a single bottleneck at a time!

Observation 2:
There is a limited number of relevant bottlenecks!
Process vs. Tool

Reduce complexity!

We propose a human driven process to enable a systematic way to success!

• Executed by humans.
• Uses tools by means of data acquisition only.

Uses one of the most powerful tools available: Your brain!

You are a investigator making sense of what’s going on.
Performance Engineering Process: Analysis

Step 1 Analysis: Understanding observed performance
Step 2 **Formulate Model**: Validate pattern and get quantitative insight.
Step 3 Optimization: Improve utilization of offered resources.
Performance pattern classification

1. Maximum resource utilization  
   (computing at a bottleneck)

2. Optimal use of parallel resources

3. Hazards  
   (something “goes wrong”)

4. Use of most effective instructions

5. Work related  
   (too much work or too inefficiently done)
<table>
<thead>
<tr>
<th>Pattern</th>
<th>Performance behavior</th>
<th>Metric signature, LIKWD performance group(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bandwidth saturation</td>
<td>Saturating speedup across cores sharing a data path</td>
<td>Bandwidth meets BW of suitable streaming benchmark (MEM, L3)</td>
</tr>
<tr>
<td>ALU saturation</td>
<td>Throughput at design limit(s)</td>
<td>Good (low) CPI, integral ratio of cycles to specific instruction count(s) (FLOPS_**, DATA, CPI)</td>
</tr>
<tr>
<td>Bad ccNUMA page placement</td>
<td>Bad or no scaling across NUMA domains, performance improves with interleaved page placement</td>
<td>Unbalanced bandwidth on memory interfaces / High remote traffic (MEM)</td>
</tr>
<tr>
<td>Load imbalance / serial fraction</td>
<td>Saturating/sub-linear speedup</td>
<td>Different amount of “work” on the cores (FLOPS_*); note that instruction count is not reliable!</td>
</tr>
</tbody>
</table>
## Patterns (II): Hazards

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Performance behavior</th>
<th>Metric signature, LIKWID performance group(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>False sharing of cache lines</td>
<td>Large discrepancy from performance model in parallel case, bad scalability</td>
<td>Frequent (remote) CL evicts (CACHE)</td>
</tr>
<tr>
<td>Pipelining issues</td>
<td>In-core throughput far from design limit, performance insensitive to data set size</td>
<td>(Large) integral ratio of cycles to specific instruction count(s), bad (high) CPI (FLOPS_*, DATA, CPI)</td>
</tr>
<tr>
<td>Control flow issues</td>
<td>See above</td>
<td>High branch rate and branch miss ratio (BRANCH)</td>
</tr>
<tr>
<td>Micro-architectural anomalies</td>
<td>Large discrepancy from simple performance model based on LD/ST and arithmetic throughput</td>
<td>Relevant events are very hardware-specific, e.g., memory aliasing stalls, conflict misses, unaligned LD/ST, requeue events</td>
</tr>
<tr>
<td>Latency-bound data access</td>
<td>Simple bandwidth performance model much too optimistic</td>
<td>Low BW utilization / Low cache hit ratio, frequent CL evicts or replacements (CACHE, DATA, MEM)</td>
</tr>
</tbody>
</table>
### Patterns (III): Work-related

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Performance behavior</th>
<th>Metric signature, LIKWID performance group(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Synchronization overhead</strong></td>
<td>Speedup going down as more cores are added / No speedup with small problem sizes / Cores busy but low FP performance</td>
<td>Large non-FP instruction count (growing with number of cores used) / Low CPI (FLOPS_, CPI)</td>
</tr>
<tr>
<td><strong>Instruction overhead</strong></td>
<td>Low application performance, good scaling across cores, performance insensitive to problem size</td>
<td>Low CPI near theoretical limit / Large non-FP instruction count (constant vs. number of cores) (FLOPS_, DATA, CPI)</td>
</tr>
<tr>
<td><strong>Excess data volume</strong></td>
<td>Simple bandwidth performance model much too optimistic</td>
<td>Low BW utilization / Low cache hit ratio, frequent CL evicts or replacements (CACHE, DATA, MEM)</td>
</tr>
<tr>
<td><strong>Code composition</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expensive instructions</td>
<td>Similar to instruction overhead</td>
<td>Many cycles per instruction (CPI) if the problem is large-latency arithmetic</td>
</tr>
<tr>
<td>Ineffective instructions</td>
<td></td>
<td>Scalar instructions dominating in data-parallel loops (FLOPS_, CPI)</td>
</tr>
</tbody>
</table>
Where to start

Look at the code and understand what it is doing!

Scaling runs:
- Scale #cores inside ccNUMA domain
- Scale across ccNUMA domains
- Scale working set size (if possible)

HPM measurements:
- Memory Bandwidth
- Instruction decomposition: Arithmetic, data, branch, other
- SIMD vectorized fraction
- Data volumes inside memory hierarchy
- CPI
Pattern: Bandwidth Saturation

1. Perform scaling run inside ccNUMA domain
2. Measure memory bandwidth with HPM
3. Compare to micro benchmark with similar data access pattern

Measured bandwidth spmv:
45964 MB/s
Synthetic load benchmark:
47022 MB/s
## Pattern: Instruction Overhead

1. Perform a HPM instruction decomposition analysis
2. Measure resource utilization
3. Static code analysis

<table>
<thead>
<tr>
<th>Instruction decomposition</th>
<th>Inlining failed</th>
<th>Inefficient data structures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic FP</td>
<td>12%</td>
<td>21%</td>
</tr>
<tr>
<td>Load/Store</td>
<td>30%</td>
<td>50%</td>
</tr>
<tr>
<td>Branch</td>
<td>24%</td>
<td>10%</td>
</tr>
<tr>
<td>Other</td>
<td>34%</td>
<td>19%</td>
</tr>
</tbody>
</table>

C++ codes which suffer from overhead (Inlining problems, complex abstractions) need a lot more overall instructions related to the arithmetic instructions

- Often (but not always) “good” (i.e., low) CPI
- Low-ish bandwidth
- Low # of floating-point instructions vs. other instructions
# Pattern: Inefficient Instructions

1. **HPM measurement: Relation packed vs. scalar instructions**
2. **Static assembly code analysis: Search for scalar loads**

<table>
<thead>
<tr>
<th>Event</th>
<th>core 0</th>
<th>core 1</th>
<th>core 2</th>
<th>core 3</th>
<th>core 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSTR_RETIRED_ANY</td>
<td>2.1945e+11</td>
<td>1.7674e+11</td>
<td>1.76255e+11</td>
<td>1.75728e+11</td>
<td>1.75578e+11</td>
</tr>
<tr>
<td>CPU_CLK_UNHALTED_CORE</td>
<td>1.4396e+11</td>
<td>1.28759e+11</td>
<td>1.28846e+11</td>
<td>1.28898e+11</td>
<td>1.28905e+11</td>
</tr>
<tr>
<td>CPU_CLK_UNHALTED_REF</td>
<td>1.20204e+11</td>
<td>1.0895e+11</td>
<td>1.09024e+11</td>
<td>1.09067e+11</td>
<td>1.09074e+11</td>
</tr>
<tr>
<td>FP_COMP_OPS_EXE_SSE_FP_PACKED_DOUBLE</td>
<td>1.1169e+09</td>
<td>1.09639e+09</td>
<td>1.09739e+09</td>
<td>1.10112e+09</td>
<td>1.10032e+09</td>
</tr>
<tr>
<td>FP_COMP_OPS_EXE_SSE_FP_SCALAR_DOUBLE</td>
<td>3.62746e+10</td>
<td>3.45789e+10</td>
<td>3.45446e+10</td>
<td>3.44553e+10</td>
<td>3.44829e+10</td>
</tr>
<tr>
<td>SIMD_FP_256_PACKED_DOUBLE</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- Small fraction of packed instructions
- No AVX

- There is usually no counter for packed vs scalar (SIMD) loads and stores.
- Also the compiler usually does not distinguish!

Only solution: Inspect code at assembly level.