Software Engineering for Performance

Rob Cameron

Computing Science 276
School of Computing Science
Simon Fraser University

March 5, 2019
Performance Issues

- Execution time: CPU time taken to complete a task.
- Response time: time to respond to user queries.
- Throughput: data processing rate, e.g. GB/sec.
- Memory requirements: main memory, secondary storage.
Performance Issues

- Execution time: CPU time taken to complete a task.
- Response time: time to respond to user queries.
- Throughput: data processing rate, e.g. GB/sec.
- Memory requirements: main memory, secondary storage.

Strategic Performance Engineering

- Analyze requirements to determine potential performance concerns.
- Choose algorithms and data structures to best address concerns.
- Put small-scale performance concerns aside during initial implementation.
- Consider performance improvement engineering, if necessary.
Memory Hierarchy

- Systems generally have large main memory stores (> 1 GB).
- Access time to main memory is slow, often 100-300 CPU cycles.
- On-chip memory caches have faster access.
  - L1 cache, e.g., 32 KB, 2-4 CPU cycles
  - L2 cache, e.g., 256KB, 8-15 CPU cycles
  - L3 cache, e.g., 2MB, 40-60 CPU cycles
Memory Usage Planning

Memory Hierarchy

- Systems generally have large main memory stores (> 1 GB).
- Access time to main memory is slow, often 100-300 CPU cycles.
- On-chip memory caches have faster access.
  - L1 cache, e.g., 32 KB, 2-4 CPU cycles
  - L2 cache, e.g., 256KB, 8-15 CPU cycles
  - L3 cache, e.g., 2MB, 40-60 CPU cycles

Memory Hierarchy Implications

- Large arrays are costly.
- Sequential access patterns much better than random access.
- Multiprocessing - may have cache contention between processes.
The Cost of Special Cases

Premature Special Case Optimization

```java
if (x > 10) {
    // This code is safe to execute in general, but
    // we skip these 10 instructions for x <= 10,
    // which is true about 40% of the time.
    ....
}
```

Branch Misprediction Cost


Modern processors execute pipelines of instructions. If tests: two possible pipeline continuations. Processor must predict which branch is taken to load pipeline. Mispredictions may cost 15 CPU cycles: about 30 instructions.
The Cost of Special Cases

Premature Special Case Optimization

if (x > 10) {
    // This code is safe to execute in general, but
    // we skip these 10 instructions for x <= 10,
    // which is true about 40% of the time.
    ....
}

Branch Misprediction Cost

- Above optimization actually decreases performance.
- Reason: branch misprediction.
- Modern processors execute pipelines of instructions.
- If tests: two possible pipeline continuations.
- Processor must predict which branch is taken to load pipeline.
- Mispredictions may cost 15 CPU cycles: about 30 instructions.
Performance improvement is the idea of restructuring software specifically to focus on improvements to performance.

- start with working software
- perform performance improvement transformations
- verify functionality
Performance Improvement

Performance improvement is the idea of restructuring software specifically to focus on improvements to performance.

- start with working software
- perform performance improvement transformations
- verify functionality

Measure First, Optimize Later

- Identify the performance-critical 10% of your program.
- Typically, over 90% of execution time is taken by this performance-critical 10%.
- Premature optimization is evil.
  - Low-level coding techniques complicate program structure.
  - Low-level coding techniques are error-prone.
Program Profiling

- breaks down the execution time spent in different routines
- often implemented by program counter sampling
- can use instrumentation added to code
Program Measurement

Program Profiling
- breaks down the execution time spent in different routines
- often implemented by program counter sampling
- can use instrumentation added to code

Processor Performance Counters
- Processors have various performance counters built-in.
  - Timestamp counter can be used for precise CPU cycle counts.
  - Branch and misprediction counters give stats on branches.
  - Cache miss counters identify when data must be loaded from memory.
- Linux `perf` tools provide access to these counters.
## Test Cases First!

- Test cases validate current functionality.
- Performance improvement steps should continue to pass all tests.
- New test cases should be written if changes testing is inadequate.
- Test failures: revert the changes and try again.
**Test Cases First!**

- Test cases validate current functionality.
- Performance improvement steps should continue to pass all tests.
- New test cases should be written if changes to testing are inadequate.
- Test failures: revert the changes and try again.

**Small Steps**

- Performance improvement can be done in multiple small steps.
- Each step should preserve correctness: pass all tests.
Parabix Concept

- Programming framework for high-performance data stream processing.
- Employs novel algorithms based on \textit{bitwise data parallelism}.
  - Process 128 bytes at a time using 128 bit registers (SSE2).
- Fully utilizes processor wide vector instructions (SIMD).
- Automatically uses multiple cores even on single data streams.
Parabix Concept

- Programming framework for high-performance data stream processing.
- Employs novel algorithms based on \textit{bitwise data parallelism}.
  - Process 128 bytes at a time using 128 bit registers (SSE2).
- Fully utilizes processor wide vector instructions (SIMD).
- Automatically uses multiple cores even on single data streams.

Parabix Scalability

- Parabix scales to use available SIMD register width.
- Parabix scales to use multiple cores.
- No changes to application programs required!
icgrep 1.8

- Full-featured grep implementation using Parabix algorithms.
- Posix REs: Basic or Extended
  - All features except backreferences.
  - Restricted backreference support under development.
- Perl-compatible REs (PCRE)
icgrep 1.8

- Full-featured grep implementation using Parabix algorithms.
- Posix REs: Basic or Extended
  - All features except backreferences.
  - Restricted backreference support under development.
- Perl-compatible REs (PCRE)

UTS #18 - Unicode Regular Expressions

- Full Unicode property support.
- Set operations, e.g., `\p{Greek}&&\p{upper case}]`
- Grapheme clusters and grapheme cluster mode.
- Name property with regexp values `\p{name=/AIRPLANE/}`
- Most advanced Unicode support of any grep tool.
Example: IP address regex

\[(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\]\{3\}

- Data source: 620 MB Wikibooks document set (15 languages)
Example: IP address regex

\[
(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)(\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)){3}
\]

Data source: 620 MB Wikibooks document set (15 languages)

### Performance Results

<table>
<thead>
<tr>
<th>Program</th>
<th>Processor</th>
<th>SIMD</th>
<th>Instructions</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>grep -E</td>
<td>i7-3770 @ 3.4 GHz</td>
<td>SSE2</td>
<td>232,079 M</td>
<td>21.3 s</td>
</tr>
<tr>
<td></td>
<td>i3-5010U @ 2.1 GHz</td>
<td>AVX2</td>
<td>232,423 M</td>
<td>39.5 s</td>
</tr>
<tr>
<td></td>
<td>W-2102 @ 2.9 GHz</td>
<td>AVX-512</td>
<td>232,081 M</td>
<td>25.6 s</td>
</tr>
<tr>
<td>icgrep</td>
<td>i7-3770 @ 3.4 GHz</td>
<td>SSE2</td>
<td>3,720 M</td>
<td>0.49 s</td>
</tr>
<tr>
<td></td>
<td>i3-5010U @ 2.1 GHz</td>
<td>AVX2</td>
<td>2,193 M</td>
<td>0.49 s</td>
</tr>
<tr>
<td></td>
<td>W-2102 @ 2.9 GHz</td>
<td>AVX-512</td>
<td>1,349 M</td>
<td>0.32 s</td>
</tr>
<tr>
<td></td>
<td>W-2102 (2 cores)</td>
<td>AVX-512</td>
<td>1,388 M</td>
<td>0.20 s</td>
</tr>
</tbody>
</table>
Parallel Techniques Beat Fast Byte-at-a-Time

- Parabix applications can achieve throughput over 1 GB/sec.
- Sequential byte-at-a-time loops cannot achieve this performance.
- Future high-performance software will use parallel techniques.
- Don’t waste time with low-level sequential coding!
Parallel Techniques Beat Fast Byte-at-a-Time

- Parabix applications can achieve throughput over 1 GB/sec.
- Sequential byte-at-a-time loops cannot achieve this performance.
- Future high-performance software will use parallel techniques.
- Don’t waste time with low-level sequential coding!

Parabix Research Repository

- cs-git-research.cs.surrey.sfu.ca/cameron/parabix-devel
- wiki: parabix.costar.sfu.ca
- Built on top of the LLVM compiler infrastructure.
- Participation welcomed!
Traditional regular expression technology processes one code unit at a time using DFA, NFA or backtracking implementations.

Instead consider a bitwise data parallel approach.

Byte-oriented data is first transformed to 8 parallel bit streams (Parabix transform).

Bit stream $j$ consists of bit $j$ of each byte.

Load 128-bit SIMD registers to process 128 positions at a time in bitwise data parallel fashion (SSE2, ARM Neon, ...).

Or use 256-bit AVX2 registers of newer Intel processors.

Process using bitwise logic, shifting and addition.

Parabix methods have previously been used to accelerate Unicode transcoding and XML parsing.
Unbounded Stream Abstraction

- Program operations as if *all positions in the file are to be processed simultaneously.*
- Unbounded bitwise parallelism.
- Pablo compiler technology maps to block-by-block processing.
- Information flows between blocks using carry bits.
- LLVM compiler infrastructure for Just-in-Time compilation.
- Custom LLVM improvements further accelerate processing.
Marker Streams

- Marker stream $M_i$ indicates the positions that are reachable after item $i$ in the regular expression.
- Each marker stream $M_i$ has one bit for every input byte in the input file.
- $M_i[j] = 1$ if and only if a match to the regular expression up to and including item $i$ in the expression occurs at position $j - 1$ in the input stream.
- Conceptually, marker streams are computed in parallel for all positions in the file at once (bitwise data parallelism).
- In practice, marker streams are computed block-by-block, where the block size is the size of a SIMD register in bits.
Consider matching regular expression a\[0-9\]*\[z9\] against the input text below.

input data    a453z--b3z--az--a12949z--ca22z7--
Consider matching regular expression $a[0-9]*[z9]$ against the input text below.

$M_1$ marks positions after occurrences of $a$.

<table>
<thead>
<tr>
<th>input data</th>
<th>a453z--b3z--az--a12949z--ca22z7--</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_1$</td>
<td>.1................1...1............1.....</td>
</tr>
</tbody>
</table>
Consider matching regular expression \(a[0-9]*[z9]\) against the input text below.

- \(M_1\) marks positions after occurrences of \(a\).
- \(M_2\) marks positions after occurrences of \(a[0-9]*\).

<table>
<thead>
<tr>
<th>input data</th>
<th>a453z--b3z--az--a12949z--ca22z7--</th>
</tr>
</thead>
<tbody>
<tr>
<td>(M_1)</td>
<td>.1................1...1.............1.....</td>
</tr>
<tr>
<td>(M_2)</td>
<td>.1111...........1...111111.....111....</td>
</tr>
</tbody>
</table>
Consider matching regular expression \(a[0-9]*z9\) against the input text below.

\(M_1\) marks positions after occurrences of \(a\).

\(M_2\) marks positions after occurrences of \(a[0-9]*\).

\(M_3\) marks positions after occurrences of \(a[0-9]*z9\).

**input data**

\[a453z--b3z--az--a12949z--ca22z7--\]

\(M_1\)

\[.1........1...1........1.....\]

\(M_2\)

\[.1111........1...111111....111...\]

\(M_3\)

\[.....1........1.....1.11......1..\]
Combining 8 bits of a code unit gives a character class stream.

**CCC**(cc_a = [a])

- temp1 = (bit[1] &~ bit[0])
- temp3 = (temp1 & temp2)
- temp6 = (temp5 &~ temp4)
- cc_a = (temp3 & temp6)
Ranges of characters are often very simple to compute.

\[
\text{CCC}(\text{cc}_0_9 = [0-9])
\]

\[
\begin{align*}
\text{temp7} &= (\text{bit}[0] \mid \text{bit}[1]) \\
\text{temp8} &= (\text{bit}[2] \& \text{bit}[3]) \\
\text{temp9} &= (\text{temp8} \&\!\!\!\& \text{temp7}) \\
\text{temp10} &= (\text{bit}[5] \mid \text{bit}[6]) \\
\text{temp11} &= (\text{bit}[4] \& \text{temp10}) \\
\text{cc}_0_9 &= (\text{temp9} \&\!\!\!\& \text{temp11})
\end{align*}
\]
Multiple definitions use common subexpressions.

\[
\text{CCC}(\text{cc}_z9 = [z9])
\]

\[
\begin{align*}
\text{temp12} &= \text{bit}[4] \& \sim \text{bit}[5] \\
\text{temp13} &= (\text{temp12} \& \text{temp5}) \\
\text{temp14} &= (\text{temp9} \& \text{temp13}) \\
\text{temp15} &= (\text{temp1} \& \text{temp8}) \\
\text{temp16} &= \text{bit}[6] \& \sim \text{bit}[7] \\
\text{temp17} &= (\text{temp12} \& \text{temp16}) \\
\text{temp18} &= (\text{temp15} \& \text{temp17}) \\
\text{cc}_z9 &= (\text{temp14} \mid \text{temp18})
\end{align*}
\]
Matching Character Class Repetitions with MatchStar

\[
\text{MatchStar} (M; \mathcal{C}) = (((M^\mathcal{C}) + \mathcal{C}) \mathcal{C})_M
\]

Consider \(M_2 = \text{MatchStar}(M_1; \mathcal{C})\)

Use addition to scan each marker through the class. Bits that change represent matches. We also have matches at start positions in \(M_1\).

input data:

```
\text{a453z--b3z--az--a12949z--ca22z7--}
```

\(\mathcal{C} = [0-9]\)

\(T_0 = M_1^{\mathcal{C}}\)

\(T_1 = T_0 + \mathcal{C}\)

\(T_2 = T_1 \mathcal{C}\)

\(M_2 = T_2 \_ M_1\)
Matching Character Class Repetitions with MatchStar

\[
\text{MatchStar}(M, C) = \left(( (M \land C) + C \right) \oplus C) \lor M
\]

Consider \( M_2 = \text{MatchStar}(M_1; C) \)

Use addition to scan each marker through the class. Bits that change represent matches.

We also have matches at start positions in \( M_1 \).

```plaintext
input data
a453z--b3z--az--a12949z--ca22z7--

M_1...........1...1.........1.......

C = [0-9].111....1........11111.....11.1..

T_0 = M_1^C.........1.........1.......

T_1 = T_0 + C....1...1.............1......11..

T_2 = T_1 C.1111............111111....111...

M_2 = T_2 _ M_1.1111........1...111111....111...
```
Matching Character Class Repetitions with MatchStar

- \( \text{MatchStar}(M, C) = (((M \land C) + C) \oplus C) \lor M \)
- Consider \( M_2 = \text{MatchStar}(M_1, C) \)

<table>
<thead>
<tr>
<th>Input data</th>
<th>a453z--b3z--az--a12949z--ca22z7--</th>
</tr>
</thead>
<tbody>
<tr>
<td>( M_1 )</td>
<td>.1........................1...1.............1.............1......</td>
</tr>
<tr>
<td>( C = [0-9] )</td>
<td>.111.....1.............111111.....1111...</td>
</tr>
</tbody>
</table>
**MatchStar Class Repetitions with MatchStar**

- \( \text{MatchStar}(M, C) = (((M \wedge C) + C) \oplus C) \vee M \)
- Consider \( M_2 = \text{MatchStar}(M_1, C) \)
- Use addition to scan each marker through the class.

```
input data       a453z--b3z--az--a12949z--ca22z7--
M_1              .1................1........1........1..
C = [0-9]        .111....1........11111.....11.1..
T_0 = M_1 \wedge C   .1................1........1........1..
T_1 = T_0 + C      ....1...1........1........1.....11..
```
### Matching Character Class Repetitions with MatchStar

- \( \text{MatchStar}(M, C) = (((M \land C) + C) \oplus C) \lor M \)
- Consider \( M_2 = \text{MatchStar}(M_1, C) \)
- Use addition to scan each marker through the class.
- Bits that change represent matches.

**Input Data**

<table>
<thead>
<tr>
<th>( M_1 )</th>
<th>a453z--b3z--az--a12949z--ca22z7--</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C = [0-9] )</td>
<td>.1.............1...1.............1.....</td>
</tr>
<tr>
<td>( T_0 = M_1 \land C )</td>
<td>.1.............1........11111........11.1..</td>
</tr>
<tr>
<td>( T_1 = T_0 + C )</td>
<td>....1...1.............1......11...</td>
</tr>
<tr>
<td>( T_2 = T_1 \oplus C )</td>
<td>.1111.............111111........111...</td>
</tr>
</tbody>
</table>
Matching Character Class Repetitions with MatchStar

- \text{MatchStar}(M, C) = (((M \land C) + C) \oplus C) \lor M

- Consider \(M_2 = \text{MatchStar}(M_1, C)\)

- Use addition to scan each marker through the class.
- Bits that change represent matches.
- We also have matches at start positions in \(M_1\).

---

<table>
<thead>
<tr>
<th>input data</th>
<th>a453z--b3z--az--a12949z--ca22z7--</th>
</tr>
</thead>
<tbody>
<tr>
<td>(M_1)</td>
<td>.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1</td>
</tr>
<tr>
<td>(C = [0-9])</td>
<td>.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1</td>
</tr>
<tr>
<td>(T_0 = M_1 \land C)</td>
<td>.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1</td>
</tr>
<tr>
<td>(T_1 = T_0 + C)</td>
<td>.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1</td>
</tr>
<tr>
<td>(T_2 = T_1 \oplus C)</td>
<td>.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1</td>
</tr>
<tr>
<td>(M_2 = T_2 \lor M_1)</td>
<td>.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1</td>
</tr>
</tbody>
</table>
The rules for bitwise data parallel regular expression matching can be summarized by these equations.

\[
\begin{align*}
\text{Match}(m, C) &= \text{Advance}(\text{CharClass}(C) \land m) \\
\text{Match}(m, RS) &= \text{Match}(\text{Match}(m, R), S) \\
\text{Match}(m, R|S) &= \text{Match}(m, R) \lor \text{Match}(m, S) \\
\text{Match}(m, C*) &= \text{MatchStar}(m, \text{CharClass}(C)) \\
\text{Match}(m, R*) &= m \lor \text{Match}(\text{Match}(m, R), R*) \\
\text{Advance}(m) &= m + m \\
\text{MatchStar}(m, C) &= (((m \land C) + C') \oplus C') \lor m
\end{align*}
\]

The recursive equation is implemented with a while loop.
Compilation Architecture

1) RegEx AST
   - RegEx Parser
   - RegEx Transformations
   - RegEx Compiler

2) Parabix
   - Parabix Transformations
   - Parabix Compiler

3) LLVM
   - LLVM Compiler

Dynamically-Generated Match Function
Execution Architecture

Streaming Input Data

- Transposition

Required Streams Generator

- Dynamic Matcher
- Match Scanner

Named Property Library

Streaming Output Result