Profiling & Optimization

- Introduction
- Analysis/Design
- Build
- Tests
- Debug
- Profiling
- Project management
- Documentation
- Versioning system
- IDE
- GForge
- Conclusion
Outline

• Profiling

• Tools

• Optimization
Profiling

No optimization without profiling

Not everyone has a P4 @ 3Ghz

“We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.” Donald Knuth
Profiling: When?

• To choose among several algorithms for a given problem

• To check the awaited behavior at runtime

• To find the parts of the code to be optimized
Profiling: How?

- On fully implemented (and also tested) code
- On a “release” version (optimized by the compiler, …)
- On representative data
- The optimization cycle:

```
Start -> Measurement -> Locate Bottleneck -> Modify Code

Yes -> Improvement Satisfying?

No -> Check for Improvement (Runtime)

Finished
```
What do we measure?

• Understand what’s going on:
  • OS: scheduling, memory management, hard drives, network
  • Compiler: optimization
  • CPU architecture, chipset, memory
  • Libraries used

• If an application is limited by its I/O, useless to improve the calculation part.

• Here we’ll limit ourselves to CPU & memory performance.
Measurement methods

- Manual
- Source instrumentation
- Statistical measure (sampling)
- Simulation
- Hardware counters
Outline

• Profiling

• Tools

• Optimization
## Tools

<table>
<thead>
<tr>
<th>Tools</th>
<th>Manual</th>
<th>Instr.</th>
<th>Sampling</th>
<th>Simulation</th>
<th>Hardware Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>▪ system timers</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>▪ gprof / gcc –pg</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>▪ valgrind(callgrind)/kcachegrind</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>IBM Rational quantify</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Oprofile</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Vtune</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>PAPI (Performance API)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>JVMPI (Java Virtual Machine Profiler)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>▪ runhprof,</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>▪ Eclipse TPTP (Test&amp;Performance)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>▪ OptimizeIT, JProbe, JMP (Memory Profiler)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shark (MacOS X)</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
# Tools

<table>
<thead>
<tr>
<th>Tools</th>
<th>Manual</th>
<th>Instr.</th>
<th>Sampling</th>
<th>Simulation</th>
<th>Hardware Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>system timers</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gprof / gcc -pg</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>valgrind(callgrind)/kcachegrind</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IBM Rational quantify</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Oprofile</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Vtune</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>PAPI (Performance API)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>JVMPI (Java Virtual Machine Profiler)</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>• runhprof,</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Eclipse TPTP (Test&amp;Performance)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• OptimizeIT, JProbe, JMP (Memory Profiler)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shark (MacOS X)</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Using the tools: system timers

• You have to know the timer’s resolution

• Windows:
  • QueryPerformanceCounter()/QueryPerformanceFrequency()

• Linux/Unix:
  • gettimeofday()
  • clock()

• Java:
  • System.currentTimeMillis()
  • System.currentTimeNanos()

• Intel CPU counter: RDTSC (ReaD Time Stamp Counter)
## Tools

<table>
<thead>
<tr>
<th>Tool</th>
<th>Manual</th>
<th>Instr.</th>
<th>Sampling</th>
<th>Simulation</th>
<th>Hardware Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>system timers</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gprof / gcc -pg</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>valgrind(callgrind)/kcachegrind</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>IBM Rational quantify</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Oprofile</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Vtune</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>PAPI (Performance API)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>JVMPI (Java Virtual Machine Profiler)</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>• runhprof,</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Eclipse TPTP (Test&amp;Performance)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• OptimizeIT, JProbe, JMP (Memory Profiler)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shark (MacOS X)</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
Using the tools: gprof

gprof (compiler generated instrumentation):
- instrumentation: count the function calls
- temporal sampling
- compile with `gcc -pg`
- create `gmon.out` file at runtime

Drawbacks
- line information not precise
- needs a complete recompilation
- results not always easy to analyze for large software
Using the tools : gprof
Using the tools: gprof
<table>
<thead>
<tr>
<th>Tools</th>
<th>Manual</th>
<th>Instr.</th>
<th>Sampling</th>
<th>Simulation</th>
<th>Hardware Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>system timers</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gprof / gcc -pg</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td><strong>valgrind(callgrind)/kcachegrind</strong></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>IBM Rational quantify</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Oprofile</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Intel Vtune</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>PAPI (Performance API)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td><strong>JVMPI (Java Virtual Machine Profiler)</strong></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>• runhprof,</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Eclipse TPTP (Test&amp;Performance)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• OptimizeIT, JProbe, JMP (Memory Profiler)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shark (MacOS X)</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Using the tools: callgrind

callgrind/kcacheGrind: [http://kcacheGrind.sf.net/cgi-bin/show.cgi](http://kcacheGrind.sf.net/cgi-bin/show.cgi)
  • cache simulator on top of valgrind
  • CPU simulation: estimate CPU cycles for each line of code
  • analyze the data more easily with kcacheGrind

Drawbacks
  • time estimates can be inaccurate
  • measure only the user part of the code
  • analyzed software is 20-100 times slower, uses huge amount of memory.
Using the tools : callgrind

callgrind/kcachegrind usage :

- on already compiled software :
  valgrind --tool=callgrind prog
- generates callgrind.out.xxx
- analyzed with callgrind_annotate or kcachegrind
- to be usable in spite of its slowness :
  do not simulate cache usage : --simulate-cache=no
  start instrumentation only when needed :
  --instr-atstart=no / callgrind_control -i on
Using the tools: callgrind
Using the tools: massif

Massif (heap profiler): [http://valgrind.org/info/tools.html#massif](http://valgrind.org/info/tools.html#massif)

- Another Valgrind tool to be used on compiled software:
  - `valgrind --tool=massif prog`
- Generates `massif.xxx.ps`: memory usage vs. time
- Generates `massif.xxx.txt`: which part of code uses what
Using the tools: massif
## Tools

<table>
<thead>
<tr>
<th>Tool</th>
<th>Manual</th>
<th>Instr.</th>
<th>Sampling</th>
<th>Simulation</th>
<th>Hardware Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>system timers</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gprof / gcc -pg</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>valgrind(callgrind)/kcachegrind</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>IBM Rational quantify</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Oprofile</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Vtune</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>PAPI (Performance API)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td><strong>JVMPI (JVM Profiler)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• runhprof,</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Eclipse TPTP (Test&amp;Performance)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shark (MacOS X)</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
Using the tools: runhprof

### java/runhprof

- SUN’s JVM extension
  - `java -Xrunhprof:cpu=samples,depth=6,thread=y prog`
- generates `java.hprof.txt` file
- analyzed with `PerfAnal.jar java.hprof.txt`
- memory:
  - `java -Xrunhprof:heap=all prog`

### Drawbacks

- coarse sampling
- does not use all the possibilities of JVMPI
Using the tools: runhprof
Using the tools:
Eclipse Test & Performance Tools Platform

TPTP: tools integrated into Eclipse
Supports only Java (for the moment)
Profiling of local or distributed software
Using the tools:
Eclipse Test & Performance Tools Platform
Using the tools:
Eclipse Test & Performance Tools Platform
Outline

• Profiling

• Tools

• Optimization
Optimization

- No premature optimization
- Keep the code maintainable
- Do not over-optimize
Optimization (continued)

- Find a better algorithm
  - the constant factor of the complexity can be significant
- Memory access: first cause of slowness
- Use already optimized libraries
- Limit the number of calls to expensive functions
- Write performance benchmarks/tests
  - allows one to check that the performance has not degraded
- The bottleneck moves at each optimization step
  - example: I/O can become blocking
Optimization example

- **Optimization example: image inversion (5000x5000)**
  ```c
  for (int x = 0; x < w; x++)
    for (int y = 0; y < h; y++)
      data[y][x] = 255 - data[y][x];
  ```

- Without compiler optimization: 435 ms
- Compiler optimization (-O3): 316 ms
- Improve memory access locality: 107 ms
- Suppress double dereference on `data`: 94 ms
- Constant code outside of the loop: 63 ms (~7x)
- OpenMP parallelization: 38 ms
- Using MMX assembly: 26 ms
Conclusion

• No premature optimization

• Know your profiling tool

• Keep the code maintainable

• Do not over-optimize
Questions ?