Performance Optimization: Simulation and Real Measurement

Josef Weidendorfer

KDE Developer Conference 2004
Ludwigsburg, Germany
Agenda

- Introduction
- Performance Analysis
- Profiling Tools: Examples & Demo
- KCachegrind: Visualizing Results
- What’s to come …
Introduction

• Why Performance Analysis in KDE?
  – Key to useful Optimizations
  – Responsive Applications required for Acceptance
  – Not everybody owns a P4 @ 3 GHz

• About Me
  – Supporter of KDE since Beginning (“KAbalone”)
  – Currently at TU Munich, working on Cache Optimization for Numerical Code & Tools
Agenda

• Introduction
• Performance Analysis
  – Basics, Terms and Methods
  – Hardware Support
• Profiling Tools: Examples & Demo
• KCachegrind: Visualizing Results
• What’s to come …
Performance Analysis

• Why to use…
  – Locate Code Regions for Optimizations (Calls to time-intensive Library-Functions)
  – Check for Assumptions on Runtime Behavior (same Paint-Operation multiple times?)
  – Best Algorithm from Alternatives for a given Problem
  – Get Knowledge about unknown Code (includes used Libraries like KDE-Libs/QT)
Performance Analysis (Cont’d)

• How to do…
  • At End of (fully tested) Implementation
  • On Compiler-Optimized Release Version
  • With typical/representative Input Data
  • Steps of Optimization Cycle

Start → Measurement → Locate Bottleneck → Modify Code

Yes
Finshed → Improvement Satisfying?

No
Check for Improvement (Runtime) → Finished
Performance Analysis (Cont’d)

- Performance Bottlenecks (sequential)
  - Logical Errors: Too often called Functions
  - Algorithms with bad Complexity or Implementation
  - Bad Memory Access Behavior (Bad Layout, Low Locality)
  - Lots of (conditional) Jumps, Lots of (unnecessary) Data Dependencies, ...

Too low-level for GUI Applications?
Performance Measurement

• Wanted:
  – Time Partitioning with
    • Reason for Performance Loss (Stall because of…)
    • Detailed Relation to Source (Code, Data Structure)
  – Runtime Numbers
    • Call Relationships, Call Numbers
    • Loop Iterations, Jump Counts
  – No Perturbation of Results b/o Measurement
Measurement - Terms

• Trace: Stream of Time-Stamped Events
  • Enter/Leave of Code Region, Actions, …
    Example: Dynamic Call Tree

• Huge Amount of Data (Linear to Runtime)
• Unneeded for Sequential Analysis (?)
Measurement – Terms (Cont’d)

- Profiling (e.g. Time Partitioning)
  - Summary over Execution
    - Exclusive, Inclusive
    - Cost / Time, Counters
    - Example: DCT → DCG (Dynamic Call Graph)
  - Amount of Data
    - Linear to Code Size

![Diagram showing a dynamic call graph with nodes A, B, C, D, and arrows indicating calls and counts. The graph shows nodes with numbers indicating the counts for exclusive and inclusive measurements. The diagram illustrates the concept of profiling with example graphs and terms related to measurement.]
Methods

- Precise Measurements
  - Increment Counter (Array) on Event
  - Attribute Counters to
    - Code / Data
- Data Reduction Possibilities
  - Selection (Event Type, Code/Data Range)
  - Online Processing (Compression, ...)
- Needs Instrumentation (Measurement Code)
Methods - Instrumentation

- Manual
- Source Instrumentation
- Library Version with Instrumentation
- Compiler
- Binary Editing
- Runtime Instrumentation / Compiler
- Runtime Injection
Methods (Cont’d)

• Statistical Measurement (“Sampling”)
  – TBS (Time Based), EBS (Event Based)
  – Assumption: Event Distribution over Code Approximated by checking every N-th Event
  – Similar Way for Iterative Code: Measure only every N-th Iteration

• Data Reduction Tunable
  – Compromise between Quality/Overhead
Methods (Cont’d)

• Simulation
  – Events for (not existant) HW Models
  – Results not influenced by Measurement
  – Compromise Quality / Slowdown
    • Rough Model = High Discrepancy to Reality
    • Detailed Model = Best Match to Reality
      But: Reality (CPU) often unknown…
  – Allows for Architecture Parameter Studies
Hardware Support

- Monitor Hardware
  - Event Sensors (in CPU, on Board)
  - Event Processing / Collection / Storing
    - Best: Separate HW
    - Comprmise: Use Same Resources after Data Reduction
  - Most CPUs nowadays include Performance Counters
Performance Counters

• Multiple Event Sensors
  – ALU Utilization, Branch Prediction, Cache Events (L1/L2/TLB), Bus Utilization

• Processing Hardware
  – Counter Registers
    • Itanium2: 4, Pentium-4: 18, Opteron: 8
      Athlon: 4, Pentium-II/III/M: 2, Alpha 21164: 3
Performance Counters (Cont’d)

• Two Uses:
  – Read
    • Get Precise Count of Events in Code Regions by Enter/Leave Instrumentation
  – Interrupt on Overflow
    • Allows Statistical Sampling
    • Handler Gets Process State & Restarts Counter

• Both can have Overhead
• Often Difficult to Understand
Agenda

• Introduction
• Performance Analysis
• Profiling Tools: Examples & Demo
  – Callgrind/Calltree
  – OProfile
• KCachegrind: Visualizing Results
• What’s to come …
Tools - Measurement

• Read Hardware Performance Counters
  – Specific: PerfCtr (x86), Pfmon (Itanium), perfex (SGI)
    Portable: PAPI, PCL

• Statistical Sampling
  – PAPI, Pfmon (Itanium), OProfile (Linux),
    VTune (commercial - Intel), Prof/GProf (TBS)

• Instrumentation
  – GProf, Pixie (HP/SGI), VTune (Intel)
  – DynaProf (Using DynInst), Valgrind (x86 Simulation)
Tools – Example 1

• **GProf (Compiler generated Instr.):**
  - Function Entries increment Call Counter for (caller, called)-Tupel
  - Combined with Time Based Sampling
  - Compile with “gcc –pg ...”
  - Run creates “gmon.out”
  - Analyse with “gprof ...”
  - Overhead still around 100%

• **Available with GCC on UNIX**
Tools – Example 2

• Callgrind/Calltree (Linux/x86), GPL
  – Cache Simulator using Valgrind
  – Builds up Dynamic Call Graph
  – Comfortable Runtime Instrumentation
  – http://kcachegrind.sf.net

• Disadvantages
  – Time Estimation Inaccurate
    (No Simulation of modern CPU Characteristics!)
  – Only User-Level
Tools – Example 2 (Cont’d)

- Callgrind/Calltree (Linux/x86), GPL
  - Run with “callgrind prog”
  - Generates “callgrind.out.xxx”
  - Results with “callgrind_annotate” or “kcachegrind”
  - Cope with Slowness of Simulation:
    - Switch of Cache Simulation: --simulate-cache=no
    - Use “Fast Forward”:
      --instr-atstart=no / callgrind_control –i on

- DEMO: KHTML Rendering…
Tools – Example 3

• OProfile
  – Configure (as Root: oprof_start, ~/.oprofile/daemonrc)
  – Start the OProfile daemon (opcontrol -s)
  – Run your code
  – Flush Measurement, Stop daemon (opcontrol –d/-h)
  – Use tools to analyze the profiling data
    `oproreport`: Breakdown of CPU time by procedures
    (better: oproreport –gdf | op2calltree)

• DEMO: KHTML Rendering…
Agenda

• Introduction
• Performance Analysis
• Profiling Tools: Examples & Demo
  • KCachegrind: Visualizing Results
    – Data Model, GUI Elements, Basic Usage
    – DEMO
• What’s to come …
KCacheGrind – Data Model

- Hierarchy of Cost Items (=Code Relations)
  - Profile Measurement Data
  - Profile Data Dumps
  - Function Groups:
    - Source files, Shared Libs, C++ classes
  - Functions
  - Source Lines
  - Assembler Instructions
KCachegrind – GUI Elements

• List of Functions / Function Groups
• Visualizations for an Activated Function

• DEMO
Agenda

- Introduction
- Performance Analysis
- Profiling Tools: Examples & Demo
- KCachegrind: Visualizing Results
- What’s to come …
  - Callgrind
  - KCachegrind
What’s to come

• Callgrind
  – Free definable User Costs
    (“MyCost += arg1” on Entering MyFunc)
  – Relation of Events to Data Objects/Structures
  – More Optional Simulation (TLB, HW Prefetch)
What’s to come (Cont’d)

• KCachegrind
  – Supplement Sampling Data with Inclusive Cost via Call-Graph from Simulation
  – Comparation of Measurements
  – Plugins for
    • Interactive Control of Profiling Tools
    • Visualizations

• Visualizations for Data Relation
Finally…

THANKS FOR LISTENING