Cache Performance Analysis with Callgrind and KCachegrind

Parallel Performance Analysis Course, 31 October, 2010
King Abdullah University of Science and Technology, Saudi Arabia

Josef Weidendorfer

Computer Architecture I-10, Department of Informatics
Technische Universität München, Germany
Outline

• Background

• Callgrind and \{Q,K\}Cachegrind
  – Measurement
  – Visualization

• Demo
  – Example: Matrix Multiplication
This Talk is about Sequential Performance

Sequential vs. parallel performance

- conceptually orthogonal: performance improvement of sequential code parts always helps, but
- better optimized sequential code sometimes more difficult to parallelize
- parallel code: resources shared among threads/processes
  - on multicore: higher bandwidth requirement to main memory
  - use of shared caches: cores compete for space vs. cores prefetch for each other
Background

- sequential performance bottlenecks
  - logical errors (unneeded/redundant function calls)
  - bad algorithm (high complexity or huge “constant factor”)
  - bad exploitation of available resources

- how to improve sequential performance
  - use tuned libraries where available
  - check for above obstacles ➔ always by use of analysis tools
Sequential Performance Analysis Tools

• count occurrences of events
  – resource exploitation is related to events
  – SW-related: function call, OS scheduling, ...
  – HW-related: FLOP executed, memory access, cache miss, time spent for an activity (like running an instruction)

• relate events to source code
  – find code regions where most time is spent
  – check for improvement after changes
  – „Profile data“: histogram of events happening at given code positions
  – inclusive vs. exclusive cost
How to measure Events (1)

• target
  – real hardware
    • needs sensors for interesting events
    • for low overhead: hardware support for event counting
    • difficult to understand because of unknown micro-architecture, overlapping and asynchronous execution
  – machine model
    • events generated by a simulation of a (simplified) hardware model
    • no measurement overhead: allows for sophisticated online processing
    • simple models relatively easy to understand

• both methods (real vs. model) has advantages & disadvantages, but reality matters in the end
How to measure Events (2)

- SW-related
  - instrumentation (= insertion of measurement code)
    - into OS / application, manual/automatic, on source/binary level
    - on real HW: always incurs overhead which is difficult to estimate

- HW-related
  - read Hardware Performance Counters
    - gives exact event counts for code ranges
    - needs instrumentation
  - statistical: Sampling
    - event distribution over code approximated by checking every N-th event
    - hardware notifies only about every N-th event  ➔ Influence tunable by N
Architectural Performance Problem Today: Main Memory

- access latency ~ 200 cycles
  - 400 FLOP wasted for one main memory access
  - Solution:
    - Memory controller on chip
    - Exploit fast caches (Locality of accesses!)
    - Prefetch data (automatically)

- bandwidth available for one chip ~ 3 – 30 GB/s
  - all cores have to share the bandwidth
  - can prevent effective prefetching
  - solution:
    - Share data in caches among cores
    - Keep working set in cache (temporal locality!)
    - use good data layout (spatial locality!)
Typical Cache Optimizations: Reordering Accesses

- **Blocking**
  - Array time
  - Array time

- **Interweaving**
  - Array time
  - Array time

- Also in multiple dimensions
- Data dependencies of algorithm have to be maintained
Callgrind

Cache Simulation with Call-Graph Relation
Callgrind: Basic Features

- based on Valgrind
  - runtime instrumentation infrastructure (no recompilation needed)
  - dynamic binary translation of user-level processes
  - Linux/AIX/OS X on x86, x86-64, PPC32/64, ARM (VG 3.6: ~10/2010)
  - correctness checking & profiling tools on top
    - “memcheck”: accessibility/validity of memory accesses
    - “helgrind” / ”drd”: race detection on multithreaded code
    - “cachegrind”/”callgrind”: cache & branch prediction simulation
    - “massif”: memory profiling
  - Open source (GPL)
  - www.valgrind.org
Callgrind: Basic Features

• part of Valgrind since 3.1
  – Open Source, GPL

• measurement
  – profiling via machine simulation (simple cache model)
  – instruments memory accesses to feed cache simulator
  – hook into call/return instructions, thread switches, signal handlers
  – instruments (conditional) jumps for CFG inside of functions

• presentation of results: callgrind_annotate / {Q,K}Cacheegrind
Pro & Contra (i.e. Simulation vs. Real Measurement)

• usage of Valgrind
  – driven only by user-level instructions of one process
  – slowdown (call-graph tracing: 15-20x, + cache simulation: 40-60x)
    • “fast-forward mode”: 2-3x
  ✓ allows detailed (mostly reproducible) observation
  ✓ does not need root access / can not crash machine

• cache model
  – “not reality”: synchronous 2-level inclusive cache hierarchy
    (size/associativity taken from real machine, always including LLC)
  ✓ easy to understand / reconstruct for user
  ✓ reproducible results independent on real machine load
  ✓ derived optimizations applicable for most architectures
Callgrind: Advanced Features

- interactive control (backtrace, dump command, …)
- “fast forward”-mode to get to quickly interesting code phases
- application control via “client requests” (start/stop, dump)

- avoidance of recursive function call cycles
  - cycles are bad for analysis (inclusive costs not applicable)
  - add dynamic context into function names (call chain/recursion depth)

- best-case simulation of simple stream prefetcher
- usage of cache lines before eviction
- optional branch prediction
Callgrind: Usage

- valgrind -tool=callgrind [callgrind options] yourprogram args
- cache simulator: `--simulate-cache=yes`
- branch prediction simulation (VG 3.6): `--simulate-cache=yes`
- enable for machine code annotation: `--dump-instr=yes`
- start in “fast-forward”: `--instr-atstart=yes`
  - switch on event collection: `callgrind_control -i on`
- spontaneous dump: `callgrind_control -d [dump identification]`
- current backtrace of threads (interactive): `callgrind_control -b`
- separate dumps per thread: `--separate-threads=yes`
- jump-tracing in functions (CFG): `--collect-jumps=yes`
{Q,K}Cachegrind

Graphical Browser for Profile Visualization
Features

• open source, GPL
• kcachegrind.sf.net (release of pure Qt version pending)
• included with KDE3 & KDE4

• visualization of
  – call relationship of functions (callers, callees, call graph)
  – exclusive/Inclusive cost metrics of functions
    • grouping according to ELF object / source file / C++ class
  – source/assembly annotation: costs + CFG
  – arbitrary events counts + specification of derived events

• callgrind support (file format, events of cache model)
Usage

• **kcachegrind callgrind.out.<pid>**

• **left: “Dockables”**
  - list of function groups
    - groups according to
      - library (ELF object)
      - source
      - class (C++)
    - list of functions with
      - inclusive
      - exclusive costs

• **right: visualization panes**
Visualization panes for selected function

- List of event types
- List of callers/callees
- Treemap visualization
- Call Graph
- Source annotation
- Assembly annotation
Upcoming ...

• callgrind
  – event relation to data structures
  – command line tool for measurement merging & results
  – multicore cache simulation (detection of data sharing)

• KCachegrind
  – pure Qt version (Windows/OS X)
  – call graph context visualization
Weidendorfer: Callgrind / KCachegrind
Demo
Getting started

• Try it out yourself on Live-DVD
  – see /usr/local/packages/kcacheegrind/README

• Test: What happens in „/bin/ls“?
  – valgrind --tool=callgrind ls /usr/bin
  – qcachegrind
  – What function takes most instruction executions? Purpose?
  – Where is the main function?
Detailed analysis of matrix multiplication

• Kernel for $C = A \times B$
  – Side length $N \Rightarrow N^3$ multiplications + $N^3$ additions

\[
C_{i,j} = \sum_{k=1}^{N} A_{i,k} \times B_{k,j}
\]

– 3 nested loops $(i,j,k)$: Best index order?
– Optimization for large matrixes: Blocking
Detailed analysis of matrix multiplication

- To try out...
  - `cp -r /usr/local/packages/kcacheegrind/example-mm`
  - `make CFLAGS=’-O2 -g’`
  - Timing of orderings (e.g. size 800): `./mm 800`
  - Cache behavior for small matrix (fitting into cache):
    `valgrind --tool=callgrind --simulate-cache=yes ./mm 300`
  - How good is L1/L2 exploitation of the MM versions?
  - Large matrix (mm800/callgrind.out). How does blocking help?