Cache Performance Analysis with Callgrind and KCachegrind

9th VI-HPS Tuning Workshop
April 2012, Versailles

Josef Weidendorfer

Computer Architecture I-10, Department of Informatics
Technische Universität München, Germany
Focus: Cache Simulation using a Simple Machine Model

Why simulation?
- reproducability
- no influence of tool on results
- allows to collect information not possible with real hardware
- can not crash machine

Focus only on cache / a simple model really enough?
- **no**: if real measurement shows cache issues, use sim. for details
- bad cache exploitation dominates: you can ignore other bottlenecks
- benefits of simple models:
  - easy to understand, still captures most problems, faster simulation…
Outline

• Background

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

• Hands-On
  – Example: Matrix Multiplication
Single Node Performance: Cache Exploitation is Important

- „Memory Wall“

- Access Latencies
  - modern x86 processors: ~200 cycles \( \Rightarrow \) 400 FLOP wasted…
Single Node Performance: Cache Exploitation is Important

This will be true also in the future
- latency of main memory access does not improve
- bandwidth to main memory increases slower than compute power
  - multicore, accelerators
- power consumption  [Keynote Dongarra, PPAM 2011]
Caches do their Job transparently...

Caches work because all programs expose access locality
- temporal (hold recently used data) / spatial (work on blocks of memory)

The “Principle of Locality” is not enough... ➔ “Cache optimization”

![Chart showing reasons for performance loss for SPEC2000](Beyls_Hollander_ICCS_2004)
How to do Cache Optimization on Parallel Code

• Analyse sequential code phases
  – optimization of sequential phases should always improve runtime
  – no need to strip down to sequential program

• Influences of threads/tasks on cache exploitation
  – on multicore: all cores share bandwidth to main memory
  – use of shared caches:
    cores compete for space vs. cores prefetch for each other
  – slowdown because of “false sharing”
  – not easy to get with hardware performance counters
    • research topic (parallel simulation with acceptable slowdown)
Go Sequential (just for a few minutes)...

• 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 make it easier to understand the problem and to think about solution

• both methods (real vs. model) have 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 \( \Rightarrow \) Influence tunable by N
Back to the Memory Wall

- **Solution for**
  - **access latency**
    - exploit fast caches: improve locality of data
    - allow hardware to prefetch data (use access patterns easy to predict)
    - memory controller on chip (standard today)
  - **low bandwidth**
    - share data in caches among cores
    - keep working set in cache (temporal locality)
    - use good data layout (spatial locality)
    - if memory accesses are unavoidable: duplicate data in NUMA nodes

Weidendorfer: Callgrind / KCachegrind
Cache Optimization: Reordering Accesses

- Blocking

- Also in multiple dimensions
- Data dependencies of algorithm have to be maintained
- Multi-core: consecutive iterations on cores with shared cache
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 (since VG 3.6)

  – 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
  - extension of the VG tool cachegrind (dynamic call graph, simulator extensions, more control)

- 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}Cachegrind`
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
Callgrinds Cache Model vs. CURIE

- **Cachegrind**
  - basic parameters adjustable: size, line size, associativity
    (for time estimation in KCachegrind: editable formula for latencies)
  - dedicated 2 levels, all fixed LRU
  - write back vs. write through does not matter for hit/miss counts
  - optional L2 stream prefetcher

- **CURIE**: 360 nodes with 4 sockets using Intel-X7560 (Nehalem-EX, 2.26GHz, 8 cores)
  - inclusive, L1 D/I 32kB, L2 256 kB, L3 shared 24 MB
  - Callgrind only simulates L1 and L3 (= LLC) ➔ L3 hit count too high
Callgrind: Advanced Features

- interactive control (backtrace, dump command, …)
- “fast forward”-mode to quickly get at 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
- byte-wise usage of cache lines before eviction
- branch prediction (since VG 3.6)
- optionally measures time spent in system calls (useful for MPI)
Callgrind: Usage

- valgrind -tool=callgrind [callgrind options] yourprogram args
- cache simulator: --cache-sim=yes
- branch prediction simulation (since VG 3.6): --branch-sim=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
- time in system calls: --collect-systime=yes
- byte-wise usage within cache lines: --cacheuse=yes
{Q,K}Cachegrind

Graphical Browser for Profile Visualization
Features

- open source, GPL
- kcachegrind.sf.net (recent versions includes pure Qt version, able to run on Linux / OS-X / Windows)
- 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 (can load cachegrind data)
Usage

- qcachegrind callgrind.out.<pid>

- left: “Dockables”
  - list of function 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
Expected future additions…

- More abstract metrics / visualizations
  - reuse distance histograms: which accesses need which cache sizes?
  - histogram on spatial cache line use
  - predictability of main memory accesses

- Effects on multicore
  - data sharing among cores
  - frequent invalidations in private L1

Weidendorfer: Callgrind / KCachegrind
Weidendorfer: Callgrind / KCachegrind
Hands-on
Getting started

• Try it out yourself (on CURIE)
  – module add kcachegrind

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

  – Now run with cache simulation: --cache-sim=yes
Detailed analysis of matrix multiplication

• Kernel for \( C = A \times B \)
  - Side length \( N \Rightarrow N^3 \) multiplications + \( N^3 \) additions

\[
\begin{align*}
C & = A \times B \\
C[k][i] & = a[k][j] \times b[j][i]
\end{align*}
\]
  - 3 nested loops \((i,j,k)\): Best index order?
  - Optimization for large matrixes: Blocking
Detailed analysis of matrix multiplication

- To try out...
  - `cp -r /tmp/kcg-example`
  - `make CFLAGS='-O2 -g'`
  - Timing of orderings (e.g. size 512): `.mm 512`
  - Cache behavior for small matrix (fitting into cache):
    `valgrind --tool=callgrind --cache-sim=yes .mm 300`
  - How good is L1/L2 exploitation of the MM versions?
  - Large matrix (800, pregenerated callgrind.out).
    How does blocking help?
How to run with MPI

• On CURIE
  
  module add kcachegrind  
  export OMP_NUM_THREADS=4  
  mpiexec -n 4 valgrind --tool=callgrind --cache-sim=yes \  
  --separate-threads=yes ./bt-mz_B.4

• reduce iterations in BT_MZ
  – sys/setparams.c, write_bt_info, set niter = 5

• load all profile dumps at once:
  – run in new directory, “qcachegrind callgrind.out”