Single Processor Performance Analysis with Callgrind / KCachegrind

1st Parallel Tools Workshop, HLRS, Stuttgart, July 9-10 2007

Josef Weidendorfer

Lehrstuhl für Rechnertechnik und Rechnerorganisation
Institut für Informatik, Technische Universität München

together with Rainer Keller, HLRS

Short Introduction

- Optimizing resource consumption
- Resource consumption related to events

- Measurement methods
  - Exact event counts: needs instrumentation
  - Statistical: Sampling
  - Simulation

Callgrind: Basic Features

- Based on Valgrind
  - Runtime instrumentation infrastructure (no recompilation needed)
  - Binary translation of user-level processes
  - Linux/AIX on x86, x86-64, PPC32, PPC64
  - Debugging/Profiling tools on top (using architecture independent IR)
- Measurement Method
  - Call-Graph Profiling via Simulation
  - Simple Cache Model (synthetic events, derived from Cachegrind)
  - Instrument memory accesses to feed cache simulator
  - Hook into Call/Return instructions, thread switches / signal handlers
    - build up DCG (dynamic call graph) separate for each thread
    - Store events according to call chains
  - Instrument (conditional) jumps for CFG inside of functions

Callgrind: Pro and Contra

- Usage of Valgrind
  - Driven only by user-level instructions of one process
  - Slowdown (Call-graph tracing: 15-20x, + cache simulation: 40-60x)
    - But: “Fast-forward mode”: 2-3x
  - Allows detailed observation (arbitrary metrics like reuse distance)
  - 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)
  - Easy to understand/reconstruct for user
  - Reproducible results independent on real machine load
  - Derived optimizations still applicable for most architectures

Callgrind: Advanced Features

- Interactive Control (backtrace, dump command, …)
- Application control via client requests (start/stop, dump)
- Options for Cycle Avoidance in DCG
  - Call chain coded into symbol name
  - Recursion depth coded into symbol name
- Best case simulation of simple L2 stream prefetcher
- Temporal/Spatial locality metrics for memory accesses

Callgrind: Usage

- `valgrind –tool=callgrind [callgrind options] yourprogram args`
- Cache simulator: `-simulate-cache=yes`
- Jump-tracing in functions (CFG): `--collect-jumps=yes`
- Separate dumps per thread: `--separate-threads=yes`
- Start in “fast-forward”: `--instr-atstart=yes`
  - Switch on event collection: `callgrind_control –i on`
- Current backtrace of threads (interactive): `callgrind_control –b`
- Spontaneous dump: `callgrind_control –d [dump identification]`
- Prefetch simulation: `--simulate-hwpref=yes`
- Temporal/spatial locality metrics: `--cacheuse=yes`

KCachegrind: Features

- 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
- Supported format
  - Currently only ASCII Callgrind format
  - Converters for OProfile, Hprof (JAVA), Phyton/PHP profilers
- Special Callgrind support:
  - Derived event “cycle estimation” (very rough, formula can be edited)
    - Executed instructions + 10 * L1 misses + 100 * L2 misses
  - Interactive dump request

KCachegrind: Usage

- `kcachegrind callgrind.out.<pid>`
- Left: “Dockables”
  - List of functions with inclusive/exclusive costs
  - List of function groups
  - Overview of multiple dumps (“parts”)
- Right: Visualization panes
  - List of event types
  - List of (all) callers/callees
  - Caller/Callee Treemap visualization
  - Call Graph
  - Source/Assemly annotation
- Advanced features
  - Simultaneous visualization of 2 functions: splitted pane
  - Pane layouts (Store/Switch between)
- See demo/tutorial

Future

- Callgrind
  - Advanced metrics: Stack reuse distance (size of working set)
    Memory bandwidth requirement
  - Multicore Cache Simulation
  - Event relation to data structures
  - Automatic context refinement (event difference in iterative func. calls)
  - Command line tool for measurements merging & ASCII results
- Callgrind format
  - Optional integration of source/assembly
- KCachegrind
  - Compare modus inside of one view
  - More import filters (gprof, pfmon, …)
  - Combination of measurement data from different tools
  - Diagrams with time axis (for “dump every second”)
  - Visualization of CFG (with loop detection)

Practical (1)

- Setup your environment:
  - `module load compiler/intel`
  - `module load valgrind`
  - `cd KCACHEGRIND_VALGRIND`
- Simple „test“ run: What happens in „/bin/ls“ ?
  - `valgrind --tool=callgrind ls`
  - `kcachegrind`
  - What function takes most instruction executions?
  - Where is the main function?

Practical (2)

- Interactivity... What happens in GUI code ?
  - **`valgrind --tool=callgrind xclock`**
  - Get backtrace of current execution: **`callgrind_control -b`**
  - Dump profile at custom points: **`... -d [some_description]`**
- What happens in OpenMP code ?
  - Check out **`mandelbrot.c`**, uncomment first OpenMP directive
  - **`icc -g -openmp mandelbrot.c -o mandelbrot`**
  - **`valgrind --tool=callgrind –separate-threads=yes \ ./mandelbrot`**
  - **`kcachegrind [callgrind.out.<pid>]`**
  - Rerun with dynamic scheduling (2nd directive in source!)