# Single Processor Performance

# Performance Analysis with Callgrind / KCachegrind

Multicore-Training, LRZ, Garching, July 23, 2007

Josef Weidendorfer

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

# Informal History

- I got interested in Valgrind around 2003
- Call graph tracing missing
  - seemed very easy to add to existing cache simulator „cachegrind“
- Now used in research, adding advanced features to simulator
- Currently to be extended for multicore

- Used in lab courses
  - no special setup / root access required
  - Simple cache model
    - Easy to understand
    - Results can be checked against model

# Outline

- Part 1: Background
  - Methods for Sequential Performance Analysis
  - Typical Cache Optimizations
- Part 2: Callgrind & KCachegrind
  - Valgrind, Pro & Contra, Features
  - Screenshot Tour
- Part 3: Demo

# Performance Analysis

General Goal:

Optimize for minimal resource consumption of an application

- Time
- Memory

- Locate code regions for Improvements
- Check correctness of assumptions on runtime behavior
- Chose best algorithm from alternatives for a problem
- Get knowledge about unknown code

# Performance Analysis

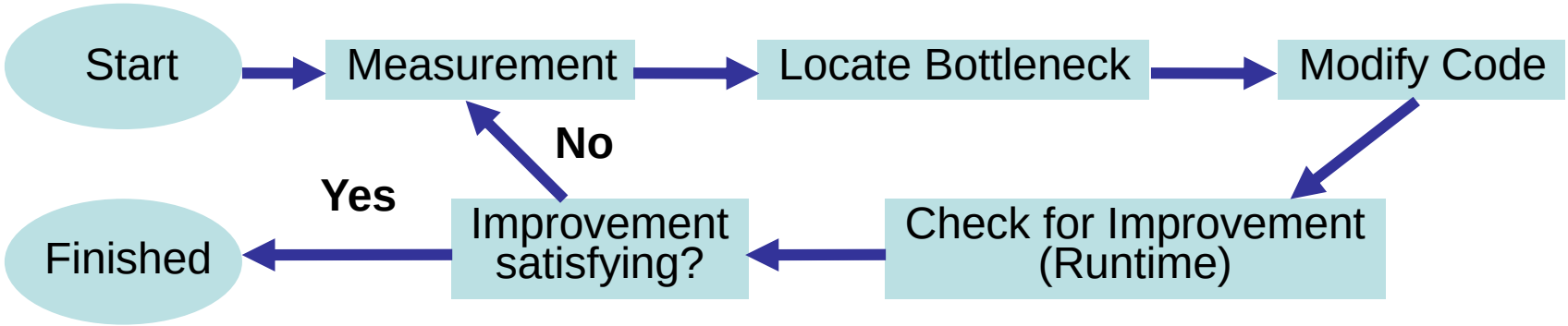
- Bottlenecks in sequential code

- Logical errors, e.g. redundant function calls
- Bad complexity / bad implementation of algorithms
- Bad cache utilization (low locality)
- Unpredictable jumps,  unnecessary data dependencies ...

# Performance Analysis

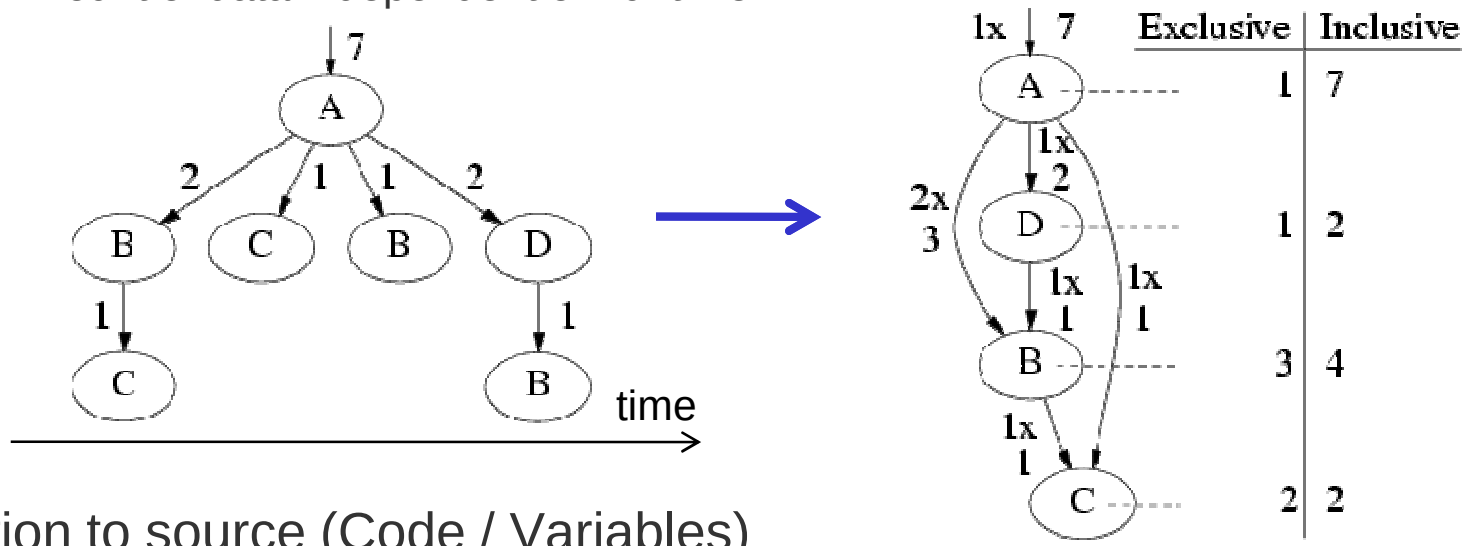
- How
  - At End of (fully tested) Implementation
  - On Compiler-Optimized Release Version
  - With typical / representative Input Data

- Steps of Optimization Cycle



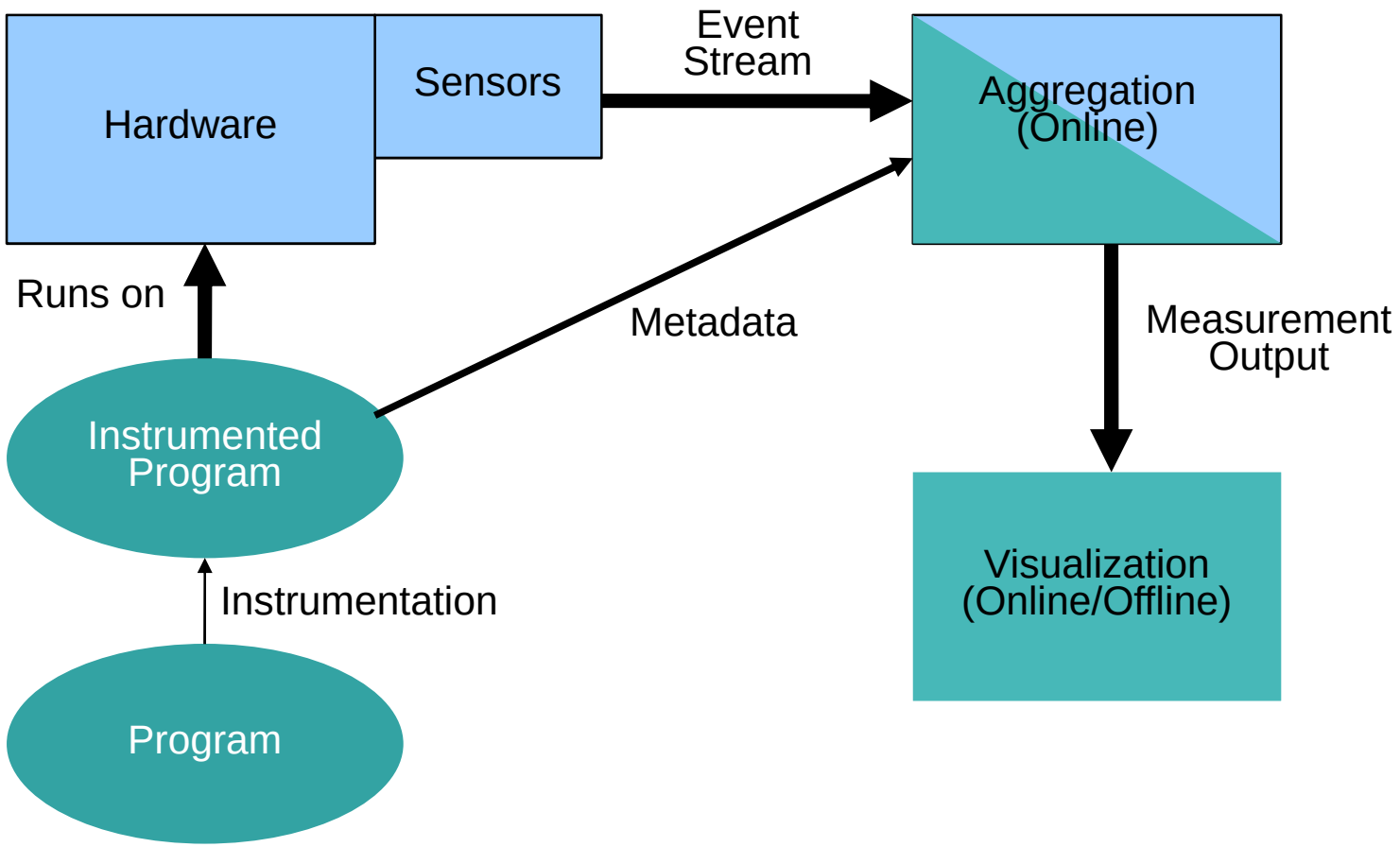
# Performance Measurement

- Observe suitable events
  - Clock ticks, cache misses, function calls, jumps …
  - Aggregation enough for sequential code
    - Event profile instead of event trace, call graph (DCG) instead of call tree
    - Amount of data independent on runtime




- Attribution to source (Code / Variables)

# Measurement Overview



# Measurement methods

- Exact event counts for code ranges
  - Reset/Read hardware performance counters
  - Needs instrumentation (= insertion of measurement code)
  - Overhead fixed, depending on instrumentation
- Statistical: Sampling
  - Hardware notifies only about every N-th event
  - Assumption:
    Event distribution over code approximated
    by checking every N-th event only
  - Overhead tunable by N
- Simulation
  - Observe events generated by a (simplified) hardware model
  - No overhead: allows for sophisticated online processing

# Hardware Performance Counters

- Every Processor now has documented counters
- Most important
  - Clock ticks
  - Instructions retired
  - Last level cache misses
  - Floating Point Operations retired

- Issues
  - Hardware Access Needed (Misconfiguration can crash machine)
  - Reflect reality
    - Provide details about events in undocumented microarchitecture
    - Semantics not easy to understand
  - Limited Resource

# Tools - Measurement

- Read Hardware Performance Counters
  - Specific: PerfCtr (x86), Pfmon (Itanium), perfex (SGI), DCPI (Alpha)
  - Portable: PAPI, PCL
- Statistical Sampling
  - PAPI, DCPI, Pfmon (Itanium), OProfile (Linux), VTune (commercial - Intel), Prof/Gprof
- Instrumentation
  - GProf, Pixie (HP/SGI), Atom (Alpha), VTune (Intel) DynaProf (Using DynInst), Valgrind (x86 – Simulation)

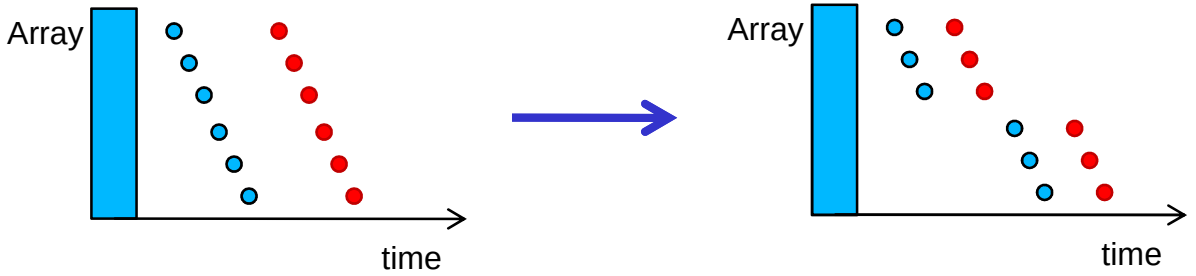
# Typical Cache Optimizations

- Benefit of caches
  - Exploit locality of memory accesses (temporal / spatial)
  - Lowers access latency by putting data copies into fast memory
    - Keep recently used copies (accounts for temporal locality)
    - Block oriented (accounts for spatial locality)

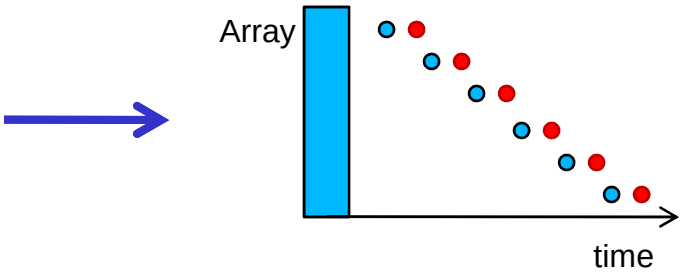
- Optimization strategies
  - Improve temporal locality by reordering accesses
  - Improve spatial locality by changing data layout
  - Prefetch data needed in the future

# Typical Cache Optimizations: Reordering Accesses

- Blocking




- Interweaving




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

# Callgrind & KCachegrind

# 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)
  - Best known for “memcheck”
    (tracks accessability/validity of memory accesses)

  - Open source (GPL)
  - www.valgrind.org

# Callgrind: Basic Features

- Part of Valgrind since 3.1
  - Open Source, GPL

- 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 applicable for most architectures

# Callgrind: Advanced Features

- Interactive Control (backtrace, dump command, …)
- Application control via client requests (start/stop, dump)

- Options for avoidance of function call cycles
  - Cycles are bad for analysis (inclusive costs not applicable)
  - Add context info into function names (call chain/rec. depth)

- Best case simulation of simple L2 stream prefetcher
- Usage of cache lines before eviction
  - Relates to temporal/spatial locality
  - How often used / How many bytes unused

# 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`
- Cacheline usage: `--cacheuse=yes`

# KCachegrind: Features

- Open Source, GPL
- kcachegrind.sf.net
- Packed with KDE (since 3.1) - kdesdk

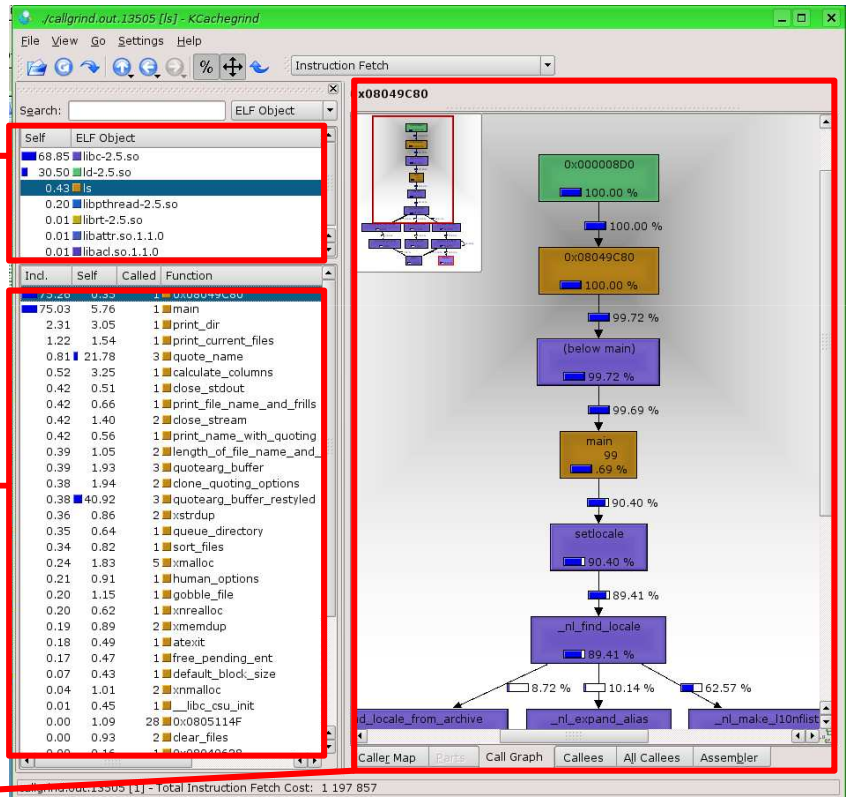
- Visualization of
  - Call relationship of functions (callers, callees, call graph)
  - Exclusive/Inclusive cost metrics of functions
    - Grouping according to ELF object / source file / C++ class
  - Source/Assembly annotation: costs + CFG
  - Arbitrary events counts + specification of derived events

# KCachegrind: Features

- 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

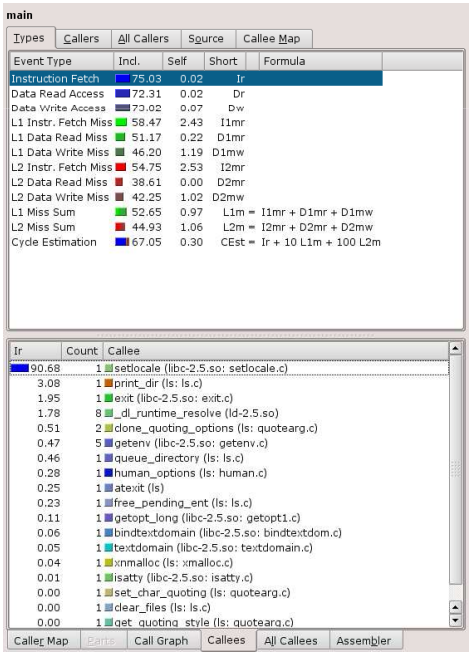
# KCachegrind: Usage

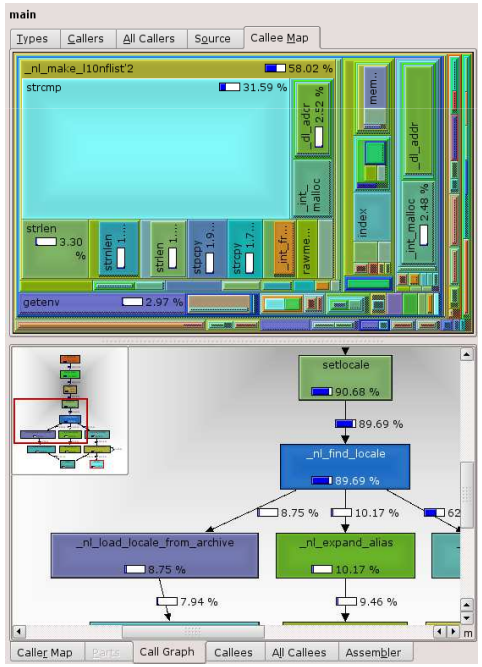
- `kcachegrind callgrind.out.<pid>`
- Left: “Dockables”
  - List of function groups
    Groups according to
    - Library (ELF object)
    - Source
    - Class (C++)
  - List of functions with
    - Inclusive
    - exclusive costs
- Right: Visualization panes

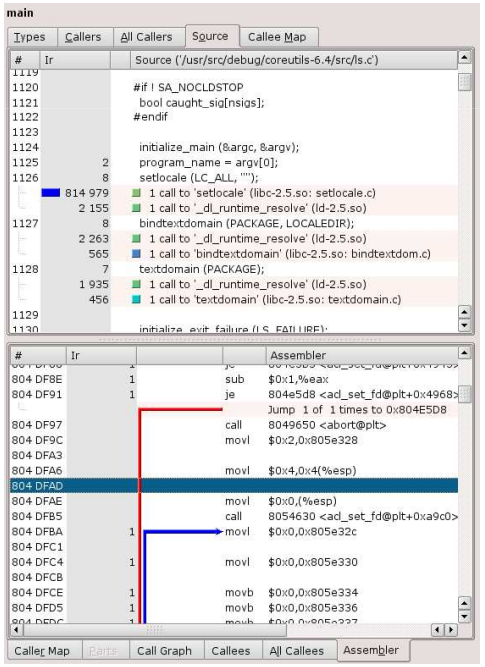


# KCachegrind: Visualization panes for selected function

- List of event types
- List of callers/callees
- Treemap visualization
- Call Graph
- Source annotation
- Assemly annotation



# Demo

# Demo (1)

- Simple „test“ run: What happens in „/bin/ls“ ?
  - **`valgrind --tool=callgrind ls`**
  - **`kcachegrind`**
  - What function takes most instruction executions?
  - Where is the main function?

- 2D blocking for matrix multiplication

# Demo (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!)

# Backup Side: Future Plans

- 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)