

Cache Performance Analysis with Callgrind and KCachegrind

10th VI-HPS Tuning Workshop
October 2012, Garching

Josef Weidendorfer

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

Focus: Cache Simulation using a Simple Machine Model

Why simulation?

- reproducibility
- 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?

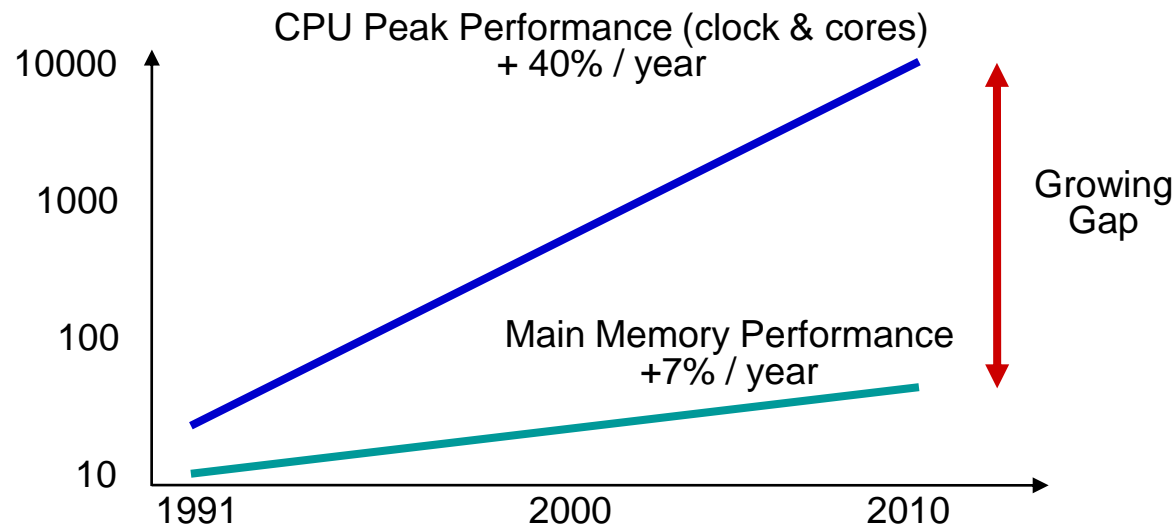
- **no**: if real measurement shows cache issues, use sim. for details
- bad cache exploitation dominates: you can ignore other bottlenecks
- benefits of simple models:
 - easy to understand, still captures most problems, faster simulation...

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 on modern x86 processors ~ 200 cycles
→ 400 FLOP wasted (much more with SSE/AVX)

Single Node Performance: Cache Exploitation is Important

This will be true also in the future

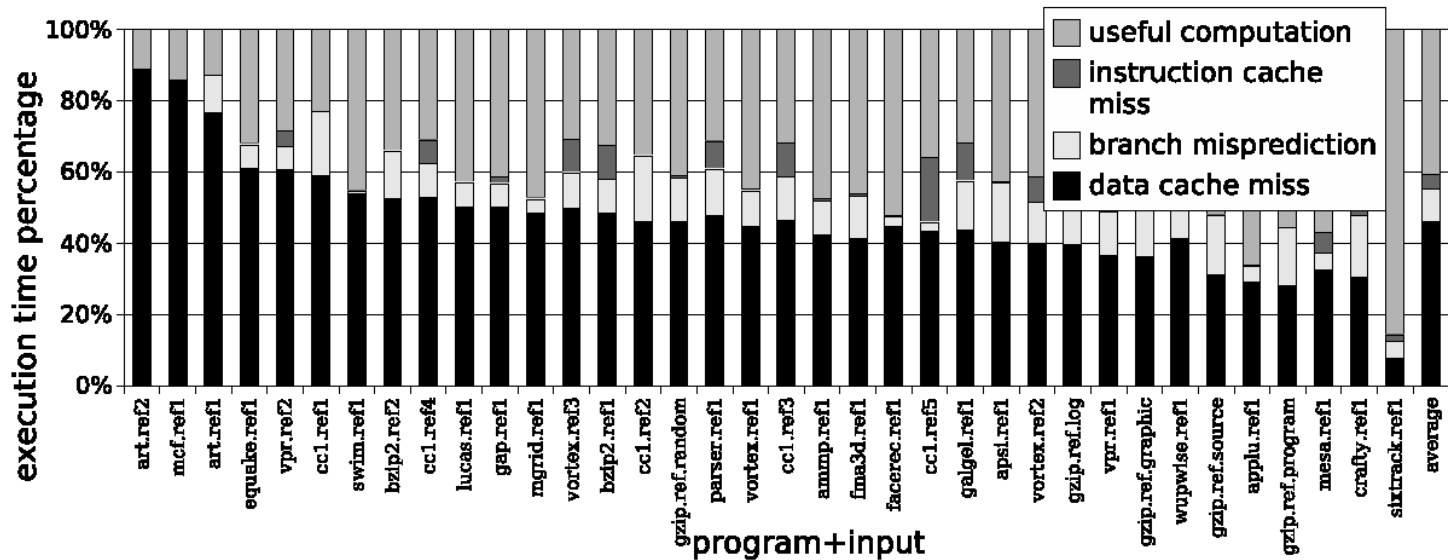
- latency of main memory access does not improve
- bandwidth to main memory increases slower than compute power
 - multicore, accelerators
- power consumption [Keynote Dongarra, PPAM 2011]
 - DP FMADD: 100 pJ (today) → 10 pJ (expected 2018)
 - DP Read DRAM: 4800 pJ (today) → 1920 pJ (expected 2018)

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”



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 multi-core: 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)

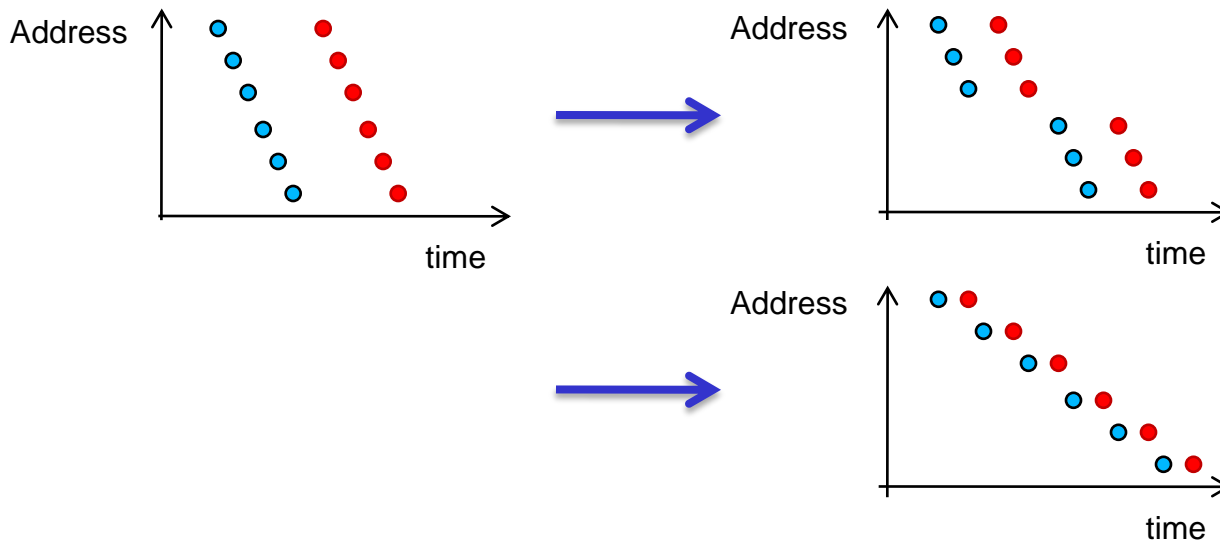
- SW-related
 - instrumentation (= insertion of measurement code)
 - into OS / application, manual/automatic, on source/binary level
 - on real HW: always incurs overhead which is difficult to estimate
- HW-related
 - read Hardware Performance Counters
 - gives exact event counts for code ranges
 - needs instrumentation
 - statistical: Sampling
 - event distribution over code approximated by checking every N-th event
 - hardware notifies only about every N-th event → 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

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 Capturing

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, MIPS
 - 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. SuperMIG/SuperMUC

- Cachegrind
 - basic parameters adjustable: size, line size, associativity
(for time estimation in KCachegrind: editable formula for latencies)
 - L1 / LL (2 levels), inclusive, all fixed LRU, shared among all threads
 - write back vs. write through does not matter for hit/miss counts
 - optional L2 stream prefetcher
- SuperMUC node: 2 sockets, Intel E5-2680 (SandyBridge, 8 core)
SuperMIG node: 4 sockets, Intel E7-4870 (Westmere-EX, 10 core)
 - strictly inclusive, private L1 D/I 32kB, private L2 256 kB,
L3 shared 30 MB (W-EX) / 20 MB (SB)
 - Callgrind only simulates L1 and L3 (= LLC) → LL hit count higher

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:** `--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

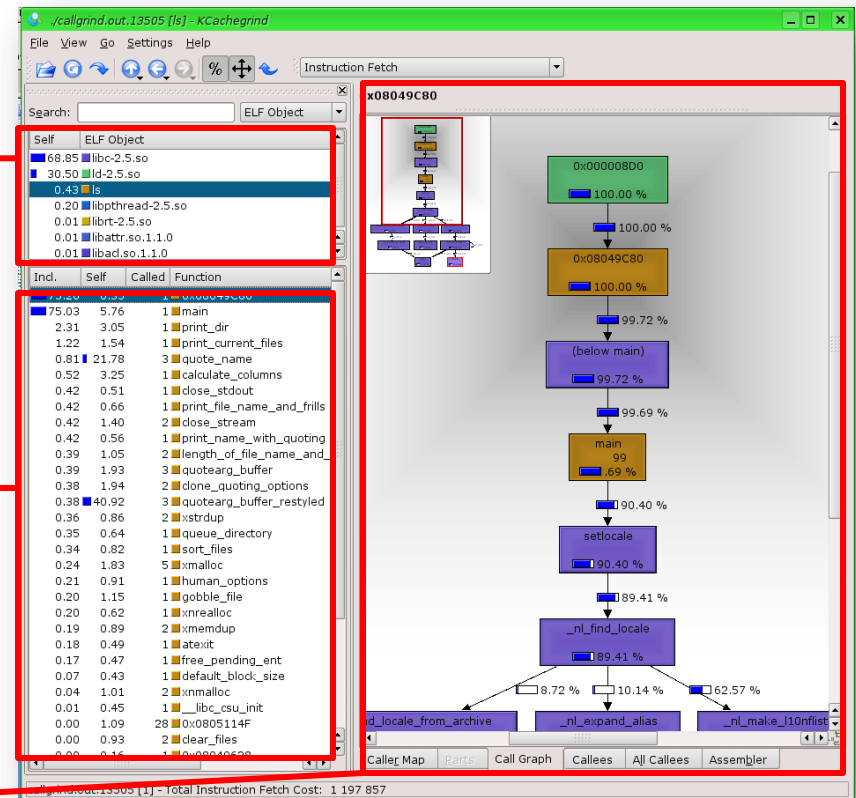
Graphical Browser for Profile Visualization

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 kcachegrind data)

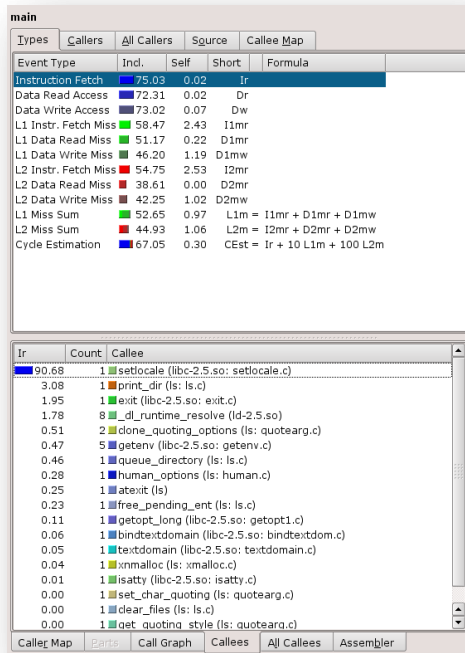
Usage

- `qcachegrind 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

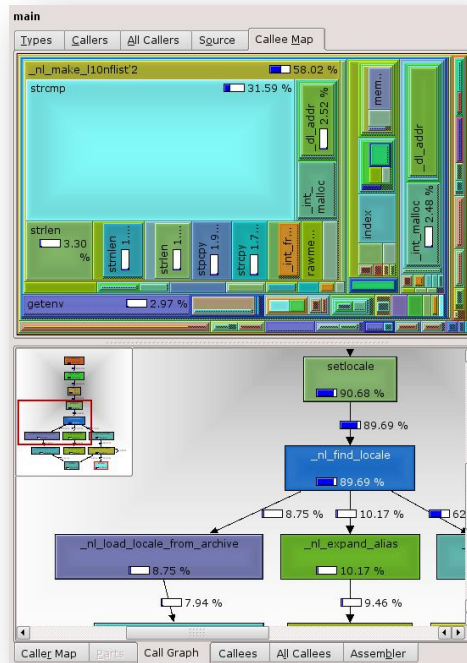
- List of event types
- List of callers/callees
- Treemap visualization
- Call Graph
- Source annotation
- Assembly annotation



The screenshot shows the 'main' window with two panes. The top pane, titled 'Event Type', displays a table of performance events. The bottom pane, titled 'Caller Map', shows a list of callers and their counts.

Event Type	Ind.	Self	Thr	Formula
Instruction Fetch	75.08	0.00	0.00	
Data Read Access	72.31	0.02	Dr	
Data Write Access	73.02	0.07	Dw	
L1 Instr. Fetch Miss	58.47	2.43	L1mr	
L1 Data Read Miss	51.17	0.22	D1mr	
L1 Data Write Miss	46.20	1.19	D1mw	
L2 Instr. Fetch Miss	54.75	2.53	L2mr	
L2 Data Read Miss	38.61	0.00	D2mr	
L2 Data Write Miss	42.25	1.02	D2mw	
L1 Miss Sum	52.65	0.97	L1m = I1mr + D1mr + D1mw	
L2 Miss Sum	44.93	1.06	L2m = I2mr + D2mr + D2mw	
Cycle Estimation	67.05	0.30	CEst = Ir + 10 L1m + 100 L2m	

Ir	Count	Callee
90.68	1	setlocale (libc-2.5.so: setlocale.c)
3.08	1	print_dir (ls: ls.c)
1.95	1	exit (libc-2.5.so: exit.c)
1.78	8	_dl_runtime_resolve (ld-2.5.so)
0.51	2	done_quoting_options (ls: quotearg.c)
0.47	5	getenv (libc-2.5.so: getenv.c)
0.46	1	queue_directory (ls: ls.c)
0.28	1	human_options (ls: human.c)
0.25	1	atexit (ls)
0.23	1	free_pending_ent (ls: ls.c)
0.11	1	getopt_long (libc-2.5.so: getopt1.c)
0.06	1	bindtextdomain (libc-2.5.so: bindtextdom.c)
0.05	1	textdomain (libc-2.5.so: textdomain.c)
0.04	1	xrnmalloc (ls: xrnmalloc.c)
0.01	1	isatty (libc-2.5.so: isatty.c)
0.00	1	set_char_quoting (ls: quotearg.c)
0.00	1	clear_files (ls: ls.c)
0.00	1	get_quotino_style (ls: quotearg.c)



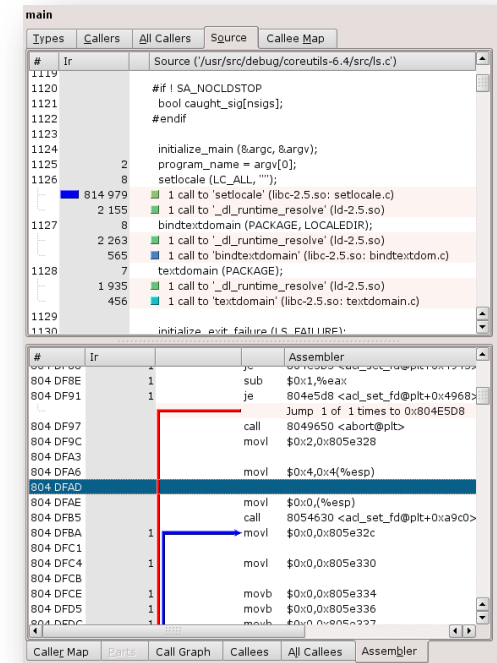
The screenshot shows the 'main' window with two panes. The top pane is a treemap visualization showing the distribution of time across different functions. The bottom pane is a call graph showing the relationships between functions.

Treemap Data:

- setlocale: 58.02%
- strcpy: 31.59%
- strlen: 8.30%
- strncpy: 1.19%
- strncpy: 1.17%
- getenv: 2.97%
- malloc: 2.52%
- malloc: 2.48%
- index: 2.48%
- malloc: 2.48%

Call Graph Data:

- setlocale (90.68%)
 - _nl_find_locale (89.69%)
 - _nl_load_locale_from_archive (8.75%)
 - _nl_expand_alias (10.17%)



The screenshot shows the 'main' window with two panes. The top pane shows source code with annotations. The bottom pane shows assembly code with annotations.

Source Code:

```

1119 Ir Source (/usr/src/debug/coreutils-6.4/src/ls.c)
1120   #if ! SA_NOCLDSTOP
1121   bool caught_sig[nsigs];
1122   #endif
1123
1124   initialize_main (@argc, @argv);
1125   program_name = argv[0];
1126   setlocale (LC_ALL, "");
1127   814 979 1 call to 'setlocale' (libc-2.5.so: setlocale.c)
1128   2 155 1 call to '_dl_runtime_resolve' (ld-2.5.so)
1129   8 263 1 call to 'bindtextdomain' (libc-2.5.so: bindtextdom.c)
1130   565 1 call to '_dl_runtime_resolve' (ld-2.5.so)
1131   7 textdomain (PACKAGE);
1132   1 935 1 call to '_dl_runtime_resolve' (ld-2.5.so)
1133   456 1 call to 'textdomain' (libc-2.5.so: textdomain.c)
1134
1135 initialize_exit_failure (LS_FATALREPLY);
    
```

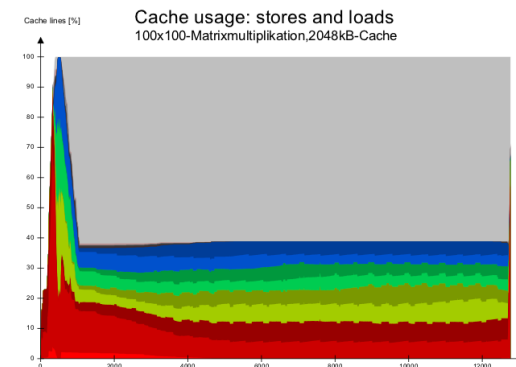
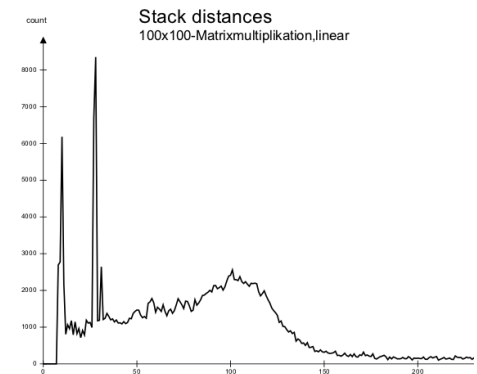
Assembly Code:

```

804 DF8E 1 sub $0x1,%eax
804 DF91 1 je 804+5d8 <ad_set_fd@plt+0x4968>
804 DF97 1 call 8049650 <abort@plt>
804 DF9C 1 movl $0x2,0x805e328
804 DFA3 1 movl $0x4,0x4(%esp)
804 DFA6 1 movl $0x4,0x4(%esp)
804 DFAD 1 movl $0x0,(%esp)
804 DFAE 1 call 8054630 <ad_set_fd@plt+0xa9c0>
804 DFBA 1 movl $0x0,0x805e32c
804 DFC1 1 movl $0x0,0x805e330
804 DFC4 1 movl $0x0,0x805e330
804 DFC8 1 movb $0x0,0x805e334
804 DFCE 1 movb $0x0,0x805e336
804 DFDC 1 movb $0x0,0x805e337
    
```


To come soon ...

- 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



Call

The image displays two side-by-side screenshots of the Callgrind/KCachegrind tool. The top portion of each screenshot shows a call stack table with columns for 'Incl.', 'Self', 'Called', 'Function', and 'Location'. A red box highlights the top five entries in both tables, which are identical:

Incl.	Self	Called	Function	Location
56.63	49.73	41 246	strcoll_l	libc-2.11.2.so: strcoll_l.c, allocallim.h, weight.h
49.29	1.65	3 801	mpsort_with_tmp2	ls: mpsort.c, string3.h
30.37	0.24	1	print_current_files	ls: ls.c, stdio.h
18.37	5.15	11 592	quote_name	ls: ls.c
11.02	3.41	7 728	via length_of_file_name ...	
7.35	1.74	3 864	via print_name_with_qu...	

Below the call stack, the 'Flat Profile' section is visible, showing a search bar and a table with the same data as the call stack. The bottom portion of each screenshot shows a call graph for the 'quote_name' function. The left graph shows a detailed view with various sub-functions and their call counts (e.g., 'length_of_file_name_and_frills' called 3 669 times). The right graph shows the same call graph but with a different color scheme and some nodes highlighted in grey, possibly indicating a different state or filter.

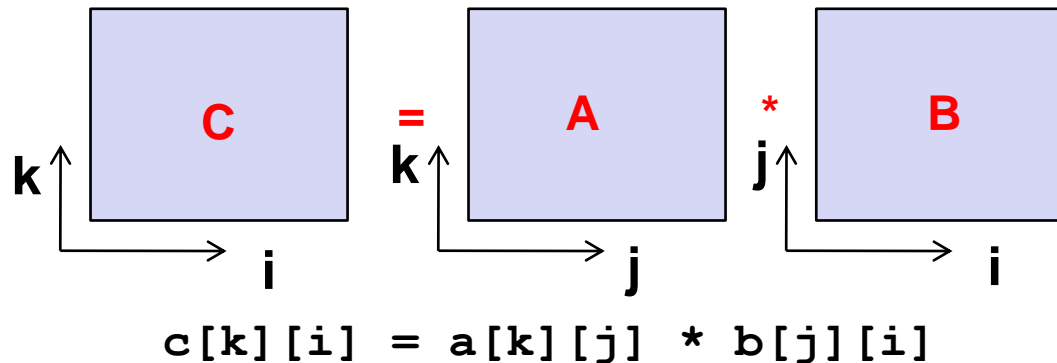
Hands-on

Getting started

- Try it out yourself
 - `module load UNITE`
 - `module load 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 $C = A * B$
 - Side length $N \rightarrow N^3$ multiplications + N^3 additions



- 3 nested loops (i,j,k): Best index order?
- Optimization for large matrixes: Blocking

Detailed analysis of matrix multiplication

- To try out...
 - `cp -r /lrz/sys/smuc_tools/tutorial/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

```
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”

?

Q&A

?

The central graphic features the text 'Q&A' in a large, bold, black sans-serif font. The ampersand is rendered in a lighter gray color. Two large, gray question marks are positioned around the text: one in the upper right and one in the lower left.

Contact
Josef Weidendorfer
TUM, Informatics I-10
weidendo@in.tum.de
+49 89 289 18454