

Cache Performance Analysis with Callgrind and KCachegrind

VI-HPS Tuning Workshop 8
September 2011, Aachen

Josef Weidendorfer

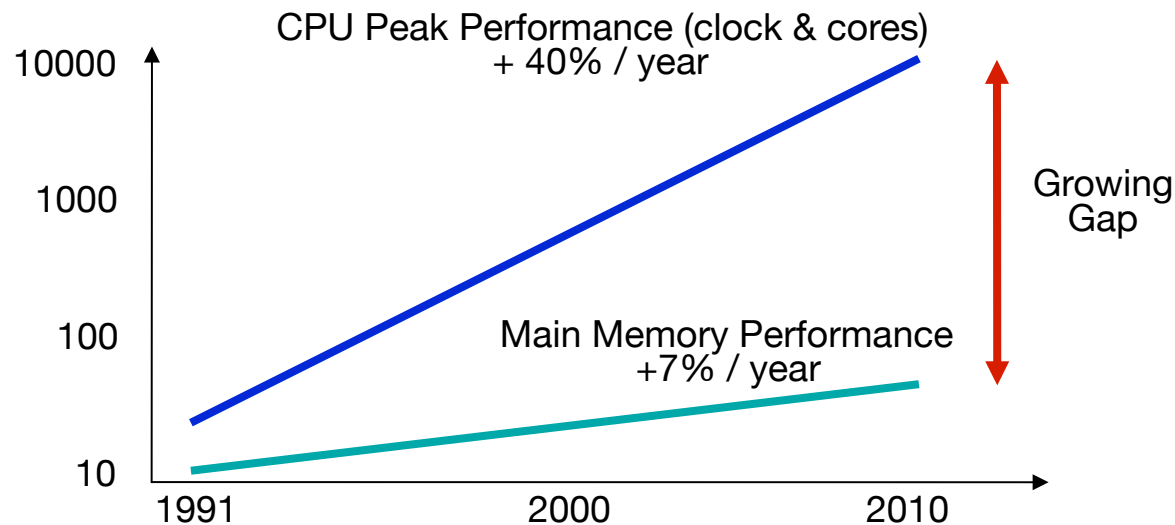
Computer Architecture I-10, Department of Informatics
Technische Universität München, Germany

Outline

- Background
- Callgrind and {Q,K}Cachegrind
 - Measurement
 - Visualization
- Hands-On
 - Example: Matrix Multiplication

Single Node Performance: Cache Exploitation is Important

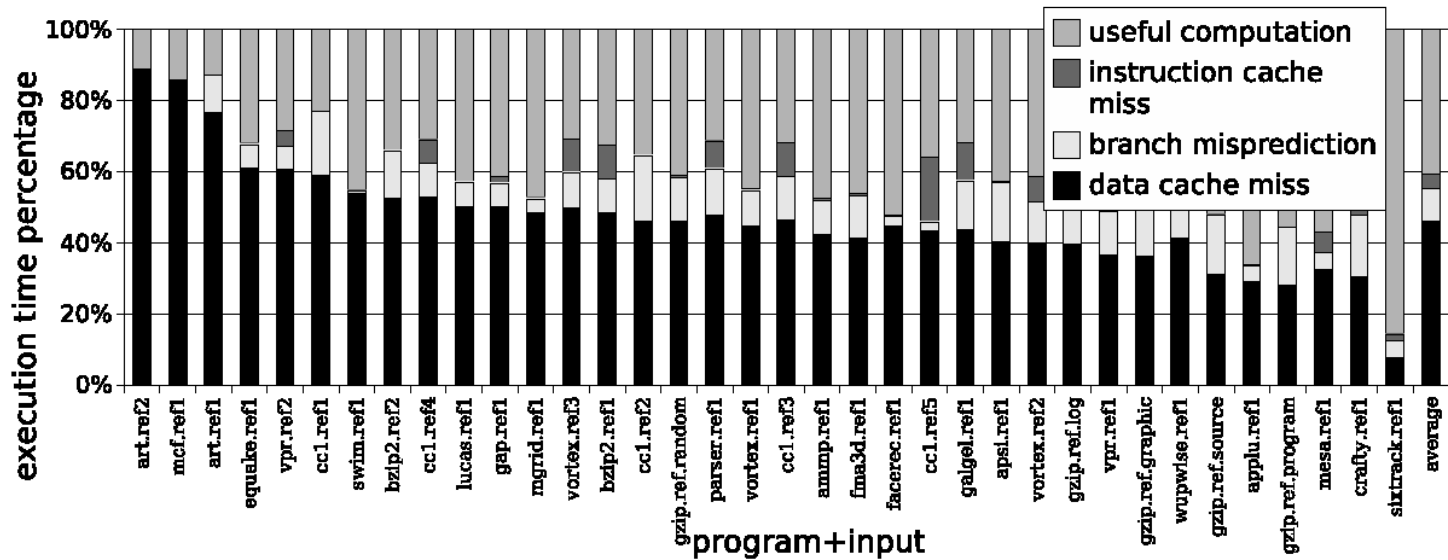
- „Memory Wall“



- Access Latencies:
 - modern x86 processors: ~ 200 cycles → 400 FLOP wasted...

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
 - does not need to strip down to sequential program
- Influences of threads/tasks on cache exploitation
 - on multicore: higher bandwidth requirement 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
 - better use simulation vs. impractical because of huge slowdown
 - research topic (worst case false sharing / OpenMP record/replay)

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 relatively easy to understand
- both methods (real vs. model) have advantages & disadvantages, but reality matters in the end

How to measure Events (2)

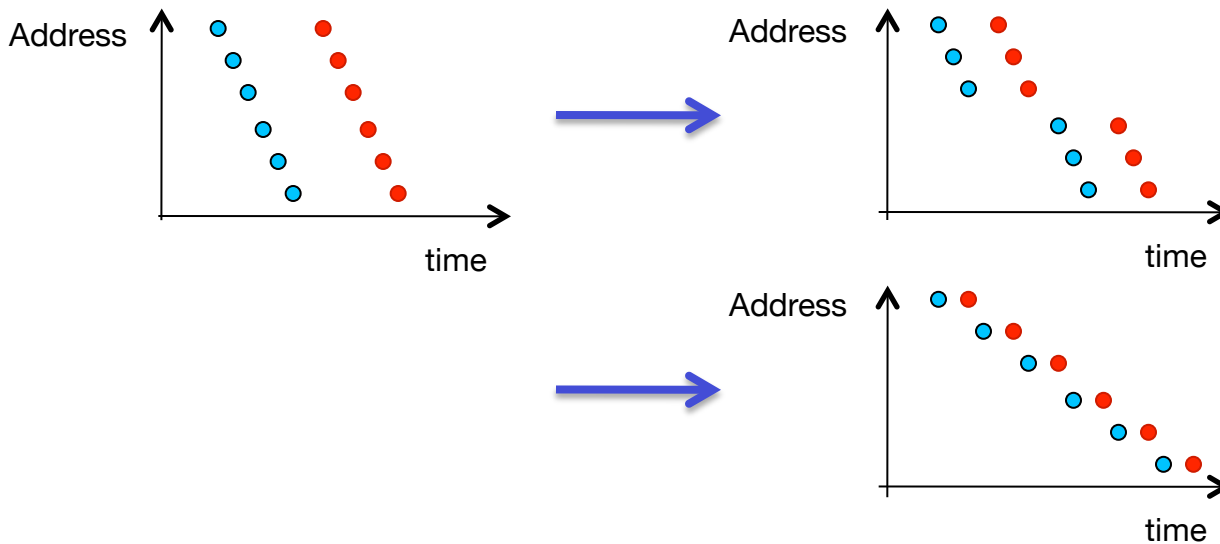
- SW-related
 - instrumentation (= insertion of measurement code)
 - into OS / application, manual/automatic, on source/binary level
 - on real HW: always incurs overhead which is difficult to estimate
- HW-related
 - read Hardware Performance Counters
 - gives exact event counts for code ranges
 - needs 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
 - prefetch data (automatically / SW prefetching) [on BG/P: sequential accesses]
 - memory controller on chip (standard today on modern x86, also BG/P)
 - low bandwidth (not so much a problem on BG/P)
 - share data in caches among cores
 - keep working set in cache (temporal locality)
 - use good data layout (spatial locality)

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 (VG 3.6), not (yet) with binaries for BG/P nodes
 - 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. JUROPA / BGP

- 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
- JUROPA: Intel Xeon X5570 (Nehalem, 4 cores)
 - inclusive, L1 D/I 32kB, L2 256 kB, L3 shared 8 MB
 - Callgrind only simulates L1 and L3 (= LLC), L3 hit count too high
- BG/P
 - L1/L2 use FIFO replacement (L2 mainly buffers for prefetching), L3 shared among 4 cores
 - Recommendation: look at LLC behavior in simulation

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 (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`

{Q,K}Cachegrind

Graphical Browser for Profile Visualization

Features

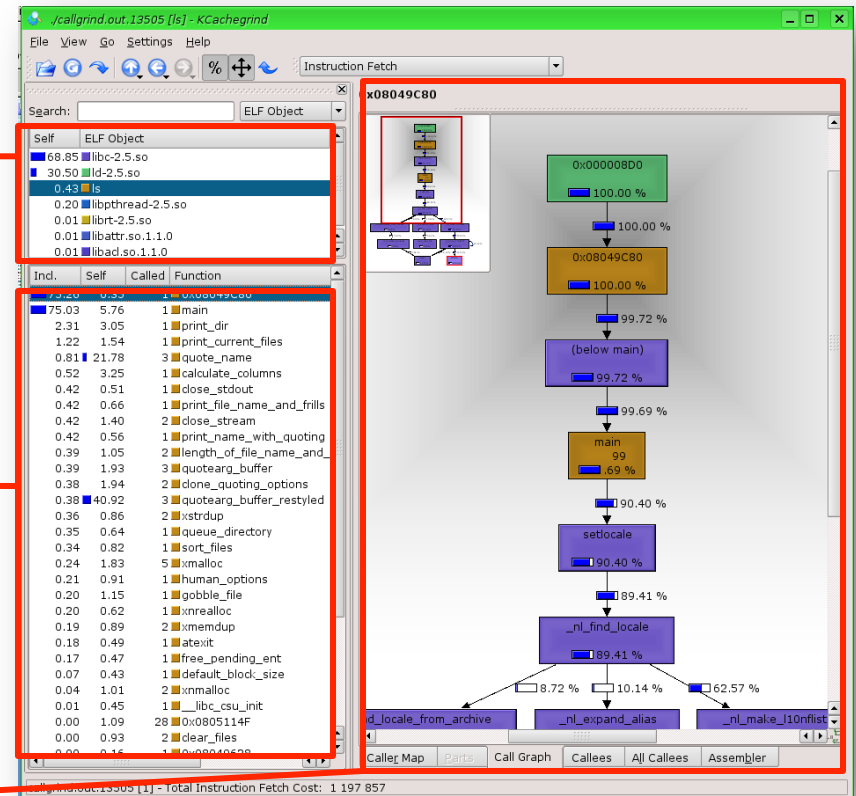
- open source, GPL
- kcachegrind.sf.net (recent release 0.7.0 includes pure Qt version, able to run on Mac OS-X/Windows)
- included with KDE3 & KDE4
- visualization of
 - call relationship of functions (callers, callees, call graph)
 - exclusive/Inclusive cost metrics of functions
 - grouping according 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)

Features

- supported format
 - currently callgrind format (support for Linux Perf. Events planned)
 - some converters available (OProfile, Java/Python/PHP profilers)
- special callgrind support:
 - derived event “cycle estimation” (very rough, formula can be edited)
 - $\text{exec. instructions} + 10 * \text{L1 misses} + 100 * \text{LL misses} + 10 * \text{Bm}$

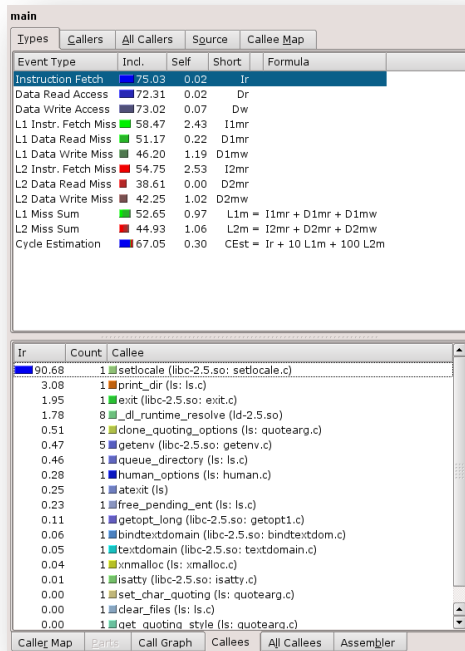
Usage

- `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

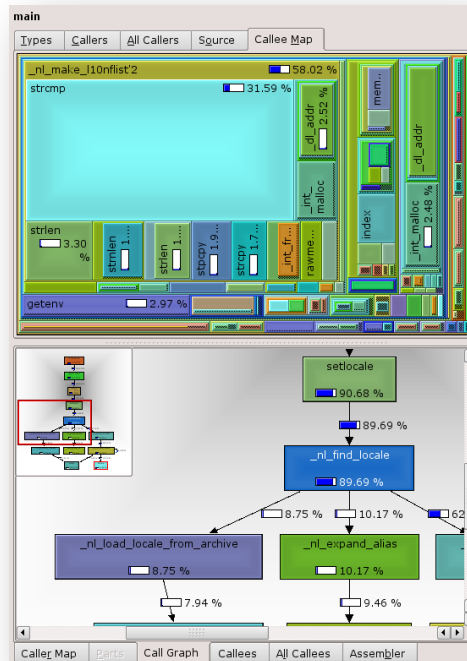
- List of event types
- List of callers/callees
- Treemap visualization
- Call Graph
- Source annotation
- Assembly annotation



The screenshot shows the 'main' window with the 'Types' pane selected. It displays a list of event types with their counts and formulas. Below it, the 'Callers' pane shows a list of callers with their counts and the callees they call.

Event Type	Incl.	Self	Short	Formula
Instruction Fetch	75.03	0.02	Ir	
Data Read Access	72.31	0.02	Dr	
Data Write Access	73.02	0.07	Dw	
L1 Instr. Fetch Miss	58.47	2.43	I1mr	
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	I2mr	
L2 Data Read Miss	39.61	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	

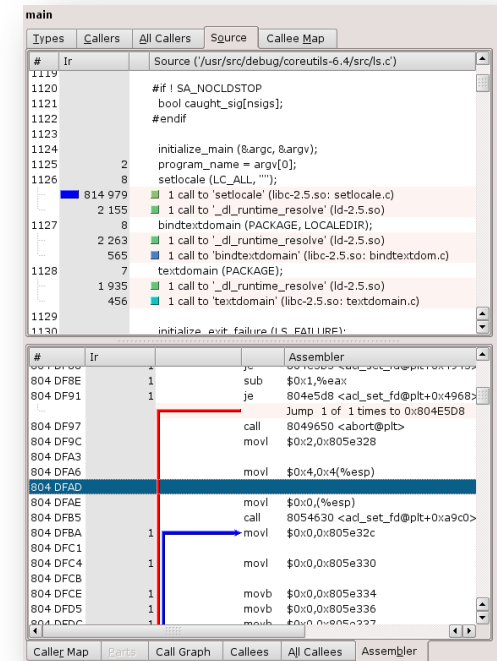
Ir	Count	Callee
90.68	1	setlocale (libc-2.5.so: setlocale.c)
3.08	1	print_dir (ls: ls.c)
1.95	1	exit (libc-2.5.so: exit.c)
1.78	8	_dl_runtime_resolve (ld-2.5.so)
0.51	2	ldone_quoting_options (ls: quotearg.c)
0.47	5	getenv (libc-2.5.so: getenv.c)
0.46	1	queue_directory (ls: ls.c)
0.28	1	human_options (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	xnmalloc (ls: xmalloc.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_quoting_style (ls: quotearg.c)



The screenshot shows the 'main' window with the 'Treemap' and 'Call Graph' panes selected. The treemap visualizes the memory usage of various functions, with 'setlocale' being the largest. The call graph shows the flow of control between functions, with 'setlocale' calling '_nl_find_locale', which in turn calls '_nl_load_locale_from_archive' and '_nl_expand_alias'.

```

graph TD
    setlocale[setlocale 90.68%] -- 89.69% --> _nl_find_locale[_nl_find_locale 89.69%]
    _nl_find_locale -- 8.75% --> _nl_load_locale_from_archive[_nl_load_locale_from_archive 8.75%]
    _nl_find_locale -- 10.17% --> _nl_expand_alias[_nl_expand_alias 10.17%]
    _nl_load_locale_from_archive -- 7.94% --> ...
    _nl_expand_alias -- 9.46% --> ...
    
```



The screenshot shows the 'main' window with the 'Source' and 'Assembler' panes selected. The source pane shows the C code for the 'main' function, and the assembler pane shows the corresponding assembly instructions. A red line highlights the call to 'setlocale' in the source code and the corresponding assembly instructions.

```

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
1128   814 979   1 call to 'setlocale' (libc-2.5.so: setlocale.c)
1129   2 155   1 call to '_dl_runtime_resolve' (ld-2.5.so)
1130   8      bindtextdomain (PACKAGE, LOCALEDIR);
1131   2 263   1 call to '_dl_runtime_resolve' (ld-2.5.so)
1132   565     1 call to 'bindtextdomain' (libc-2.5.so: bindtextdom.c)
1133   7      textdomain (PACKAGE);
1134   1 935   1 call to '_dl_runtime_resolve' (ld-2.5.so)
1135   456     1 call to 'textdomain' (libc-2.5.so: textdomain.c)
1136
1137   initialize_exit_failure (LS_FATAL_ERR);
1138
1139
1140 #
1141 Ir Assembler
1142 804 DF8E 1 sub $0x1,%eax
1143 804 DF91 1 je 804+5D8 <ad_set_fd@plt+0x4968>
1144
1145 804 DF97 call 8049650 <abort@plt>
1146 804 DF9C movl $0x2,0x805e328
1147 804 DFA3
1148 804 DFA6 movl $0x4,0x4(%esp)
1149 804 DFAD
1150 804 DFAE movl $0x0,(%esp)
1151 804 DF85 call 8054630 <ad_set_fd@plt+0xa9c0>
1152 804 DFBA movl $0x0,0x805e32c
1153 804 DFC1
1154 804 DFC4 movl $0x0,0x805e330
1155 804 DFCB
1156 804 DFCE movb $0x0,0x805e334
1157 804 DFDS movb $0x0,0x805e336
1158 804 DFDE movb $0x0,0x805e337
1159
1160 #
    
```

Call

The image displays two side-by-side screenshots of the Callgrind/KCachegrind tool. The top portion shows two 'Flat Profile' windows. The left window shows a table of function calls, with the entry 'via length of file name ...' selected. The right window shows the same table, but with 'via print_name_with_qu...' selected. Red boxes highlight these specific entries in both windows. Below the flat profiles are two 'Call Graph' windows for the 'quote_name' function. The left call graph shows a detailed view of the function's internal calls, with 'length_of_file_name_and_frills' and 'quote_name' highlighted in red. The right call graph shows the same function, but with 'length_of_file_name_and_frills' and 'quote_name' highlighted in grey, indicating they are not active in the current context. The bottom of each window shows the total instruction fetch cost: 42 191 913.

Incl.	Self	Called	Function	Location
56.63	49.73	41 246	strcoll_l	libc-2.11.2.so: strcoll_l.c, alloca.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...	

Hands-on

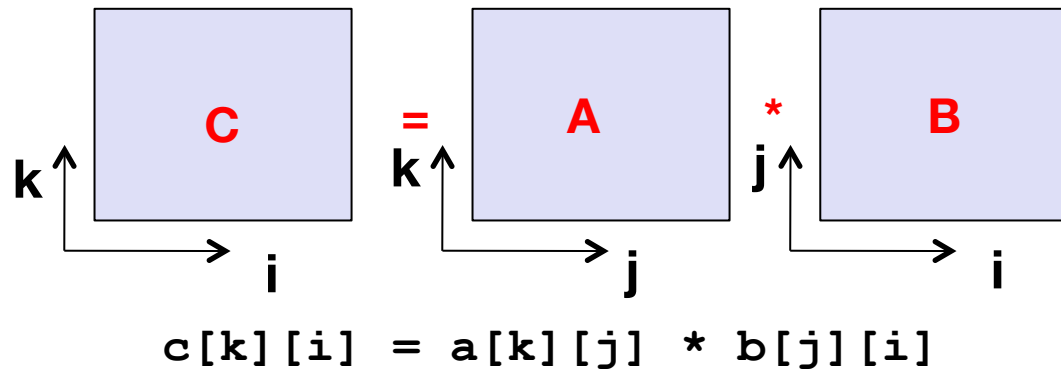
Getting started

- Try it out yourself (on JUROPA / cluster-beta)
 - module add UNITE
 - module add kcachegrind
- 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?

 - 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 ~hpclab01/tutorial/mm-vihpstw8 .`
 - `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 “cluster-beta”
module add UNITE
module add kcachegrind
export OMP_NUM_THREADS=4
mpiexec -n 4 valgrind --tool=callgrind --cache-sim=yes \
--separate-threads=yes ./bt-mz_B.4
- \leq VG 3.6.x: cache config detection on Westmere not working
 - “--l1=32768,4,64 --D1=32768,8,64 --LL=12582912,24,64”
- 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 text 'Q & A' is rendered in a large, bold, black sans-serif font. The ampersand is a lighter shade of gray. Three large, gray question marks are scattered around the central text: one in the upper right, one in the lower left, and one in the upper right.