Cache Performance Analysis with Callgrind and KCachegrind

21th VI-HPS Tuning Workshop
April 2016, Garching

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
- no special permissions needed / can not crash machine

Focus only on cache / a simple model really enough?

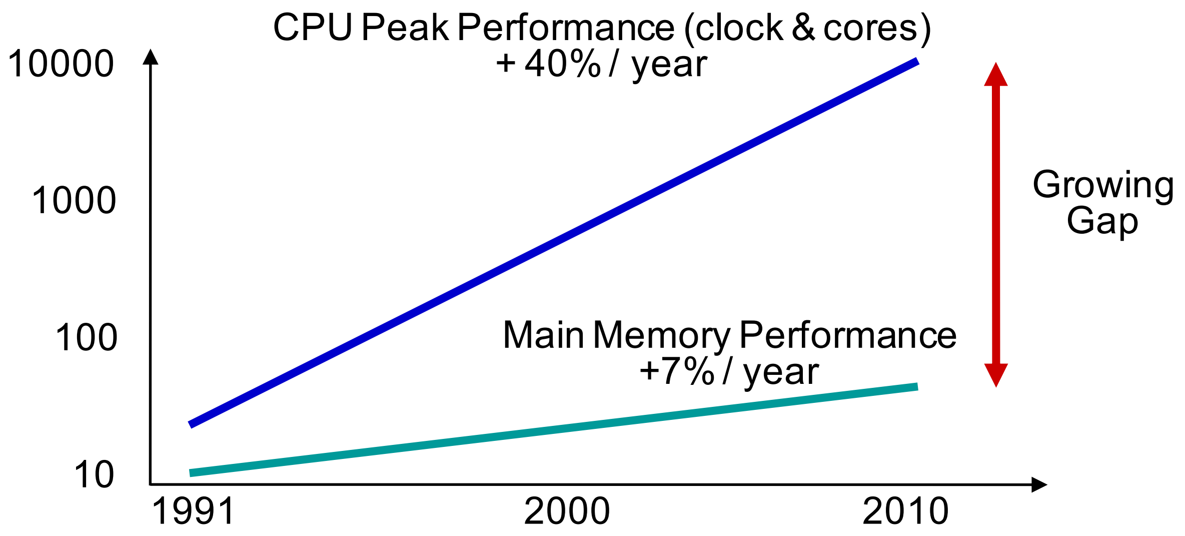
- **no**: if real measurement shows cache issues, use simulation for details
- if bad cache exploitation dominates: you can ignore other bottlenecks
- benefits of simple machine models:
  - easy to understand, still captures most problems, faster simulation…

Outline

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

Single Node Performance: Cache Exploitation is Important

- „Memory Wall"



- Worst-case (local) access latencies on modern x86 processors ~ 200 cycles
  ➔ AVX2 can do 200 * 4 (vector) * 4 (2 FMA units) = 1600 DP-FLOPs

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

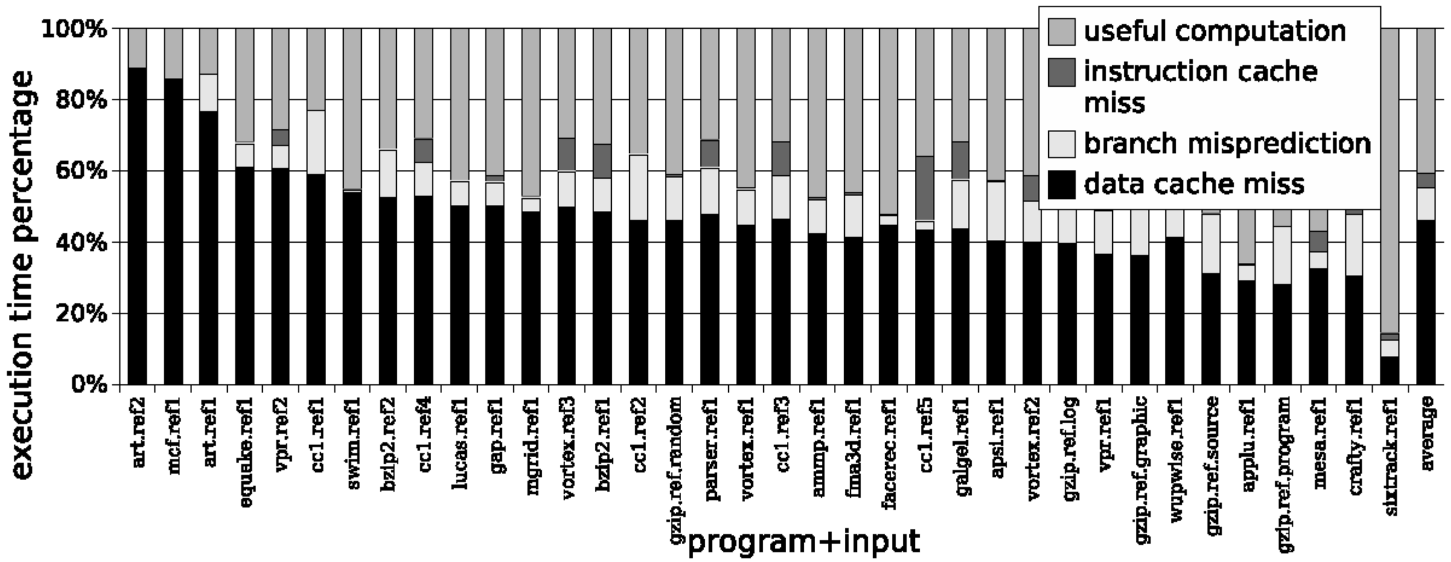
- power consumption [Keynote Dongarra, PPAM 2011]
  - DP FMADD: 100 pJ (2011) ➔ 10 pJ (expected 2018)
  - DP Read DRAM: 4800 pJ (2011) ➔ 1920 pJ (expected 2018)

Caches do their Job transparently…

Caches work because 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

- Analyze 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 measure 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 (caches, vector units, pipelining,...)

- how to improve sequential performance
  - use tuned libraries where available
  - check for above obstacles ➔ 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

- target real hardware
  - needs sensors for interesting events
  - for low overhead: hardware support for event counting
  - may be difficult to understand because of unknown micro-architecture, overlapping and asynchronous execution
- target 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

Back to the Memory Wall: Improvements

Access latency

- exploit fast caches: improve locality of data
- allow hardware to prefetch data (use access patterns which are easy to predict)
- memory controller on chip (standard today)

Low bandwidth

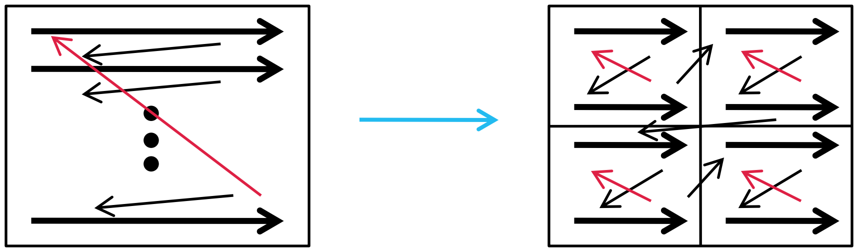
- share data in caches among cores
- keep working set in cache (temporal locality)
- use good data layout (spatial locality)
- if memory accesses are unavoidable: duplicate data in NUMA nodes

Cache Optimization (1): Reduce Number of Accesses

- use large data types (may be done by compiler)
  - vectors instead of bytes
- 1 cache line = 1 access: use full cache lines
  - alignment: crossing cache line gives two accesses
- (redundant) calculation instead of memory access
- avoid unneeded writes
  - check if a variable already has given value before writing
  - writes result in higher bandwidth needs

Cache Optimization (2): Reorder Accesses

- if possible, do sequential accesses at inner level
  - exploit full cache line
  - trigger hardware prefetcher
    (small sequential accesses: reduce accuracy of prefetcher)
- blocking: reuse data as much as possible
  - instead of multiple large sweeps over large buffer,
    split up into multiple small sweeps over buffer parts
  - useful in 1d, 2d, 3d, …



  - recursive (multi-level) blocking: "cache-oblivious":
    best use of multiple cache levels at once!
  - multi-core: consecutive iterations on cores with shared cache

Cache Optimization (3): Improve Data Layout

- group data with same access frequency and access type (read vs. write)
  - use every byte of a fetched cache line (unused data is wasted space + bandwidth)
  - AoS-to-SoA
- reorder data in memory according to traversal order in program
- avoid power-of-2 strides: may produce conflict misses
  - by padding

Callgrind

Cache Simulation with Call-Graph Capturing

Callgrind: Basic Features

Based on Valgrind

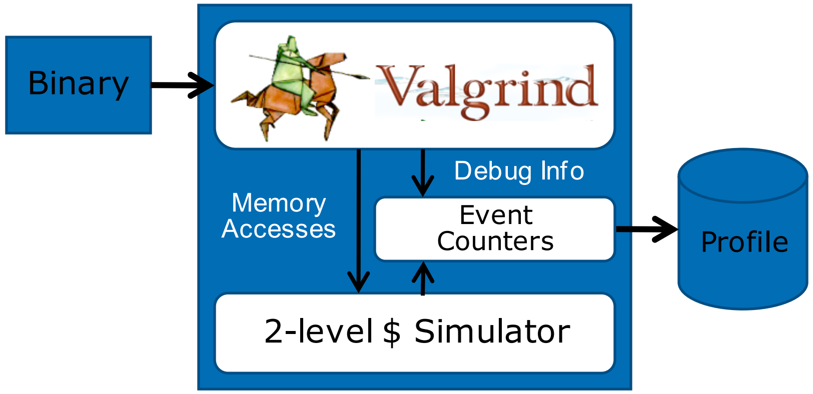
- runtime instrumentation infrastructure (no recompilation needed)
- dynamic binary translation of user-level processes
- Linux/AIX/OS X on x86, x86-64, PPC32/64, ARM, MIPS
- Open source (GPL), www.valgrind.org

- includes correctness checking & profiling tools
  - “memcheck”: accessibility/validity of memory accesses
  - “helgrind” / ”drd”: race detection on multithreaded code
  - “cachegrind”/”callgrind”: cache & branch prediction simulation
  - “massif”: memory profiling

Callgrind: Basic Features

Part of Valgrind (since 3.1)

- Open Source, GPL
- extension of 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
- serializes threads
- detailed 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)
- reproducible results independent on real machine load
- derived optimizations applicable for most architectures

Callgrinds Cache Model vs. SuperMUC / UV2

- parameters: size, line size, associativity
- L1 / LLC, inclusive, LRU, shared among threads
- write back vs. write through does not matter for hit/miss counts
- optional stream prefetcher

SuperMUC node: 2x Intel E5-2680 (SandyBridge, 8 core, 20 MB L3)
SuperMUC-2 node: 2x Intel E5-2697v3 (Haswell, 14 core, 2x18 MB L3)
UV2: 96x Intel Westmere-EX (10 core, 30 MB L3)

- private L1 (D/I a 32kB) + L2 (256 kB) per core
- L1/L2 strictly inclusive to L3, L3 shared (Haswell: half of cores see shared half of L3)

Callgrind only simulates L1 and L3 (= LLC) ➔ LLC 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)

Optional

- best-case simulation of simple stream prefetcher
- byte-wise usage of cache lines before eviction
- branch prediction
- dynamic context in function names (call chain/recursion depth)
- wallclock 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 output per thread: --separate-threads=yes
- jump-profiling 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.github.io
(includes pure Qt version, able to run on Linux / OS-X / Windows)

Visualization of

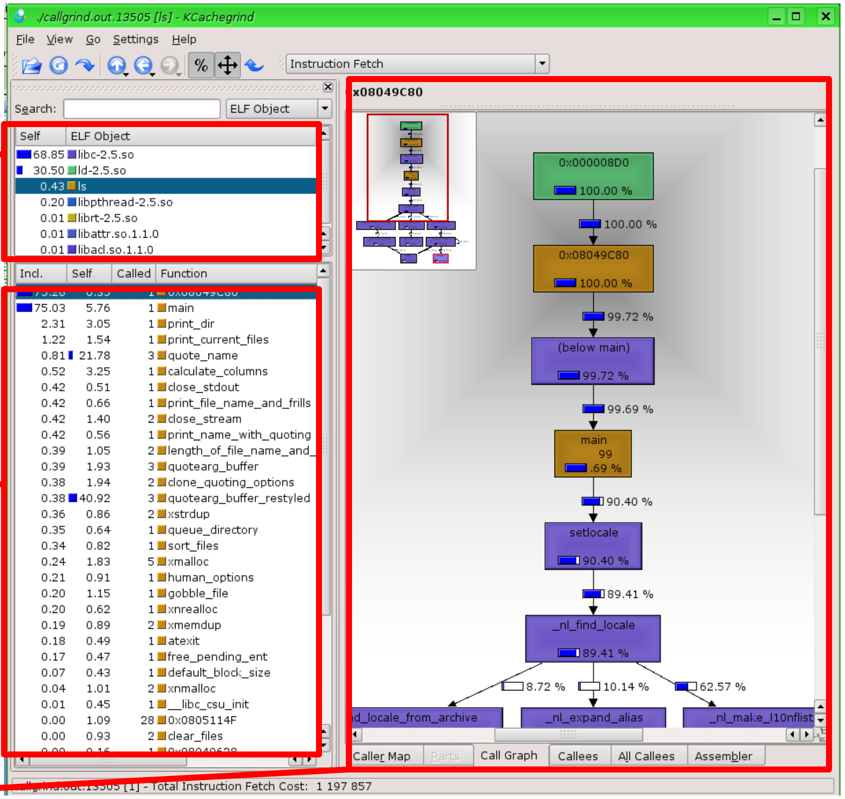
- call relationship of functions (callers, callees, call graph)
- exclusive/Inclusive cost metrics of functions
  - grouping according to ELF object / source file / C++ class
- source/assembly annotation: costs + CFG
- arbitrary events counts + specification of derived events

Callgrind support: file format, events of cache model

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



Self	ELF Object
68.85	🟪 libc-2.5.so
30.50	🟩 ld-2.5.so
0.43	🟧 ls
0.20	🟦 libpthread-2.5.so
0.01	🟨 librt-2.5.so
0.01	🟦 libattr.so.1.1.0
0.01	🟪 libacl.so.1.1.0

Incl.	Self	Called	Function
75.26	0.35	1	0x08049C80
75.03	5.76	1	main
2.31	3.05	1	print_dir
1.22	1.54	1	print_current_files
0.81	21.78	3	quote_name
0.52	3.25	1	calculate_columns
0.42	0.51	1	close_stdout
0.42	0.66	1	print_file_name_and_frills
0.42	1.40	2	close_stream
0.42	0.56	1	print_name_with_quoting
0.39	1.05	2	length_of_file_name_and_
0.39	1.93	3	quotearg_buffer
0.38	1.94	2	clone_quoting_options
0.38	40.92	3	quotearg_buffer_restyled
0.36	0.86	2	xstrdup
0.35	0.64	1	queue_directory
0.34	0.82	1	sort_files
0.24	1.83	5	xmalloc
0.21	0.91	1	human_options
0.20	1.15	1	gobble_file
0.20	0.62	1	xnrealloc
0.19	0.89	2	xmemdup
0.18	0.49	1	atexit
0.17	0.47	1	free_pending_ent
0.07	0.43	1	default_block_size
0.04	1.01	2	xnmalloc
0.01	0.45	1	__libc_csu_init
0.00	1.09	28	0x0805114F
0.00	0.93	2	clear_files
0.00	0.16	1	0x08049628

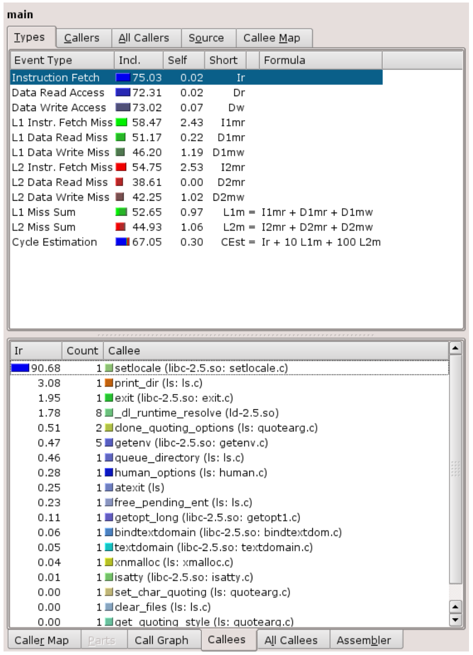
callgrind.out.13505 [1] - Total Instruction Fetch Cost: 1 197 857

Visualization panes for selected function

- List of event types
- List of callers/callees

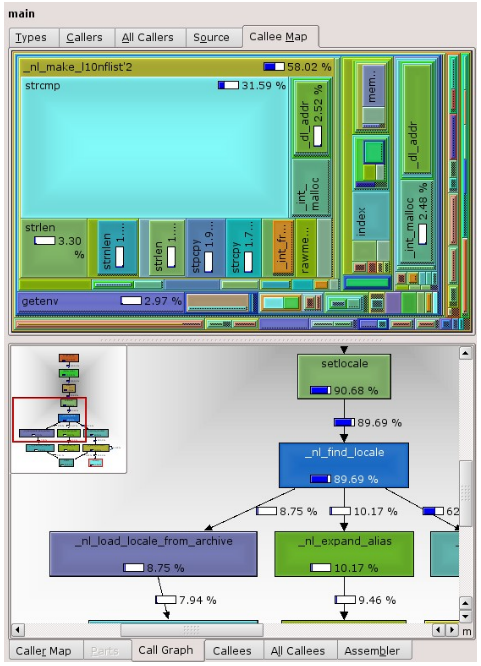
- Treemap visualization
- Call Graph

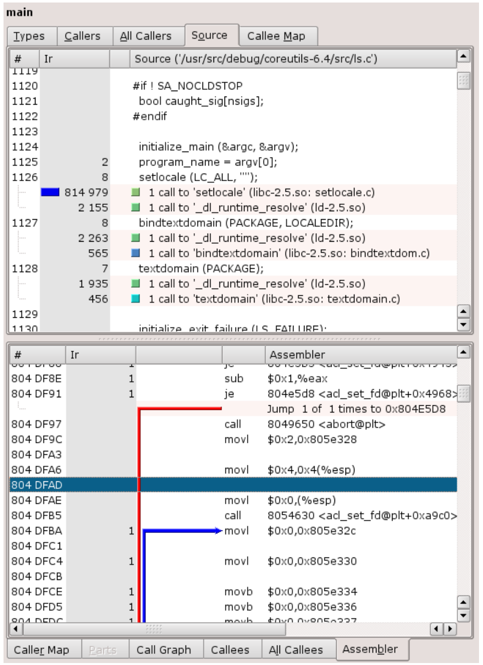
- Source annotation
- Assemly annotation



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	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	clone_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)





```
Ir        Source ('/usr/src/debug/coreutils-6.4/src/ls.c')
          #if ! SA_NOCLDSTOP
            bool caught_sig[nsigs];
          #endif

            initialize_main (&argc, &argv);
       2    program_name = argv[0];
       8    setlocale (LC_ALL, "");
 814 979      1 call to 'setlocale' (libc-2.5.so: setlocale.c)
   2 155      1 call to '_dl_runtime_resolve' (ld-2.5.so)
       8    bindtextdomain (PACKAGE, LOCALEDIR);
   2 263      1 call to '_dl_runtime_resolve' (ld-2.5.so)
     565      1 call to 'bindtextdomain' (libc-2.5.so: bindtextdom.c)
       7    textdomain (PACKAGE);
   1 935      1 call to '_dl_runtime_resolve' (ld-2.5.so)
     456      1 call to 'textdomain' (libc-2.5.so: textdomain.c)
```

```
#          Ir          Assembler
804 DF8E    1    sub   $0x1,%eax
804 DF91    1    je    804e5d8 <acl_set_fd@plt+0x4968>
                       Jump  1 of  1 times to 0x804E5D8
804 DF97         call  8049650 <abort@plt>
804 DF9C         movl  $0x2,0x805e328
804 DFA3
804 DFA6         movl  $0x4,0x4(%esp)
804 DFAD
804 DFAE         movl  $0x0,(%esp)
804 DFB5         call  8054630 <acl_set_fd@plt+0xa9c0>
804 DFBA    1    movl  $0x0,0x805e32c
804 DFC1
804 DFC4    1    movl  $0x0,0x805e330
804 DFCB
804 DFCE    1    movb  $0x0,0x805e334
804 DFD5    1    movb  $0x0,0x805e336
```

Hands-on

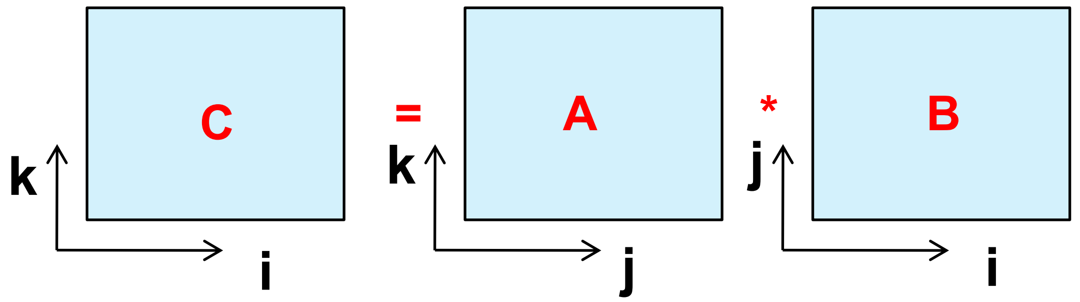
Getting started

- Try it out
  - on ice1-login/interactive job on uv2 ...
  - module load valgrind/3.10
  - cp -r /lrz/sys/courses/VIHPS21TW/kcg ~
  - GUI (ssh -X ice1-login): ~/kcg/qcachegrind
- Test: What happens in „/bin/ls“ ?
  - run valgrind --tool=callgrind ls /usr/bin
  - run ~/kcg/qcachegrind
  - function with highest instruction execution count? Purpose?
  - where is the main function?

  - run with cache simulation: --cache-sim=yes

Detailed analysis of matrix multiplication

- Kernel for C = A * B
  - Side length N ➔ N3 multiplications + N3 additions



```
c[k][i] = a[k][j] * b[j][i]
```

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

Detailed analysis of matrix multiplication

- To try out...
  - cd ~/kcg; make
  - timing of orderings (e.g. size 512): ./mm 512
  - cache behavior for small matrix (fits 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
- module load valgrind/3.10
- srun -n 4 -t 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

?

Josef Weidendorfer
TUM, Informatics I-10
weidendo@in.tum.de