

# Cache Performance Analysis with Callgrind and KCachegrind

---

---

Josef Weidendorfer <sup>1</sup>

45<sup>th</sup> VI-HPS Tuning Workshop

June 2024, Garching

---

---

<sup>1</sup> Leibniz Supercomputing Centre (LRZ)

# Focus: CPU Cache Simulation using a Simple Machine Model

---

Why simulation? (in contrast to real measurement)

- Reproducibility
- No influence of tool on results
- Allows to collect information not possible with real hardware
- No special permissions needed / cannot crash machine

## Focus: CPU Cache Simulation using a Simple Machine Model

---

Why a simple machine model?

- easier to understand
- still captures most problems
- faster simulation

A sophisticated model includes

- All pipeline stages, Out-of-Order scheduling, speculation, instr. throughput & latency
- All cache layers, coherency protocol, replacement, memory parallelism, contention, hardware prefetching, exact interleaving of accesses from cores

# Focus: CPU Cache Simulation using a Simple Machine Model

---

Why a simple cache model?

- Bottlenecks in the memory hierarchy often dominate anything else
  - You should first check this with real measurements
- Qualitative results still useful for cache optimizations

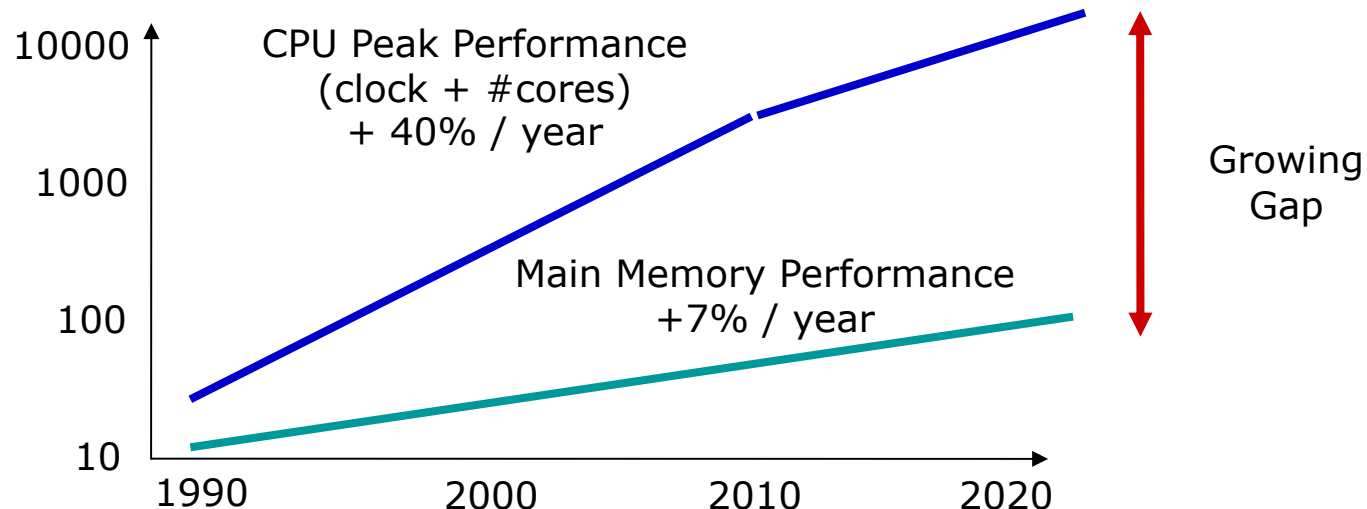
# Outline

---

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

# Single Node CPU Performance: Cache Exploitation is Important

„Memory Wall“



Access latencies to local memory on modern x86 processors  $\sim 200$  cycles

→ AVX512 can do  $200 * 8$  (vector) \* 4 (2 FMA units) = 6400 DP-FLOPs / access



# Single Node CPU Performance: Cache Exploitation is Important

---

This will mostly be true also in the future

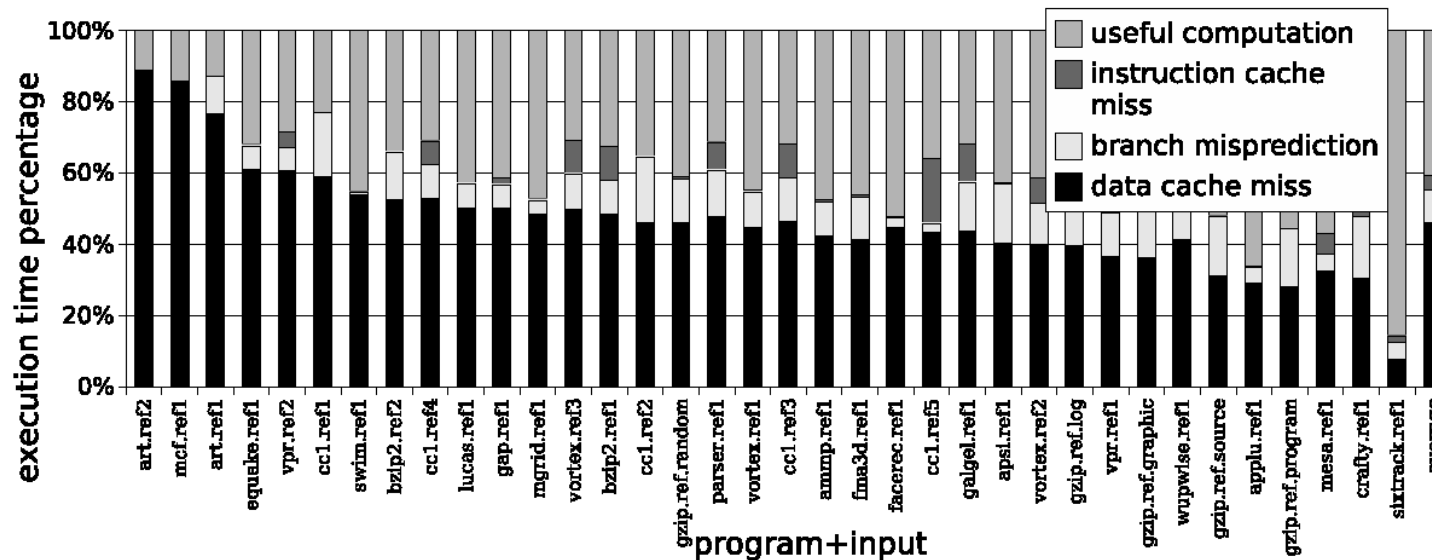
- Latency of main memory access does not improve
- Bandwidth to typical main memory (DDR<sub>x</sub>) increases slower than compute power
  - Lots of integrated cores requesting access to memory (multi-core, accelerators)
  - Improvements GDDR<sub>x</sub> / HBM<sub>x</sub>: better bandwidth, but fixed capacity, higher power (Examples: GPUs / A64FX with HBM)
- Power consumption
  - DP FMADD:                    100 pJ (2011) →                    10 pJ (2018)
  - DP Read DRAM:                4800 pJ (2011) →                1920 pJ (2018)
  - “Power Wall”: drives large system designs (lots of compute but low memory capacity → NVM)
  - “Affinity Wall”: pressure towards shorter transfer distances makes programming more complex (increased number of NUMA domains, NUCA & PIM designs)

## 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 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

## Going Sequential ...

---

- Sequential performance bottlenecks
  - Logical errors (unnneeded/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 solutions
- Both methods (real vs. model) have advantages & disadvantages, but reality matters in the end

# Back to the Memory Wall: Improvements

---

## Latency

- Exploit (fast) cache: improve locality of data
- Allow hardware to prefetch data (use access patterns which are easy to predict)
- Memory controller on chip (standard today) – be aware of NUMA

## 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
  - Predictable access pattern (stream/strided) → exploit HW prefetcher
  - Memory affinity
  - Avoid data dependencies (linked list traversals)

# Cache Optimization (1): Reduce Number of Accesses

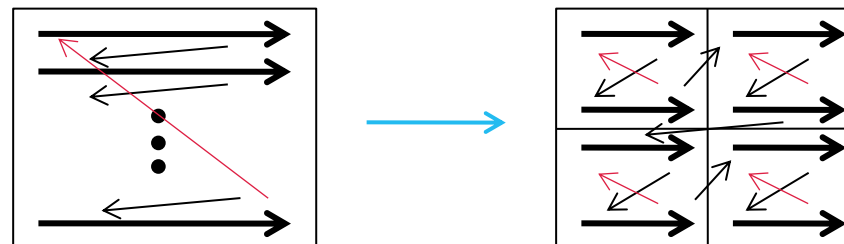
---

- 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
  - "Write-allocate" effect: higher bandwidth than expected



## Cache Optimization (2): Reorder Accesses

- If possible, do sequential accesses (in inner loop level)
  - Exploit full cache line
  - Trigger hardware prefetcher  
(small sequential accesses reduce accuracy of HW 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

---

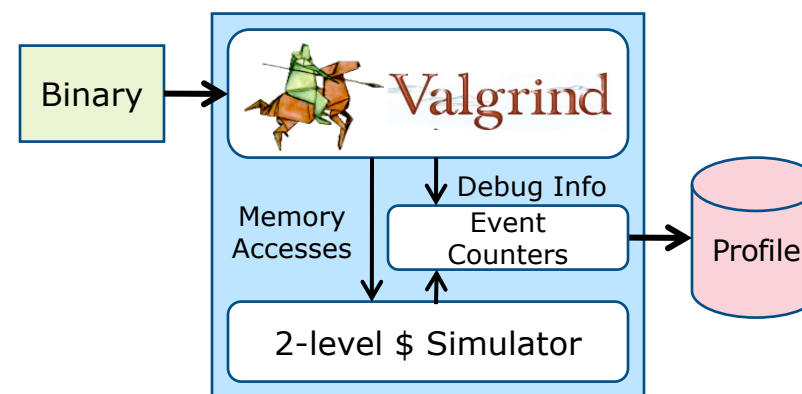
Based on Valgrind

- Runtime instrumentation infrastructure (no recompilation needed)
- Dynamic binary translation of user-level processes
- Linux/AIX/OS X on x86, x86-64, PPC32/64, ARM/ARM64, MIPS
- Open source (GPL), [www.valgrind.org](http://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

- Open Source, GPL
- Callgrind vs. 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. Xeon

---

- 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

CoolMUC2 node: 2x Intel Xeon (Haswell, each 14 cores, 35 MB L3)

- private L1 (D/I a 32kB) + L2 (256 kB) per core
- L1/L2 strictly inclusive to L3, L3 shared (SNC2: 17.5 MB L3 per core visible)

Callgrind only simulates 2 levels (L1+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)

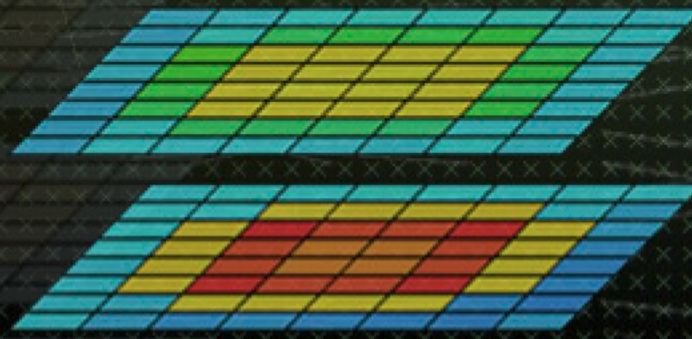
### Optional

- 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 Cheat-Sheet

---

- `"valgrind -tool=callgrind [callgrind options] <yourprogram> [args]"`
- Cache simulator: `"--cache-sim=yes"`
- Specify cache sizes: `"--L1/I1/LL=<size>,<assoc>,<linesize>"`
- 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](https://github.com/KDE/kcachegrind)

- <https://github.com/KDE/kcachegrind>
- 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

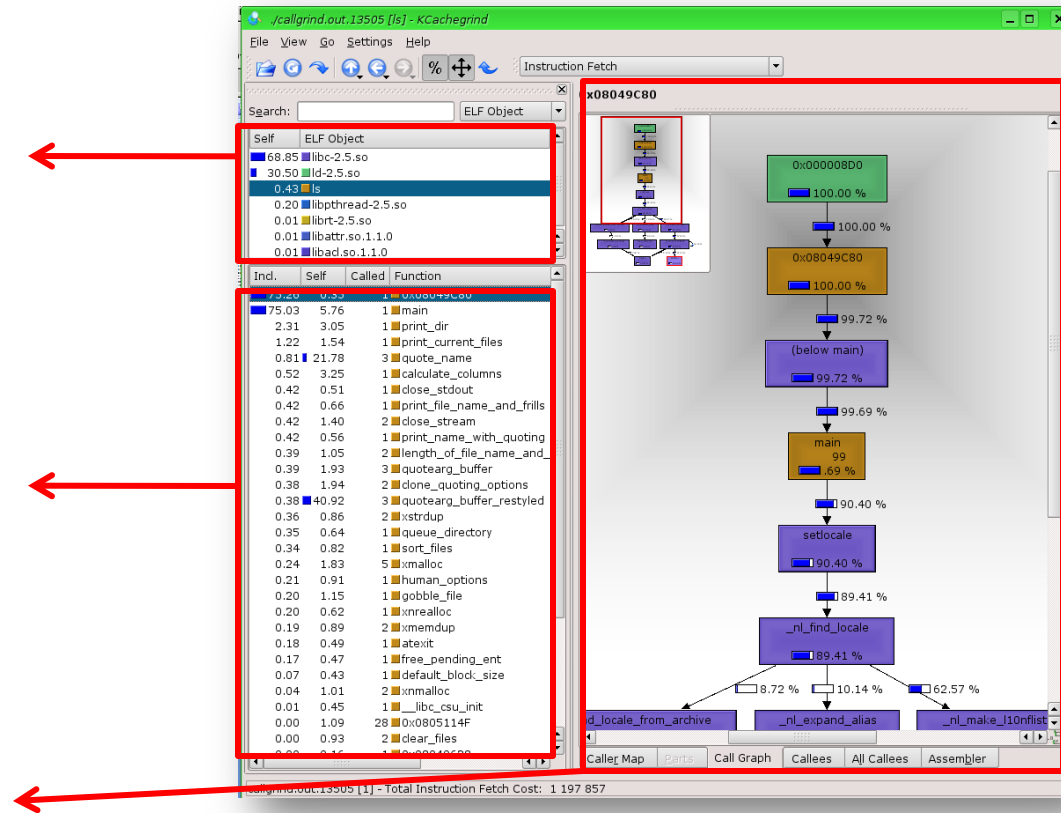
Callgrind support: file format, events of cache model



# 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

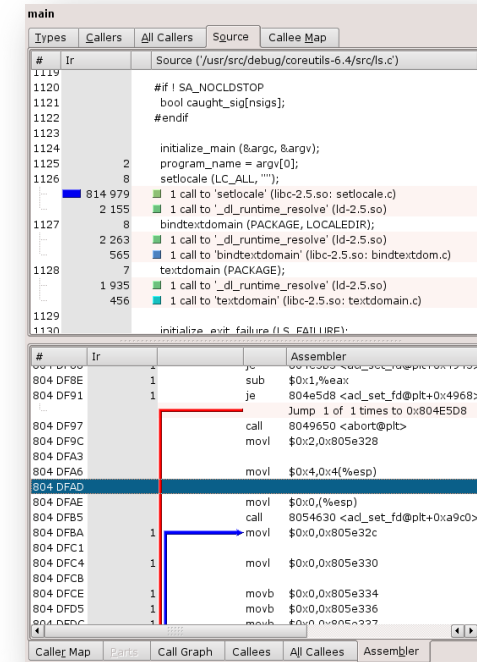
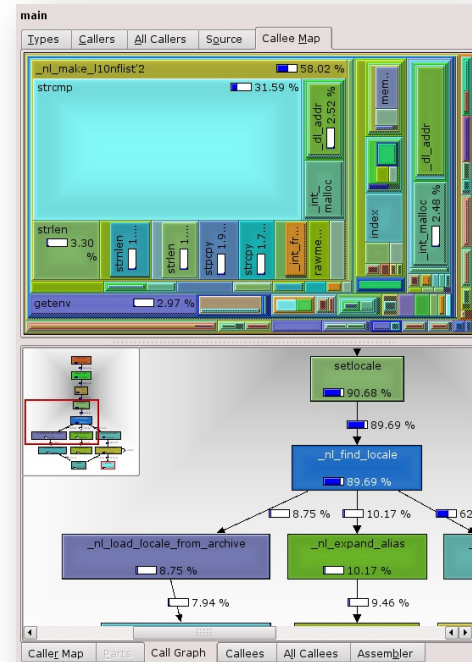
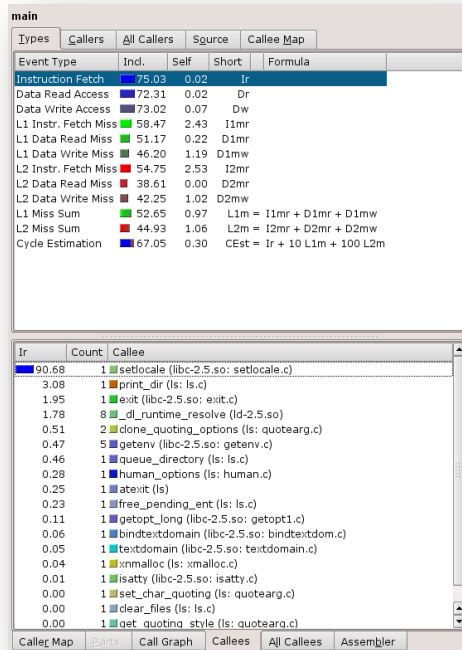


- Right: visualization panes



# Visualization panes for selected function

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



# Hands-on

## Getting started (see `/lrz/sys/courses/vihps/material/kcg`)

---

### ▪ Setup

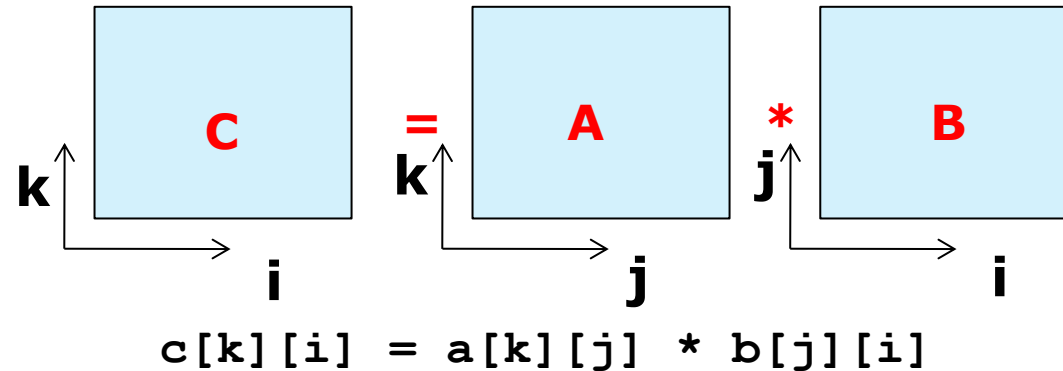
- `"ssh -X hpckursXX@lxlogin1.lrz.de"`
  - `"cp -r /lrz/sys/courses/vihps/2024/material/kcg ."`
  - `"module use /lrz/sys/courses/vihps/2024/modulefiles"`
  - `"module load qcachegrind"`
  - `"module load valgrind"`
- } `"source kcg/kcg.env"`

### ▪ Test: What happens in `./bin/ls` ?

- run `"valgrind --tool=callgrind ls /usr/bin"`
  - run `"qcachegrind"`
  - function with highest instruction execution count? Purpose?
  - where is the main function?
- 
- run with cache simulation: `"--cache-sim=yes"`

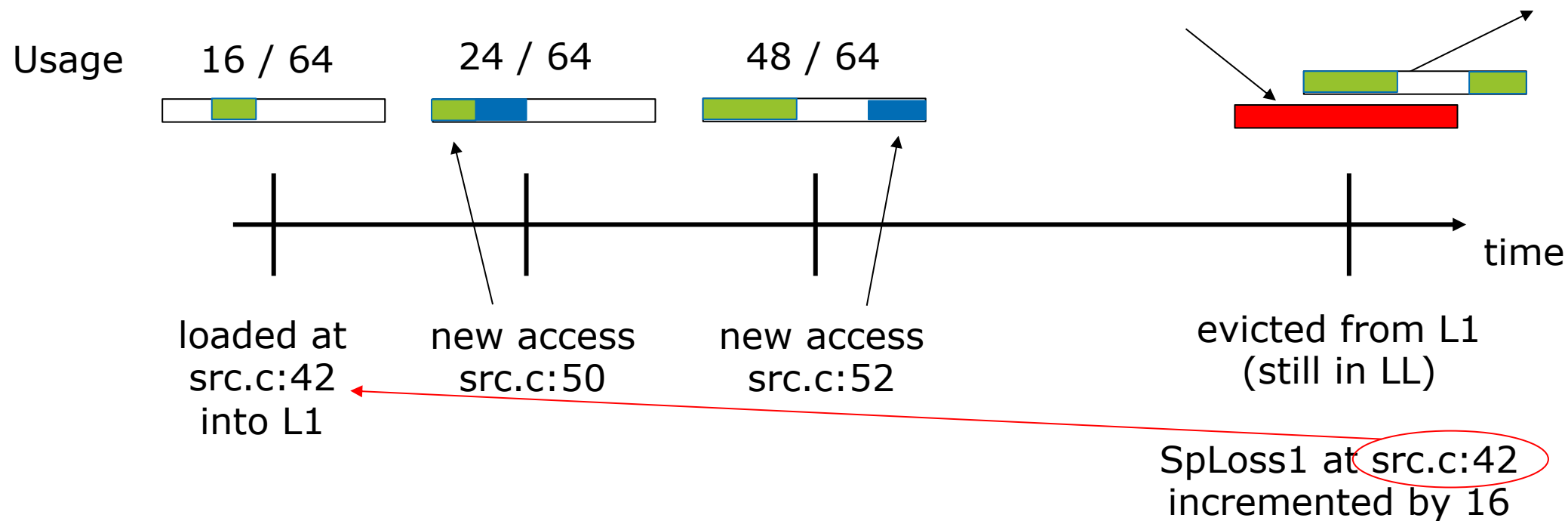
# Detailed analysis of matrix multiplication

- Kernel for  $C = A * B$ 
  - Side length  $N \rightarrow N^3$  multiplications +  $N^3$  additions



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

## Cacheline Usage



Why maintain "Loss" events? Higher Numbers should point at larger bottlenecks (here: 16B lost)

Why attribution to line loading the cacheline? No variable to attach "Loss" to, still understandable



## Detailed analysis of matrix multiplication

---

- To try out...
  - `"cd ~/kcg; make"`
  - timing of orderings (e.g. size 512): `"./mm 800"`
  - 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?
  - Warning: Login node has 17.5 MB LLC visible by cores (also used in simulation)

### Cache Line Usage

- `"valgrind --tool=callgrind --cacheuse=yes ./mm"`
- Right-click in list on Types tab, "New Event Type", double click formula column: `"64 L1m"`  
= "How much data is loaded into L1"
- compare with `"SpLoss1"` : "How much data was never accessed but loaded into L1"

Other example: 2d Jacobi solver: `jc / jc.c`



## How to run with MPI (here: interactive session)

---

- Optional: reduce iterations in BT\_MZ / use class A
  - `sys/setparams.c, write_bt_info, class A: set niter = 5`
  - `"make clean; make bt-mz CLASS=A NPROCS=2"`
- `"salloc --nodes=1 --reservation=hhps1s24"`
- `"module load valgrind"`
- `"export OMP_NUM_THREADS=2"`
- `"mpiexec -n 2 valgrind --tool=callgrind --cache-sim=yes \`  
`--separate-threads=yes bin/bt-mz_A.2"`  
(verification with fail as it assumes 200 iterations)
- load all profile dumps at once:
  - in directory you started mpiexec: `"qcachegrind callgrind.out.*"`

The graphic features the text 'Q&A' in a large, bold, sans-serif font. The 'Q' and 'A' are black, while the ampersand is grey. Three large, grey question marks are scattered around the text: one to the left, one above, and one to the right.

Josef Weidendorfer  
LRZ  
weidendo@lrz.de