Analysis and Optimization of the Memory Access Behavior of Applications

École Optimisation 2014

Université de Strasbourg

Josef Weidendorfer

Chair for Computer Architecture (LRR)
TUM, Munich, Germany

Fakultät für Informatik
der Technischen Universität München
Informatik X: Rechnertechnik und Rechnerorganisation / Parallelrechnerarchitektur
Prof. Dr. Arndt Bode , Prof. Dr. Hans Michael Gerndt

My Background

- Chair for computer architecture at CS faculty, TUM
  - how to exploit current & future (HPC) systems (multicore, accelerators)
  - programming models, performance analysis tools, application tuning
- PhD on load balancing of commercial car crash code (MPI) 2003
- Interested especially in cache analysis and optimization
  - cache simulation: Callgrind (using Valgrind)
  - applied to 2D/3D stencil codes
  - recently extended to multicore (new bottlenecks, new benefits)

Topic of this Morning: Bottleneck Memory

- Why should you care about memory performance?
- Most (HPC) applications often do memory accesses
- Good vs. bad use of the memory hierarchy can be ~ factor 100 (!)
- Example: modern processor with 3GHz clock rate, 2 sockets
  - latency to remote socket ~ 100 ns: 300 clock ticks
  - bandwidth (1 core) ~ 15 GB/s
  - compare to L1 access: latency 2-3 ticks, bandwidth ~150GB/s
- Bad memory performance easily can dominate performance (better memory performance also will speed up parallel code)

Topic of this Morning: Bottleneck Memory

Still getting more important

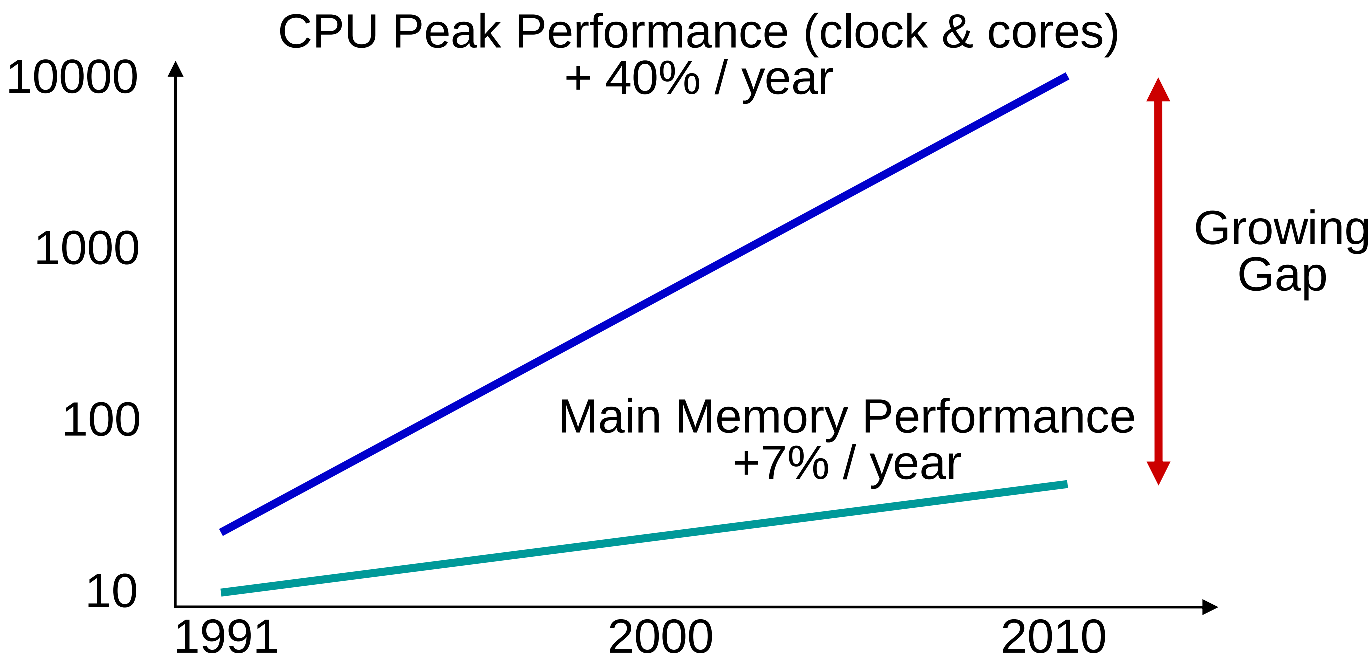
- compute power on one chip still increases
- main memory latency will stay (off-chip distance)
- bandwidth increases, but not as much as compute power

➔ Memory Wall (stated already in 1994)

In addition:

- with multi-core, cores share connection to main memory!

The Memory Wall



Access latency to main memory today up to 300 cycles

Assume 2 Flops/clock ticks ➔ 600 Flops wasted while waiting for one main memory access!

Topic of this Morning: Bottleneck Memory

- Further getting more important not only for performance, but
- for problem no.1 in the future: power consumption (Power Wall)
  - reason that we have multi-core today
  - most significant cost factor for compute centers in the future
  - users not to be charged by hours, but by energy consumption?

- Comparison computation vs. memory access [Dongarra, PPAM 2011]
  - DP FMA: 100 pJ (today) ➔ 10 pJ (estimation 2018)
  - DP Read DRAM: 4800 pJ (today) ➔ 1920 pJ (estimation 2018)
- today: for 1 memory access saved, can do 48 FMAs more
  2018: 192 FMAs more
- solution (?): do redundant calculation to avoid memory access

Outline: Part 1

The Memory Hierarchy

Caches: Why & How do they work?

Bad Memory Access Patterns

How to not exploit Caches

Cache Optimization Strategies

How to exploit Caches even better

Outline: Part 2

Cache Analysis

- Measuring on real Hardware vs. Simulation

Cache Analysis Tools

Case Studies

Hands-on

The Memory Hierarchy

Two facts of modern computer systems

- processor cores are quite fast
- main memory is quite slow

Why? Different design goals

- everybody wants a fast processor
- everybody wants large amounts of cheap memory

Why is this not a contradiction? There is a solution to bridge the gap:

- a hierarchy of buffers between processor and main memory
- often effective, and gives seemingly fast and large memory

Solution: The Memory Hierarchy

We can build very fast memory (for a processor), but

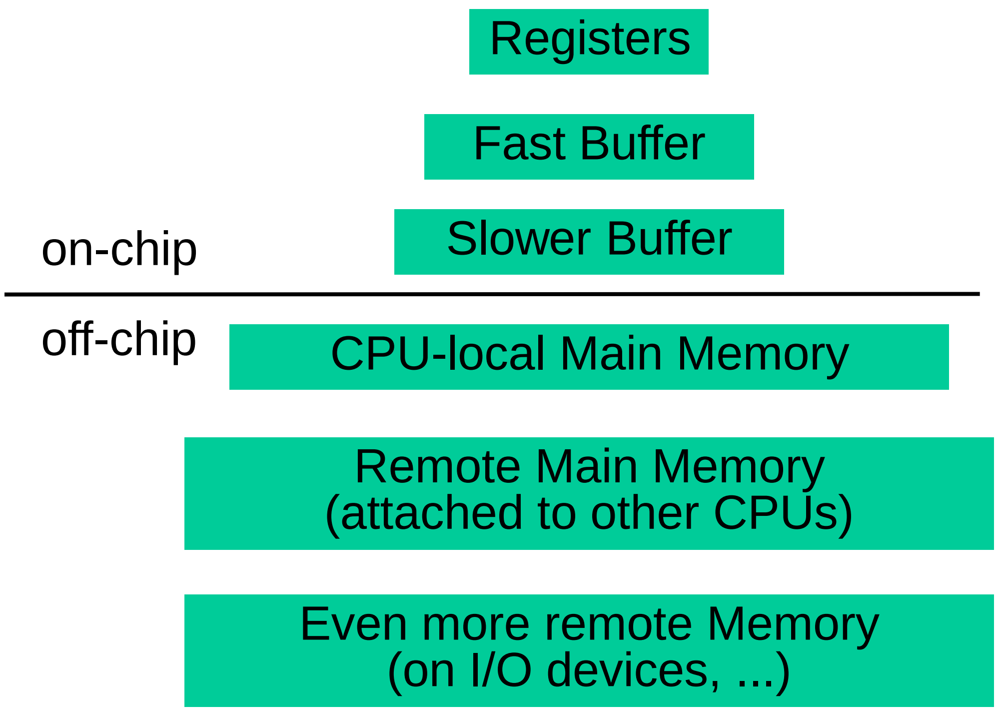
- it has to be small (only small number of cascading gates)
  - tradeoff: buffer size vs. buffer speed
- it has to be near (where data is to be used)
  - on-chip, not much space around execution units
- it will be quite expensive (for its size)
  - SRAM needs a lot more energy and space than DRAM

➔ use fast memory only for data most relevant to performance

➔ if less relevant, we can afford slower access, allowing more space

➔ this works especially well if “most relevant data” fits into fast buffer

Solution: The Memory Hierarchy



Size	Latency	Bandwidth
300 B	1	
32 kB	3	100 GB/s
4 MB	20	30 GB/s
4 GB	200	15 GB/s
4 GB	300	10 GB/s
1 TB	> 10^7	0,2 GB/s

Solution: The Memory Hierarchy

Programmers want memory to be a flat space

- registers not visible, used by compilers
- on-chip buffers are
  - not explicitly accessed, but automatically filled from lower levels
  - indexed by main memory address
  - hold copies of blocks of main memory

  ➔ not visible to programmers: caches
- transparent remote memory access provided by hardware
- extension on I/O devices by MMU & OS

Let’s concentrate on Processor Caches…

Solution: Processor Caches

Why are Caches effective? Because typical programs

- often access same memory cells repeatedly
  - temporal locality ➔ good to keep recent accessed data in cache
- often access memory cells near recent accesses
  - spatial locality ➔ good to work on blocks of nearside data (cache line)

"Principle of Locality"

So what's about the Memory Wall?

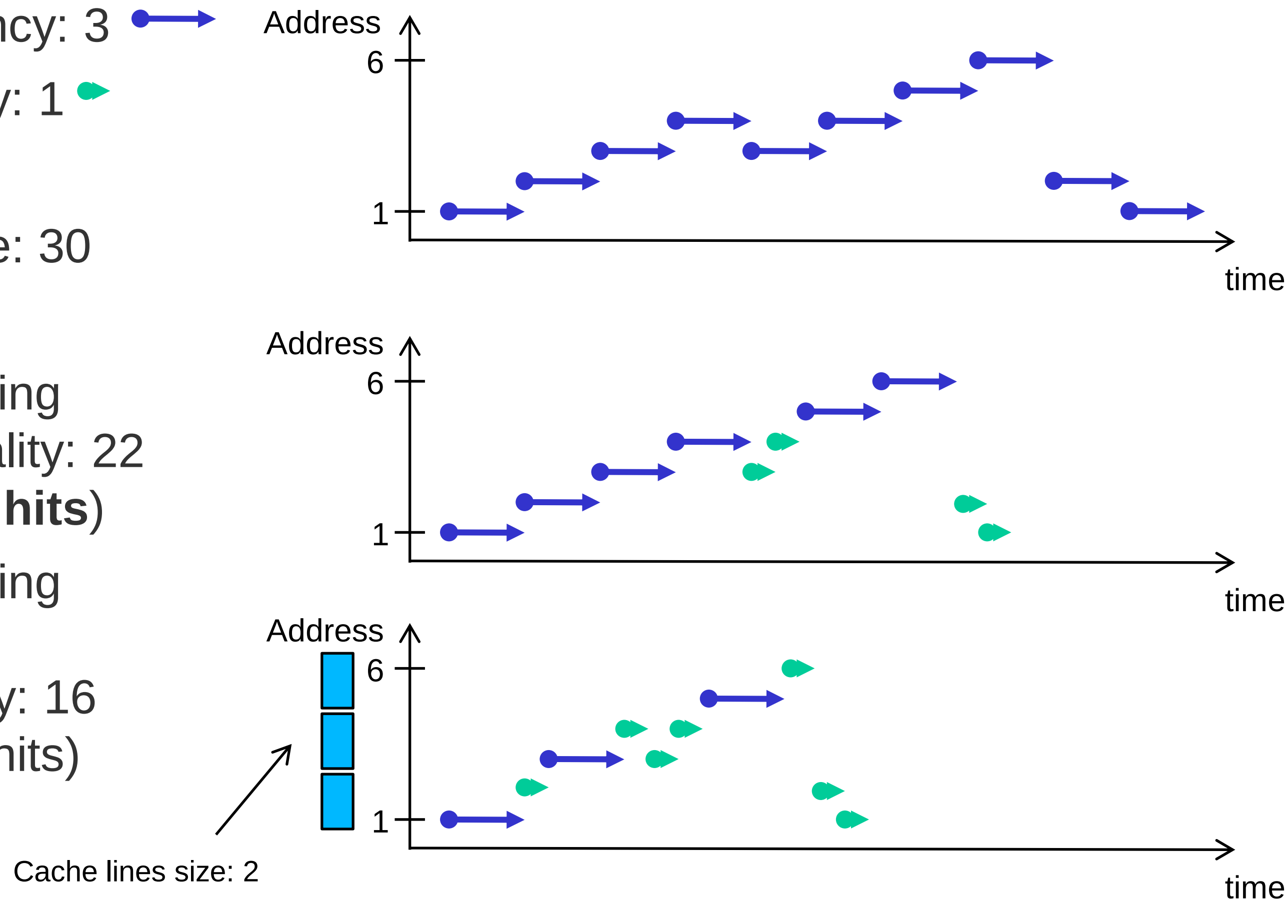
- the degree of "locality" depends on the application
- at same locality, the widening gap between processor and memory performance reduces cache effectiveness

Example: Sequence with 10 Accesses

- memory latency: 3
- cache latency: 1

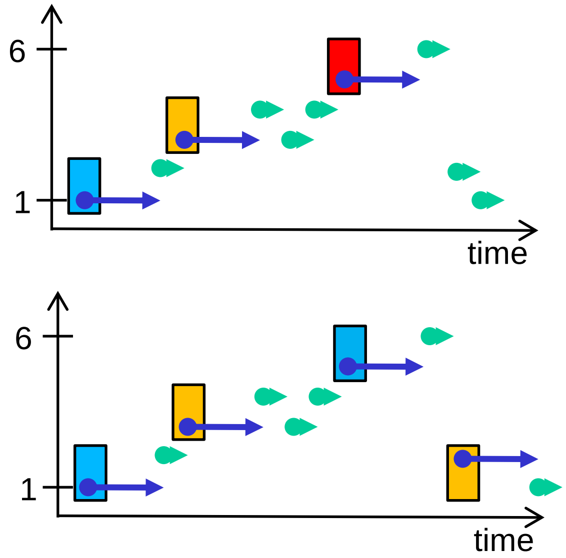
- without cache: 30

- cache exploiting temporal locality: 22 (6 **misses**, 4 **hits**)
- cache exploiting temporal and spatial locality: 16 (3 misses, 7 hits)



Basic Cache Properties (1)

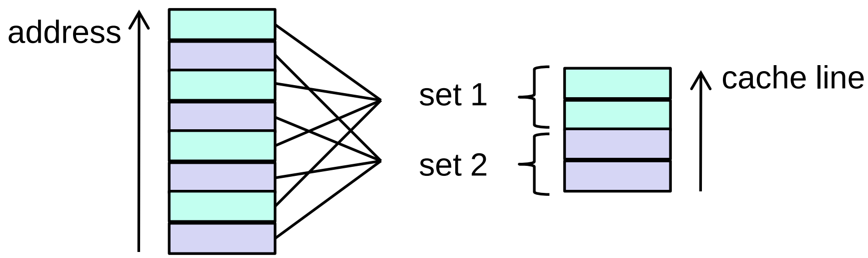
- Cache holds copies of memory blocks
  - space for one copy called “cache line” ➔ **Cache Line Size**
  - transfers from/to main memory always at line size granularity
- Cache has restricted size: **Cache Size**
  - line size 2, cache size 6 (= 3 lines)
  - line size 2, cache size 4 (=2 lines)



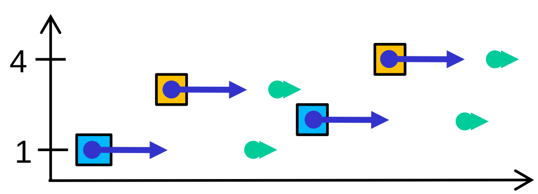
- Which copy to evict for new copy
  - **Replacement Policy**
  - Typically: Evict Least Recently Used (LRU)

Basic Cache Properties (2)

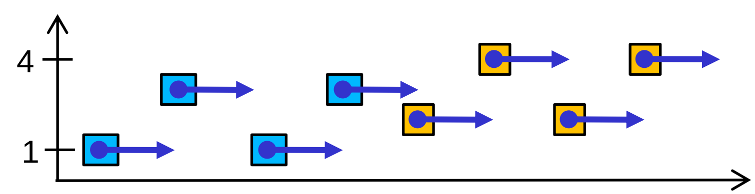
- every cache line knows the memory address it has a copy of („tag“)
- comparing all tags at every access ➔ expensive (space & energy)
- better: reduce number of comparisons per access
  - group cache lines into sets
  - a given address can only be stored into a given set
  - lines per set: **Associativity**



- example: 2 lines (■ ■) , sequence 1/3/1/3/2/4/2/4



associativity 2 („full“)

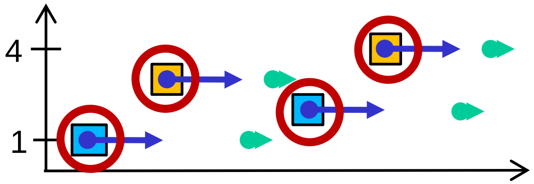


associativity 1 („direct mapped“): even ■ odd ■

Solution: Processor Caches

The "Principle of Locality" makes caches effective

- How to improve on that?
- Try to further reduce misses!



Options

- increase cache line size!
  - can reduce cache effectiveness, if not all bytes are accessed
- predict future accesses (hardware prefetcher), load before use
  - example: stride detectors (more effective if keyed by instruction)
  - allows "burst accesses" with higher netto bandwidth
  - only works if bandwidth not exploited anyway (demand vs. speculative)
  - can increase misses if prefetching is too aggressive

The Memory Hierarchy on Multi-Core

Principle of Locality often holds true across multiple threads

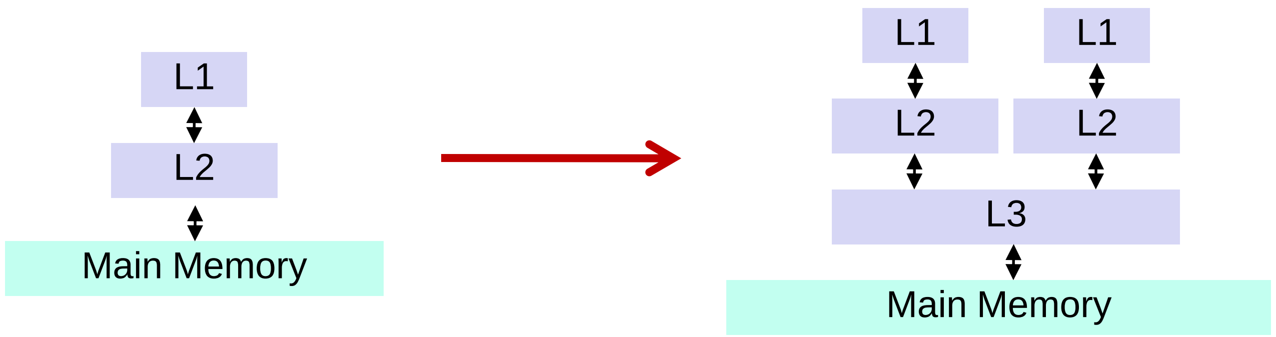
- example: threads need same vectors/matrices
- caches shared among cores can be beneficial
- sharing allows threads to prefetch data for each other

However, if threads work on different data…

- example: disjunct partitioning of data among threads
- threads compete for space, evict data of each other
- trade-off: only use cache sharing on largerst on-chip buffer

The Memory Hierarchy on Multi-Core

Typical example (modern Intel / AMD processors)



Why are there 3 levels?

- cache sharing increases on-chip bandwidth demands by cores
- L1 is very small to be very fast ➔ still lots of references to L2
- private L2 caches reduce bandwidth demands for shared L3

Caches and Multi-Processor Systems

The Cache Coherence Problem

- suppose 2 processors/cores with private caches at same level
- P1 reads a memory block X
- P2 writes to the block X
- P1 again reads from block X (which now is invalid!)

A strategy is needed to keep caches coherent

- writing to X by P2 needs to invalidate or update copy of X in P1
- cache coherence protocol
- all current multi-socket/-core systems have fully automatic cache coherence in hardware (today already a significant overhead!)

Outline: Part 1

The Memory Hierarchy

Caches: Why & How do they work?

Bad Memory Access Patterns

How to not exploit Caches

Cache Optimization Strategies

How to exploit Caches even better

Memory Access Behavior

How to characterize good memory access behavior?

Cache Hit Ratio

- percentage of accesses which was served by the cache
- good ratio: > 97%

Symptoms of bad memory access: Cache Misses

Let's assume that we can not change the hardware as countermeasure for cache misses (e.g. enlarging cache size)

Memory Access Behavior: Cache Misses

Classification:

- cold / compulsory miss
  - first time a memory block was accessed
- capacity miss
  - recent copy was evicted because of too small cache size
- conflict miss
  - recent copy was evicted because of too low associativity
- concurrency miss
  - recent copy was evicted because of invalidation by cache coherence protocol
- prefetch inaccuracy miss
  - recent copy was evicted because of aggressive/imprecise prefetching

Bad Memory Access Behavior (1)

Lots of cold misses

- each memory block only accessed once, and
- prefetching not effective because accesses are not predictable or bandwidth is fully used
- usually not important, as programs access data multiple times
- can become relevant if there are lots of context switches (when multiple processes synchronize very often)
  - L1 gets flushed because virtual addresses get invalid

Bad Memory Access Behavior (2)

Lots of capacity misses

- blocks are only accessed again after eviction due to limited size
  - number of other blocks accessed in-between (= reuse distance) > number of cache lines
  - example: sequential access to data structure larger than cache size
- and prefetching not effective

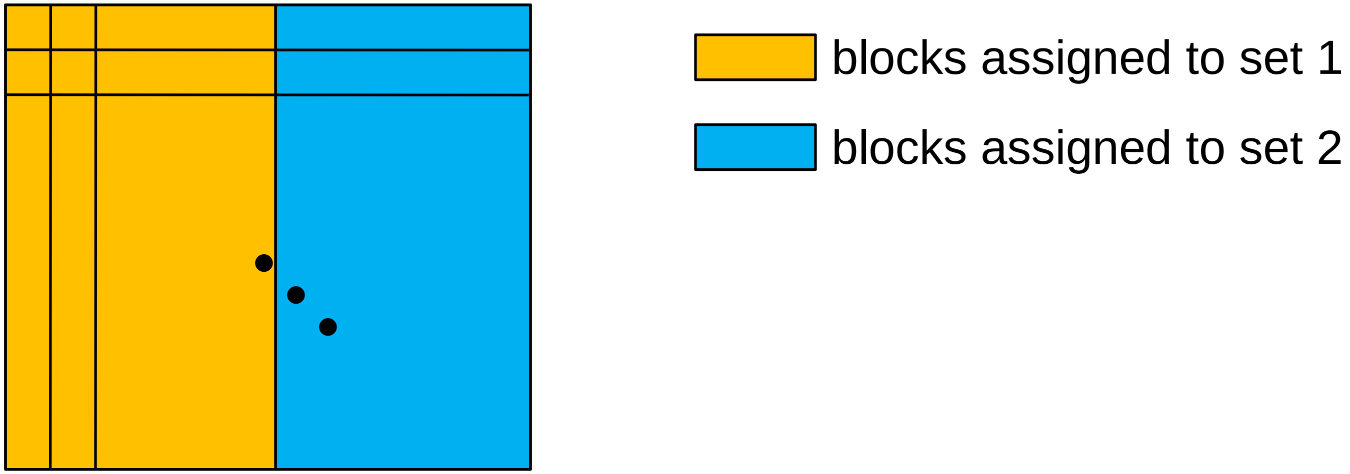
Countermeasures

- reduce reuse distance of accesses = increase temporal locality
- improve utilization inside cache lines = increase spatial locality
- do not share cache among threads accessing different data
- increase predictability of memory accesses

Bad Memory Access Behavior (3)

Lots of conflict misses

- blocks are only accessed again after eviction due to limited set size
- example:
  - matrix where same column in multiple rows map to same set
  - we do a column-wise sweep



Bad Memory Access Behavior (3)

Lots of conflict misses

- blocks are only accessed again after eviction due to limited set size

Countermeasures

- set sizes are similar to cache sizes: see last slide…
- make successive accesses cross multiple sets

Bad Memory Access Behavior (4)

Lots of concurrency misses

- lots of conflicting accesses to same memory blocks by multiple processors/cores, which use private caches
  - “conflicting access”: at least one processor is writing

Two variants: same block is used

- because processors access same data
- even though different data are accessed, the data resides in same block (= false sharing)
  - example: threads often write to nearside data
    (e.g. using OpenMP dynamic scheduling)

Bad Memory Access Behavior (4)

Lots of concurrency misses

- lots of conflicting accesses to same memory blocks by multiple processors/cores, which use private caches

Countermeasures

- reduce frequency of accesses to same block by multiple threads
- move data structures such that data accessed by different threads reside on their own cache lines
- place threads to use a shared cache

Bad Memory Access Behavior (5)

Lots of prefetch inaccuracy misses

- much useful data gets evicted due to misleading access patterns
- example: prefetchers typically “detect” stride pattern after 3-5 regular accesses, prefetching with distance 3-5
  - frequent sequential accesses to very small ranges (5-10 elements) of data structures

Countermeasures

- use longer access sequences with strides
- change data structure if an access sequence accidently looks like a stride access

Memory Access Behavior: Cache Misses

Classifications:

- kind of misses
- each cache miss needs another line to be evicted:
  is the previous line modified (= dirty) or not?
  - yes: needs write-back to memory
  - increases memory access latency

Outline: Part 1

The Memory Hierarchy

Caches: Why & How do they work?

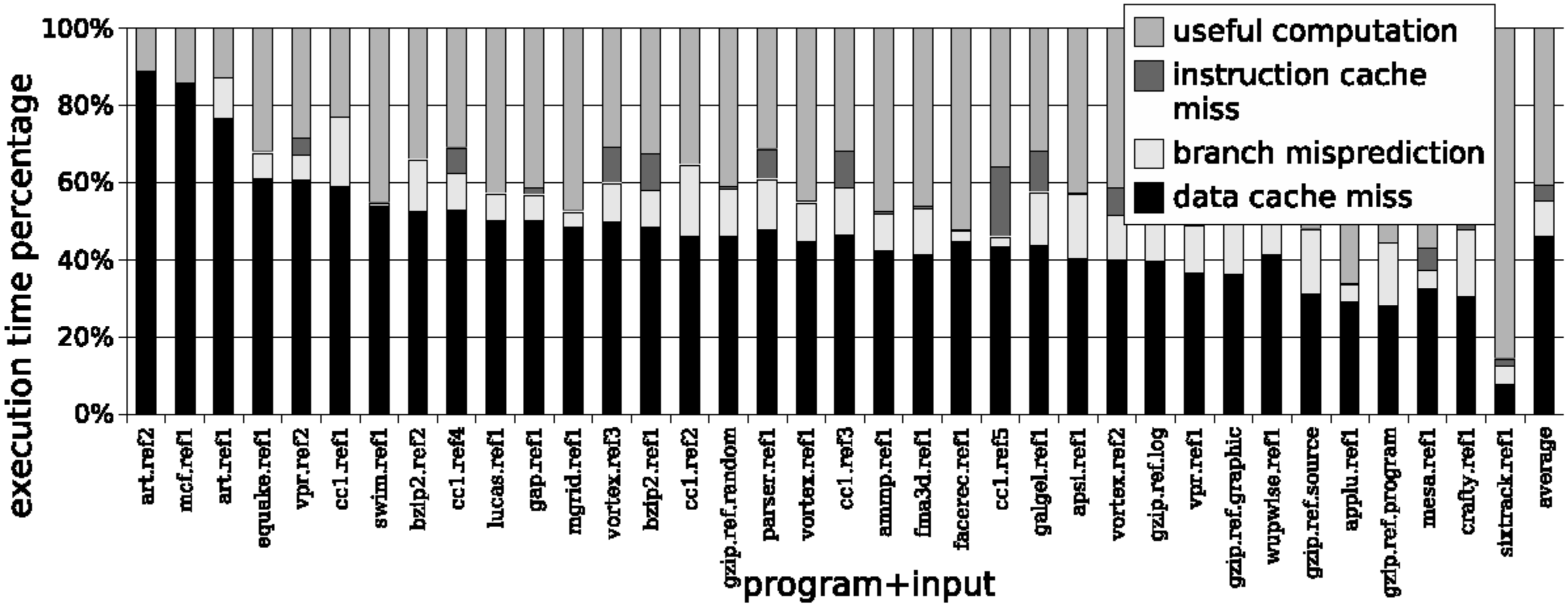
Bad Memory Access Patterns

How to not exploit Caches

Cache Optimization Strategies

How to exploit Caches even better

The Principle of Locality is not enough...



Reasons for Performance Loss for SPEC2000
[Beyls/Hollander, ICCS 2004]

Basic efficiency guidelines

Always use a performance analysis tool before doing optimizations:
How much time is wasted where because of cache misses?

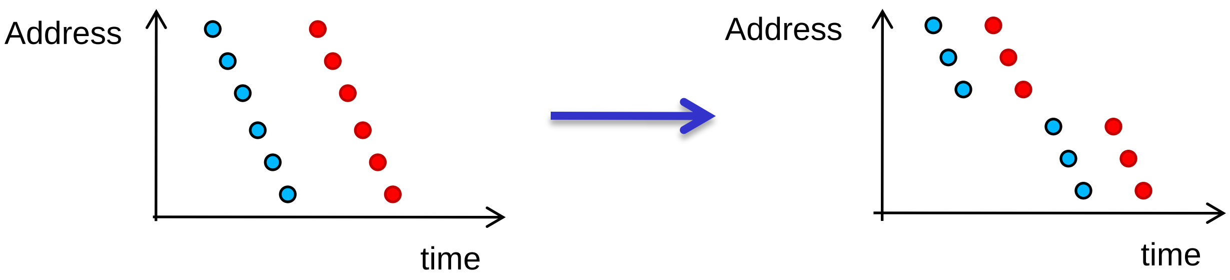
1. Choose the best algorithm
2. Use efficient libraries
3. Find good compiler and options (“-O3”, “-fno-alias” ...)
4. Reorder memory accesses
5. Use suitable data layout
6. Prefetch data

(Items 4–6: Cache Optimizations)

Warning: Conflict and capacity misses are not easy to distinguish...

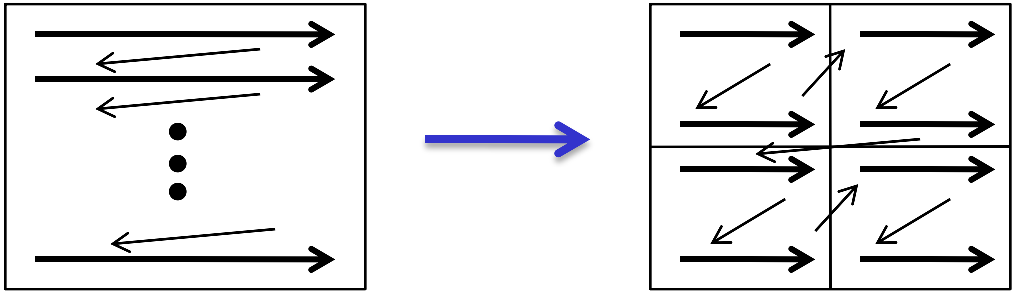
Cache Optimization Strategies: Reordering Accesses

- Blocking: make arrays fit into a cache



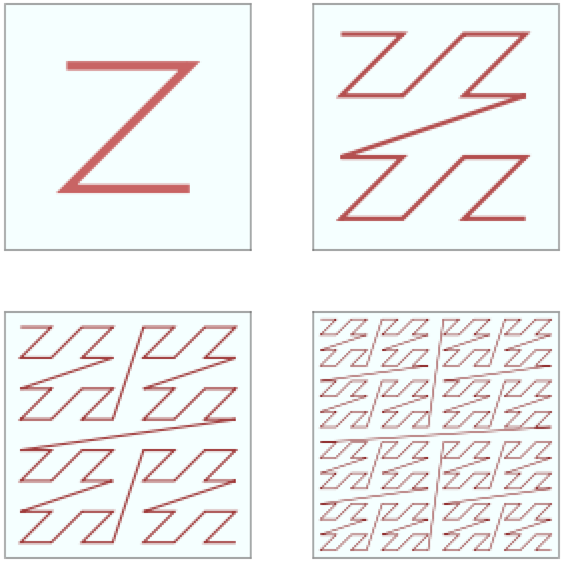
Cache Optimization Strategies: Reordering Accesses

- Blocking: make arrays fit into a cache
- Blocking in multiple dimensions (example: 2D)



Cache Optimization Strategies: Reordering Accesses

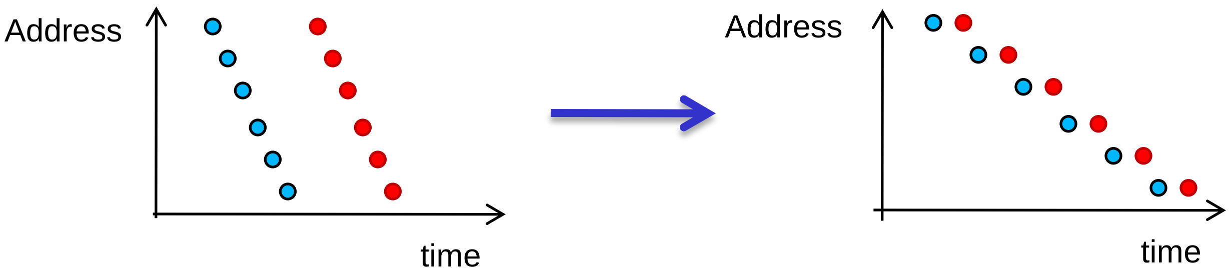
- Blocking: make arrays fit into a cache
- Blocking in multiple dimensions (example: 2D)
- Nested blocking: tune to multiple cache levels
  - can be done recursively according to a space filling curve
  - example: Morton curve (without “jumps”: Hilbert, Peano…)
  - cache-oblivious orderings/algorithms (= automatically fit to varying levels and sizes using the same code)



[http://en.wikipedia.org/wiki/Z-order_curve]

Cache Optimization Strategies: Reordering Accesses

- Extreme blocking with size 1: Interweaving



  - combined with blocking in other dimenions, results in pipeline patterns
  - On multi-core: consecutive iterations on cores with shared cache

- Block Skewing:
  Change traversal order over non-rectangular shapes

- For all reorderings: preserve data dependencies of algorithm !

Cache Optimization Strategies: Suitable Data Layout

Strive for best spatial locality

- use compact data structures
  (arrays are almost always better than linked lists!)
- data accessed at the same time should be packed together
- avoid putting frequent and rarely used data packed together
- object-oriented programming
  - try to avoid indirections
  - bad: frequent access of only one field of a huge number of objects
  - use proxy objects, and structs of arrays instead of arrays of structs
- best layout can change between different program phases
  - do format conversion if accesses can become more cache friendly
  - (also can be important to allow for vectorization)

Cache Optimization Strategies: Prefetching

Allow hardware prefetcher to help loading data as much as possible

- make sequence of memory accesses predictable
  - prefetchers can detect multiple streams at the same time (>10)
- arrange your data accordingly in memory
- avoid non-predictable, random access sequences
  - pointer-based data structures without control on allocation of nodes
  - hash tables accesses

Software controlled prefetching (difficult !)

- switch between block prefetching & computation phases
- do prefetching in another thread / core („helper thread“)

Countermeasures for Capacity Misses

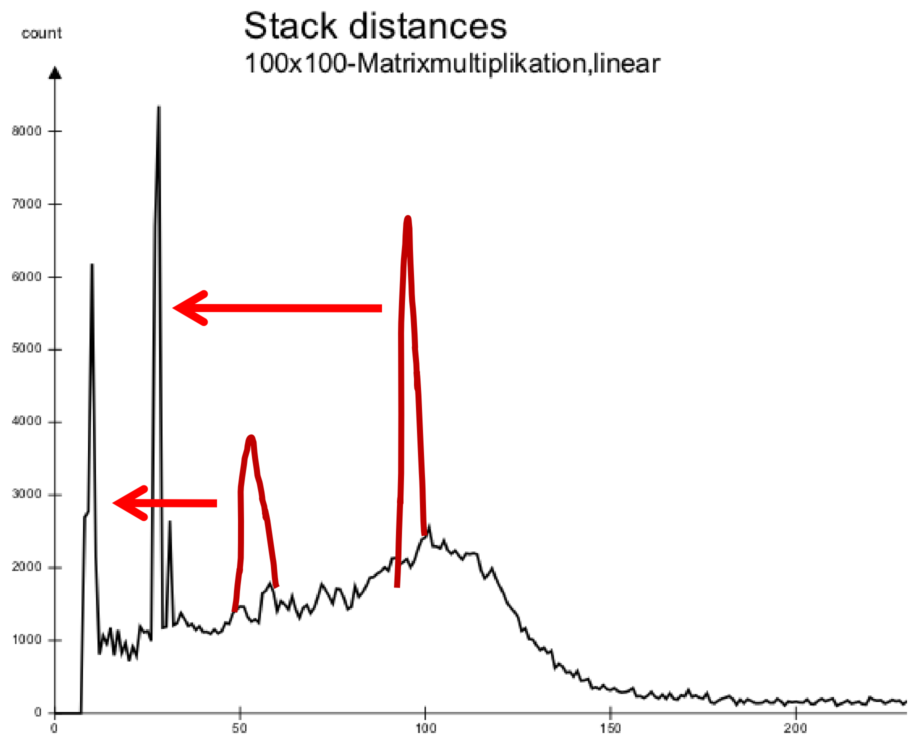
Reduce reuse distance of accesses = increase temporal locality

Strategy:

- blocking

Effectiveness can be seen by

- reduced number of misses
- in reuse distance histogram (needs cache simulator)



Countermeasures for Capacity Misses

Improve utilization inside cache lines = increase spatial locality

Strategy:

- improve data layout

Effectiveness can be seen by

- reduced number of misses
- spatial loss metric (needs cache simulator)
  - counts number of bytes fetched to a given cache level but never actually used before evicted again
- spatial access homogenity (needs cache simulator)
  - variance among number of accesses to bytes inside of a cache line

Countermeasures for Capacity Misses

Do not share cache among threads accessing different data

Strategy:

- explicitly assign threads to cores
- “sched_setaffinity” (automatic system-level tool: autopin)

Effectiveness can be seen by

- reduced number of misses

Countermeasures for Capacity Misses

Increase predictability of memory accesses

Strategy:

- improve data layout
- reorder accesses

Effectiveness can be seen by

- reduced number of misses
- performance counter for hardware prefetcher
- run cache simulation with/without prefetcher simulation

Countermeasures for Conflict Misses

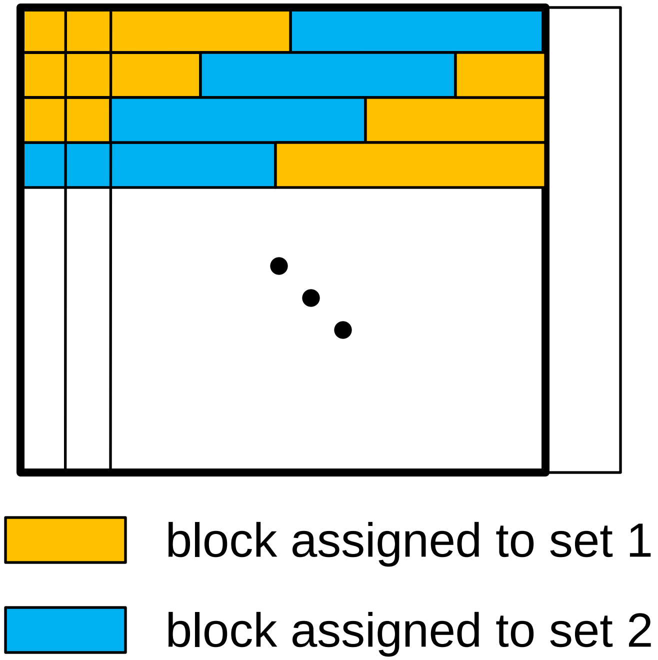
Make successive accesses cross multiple cache sets

Strategy:

- change data layout by Padding
- reorder accesses

Effectiveness can be seen by

- reduced number of misses



Countermeasures for Concurrency Misses

Reduce frequency of accesses to same block by multiple threads

Strategy:

- for true data sharing: do reductions by partial results per thread
- for false sharing (reduce frequency to zero = data accessed by different threads reside on their own cache lines)
  - change data layout by padding (always possible)
  - change scheduling (e.g. increase OpenMP chunk size)

Effectiveness can be seen by

- reduced number of concurrency misses (there is a perf. counter)

Countermeasures for Misses triggering Write-Back

Only general rule:

- Try to avoid writing if not needed

Sieve of Eratosthenes:

```
isPrim[*] = 1;
for(i=2; i<n/2; i++)
  if (isPrim[i] == 1)
    for(j=2*i; i<n; j+=i)
      isPrim[j] = 0;
```

~ 2x faster (!):

```
isPrim[*] = 1;
for(i=2; i<n/2; i++)
  if (isPrim[i] == 1)
    for(j=2*i; i<n; j+=i)
      if (isPrim[j]==1)
        isPrim[j] = 0;
```

Outline: Part 2

Cache Analysis

- Measuring on real Hardware vs. Simulation

Cache Analysis Tools

Case Studies

Hands-on

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“: histogram of events happening at given code positions
- inclusive vs. exclusive cost

How to measure Events (1)

Where?

- on 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
- using 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 have pro & contra, 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 every N-th event
  - HW notifies only about every N-th event ➔ Influence tunable by N

Outline: Part 2

Cache Analysis

Measuring on real Hardware vs. Simulation

Cache Analysis Tools

Case Studies

Hands-on

Analysis Tools

- GProf
  - Instrumentation by compiler for call relationships & call counts
  - Statistical time sampling using timers
  - Pro: available almost everywhere (gcc: -pg)
  - Contra: recompilation, measurement overhead, heuristic
- Intel VTune (Sampling mode) / Linux Perf (>2.6.31)
  - Sampling using hardware performance counters, no instrumentation
  - Pro: minimal overhead, detailed counter analysis possible
  - Contra: call relationship can not be collected
    (this is not about call stack sampling: provides better context…)
- Callgrind: machine model simulation

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

- 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

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 reproducable) 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

Callgrind: Usage

- `valgrind –tool=callgrind [callgrind options] yourprogram args`
- cache simulator: `--cache-sim=yes`
- branch prediction simulation (since 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` / Macro
- spontaneous dump: `callgrind_control –d [dump identification]`
- current backtrace of threads (interactive): `callgrind_control –b`
- separate dumps per thread: `--separate-threads=yes`
- cache line utilization: `--cacheuse=yes`
- enable prefetcher simulation: `--simulate-hwpref=yes`
- jump-tracing in functions (CFG): `--collect-jumps=yes`

KCachegrind: Features

- open source, GPL, kcachegrind.sf.net
- included with KDE3 & KDE4

Visualization of

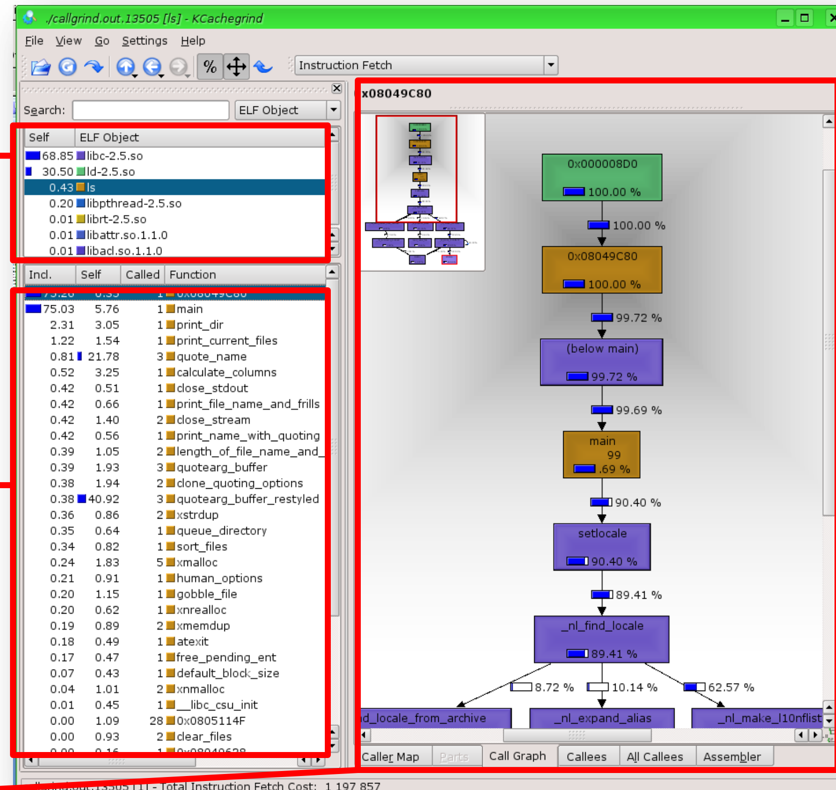
- call relationship of functions (callers, callees, call graph)
- exclusive/Inclusive cost metrics of functions
  - grouping according to ELF object / source file / C++ class
- source/assembly annotation: costs + CFG
- arbitrary events counts + specification of derived events

Callgrind support (file format, events of cache model)

KCachegrind: Usage

```
{k,q}cachegrind 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



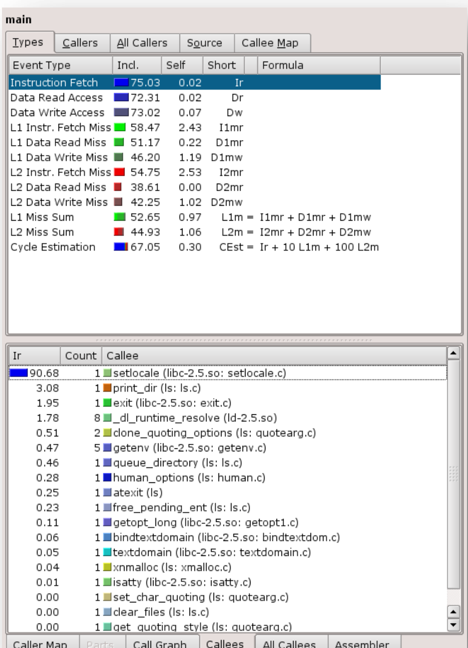
- right: visualization panes

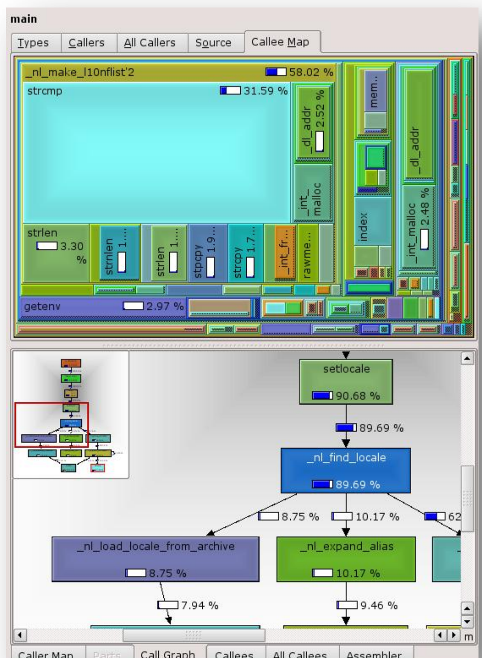
Visualization panes for selected function

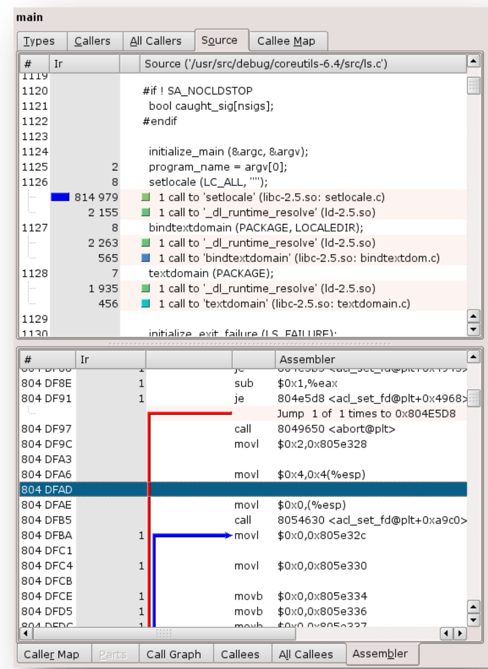
- List of event types
- List of callers/callees

- Treemap visualization
- Call Graph

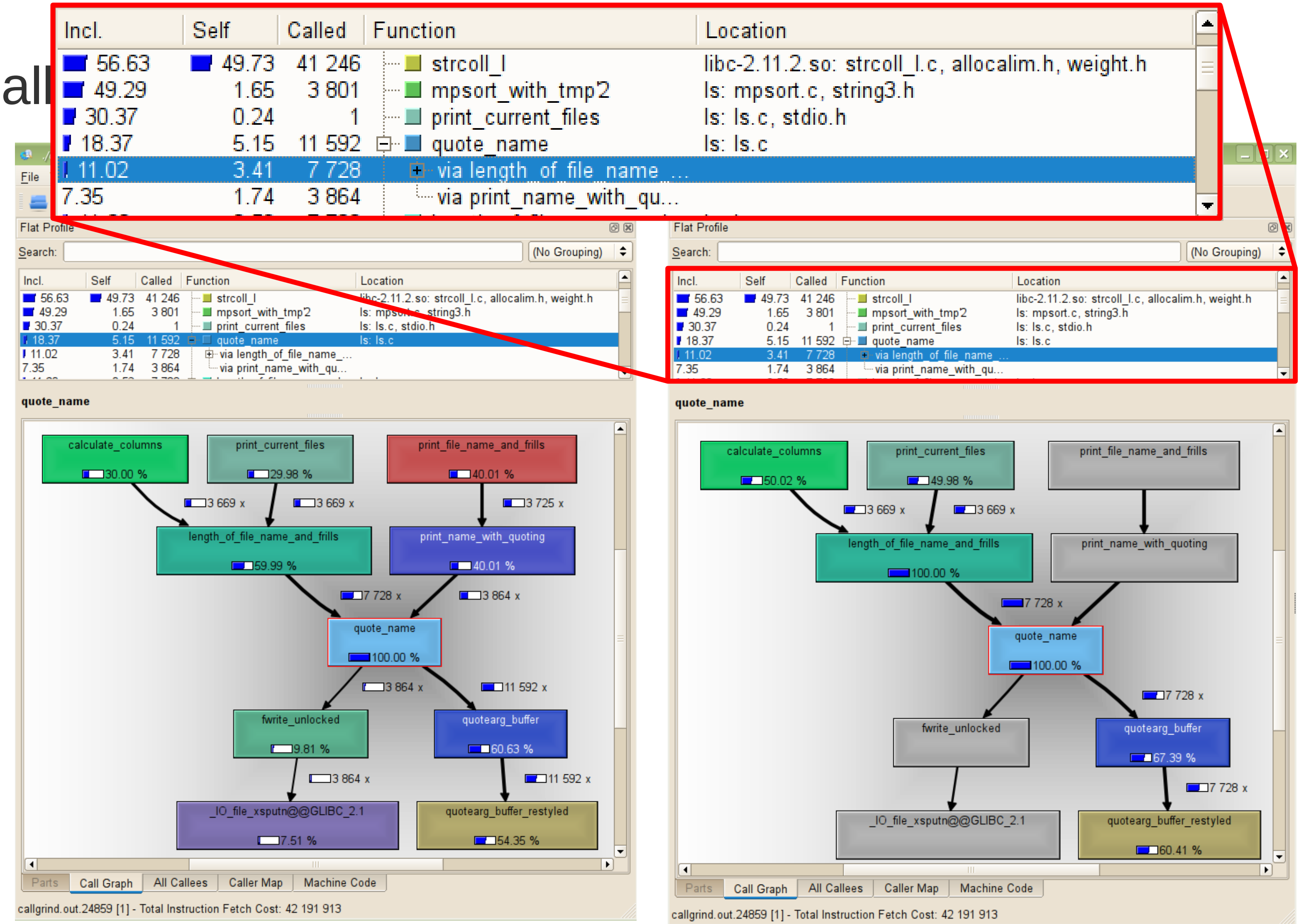
- Source annotation
- Assemly annotation







Call


Incl. Self Called Function Location
56.63 49.73 41 246 strcoll_l libc-2.11.2.so: strcoll_l.c, allocalim.h, weight.h
49.29 1.65 3 801 mpsort_with_tmp'2 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...
File
Flat Profile
Search:
(No Grouping)
quote_name
calculate_columns
30.00 %
print_current_files
29.98 %
print_file_name_and_frills
40.01 %
3 669 x
3 669 x
3 725 x
length_of_file_name_and_frills
59.99 %
print_name_with_quoting
40.01 %
7 728 x
3 864 x
quote_name
100.00 %
3 864 x
11 592 x
fwrite_unlocked
9.81 %
quotearg_buffer
60.63 %
3 864 x
11 592 x
_IO_file_xsputn@@GLIBC_2.1
7.51 %
quotearg_buffer_restyled
54.35 %
Parts
Call Graph
All Callees
Caller Map
Machine Code
callgrind.out.24859 [1] - Total Instruction Fetch Cost: 42 191 913
calculate_columns
50.02 %
print_current_files
49.98 %
print_file_name_and_frills
3 669 x
3 669 x
length_of_file_name_and_frills
100.00 %
print_name_with_quoting
7 728 x
quote_name
100.00 %
7 728 x
fwrite_unlocked
quotearg_buffer
67.39 %
7 728 x
_IO_file_xsputn@@GLIBC_2.1
quotearg_buffer_restyled
60.41 %
callgrind.out.24859 [1] - Total Instruction Fetch Cost: 42 191 913

Outline: Part 2

Cache Analysis

Measuring on real Hardware vs. Simulation

Cache Analysis Tools

Case Studies

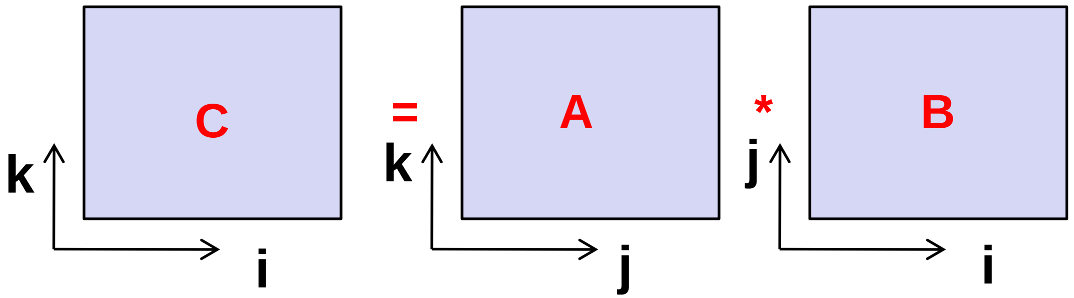
Hands-on

Case Studies

- Get ready for hands-on
  - matrix multiplication
  - 2D relaxation

Matrix Multiplication

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



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

Matrix Multiplication

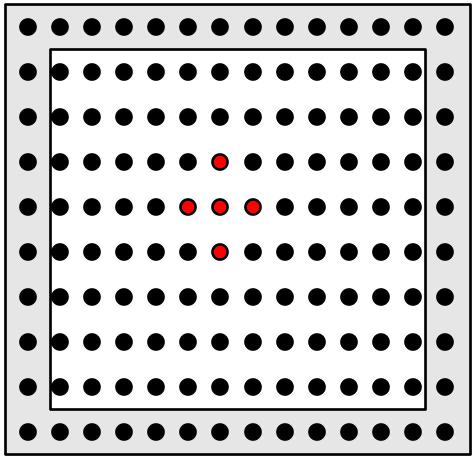
- Kernel for C = A * B
  - 3 nested loops (i,j,k): What is the best index order? Why?

```
for(i=0;i<N;i++)
  for(j=0;j<N;j++)
    for(k=0;k<N;k++)
      c[k][i] = a[k][j] * b[j][i]
```

  - blocking for all 3 indexes, block size B, N multiple of B

```
for(i=0;i<N;i+=B)
 for(j=0;j<N;j+=B)
  for(k=0;k<N;k+=B)
   for(ii=i;ii<i+B;ii++)
    for(jj=j;jj<j+B;jj++)
     for(kk=k;kk<k+B;kk++)
      c[k+kk][i+ii] =
       a[k+kk][j+jj] * b[j+jj][i+ii]
```

Iterative Solver for PDEs: 2D Jacobi Relaxation



Example: Poisson

One iteration:

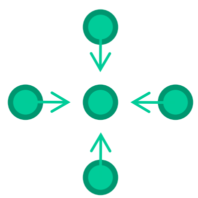
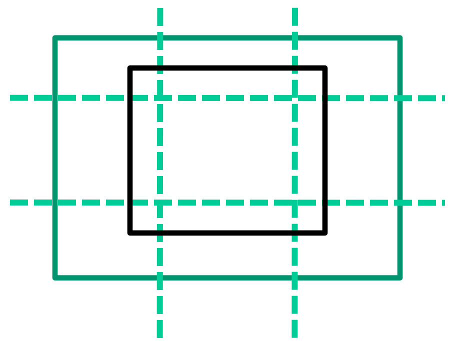
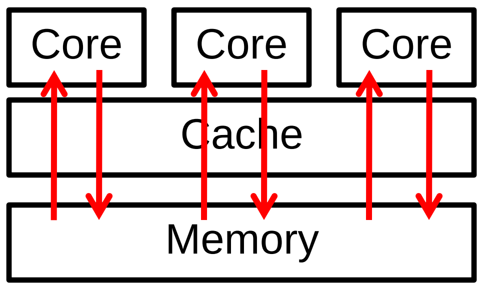
```
for(i=1;i<N-1;i++)
  for(j=1;j<N-1;j++)
    u2[i][j] = ( u[i-1][j] +
                 u[i][j-1] +
                 u[i+1][j] +
                 u[i][j+1]   )/4.0;
u[*][*] = u2[*][*];
```

Optimization: Interleave 2 iterations

- iteration 1 for row 1
- iteration 1 for row 2, iteration 2 for row 1
- iteration 1 for row 3, iteration 2 for row 2
- …

2D Jacobi: Parallel Version

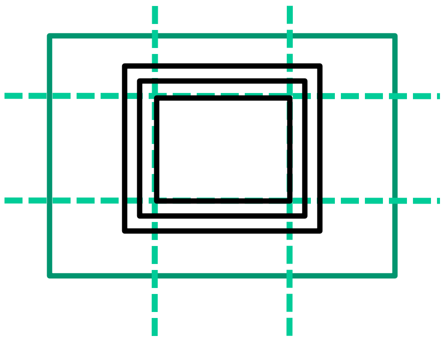
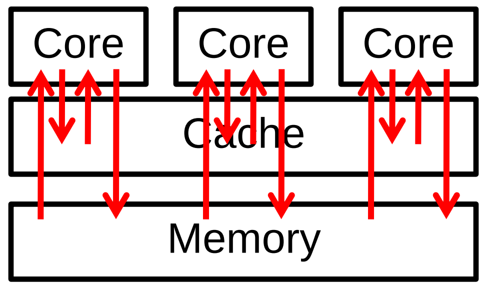
- Base version: 1 layer of ghost cells

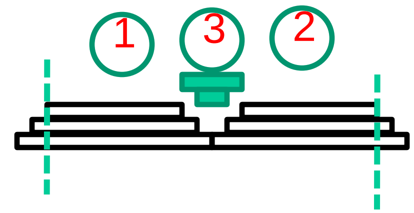




  - update: average of 4 neighbours, 4 Flops/update
  - bandwidth requirement: 16 bytes/update
  - memory-bound: 2 cores already occupy bus
  - on SuperMUC: 7 GFlop/s per node (2 sockets)

2D Jacobi: Cache Optimizations

- Spatial blocking of n iterations: n ghost layers

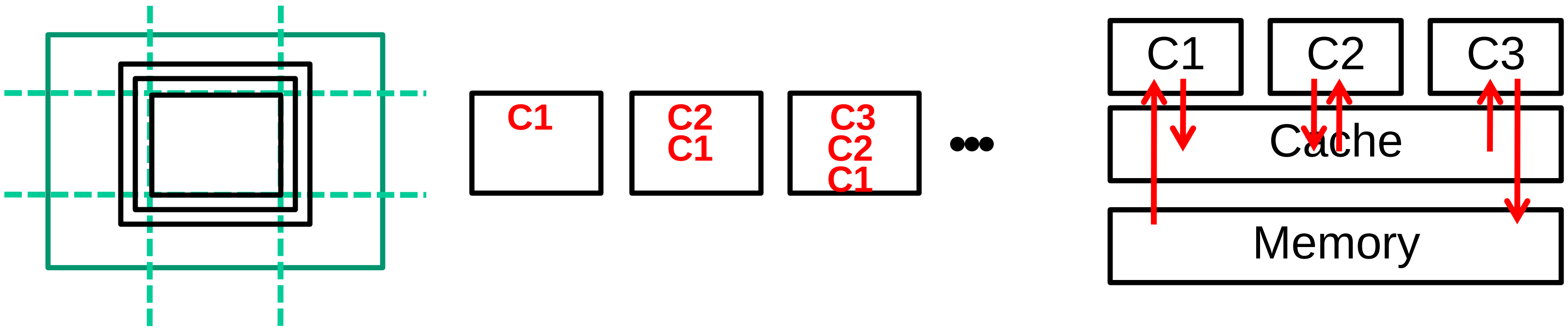
-  



- blocks fit into cache: update of inner borders ③
  reduced BW to memory ➔ better scalability
  – MPI: duplication of ghost layers, redundant computation
  – hybrid: less memory/BW, no redundant computation, enables cache-obliviousness (recursive bisection)

2D Jacobi: Cache Optimizations

- Wavefront: similar to blocking, use shared cache



- within multicore, may be combined with blk.
  - allows larger blocks, less border updates
  - not possible among MPI processes
    (matrix needs to be streamed trough cores)

Outline: Part 2

Cache Analysis

Measuring on real Hardware vs. Simulation

Cache Analysis Tools

Case Studies

Hands-on

How to run with MPI

- Run valgrind with mpirun (bt-mz: example from NAS)

  export OMP_NUM_THREADS=4

  mpirun -np 4 valgrind --tool=callgrind --cache-sim=yes \
  --separate-threads=yes ./bt-mz_B.4
- load all profile dumps at once:
  - run in new directory, “qcachegrind callgrind.out”

Getting started / Matrix Multiplication / Jacobi

- Try it out yourself (on intelnode)

  “cp -r /srv/app/kcachegrind/kcg-examples .”

  example exercises are in “exercises.txt”

- 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`**

?

Q&A

?