

# Analysis and Optimization of the Memory Access Behavior of Applications

October 8th, 2013

David Büttner (Josef Weidendorfer)

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



Fakultät für **Informatik**  
der Technischen Universität München  
Informatik X: Rechnerarchitektur 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
- Bachelor and Master in Informatics at the “Ruprecht-Karls-Universität” of Heidelberg
- Since 2010: working on my PhD
- Specialized in HPC (MPI, OpenMP, hybrid MPI-OpenMP)

## 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 can easily dominate performance (better memory performance will also speed up parallel code)

## Topic of this Morning: Bottleneck Memory

Still getting more important

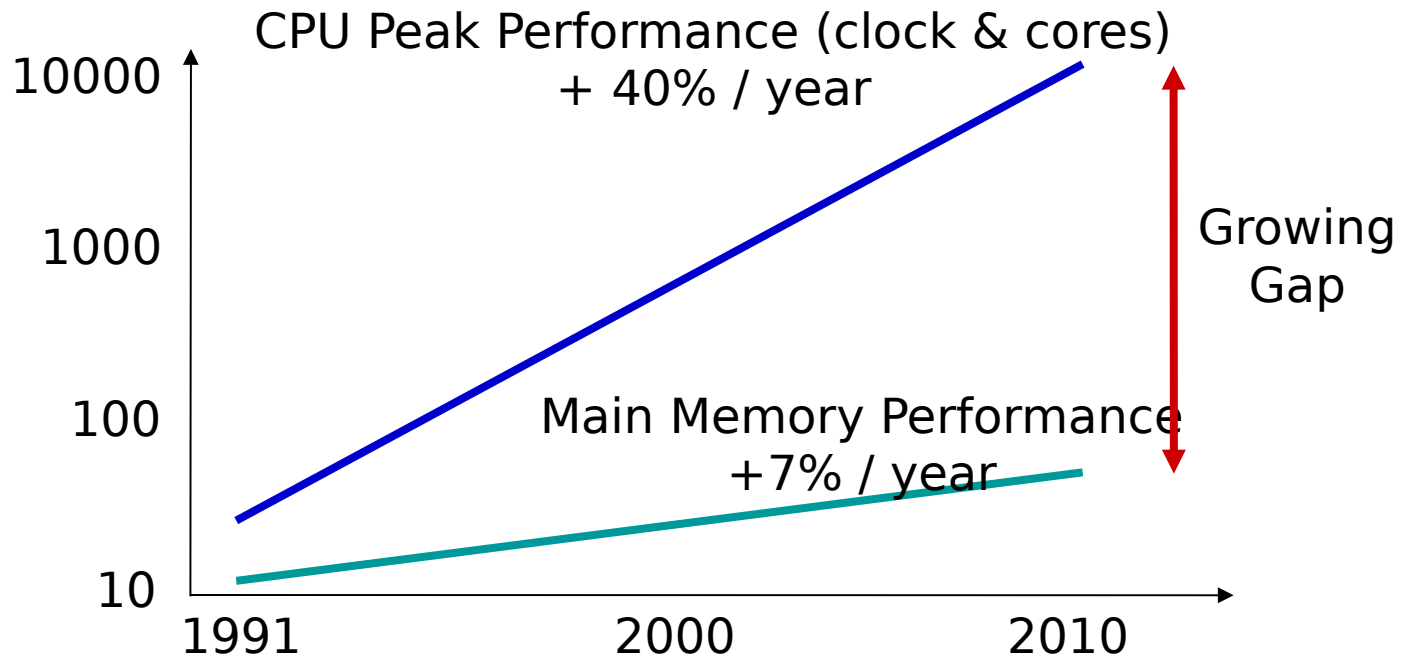
- compute power on one chip still increases
- main memory latency will stay (off-chip distance)
- bandwidth increases, but not as much as compute power

**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!



# Outline: Part 1

## **The Memory Hierarchy**

Caches: Why & How do they work?

## **Bad Memory Access Patterns**

How not to 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? The solution to bridging the gap:

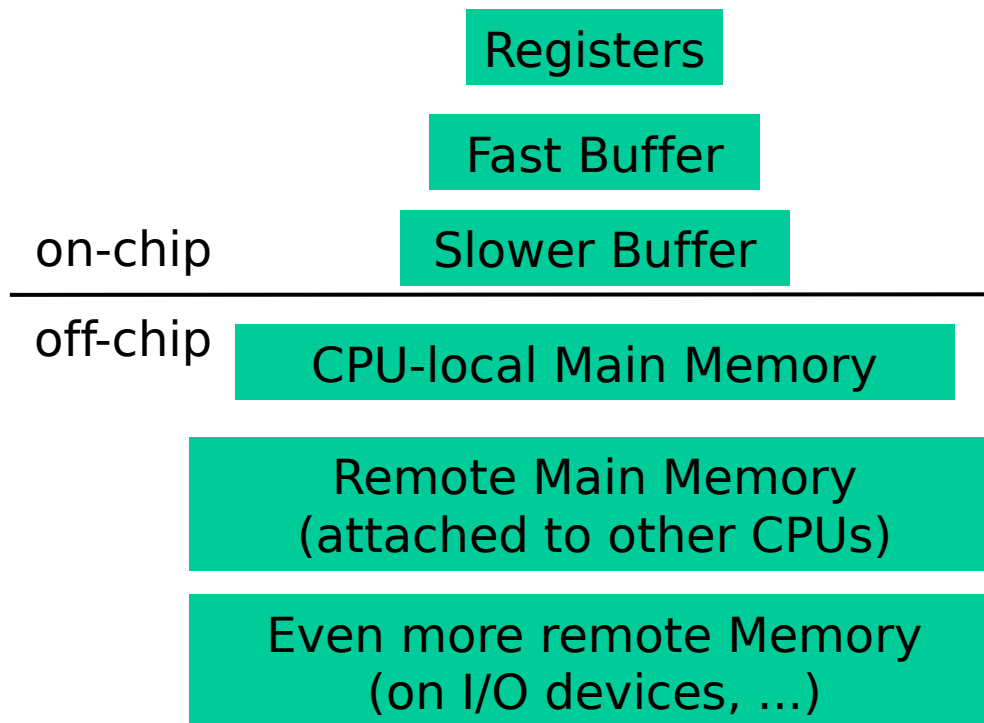
- a hierarchy of buffers between processor and main memory
- often effective, and gives seemingly fast and large memory

## 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: **cache**s
- 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)

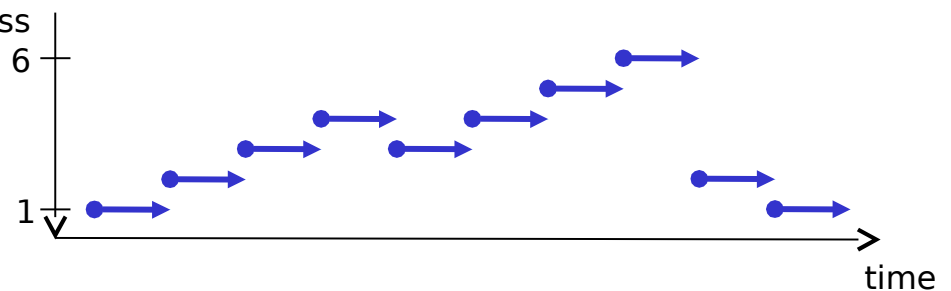
“Principle of Locality”

So what’s about the Memory Wall?

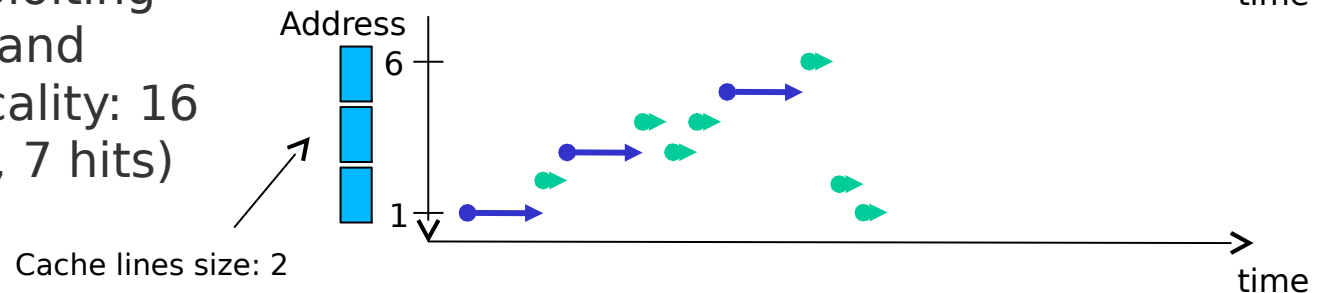
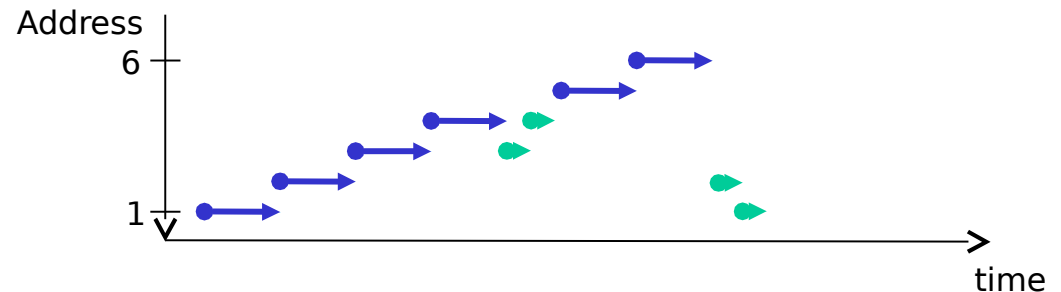
- the degree of “locality” depends on the application
- at same locality, the widening gap between processor and memory performance reduces cache effectiveness

## 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**)

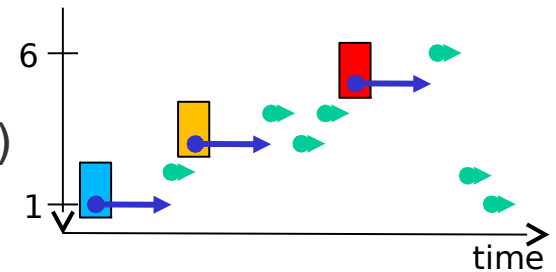


## Basic Cache Properties (1)

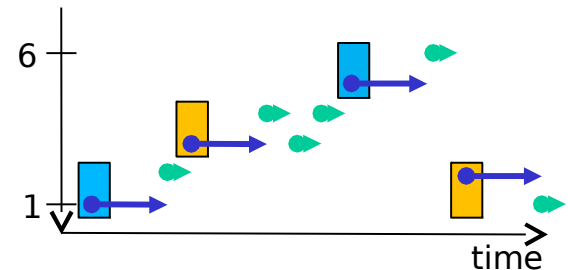
- Cache holds copies of memory blocks
  - space for one copy called “cache line”: **Cache Line Size**
  - transfers from/to main memory always at line size granularity

- Cache has restricted size: **Cache Size**

- line size 2, cache size 6 (= 3 lines  )



- line size 2, cache size 4 (=2 lines  )

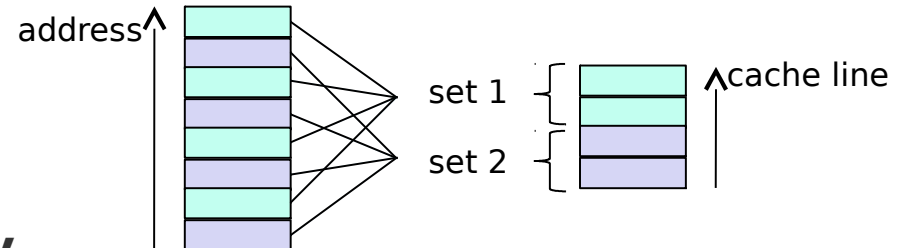


- Which copy to evict for new copy
  - **Replacement Policy**
  - Typically: Evict Least Recently Used (LRU)

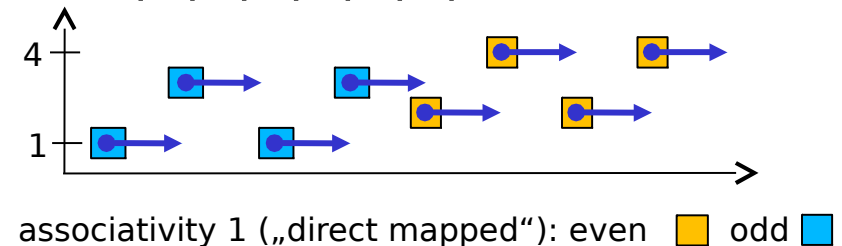
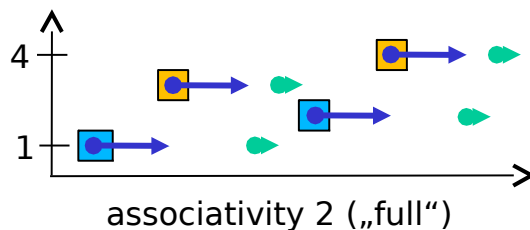
## Basic Cache Properties (2)

- 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

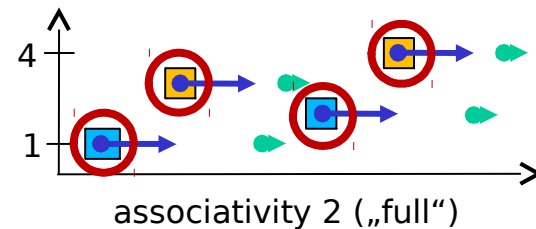




## Solution: Processor Caches

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

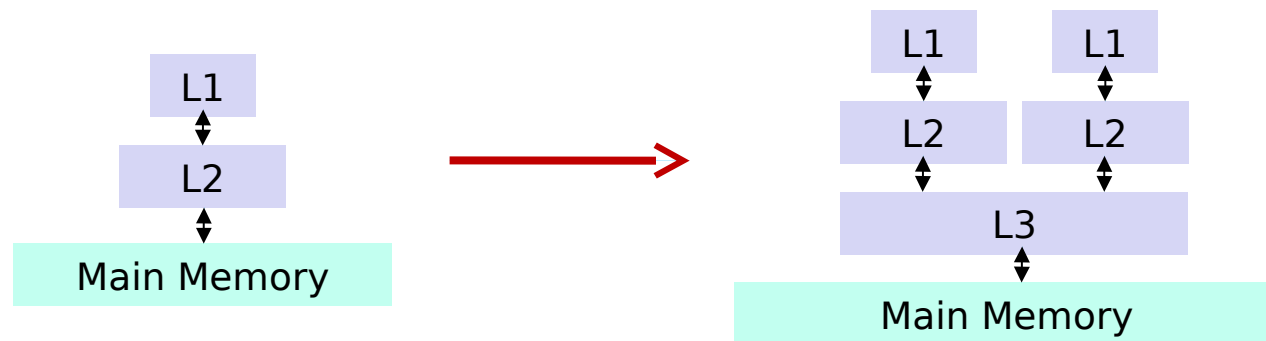
- example: threads need same vectors/matrices
- caches shared among cores can be beneficial
- sharing allows threads to prefetch data for each other

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/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

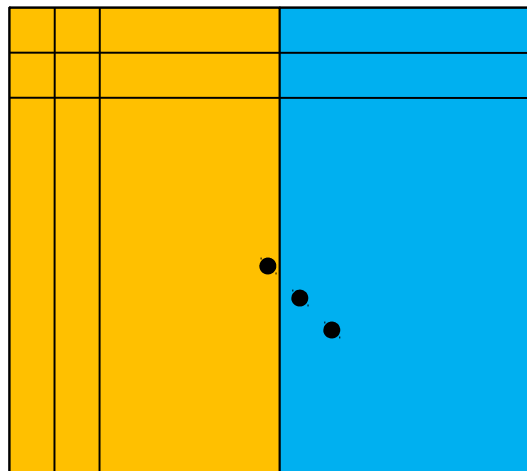
Countermeasures

- 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



 blocks assigned to set 1

 blocks assigned to set 2

## 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 accidentally 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**

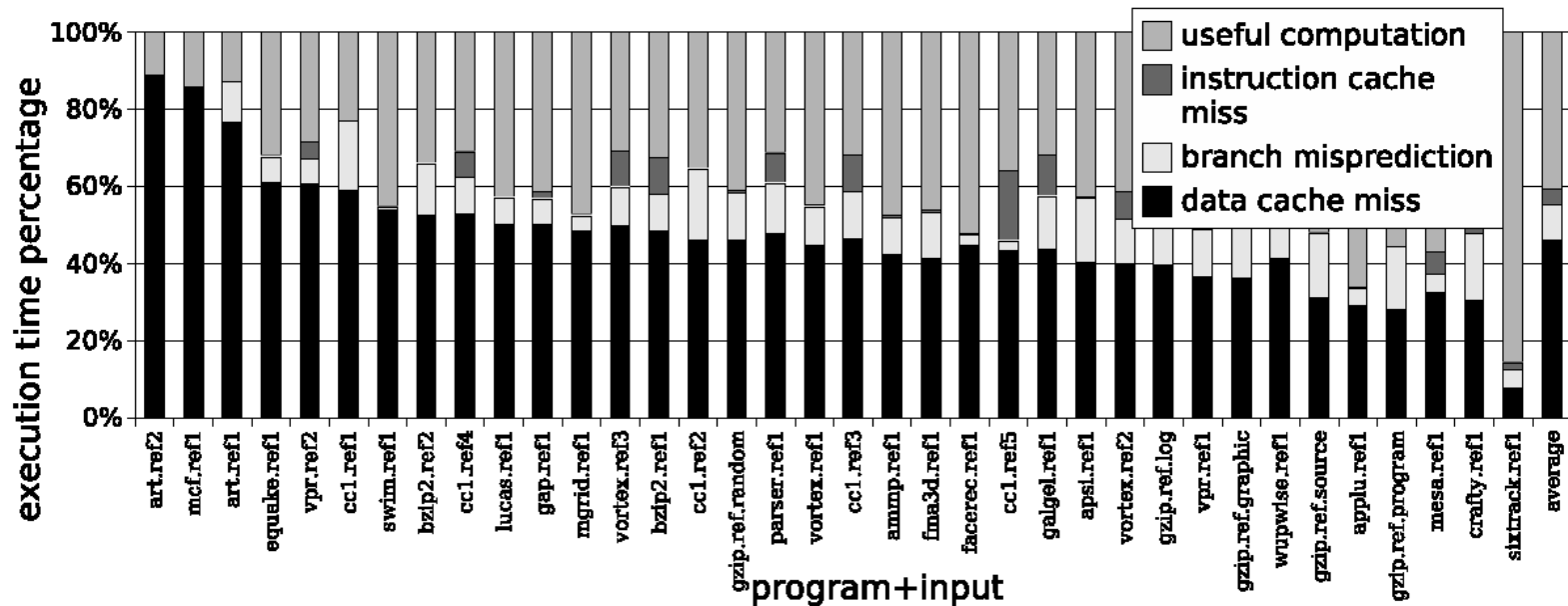
How to not exploit Caches

## **Cache Optimization Strategies**

How to exploit Caches even better



# The Principle of Locality is not enough...



## Reasons for Performance Loss for SPEC2000

[Beys/Hollander, ICCS 2004]

## Basic efficiency guidelines

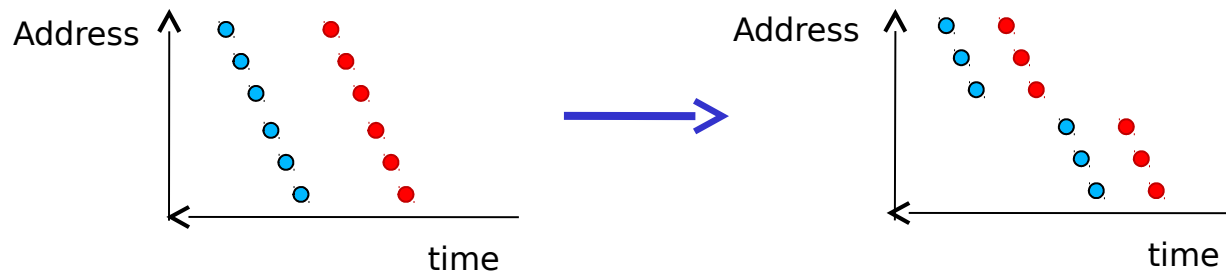
Always use a performance analysis tool before doing optimizations:  
How much time is wasted where because of cache misses?

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
- } Cache Optimizations

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

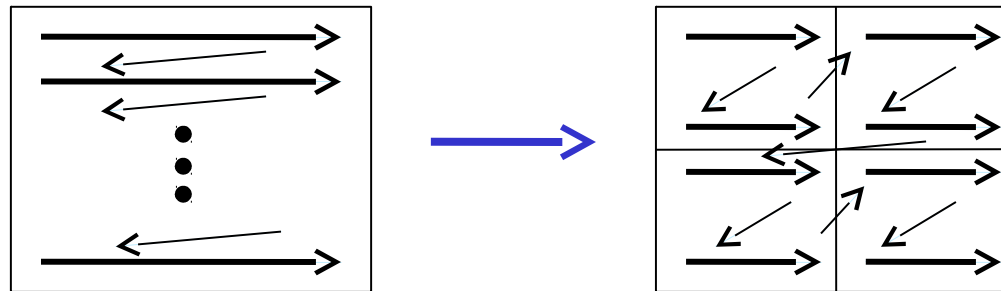
# Cache Optimization Strategies: Reordering Accesses

- Blocking: make arrays fit into a cache



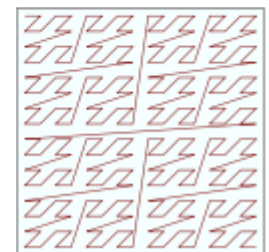
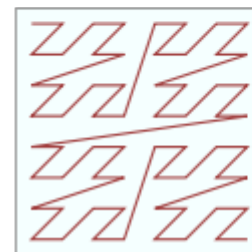
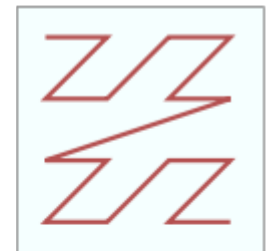
# Cache Optimization Strategies: Reordering Accesses

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



# Cache Optimization Strategies: Reordering Accesses

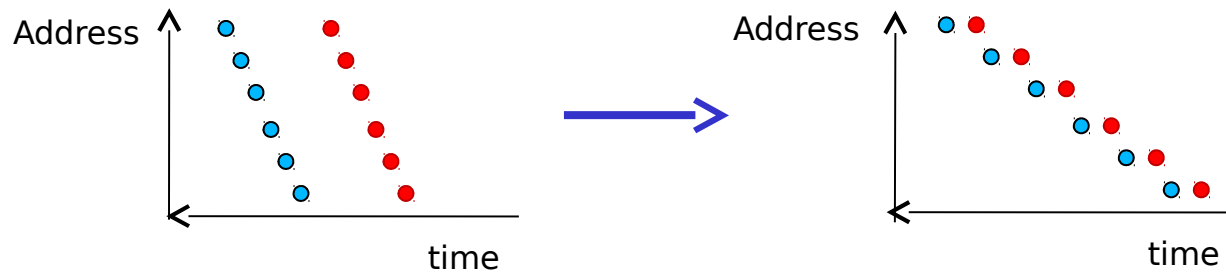
- Blocking: make arrays fit into a cache
- Blocking in multiple dimensions (example: 2D)
- Nested blocking: tune to multiple cache levels
  - can be done recursively according to a space filling curve
  - example: Morton curve (without “jumps”: Hilbert, Peano...)
  - **cache-oblivious** orderings/algorithms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



[ [http://en.wikipedia.org/wiki/Z-order\\_curve](http://en.wikipedia.org/wiki/Z-order_curve) ]

# Cache Optimization Strategies: Reordering Accesses

- Extreme blocking with size 1: Interweaving



- combined with blocking in other dimensions, 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

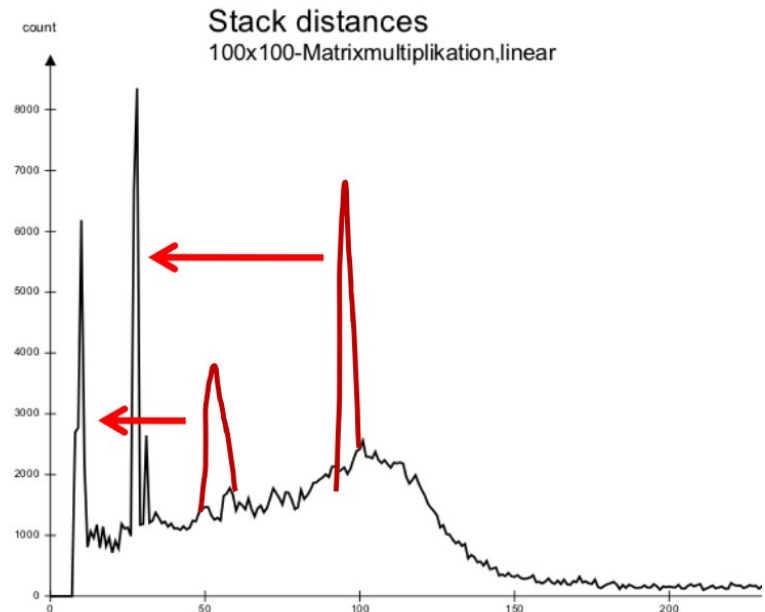
Reduce reuse distance of accesses = increase temporal locality

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 homogeneity (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_set_affinity`” (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

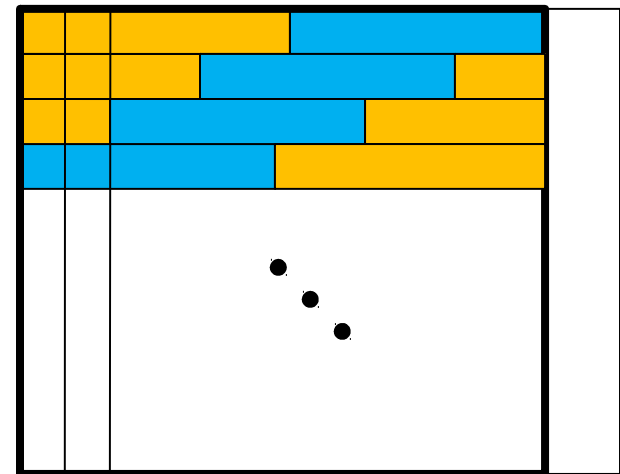
Make successive accesses cross multiple cache sets

Strategy:

- change data layout by **Padding**
- reorder accesses

Effectiveness can be seen by

- reduced number of misses



 block assigned to set 1

 block assigned to set 2

## 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](http://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 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



## 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](http://kcachegrind.sf.net)
- included with KDE3 & KDE4

### Visualization of

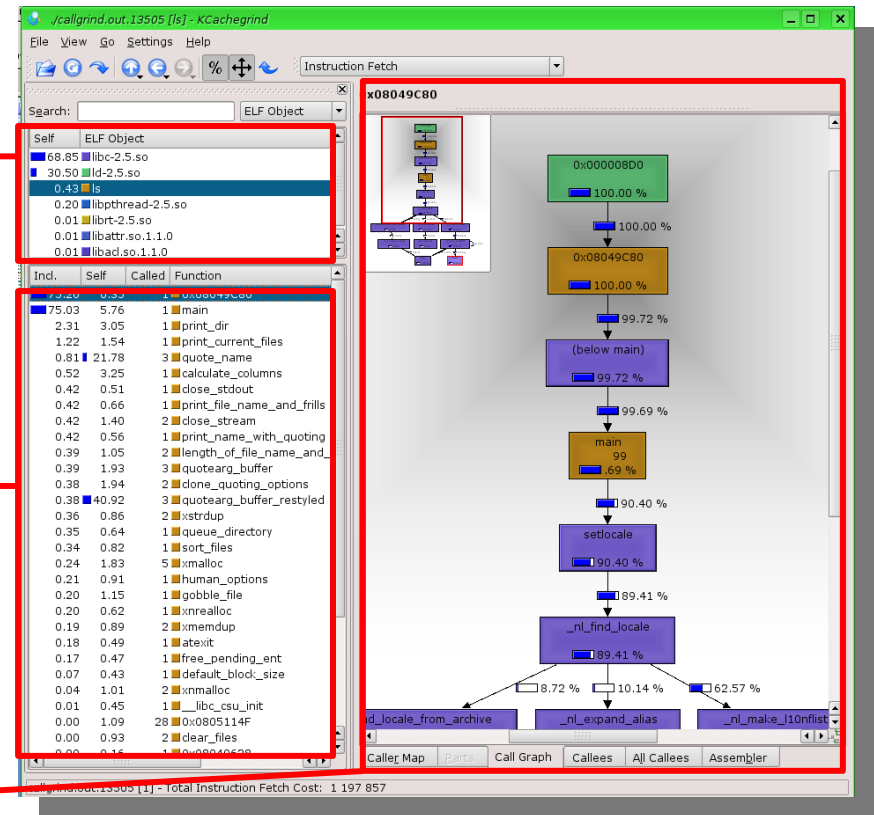
- call relationship of functions (callers, callees, call graph)
- exclusive/Inclusive cost metrics of functions
  - grouping according to ELF object / source file / C++ class
- source/assembly annotation: costs + CFG
- arbitrary events counts + specification of derived events

### 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
- Assembly annotation

**main**

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	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	xmalloc (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)

**main**

Treemap visualization showing memory access patterns for various functions. The largest block is `_nl_malloc` at 58.02%, followed by `strcpy` at 31.59%.

Call Graph visualization showing the flow of control between functions. The root node is `setlocale` (90.68%), which calls `_nl_find_locale` (89.69%). `_nl_find_locale` then branches into `_nl_load_locale_from_archive` (8.75%) and `_nl_expand_alias` (10.17%).

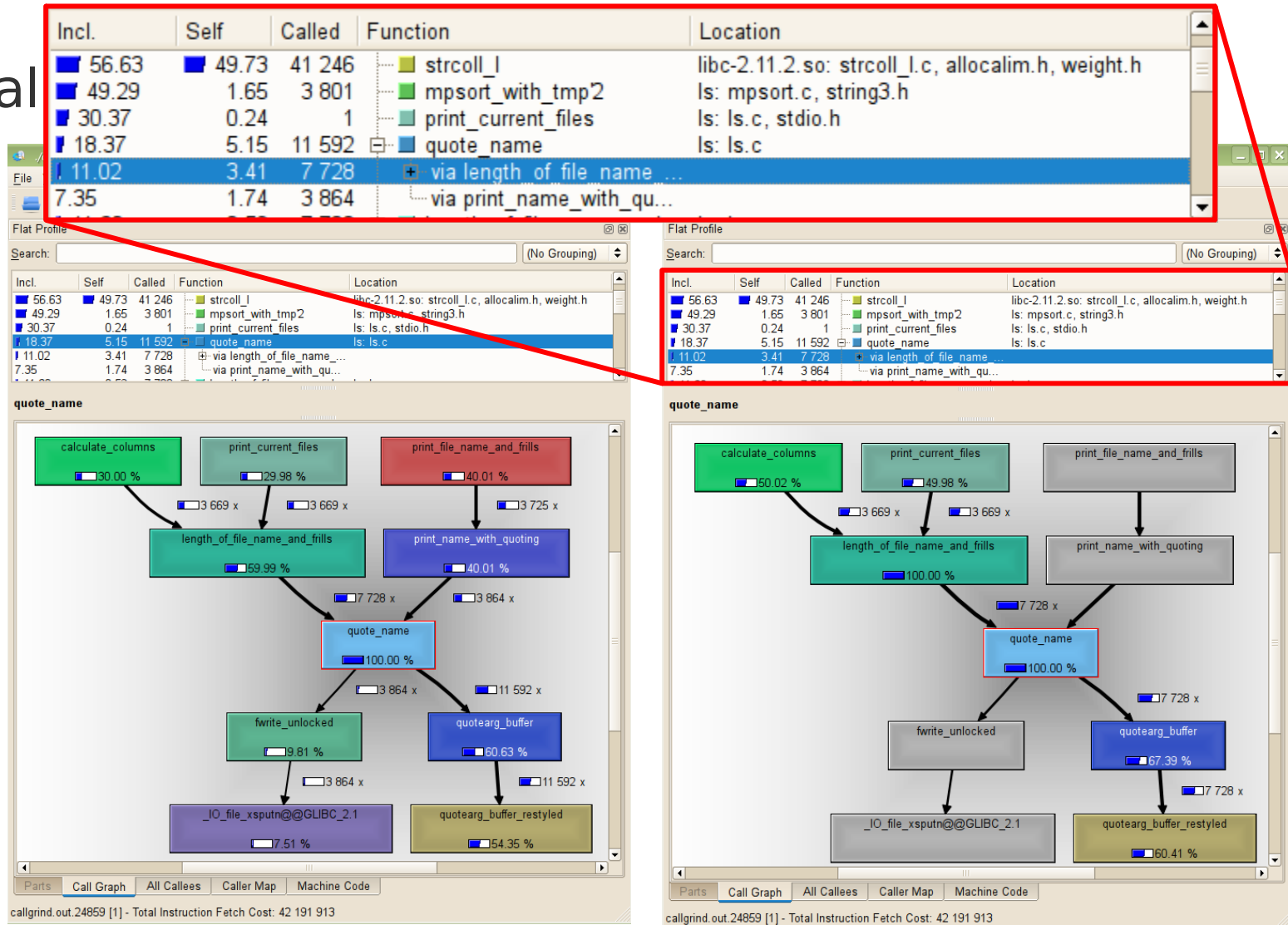
**main**

Source code snippet with annotations:

```

1119 # Source (/usr/src/debug/coreutils-6.4/src/ls.c)
1120
1121 #if ! SA_NOCLDSTOP
1122 bool caught_sig[nsigs];
1123 #endif
1124
1125 initialize_main (@argc, @argv);
1126 program_name = argv[0];
1127 setlocale (LC_ALL, "");
1128
1129 # 814 979 # 1 call to 'setlocale' (libc-2.5.so: setlocale.c)
1130 # 2 155 # 1 call to '_dl_runtime_resolve' (ld-2.5.so)
1131 # 8 # 1 call to 'bindtextdomain' (PACKAGE, LOCALEDIR);
1132 # 2 263 # 1 call to '_dl_runtime_resolve' (ld-2.5.so)
1133 # 565 # 1 call to 'bindtextdomain' (libc-2.5.so: bindtextdom.c)
1134 # 7 # 1 call to 'textdomain' (PACKAGE);
1135 # 1 925 # 1 call to '_dl_runtime_resolve' (ld-2.5.so)
1136 # 456 # 1 call to 'textdomain' (libc-2.5.so: textdomain.c)
1137
1138 initialize_exit_failure (LS_FAILURE);
1139
1140 # Assembler
1141
1142 804 DF8E 1 sub $0x1,%eax
1143 804 DF91 1 je 804e5d8 <ac_set_fd@plt+0x4968>
1144
1145 # Jump 1 of 1 times to 0x804E5D8
1146 804 DF97 1 call 8049650 <abort@plt>
1147 804 DF9C 1 movl $0x2,0x805e328
1148 804 DFA3 1 movl $0x4,0x4(%esp)
1149 804 DFA6 1 movl $0x0,0x805e330
1150 804 DFAE 1 movl $0x0,0x805e334
1151 804 DF85 1 call 8054630 <ac_set_fd@plt+0xa9c0>
1152 804 DFBA 1 movl $0x0,0x805e32c
1153 804 DFC1 1 movl $0x0,0x805e330
1154 804 DFC4 1 movl $0x0,0x805e330
1155 804 DFCB 1 movb $0x0,0x805e334
1156 804 DFCE 1 movb $0x0,0x805e336
1157 804 DFD5 1 movb $0x0,0x805e337
1158 804 DFD6 1 movb $0x0,0x805e337
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658
2659
2660
2661
2662
2663
2664
2665
2666
2667
2668
2669
2670
2671
2672
2673
2674
2675
2676
2677
2678
2679
2680
2681
2682
2683
2684
2685
2686
2687
2688
2689
2690
2691
2692
2693
2694
2695
2696
2697
2698
2699
2700
2701
2702
2703
2704
2705
2706
2707
2708
2709
2710
2711
2712
2713
2714
2715
2716
2717
2718
2719
2720
2721
2722
2723
2724
2725
2726
2727
2728
2729
2730
2731
2732
2733
2734
2735
2736
2737
2738
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2749
2750
2751
2752
2753
2754
2755
2756
2757
2758
2759
2760
2761
2762
2763
2764
2765
2766
2767
2768
2769
2770
2771
2772
2773
2774
2775
2776
2777
2778
2779
2780
2781
2782
2783
2784
2785
2786
2787
2788
2789
2790
2791
2792
2793
2794
2795
2796
2797
2798
2799
2800
2801
2802
2803
2804
2805
2806
2807
2808
2809
2810
2811
2812
2813
2814
2815
2816
2817
2818
2819
2820
2821
2822
2823
2824
2825
2826
2827
2828
2829
2830
2831
2832
2833
2834
2835
2836
2837
2838
2839
2840
2841
2842
2843
2844
2845
2846
2847
2848
2849
2850
2851
2852
2853
2854
2855
2856
2857
2858
2859
2860
2861
2862
2863
2864
2865
2866
2867
2868
2869
2870
2871
2872
2873
2874
2875
2876
2877
2878
2879
2880
2881
2882
2883
2884
2885
2886
2887
2888
2889
2890
2891
2892
2893
2894
2895
2896
2897
2898
2899
2900
2901
2902
2903
2904
2905
2906
2907
2908
2909
2910
2911
2912
2913
2914
2915
2916
2917
2918
2919
2920
2921
2922
2923
2924
2925
2926
2927
2928
2929
2930
2931
2932
2933
2934
2935
2936
2937
2938
2939
2940
2941
2942
2943
2944
2945
2946
2947
2948
2949
2950
2951
2952
2953
2954
2955
2956
2957
2958
2959
2960
2961
2962
2963
2964
2965
2966
2967
2968
2969
2970
2971
2972
2973
2974
2975
2976
2977
2978
2979
2980
2981
2982
2983
2984
2985
2986
2987
2988
2989
2990
2991
2992
2993
2994
2995
2996
2997
2998
2999
3000
3001
3002
3003
3004
3005
3006
3007
3008
3009
3010
3011
3012
3013
3014
3015
3016
3017
3018
3019
3020
3021
3022
3023
3024
3025
3026
3027
3028
3029
3030
3031
3032
3033
3034
3035
3036
3037
3038
3039
3040
3041
3042
3043
3044
3045
3046
3047
3048
3049
30
```

Call



# Outline: Part 2

## **Cache Analysis**

Measuring on real Hardware vs. Simulation

## **Cache Analysis Tools**

## **Case Studies**

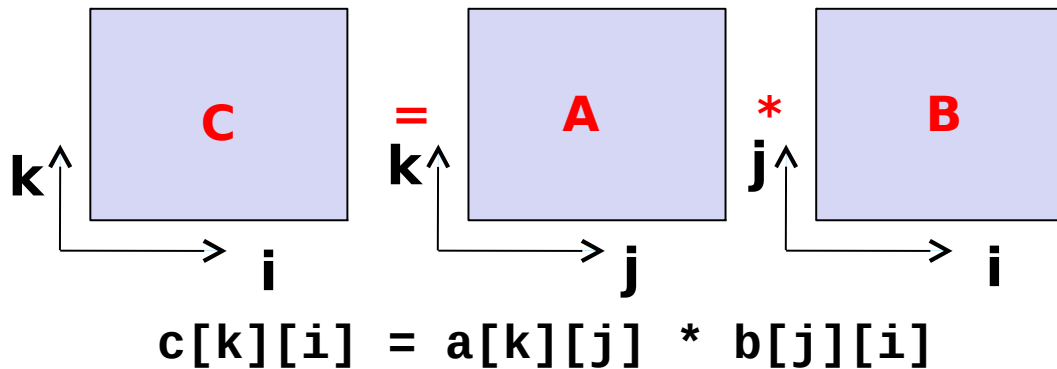
## **Hands-on**

## 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





## 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]
```

# Outline: Part 2

## **Cache Analysis**

Measuring on real Hardware vs. Simulation

## **Cache Analysis Tools**

## **Case Studies**

## **Hands-on**

?

Q&A

?