

Analysis and Optimization of the Memory Access Behavior of Applications

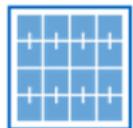
Ecole

“Méthodologie et outils d’optimisation en développement logiciel”

Fréjus, February 8, 2012

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)
- Invited by Romaric David to give a talk at this workshop

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

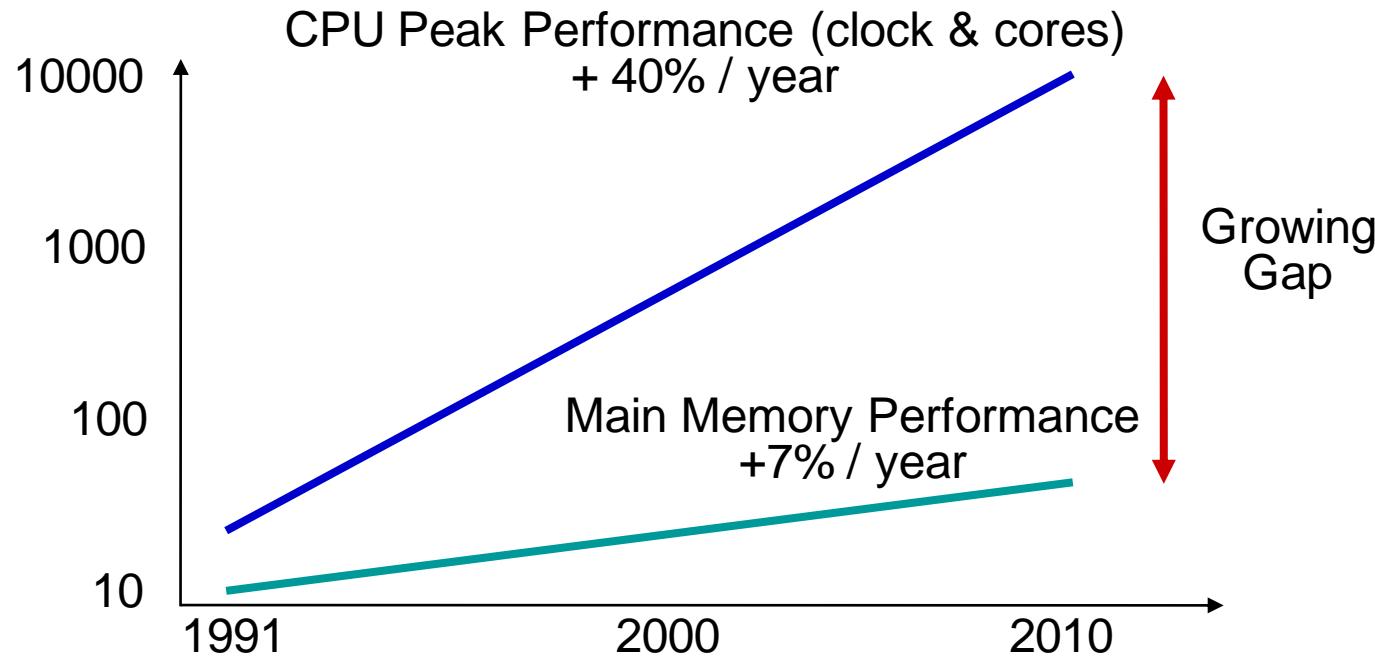
Still getting more important

- 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

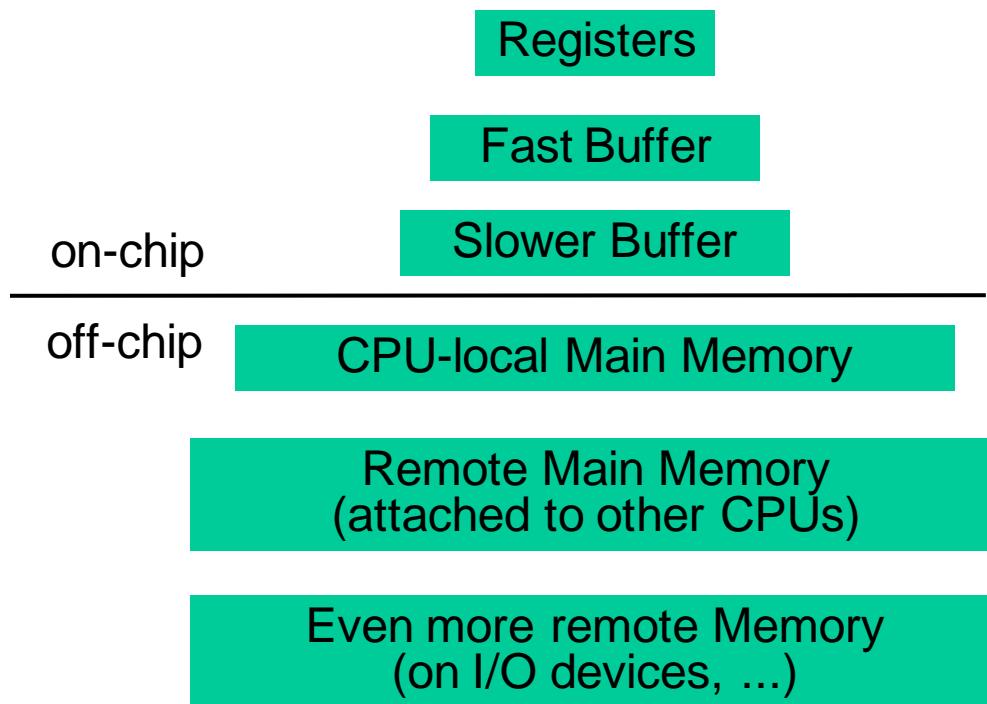
Solution: The Memory Hierarchy

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)

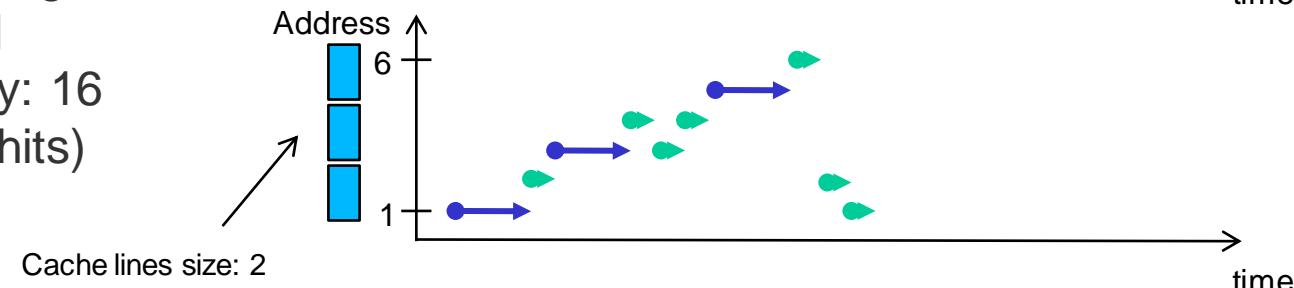
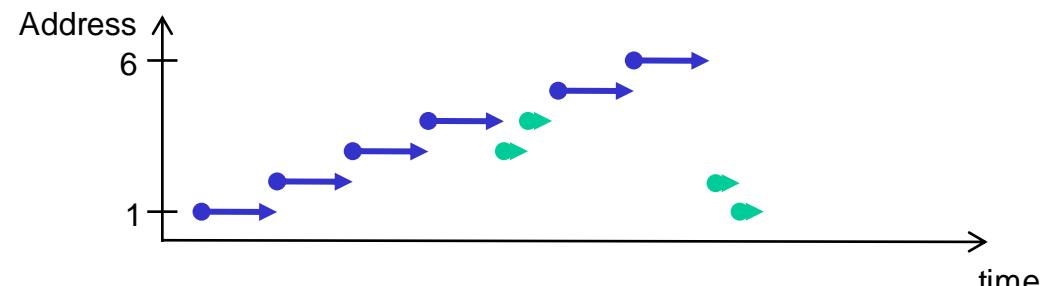
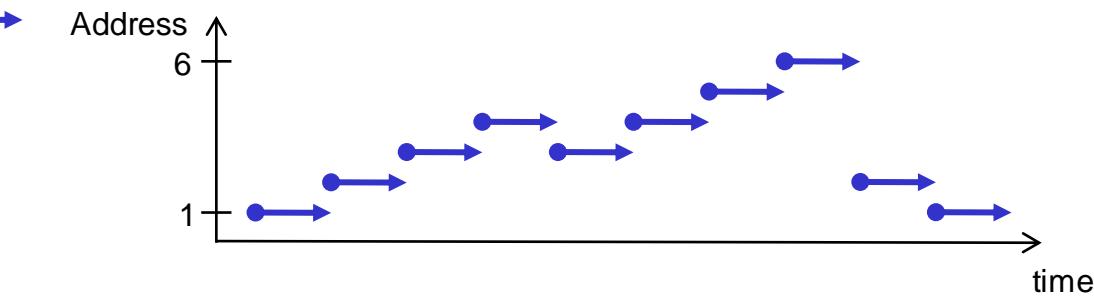
“Principle of Locality”

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

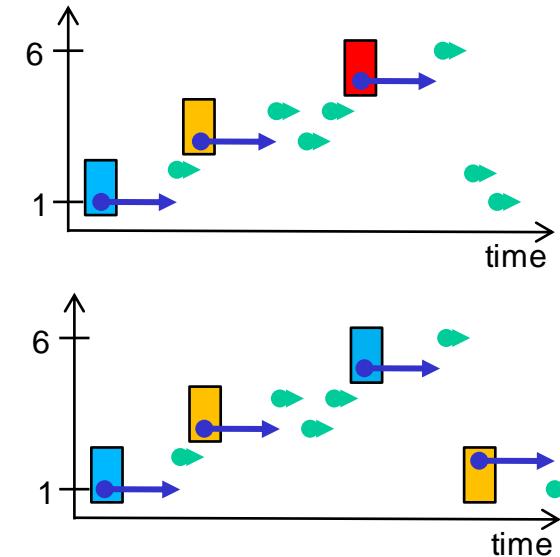
Example: Sequence with 10 Accesses

- 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)



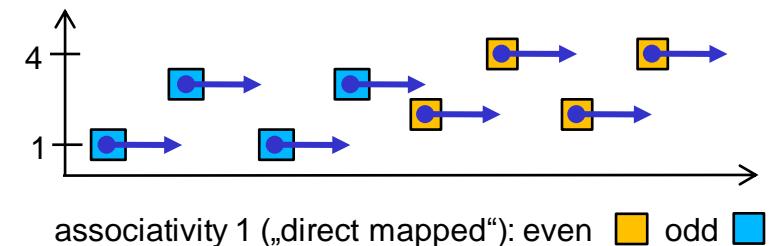
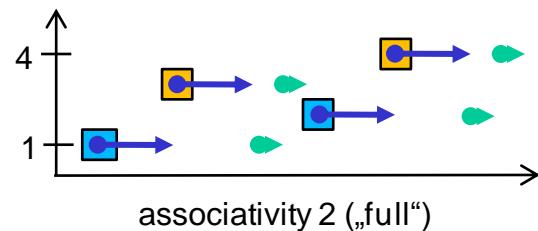
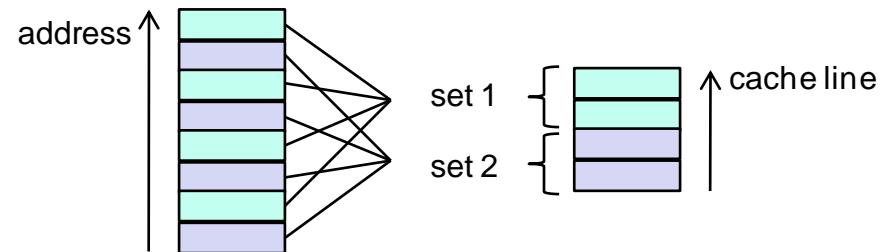
Basic Cache Properties (1)

- 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)

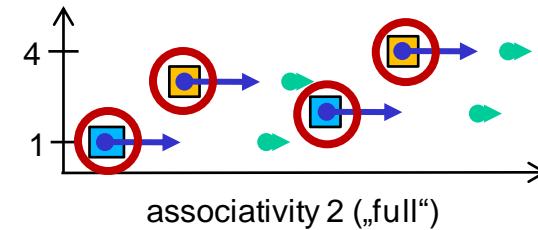
- every cache line knows the memory address it has a copy of („tag“)
- comparing all tags at every access → expensive (space & energy)
- better: reduce number of comparisons per access
 - group cache lines into sets
 - a given address can only be stored into a given set
 - lines per set: **Associativity**
- example: 2 lines (■ ■) , sequence 1/3/1/3/2/4/2/4



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

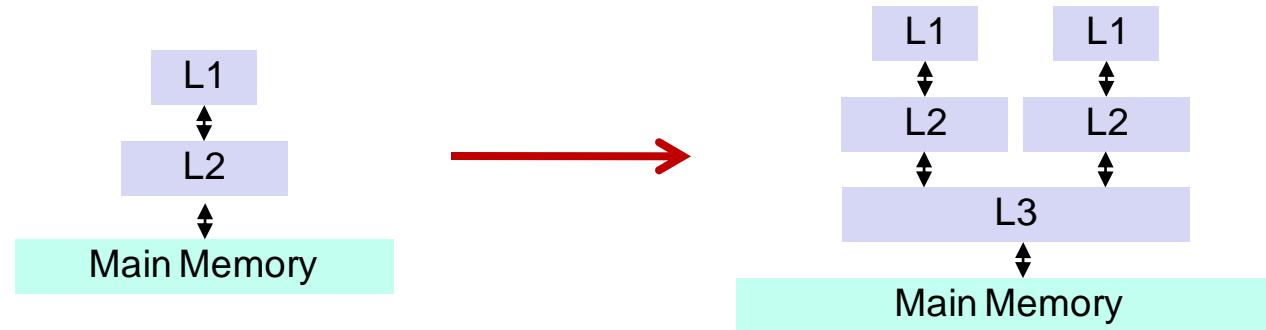
- example: threads need same vectors/matrices
- caches shared among cores can be beneficial
- sharing allows threads to prefetch data for each other

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 largest 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

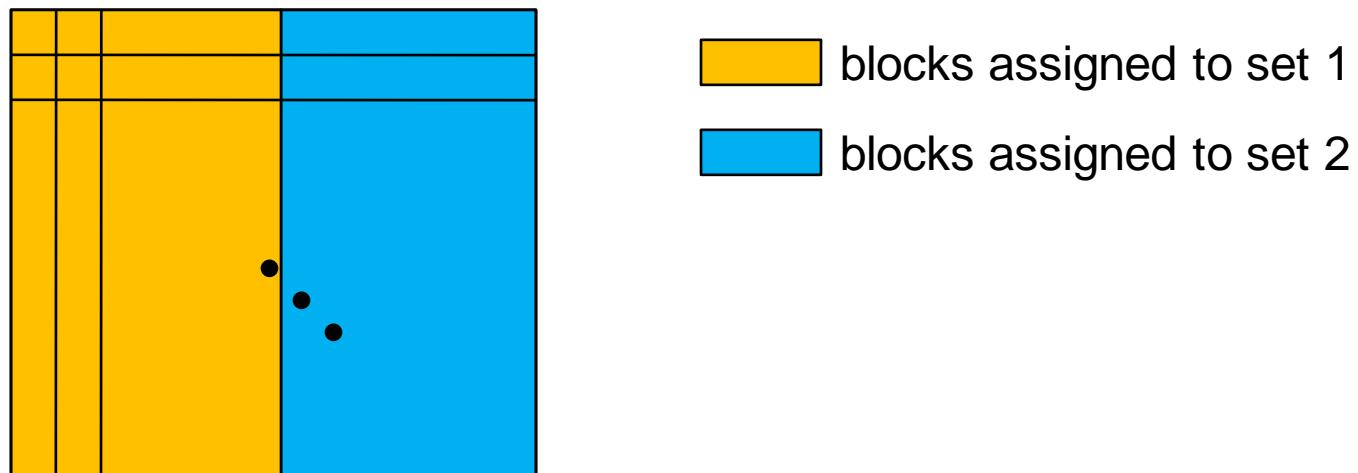
Countermeasures

- 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 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?

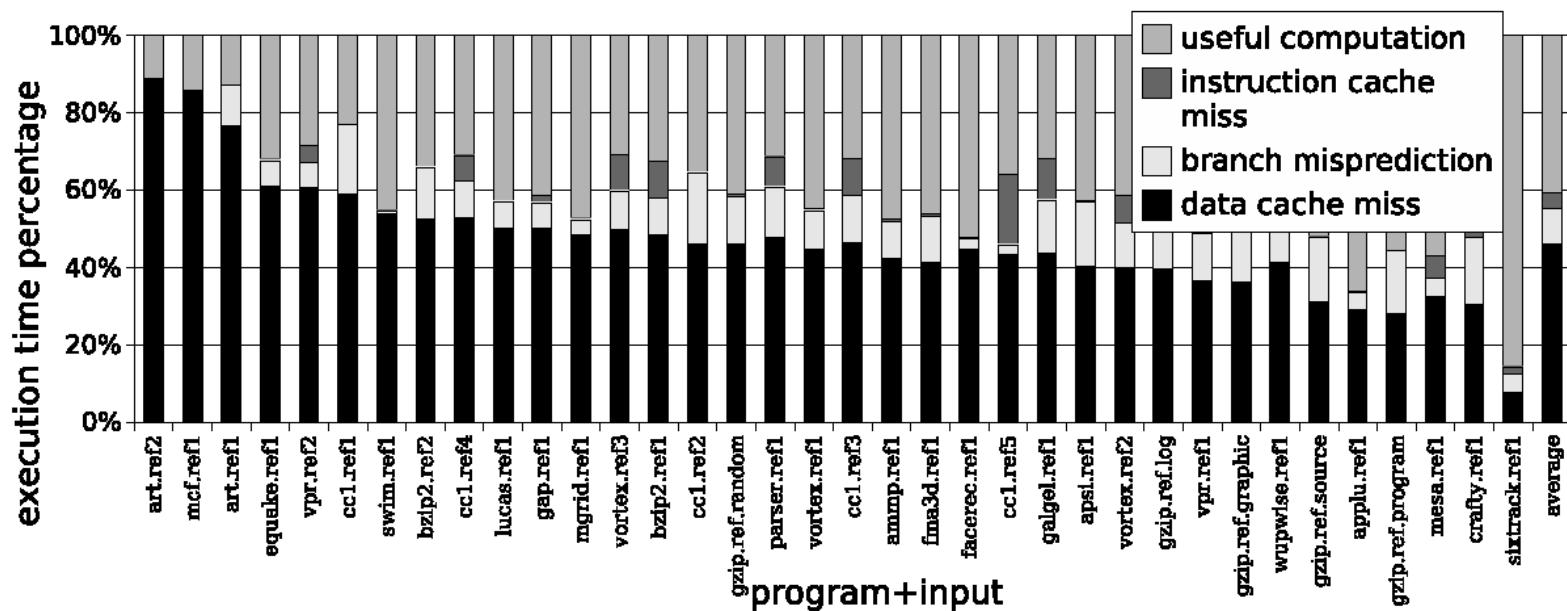
Bad Memory Access Patterns

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
[Beyls/Hollander, ICCS 2004]

Basic efficiency guidelines

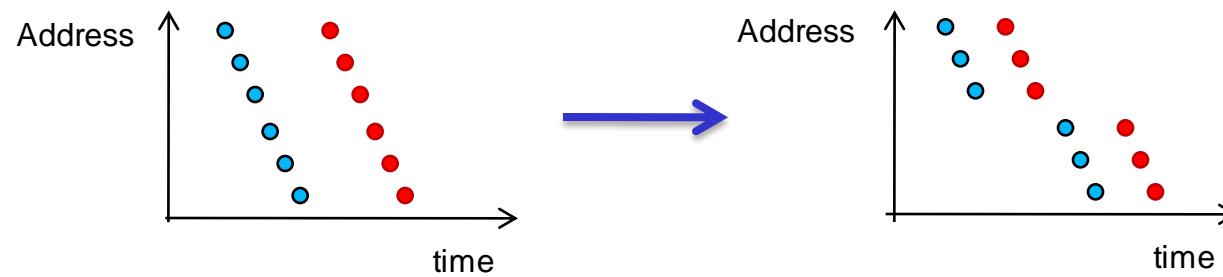
Always use a performance analysis tool before doing optimizations:
How much time is wasted where because of cache misses?

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
- } Cache Optimizations

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

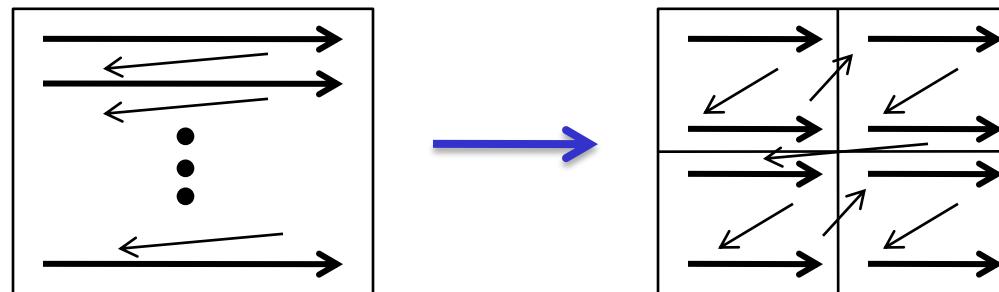
Cache Optimization Strategies: Reordering Accesses

- Blocking: make arrays fit into a cache



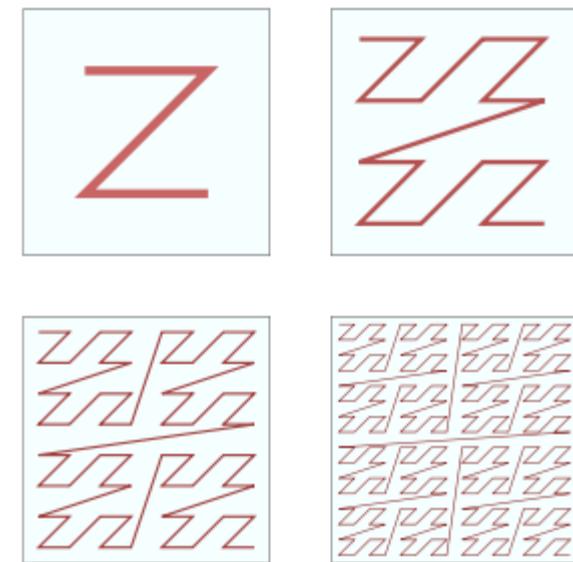
Cache Optimization Strategies: Reordering Accesses

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



Cache Optimization Strategies: Reordering Accesses

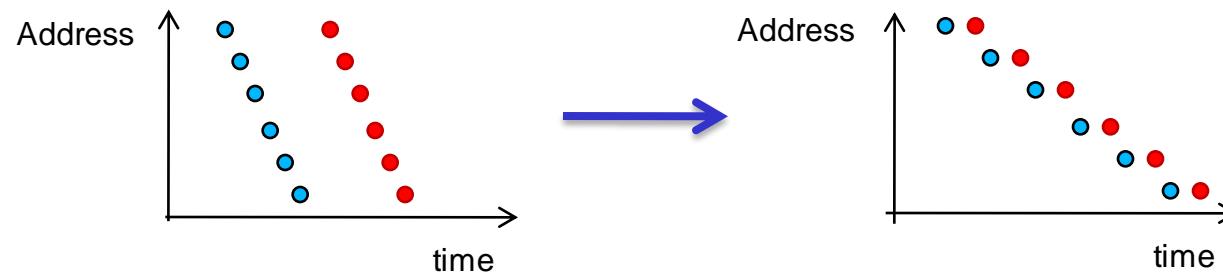
- Blocking: make arrays fit into a cache
- Blocking in multiple dimensions (example: 2D)
- Nested blocking: tune to multiple cache levels
 - can be done recursively according to a space filling curve
 - example: Morton curve (without “jumps”: Hilbert, Peano...)
 - **cache-oblivious** orderings/algorithms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

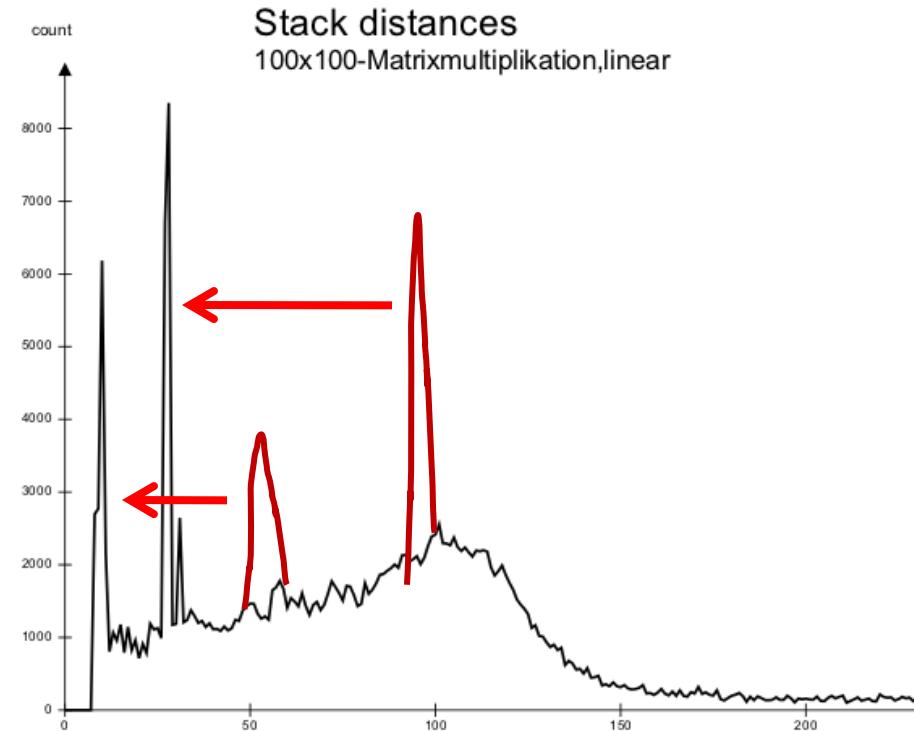
Reduce reuse distance of accesses = increase temporal locality

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_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

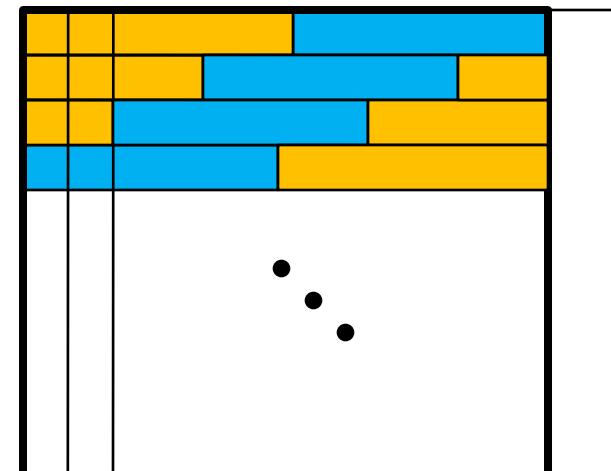
Make successive accesses cross multiple cache sets

Strategy:

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

Effectiveness can be seen by

- reduced number of misses



 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

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
- included with KDE3 & KDE4

Visualization of

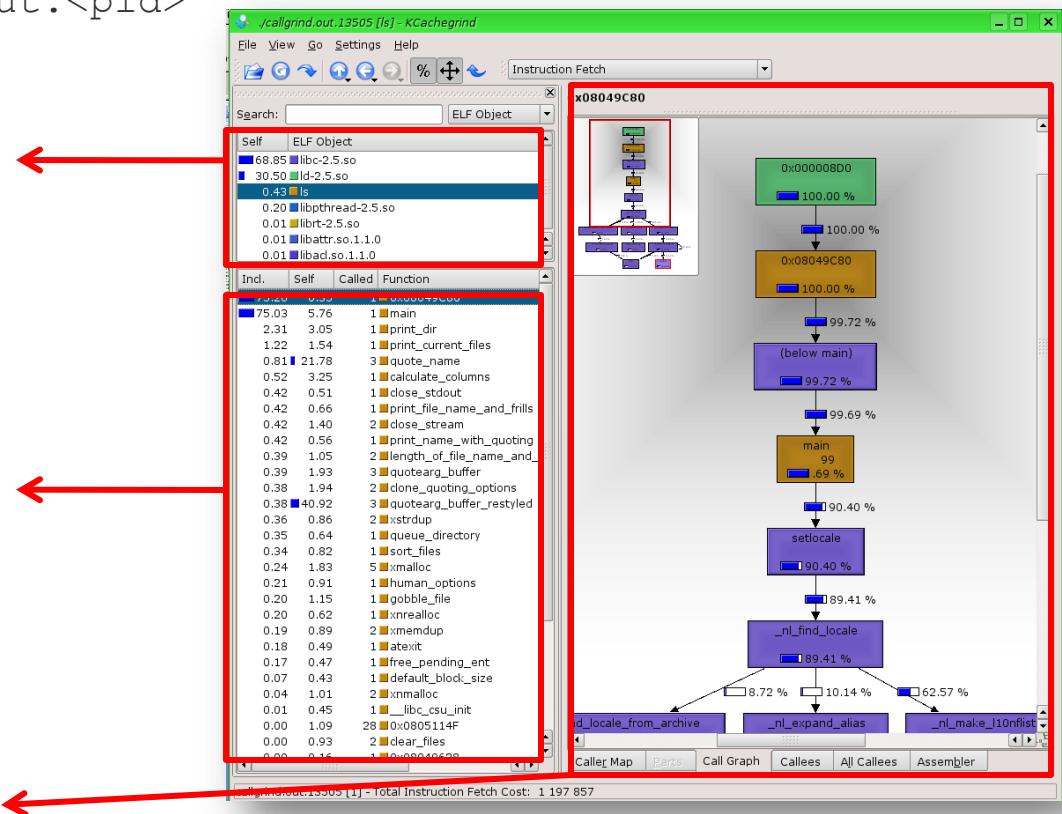
- call relationship of functions (callers, callees, call graph)
- exclusive/inclusive cost metrics of functions
 - grouping according to ELF object / source file / C++ class
- source/assembly annotation: costs + CFG
- arbitrary events counts + specification of derived events

Callgrind support (file format, events of cache model)

KCachegrind: Usage

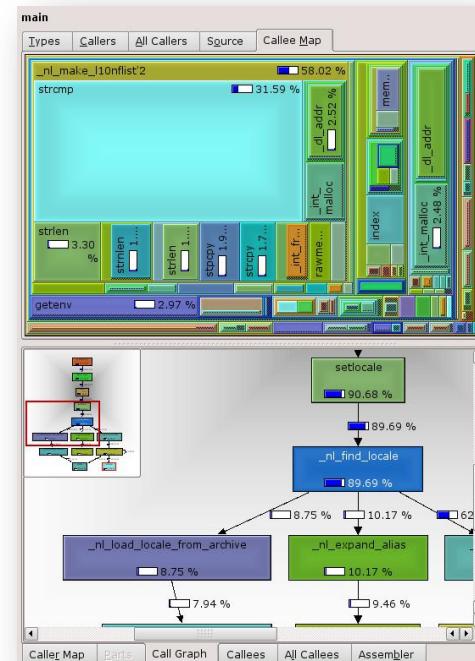
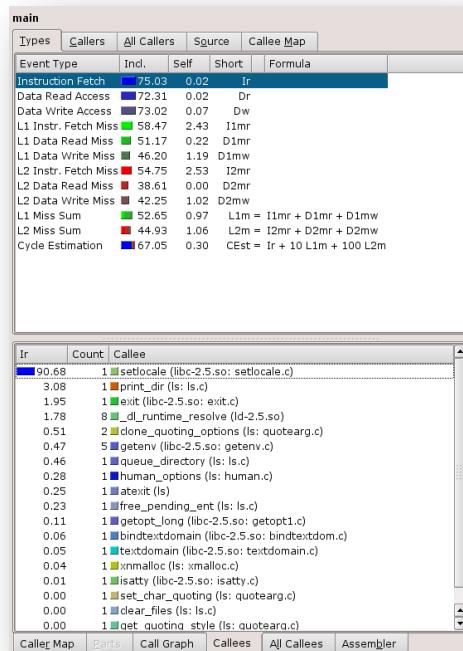
{k, q}cachegrind callgrind.out.<pid>

- left: “Dockables”
 - list of function groups according to
 - library (ELF object)
 - source
 - class (C++)
 - list of functions with
 - inclusive
 - exclusive costs
- right: visualization panes



Visualization panes for selected function

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

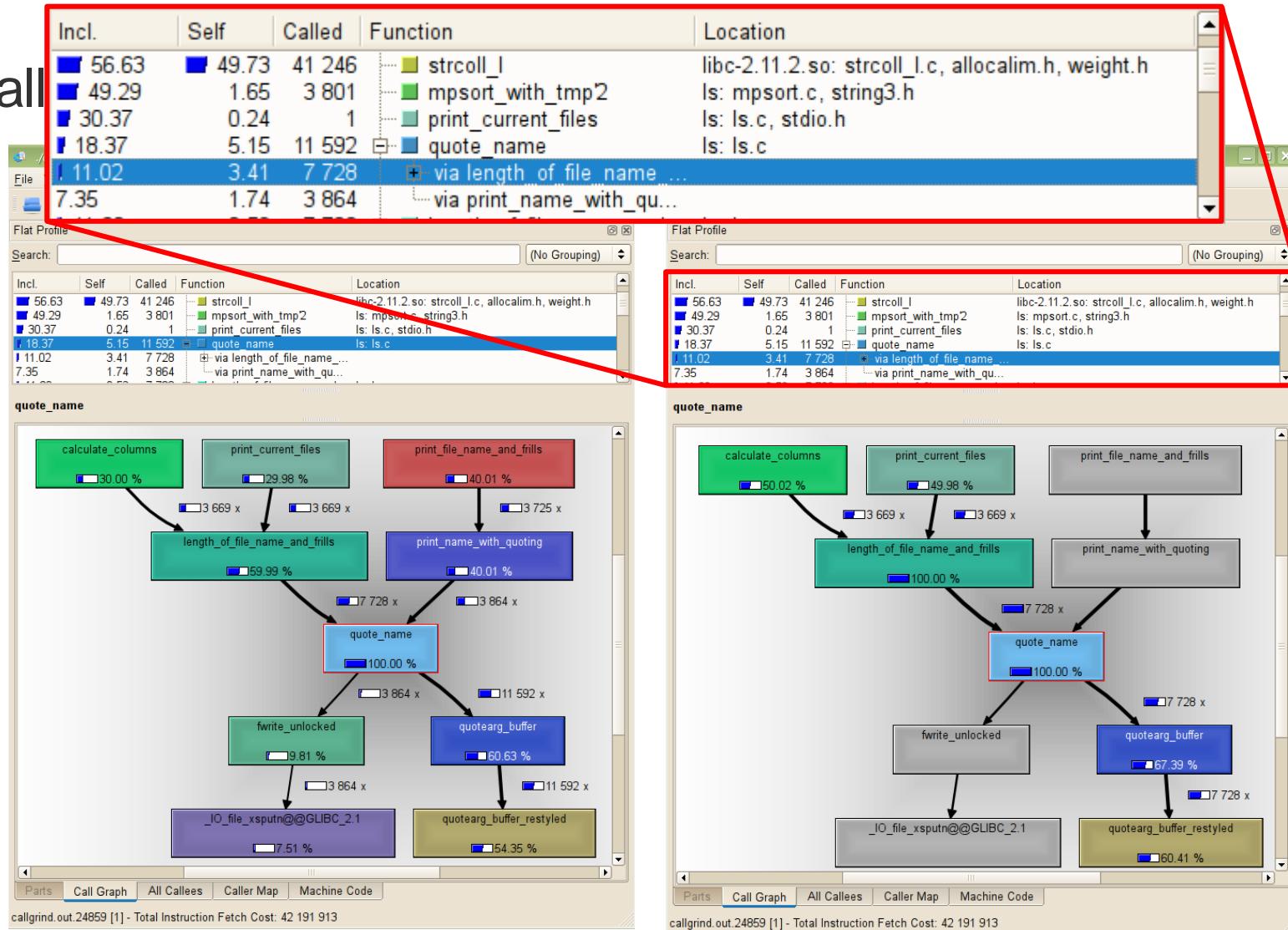


The figure shows the source code for `initialize_main` and its assembly translation. A red box highlights the `setlocale` call in the source code, which corresponds to the assembly instruction `call 804DF8E`. The assembly pane shows the assembly code for the entire function, with the highlighted line being `movl $0x4,0(%esp)`.

```

main
Types Callers All Callers Srouce Callee Map
# Ir Source ('/usr/src/debug/coreutils-6.4/src/ls.c')
1119
1120 #if ! SA_NOCLDSTOP
1121     bool caught_sig[nsigs];
1122 #endif
1123
1124     initialize_main (&argc, &argv);
1125     2 program_name = argv[0];
1126     8 setlocale (LC_ALL, "");
1127     1124 8 1 call to 'setlocale' (libc-2.5.so: setlocale.c)
1128     2 155 8 1 call to '_dl_runtime_resolve' (ld-2.5.so)
1129     2 263 8 1 call to '_dl_runtime_resolve' (ld-2.5.so)
1130     565 7 1 call to 'bindtextdomain' (libc-2.5.so: bindtextdom.c)
1131     1 935 7 1 call to 'textdomain' (PACKAGE);
1132     456 1 935 7 1 call to '_dl_runtime_resolve' (ld-2.5.so)
1133     1 456 1 1 call to 'textdomain' (libc-2.5.so: textdomain.c)
1134
1135     initialize_exit_failure (ls: EXIT_FAILURE).
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
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
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
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
18010
18011
18012
18013
18014
18015
18016
18017
18018
18019
18020
18021
18022
18023
18024
18025
18026
18027
18028
18029
18030
18031
18032
18033
18034
18035
18036
18037
18038
18039
18040
18041
18042
18043
18044
18045
18046
18047
18048
18049
18050
18051
18052
18053
18054
18055
18056
18057
18058
18059
18060
18061
18062
18063
18064
18065
18066
18067
18068
18069
18070
18071
18072
18073
18074
18075
18076
18077
18078
18079
18080
18081
18082
18083
18084
18085
18086
18087
18088
18089
18090
18091
18092
18093
18094
18095
18096
18097
18098
18099
180100
180101
180102
180103
180104
180105
180106
180107
180108
180109
180110
180111
180112
180113
180114
180115
180116
180117
180118
180119
180120
180121
180122
180123
180124
180125
180126
180127
180128
180129
180130
180131
180132
180133
180134
180135
180136
180137
180138
180139
180140
180141
180142
180143
180144
180145
180146
180147
180148
180149
180150
180151
180152
180153
180154
180155
180156
180157
180158
180159
180160
180161
180162
180163
180164
180165
180166
180167
180168
180169
180170
180171
180172
180173
180174
180175
180176
180177
180178
180179
180180
180181
180182
180183
180184
180185
180186
180187
180188
180189
180190
180191
180192
180193
180194
180195
180196
180197
180198
180199
180200
180201
180202
180203
180204
180205
180206
180207
180208
180209
180210
180211
180212
180213
180214
180215
180216
180217
180218
180219
180220
180221
180222
180223
180224
180225
180226
180227
180228
180229
180230
180231
180232
180233
180234
180235
180236
180237
180238
180239
180240
180241
180242
180243
180244
180245
180246
180247
180248
180249
180250
180251
180252
180253
180254
180255
180256
180257
180258
180259
180260
180261
180262
180263
180264
180265
180266
180267
180268
180269
180270
180271
180272
180273
180274
180275
180276
180277
180278
180279
180280
180281
180282
180283
180284
180285
180286
180287
180288
180289
180290
180291
180292
180293
180294
180295
180296
180297
180298
180299
180300
180301
180302
180303
180304
180305
180306
180307
180308
180309
180310
180311
180312
180313
180314
180315
180316
180317
180318
180319
180320
180321
180322
180323
180324
180325
180326
180327
180328
180329
180330
180331
180332
180333
180334
180335
180336
180337
180338
180339
180340
180341
180342
180343
180344
180345
180346
180347
180348
180349
180350
180351
180352
180353
180354
180355
180356
180357
180358
180359
180360
180361
180362
180363
180364
180365
180366
180367
180368
180369
180370
180371
180372
180373
180374
180375
180376
180377
180378
180379
180380
180381
180382
180383
180384
180385
180386
180387
180388
180389
180390
180391
180392
180393
180394
180395
180396
180397
180398
180399
180400
180401
180402
180403
180404
180405
180406
180407
180408
180409
180410
180411
180412
180413
180414
180415
180416
180417
180418
180419
180420
180421
180422
180423
180424
180425
180426
180427
180428
180429
180430
180431
180432
180433
180434
180435
180436
180437
180438
180439
180440
180441
180442
180443
180444
180445
180446
180447
180448
180449
180450
180451
180452
180453
180454
180455
180456
180457
180458
180459
180460
180461
180462
180463
180464
180465
180466
180467
180468
180469
180470
180471
180472
180473
180474
180475
180476
180477
180478
180479
180480
180481
180482
180483
180484
180485
180486
180487
180488
180489
180490
180491
180492
180493
180494
180495
180496
180497
180498
180499
180500
180501
180502
180503
180504
180505
180506
180507
180508
180509
180510
180511
180512
180513
180514
180515
180516
180517
180518
180519
180520
180521
180522
180523
180524
180525
180526
180527
180528
180529
180530
180531
180532
180533
180534
180535
180536
180537
180538
180539
180540
180541
180542
180543
180544
180545
180546
180547
180548
180549
180550
180551
180552
180553
180554
180555
180556
180557
180558
180559
180560
180561
180562
180563
180564
180565
180566
180567
180568
180569
180570
180571
180572
180573
180574
180575
180576
180577
180578
180579
180580
180581
180582
180583
180584
180585
180586
180587
180588
180589
180590
180591
180592
180593
180594
180595
180596
180597
180598
180599
180600
180601
180602
180603
180604
180605
180606
180607
180608
180609
180610
180611
180612
180613
180614
180615
180616
180617
180618
180619
180620
180621
180622
180623
180624
180625
180626
180627
180628
180629
180630
180631
180632
180633
180634
180635
180636
180637
180638
180639
180640
180641
180642
180643
180644
180645
180646
180647
180648
180649
180650
180651
180652
180653
180654
180655
180656
180657
180658
180659
180660
180661
180662
180663
180664
180665
180666
180667
180668
180669
180670
180671
180672
180673
180674
180675
180676
180677
180678
180679
180680
180681
180682
180683
180684
180685
180686
180687
180688
180689
180690
180691
180692
180693
180694
180695
180696
180697
180698
180699
180700
180701
180702
180703
180704
180705
180706
180707
180708
180709
180710
180711
180712
180713
180714
180715
180716
180717
180718
180719
180720
180721
180722
180723
180724
180725
180726
180727
180728
180729
180730
180731
180732
180733
180734
180735
180736
180737
180738
180739
180740
180741
180742
180743
180744
180745
180746
180747
180748
180749
180750
180751
180752
180753
180754
180755
180756
180757
180758
180759
180760
180761
180762
180763
180764
180765
180766
180767
180768
180769
180770
180771
180772
180773
180774
180775
180776
180777
180778
180779
180780
180781
180782
180783
180784
180785
180786
180787
180788
180789
180790
180791
180792
180793
180794
180795
180796
180797
180798
180799
180800
180801
180802
180803
180804
180805
180806
180807
180808
180809
180810
180811
180812
180813
180814
180815
180816
180817
180818
180819
180820
180821
180822
180823
180824
180825
180826
180827
180828
180829
180830
180831
180832
180833
180834
180835
180836
180837
180838
180839
180840
180841
180842
180843
180844
180845
180846
180847
180848
180849
180850
180851
180852
180853
180854
180855
180856
180857
180858
180859
180860
180861
180862
180863
180864
180865
180866
180867
180868
180869
180870
180871
180872
180873
180874
180875
180876
180877
180878
180879
180880
180881
180882
180883
180884
180885
180886
180887
180888
180889
180880
180881
180882
180883
180884
180885
180886
180887
180888
180889
180890
180891
180892
180893
180894
180895
180896
180897
180898
180899
180890
180891
180892
180893
180894
180895
180896
180897
180898
180899
180900
180901
180902
180903
180904
180905
180906
180907
180908
180909
180910
180911
180912
180913
180914
180915
180916
180917
180918
180919
180920
180921
1809
```

Call Graph



Outline: Part 2

Cache Analysis

Measuring on real Hardware vs. Simulation

Cache Analysis Tools

Case Studies

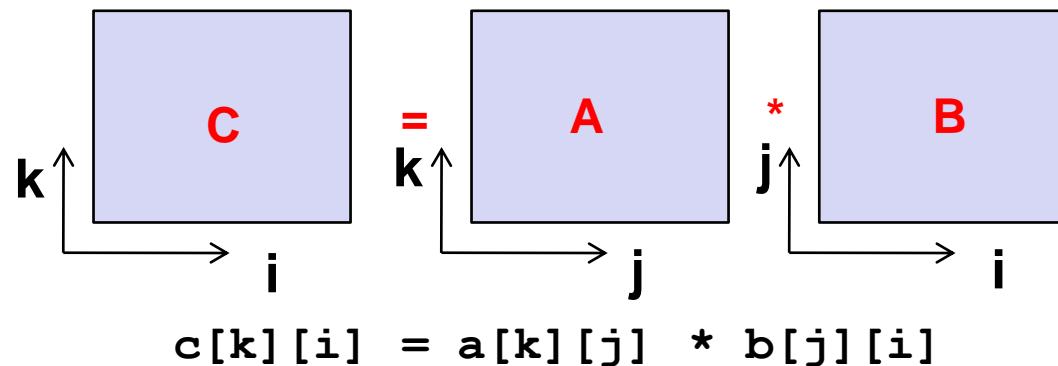
Hands-on

Case Studies

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

Matrix Multiplication

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



Matrix Multiplication

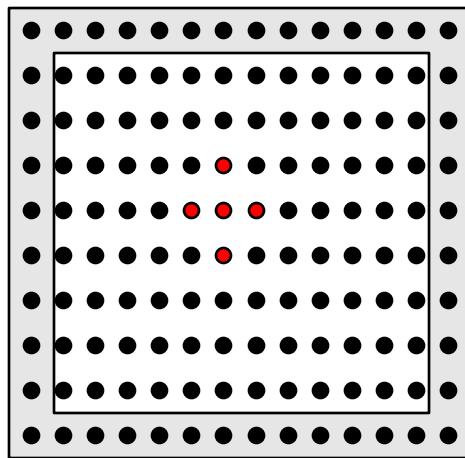
- Kernel for $C = A * B$
 - 3 nested loops (i, j, k): What is the best index order? Why?

```
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]
```

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
- ...

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