

# Cache Profiling with Callgrind

Linux/x86 Performance Practical, 17.06.2009

Zellescher Weg 12

Willers-Bau A106

Tel. +49 351 - 463 - 31945

Ulf Markwardt ([ulf.markwardt@tu-dresden.de](mailto:ulf.markwardt@tu-dresden.de))

Matthias Lieber ([matthias.lieber@tu-dresden.de](mailto:matthias.lieber@tu-dresden.de))

# Valgrind

---

- Suite of simulation-based debugging and profiling tools
- Valgrind core simulates a CPU in software
- Tools implement various tasks by adding analysis code
- Available for Linux on x86 and PowerPC platforms (both 32/64 bit)
- Open source
- Standard Linux package
- Wide acceptance, e.g. Firefox, OpenOffice, KDE use Valgrind
- <http://www.valgrind.org>

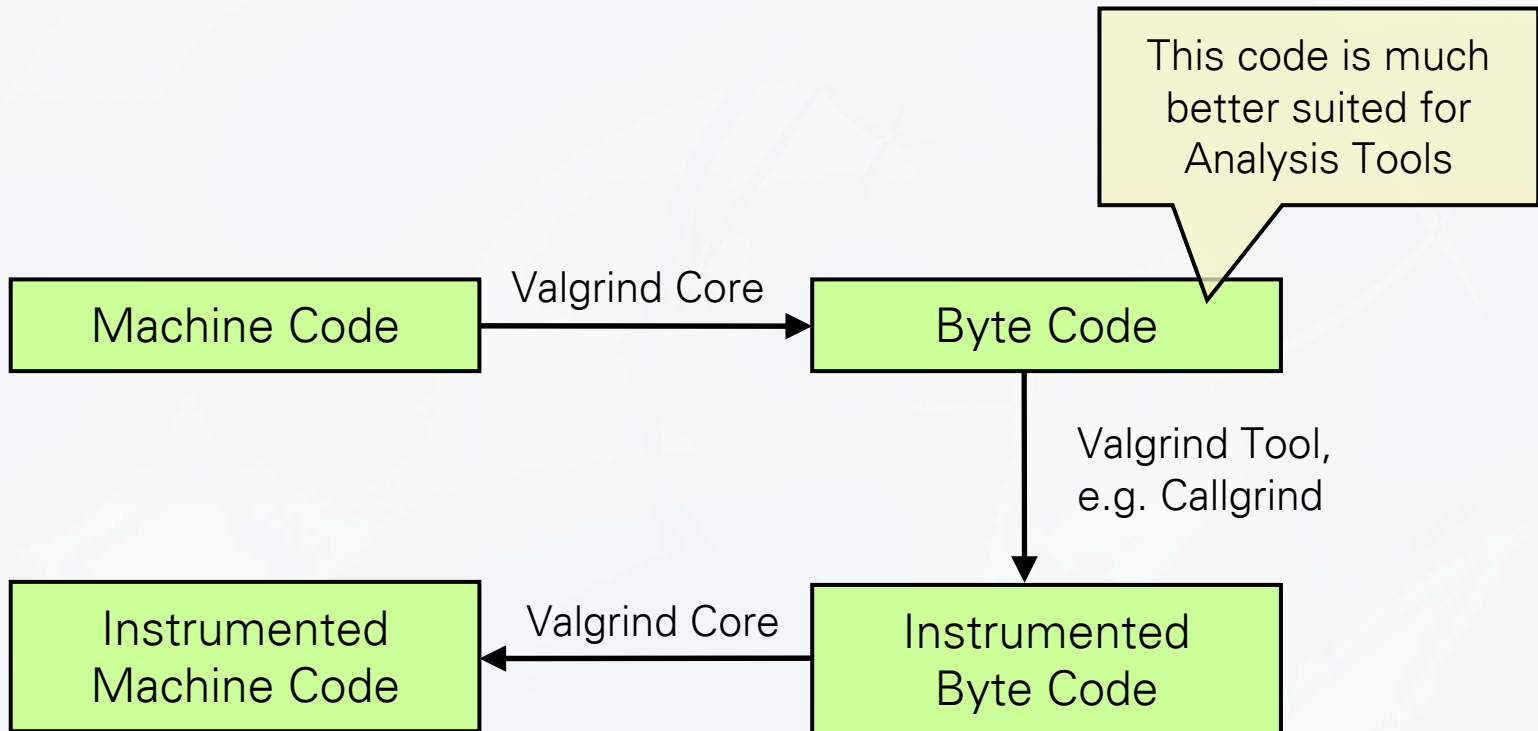
# Valgrind's Tool Suite

---

- Memcheck = “Valgrind”
  - Most prominent, detects memory management bugs
- Cachegrind
  - Cache profiler, finds source location of cache misses
- **Callgrind**
  - Cachegrind + function call graph information
- Massif
  - Heap profiler, where / how much memory allocations?
- Helgrind
  - Thread debugger

# How does Valgrind work?

- Dynamic recompilation of program's binary at runtime
- Original code never runs directly on CPU
- **Large overhead:** 10 – 100 times slower, depending on tool



# Cachegrind, Callgrind

---

- **Cachegrind** collects statistics about cache misses
- Simulates L1i, L1d, inclusive L2 cache
- Size of the caches can be specified, default is current machine's cache
- Output:
  - Total program run hit/miss count and ratio
  - Per function hit/miss count
  - Per source code line hit/miss count
- **Callgrind** is an extension of Cachegrind
  - Additional function call graph information

Very useful for performance tuning

# Callgrind Example: Program run under Callgrind

```
mliieber@phobos:~/loops
mliieber@phobos:~/loops> make
gcc -O2 -g -c loops-fast.c -o loops-fast.o
gcc -O2 -g -o loops-fast loops-fast.o
gcc -O2 -g -c loops-slow.c -o loops-slow.o
gcc -O2 -g -o loops-slow loops-slow.o
mliieber@phobos:~/loops> module load valgrind
Valgrind 3.4.1 loaded
Refer to http://valgrind.org/ for more info about the Valgrind tool suite.
mliieber@phobos:~/loops> valgrind --tool=callgrind --simulate-cache=yes ./loops-fast
==7827== Callgrind, a call-graph generating cache profiler.
==7827== Copyright (C) 2002-2008, and GNU GPL'd, by Josef Weidendorfer et al.
==7827== Using LibVEX rev 1884, a library for dynamic binary translation.
==7827== Copyright (C) 2004-2008, and GNU GPL'd, by OpenWorks LLP.
==7827== Using valgrind-3.4.1, a dynamic binary instrumentation framework.
==7827== Copyright (C) 2000-2008, and GNU GPL'd, by Julian Seward et al.
==7827== For more details, rerun with: -v
==7827==
==7827== For interactive control, run 'callgrind_control -h'.
sum = 10000.000
==7827==
==7827== Events      : Ir Dr Dw I1mr D1mr D1mw I2mr D2mr D2mw
==7827== Collected : 8144314 1031520 1013097 783 126671 125566 781 126529 125546
==7827==
==7827== I  refs:      8,144,314
==7827== I1 misses:    783
==7827== L2i misses:  781
==7827== I1 miss rate: 0.0%
==7827== L2i miss rate: 0.0%
==7827==
==7827== D  refs:      2,044,617 (1,031,520 rd + 1,013,097 wr)
==7827== D1 misses:   252,237 ( 126,671 rd + 125,566 wr)
==7827== L2d misses:  252,075 ( 126,529 rd + 125,546 wr)
==7827== D1 miss rate: 12.3% ( 12.2% + 12.3% )
==7827== L2d miss rate: 12.3% ( 12.2% + 12.3% )
==7827==
==7827== L2 refs:      253,020 ( 127,454 rd + 125,566 wr)
==7827== L2 misses:   252,856 ( 127,310 rd + 125,546 wr)
==7827== L2 miss rate: 2.4% ( 1.3% + 12.3% )
mliieber@phobos:~/loops>
```

# Callgrind Example: callgrind\_annotate

```
mliieber@phobos:~/loops
mliieber@phobos:~/loops> ls -l
total 84
-rw----- 1 mliieber zih 22776 2009-06-08 18:11 callgrind.out.7827
-rwxr-xr-x 1 mliieber zih 14208 2009-06-08 18:10 loops-fast
-rw-r--r-- 1 mliieber zih 559 2009-06-08 14:39 loops-fast.c
-rw-r--r-- 1 mliieber zih 6952 2009-06-08 18:10 loops-fast.o
-rwxr-xr-x 1 mliieber zih 14208 2009-06-08 18:10 loops-slow
-rw-r--r-- 1 mliieber zih 559 2009-06-08 14:39 loops-slow.c
-rw-r--r-- 1 mliieber zih 6944 2009-06-08 18:10 loops-slow.o
-rw-r--r-- 1 mliieber zih 275 2009-06-08 14:39 Makefile
mliieber@phobos:~/loops> callgrind_annotate callgrind.out.7827
-----
Profile data file 'callgrind.out.7827' (creator: callgrind-3.4.1)
-----
I1 cache: 65536 B, 64 B, 2-way associative
D1 cache: 65536 B, 64 B, 2-way associative
L2 cache: 1048576 B, 64 B, 8-way associative
Timerange: Basic block 0 - 2026375
Trigger: Program termination
Profiled target: ./loops-fast (PID 7827, part 1)
Events recorded: Ir Dr Dw I1mr D1mr D1mw I2mr D2mr D2mw
Events shown: Ir Dr Dw I1mr D1mr D1mw I2mr D2mr D2mw
Event sort order: Ir Dr Dw I1mr D1mr D1mw I2mr D2mr D2mw
Thresholds: 99 0 0 0 0 0 0 0 0
Include dirs:
User annotated:
Auto-annotation: off
-----
      Ir      Dr      Dw I1mr  D1mr  D1mw I2mr  D2mr  D2mw
-----
8,144,317 1,031,520 1,013,097 783 126,671 125,566 781 126,529 125,546 PROGRAM TOTALS
-----
      Ir      Dr      Dw I1mr  D1mr  D1mw I2mr  D2mr  D2mw  file:function
-----
4,007,016      4 1,000,004      2      3 125,001      2      3 125,001 /home/mliieber/loops/loops-fast.c:main [/work/home/mliieber/loops/loops-fast]
4,007,003 1,000,001      0      0 125,001      0      0 125,001      . /home/mliieber/loops/loops-fast.c:array_sum [/work/home/mliieber/loops/loops-fast]
  30,650   11,448   3,249   14   438      5   14   389      4 ???:do_lookup_x [/lib64/ld-2.3.3.so]
  20,318    5,109   2,436   33   666     264   33   628   254 ???:_dl_relocate_object [/lib64/ld-2.3.3.so]
mliieber@phobos:~/loops> █
```

# Callgrind Example: callgrind\_annotate with Source

```
mliieber@phobos:~/loops
  Ir    Dr    Dw  I1mr  D1mr  D1mw  I2mr  D2mr  D2mw
-----
-- line 3 -----
  .    .    .    .    .    .    .    .    .    .  #define N 1000
  .    .    .    .    .    .    .    .    .    .
  .    .    .    .    .    .    .    .    .    .  double array_sum(double[N][N]);
  .    .    .    .    .    .    .    .    .    .
  .    .    .    .    .    .    .    .    .    .  double array_sum(double a[N][N])
  .    .    .    .    .    .    .    .    .    .  {
  .    .    .    .    .    .    .    .    .    .    int i,j;
  1    .    .    .    .    .    .    .    .    .    .    double s;
3.001  .    .    .    .    .    .    .    .    .    .    s=0;
3.004,000  .    .    .    .    .    .    .    .    .    .  for(i=0;i<N;i++)
1.000,000 1.000,000 0 0 125,000 0 0 125,000  .    .    .    .    .    .  for(j=0;j<N;j++)
  .    .    .    .    .    .    .    .    .    .    .    s += a[i][j];
  .    .    .    .    .    .    .    .    .    .    .
  1    1    0 0 1 0 0 1  .    .    .    .    .    .    .  return s;
  .    .    .    .    .    .    .    .    .    .    .  }
  1    0    0 1 0 0 1  .    .    .    .    .    .    .  int main(int argc, char** argv)
  .    .    .    .    .    .    .    .    .    .    .  {
  .    .    .    .    .    .    .    .    .    .    .  double a[N][N];
  .    .    .    .    .    .    .    .    .    .    .  int i,j;
  .    .    .    .    .    .    .    .    .    .    .
  3.002 0 0 1 0 0 1  .    .    .    .    .    .    .  for(i=0;i<N;i++)
  .    .    .    .    .    .    .    .    .    .    .  {
3.004,000  .    .    .    .    .    .    .    .    .    .    .    for(j=0;j<N;j++)
  .    .    .    .    .    .    .    .    .    .    .    .    {
1.000,000 0 1,000,000 0 0 125,000 0 0 125,000  .    .    .    .    .    .  a[i][j]=0.01;
  .    .    .    .    .    .    .    .    .    .    .    .    }
  .    .    .    .    .    .    .    .    .    .    .    .    }
  .    .    .    .    .    .    .    .    .    .    .
  .    .    .    .    .    .    .    .    .    .    .  /* this is just to prevent the compiler
  10 3 4 0 2 1 0 2 1  .    .    .    .    .    .    .    .  from optimizing the upper loop away */
  4.771 1.181 688 198 42 21 198 42 21 => ???:printf (1x)
4.007,003 1.000,001 0 0 125,001 0 0 125,001  . => /home/mliieber/loops/loops-fast.c:array_sum (1x)
  1,048 272 101 0 59 10 0 59 10 => ???:_dl_runtime_resolve (1x)
  .    .    .    .    .    .    .    .    .    .    .
  .    .    .    .    .    .    .    .    .    .    .  return 0;
  3 1 0 0 1 0 0 1  .    .    .    .    .    .    .  }
-----
  Ir Dr Dw  I1mr  D1mr  D1mw  I2mr  D2mr  D2mw
-----
98 97 99  0 99 100  0 99 100 percentage of events annotated
mliieber@phobos:~/loops> callgrind_annotate --auto=yes callgrind.out.7827
```



# KCachegrind

callgrind.out.26724 [./loops-fast] - KCachegrind <@phobos>

File View Go Settings Help

L2 Miss Sum

Flat Profile

Search:  Source File

Self	Source File
250 007	loops-fast.c
2 843	(unknown)
2	start.S
2	elf-init.c
2	crti.S

Incl.	Self	Called	Function	Location
250 337	125 006	1	main	loops-fast: loo
125 001	125 001	1	array_sum	loops-fast: loo

### main

Types Callers All Callers Source Callee Map

#	L2m	Source ('/home/mlieber/loops/loops-fast.c')
23		
24	1	for(i=0;i<N;i++)
25		{
26		for(j=0;j<N;j++)
27		{
28	125 000	a[i][j]=0.01;
29		}
30		}
31		
32		/* this is just to prevent the compiler
33		from optimizing the upper loop away */
34	3	printf("sum = %10.3f\n",array_sum(a));
	125 001	1 call to 'array_sum' (loops-fast: loops-fast.c)
	261	1 call to 'printf' (libc.so.6)
	69	1 call to '_dl_runtime_resolve' (ld-2.3.3.so)
35		

Caller Map Parts Call Graph Callees All Callees Assembler

callgrind.out.26724 [1] - Total L2 Miss Sum Cost: 252 856

# KCachegrind

**./callgrind.out.30791 [./fd4test] - KCachegrind**

Datei Ansicht Gehe zu Einstellungen Hilfe

Abschätzung CPU-Takte

Kostenprofil

Suche:  Quelldatei

Exkl. | Quelldatei

- 504 424 fd4\_domain.F90
- 274 147 test.F90
- 262 653 fd4\_iter.F90
- 210 283 fd4\_util.F90
- 151 966 rbtree\_fd4\_block.F90
- 131 808 fd4\_vis5d.F90
- 99 097 fd4\_boundary.F90

Inkl. | Exkl. | Aufruf | Funktion

- 1 194 876 151 748 5 fd4\_iter\_mod\_mp\_fd4\_i...
- 39 601 39 601 89 fd4\_iter\_mod\_mp\_fd4\_i...
- 32 329 16 241 33 fd4\_iter\_mod\_mp\_fd4\_i...
- 26 344 26 344 64 fd4\_iter\_mod\_mp\_fd4\_i...
- 24 514 8 610 705 fd4\_iter\_mod\_mp\_fd4\_i...
- 17 714 17 714 1 fd4\_iter\_mod\_mp\_fd4\_i...
- 1 991 1 991 1 fd4\_iter\_mod\_mp\_fd4\_i...
- 1 058 404 16 fd4\_iter\_mod\_mp\_fd4\_i...

**MAIN\_**

Kostenarten Aufrufer Alle Aufrufer Quelle Aufrufkarte

for\_write\_seq\_lls\_xmit 1 471 562 3 192

for\_desc\_ret\_item 298 184 6 213

for\_format\_value cvt\_jeee\_t\_to... cvtas\_t to a 6...

for\_write\_seq\_fmt\_xmit cvt\_integer\_to\_text 636

for\_d... for\_interp\_fmt 710

for\_write\_seq\_lls... fd4\_block\_mod\_mp\_fd4\_block\_allocate\_ 1 635 364 64

for... fd4\_block\_mod\_mp\_f... 72

for... vSdCompressGrid compute\_ga\_gb 8

for... fd4\_util\_mod\_mp\_fd4\_util\_get\_array\_ 6

for... for\_write\_seq\_fmt 1 177 643

for\_acquire lun 668

for\_de... for\_write\_seq\_fmt\_xmit 751 651

for\_format\_value intel\_f... cvtas\_t to a 7... cvt\_integer\_to\_text 746

fd4\_block\_mod\_mp\_fd4\_block\_allocate\_ 358 432 69

for\_check\_mult\_o... 0x000000...

memset 617 986 1

fd4\_block\_mod\_mp\_fd4\_block\_allocate\_ 35

fd4\_block\_mod\_mp\_f... 72

fd4\_vis5d\_mod\_mp\_fd4\_vis5d\_write\_ 2

fd4\_util\_mod\_mp\_fd4\_util\_get\_array\_ 6

fd4\_boundary\_mod\_mp... 163 167

fd4\_util\_m... 198 592

vSdwrite\_ 206 355

MAIN\_ 8 725 153

fd4\_util\_mod\_mp\_fd4\_util\_put\_array\_ 587 450

fd4\_domain\_mod\_mp\_fd4\_domain\_delete\_ 126 129

fd4\_boundary\_mod\_mp\_fd4\_boundary\_zerograd\_ 99 097

fd4\_boundary\_mod\_mp\_fd4\_boundary\_set\_ 95 831

fd4\_util\_mod\_mp\_fd4\_util\_get\_array\_ 461 635

fd4\_vis5d\_mod\_mp\_fd4\_vis5d\_write\_ 695 291

for\_write\_seq\_lls\_xmit 380 503

16 094

50 877

1 635 364

383 055

1 194 876

126 129

95 831

56 550

142 042

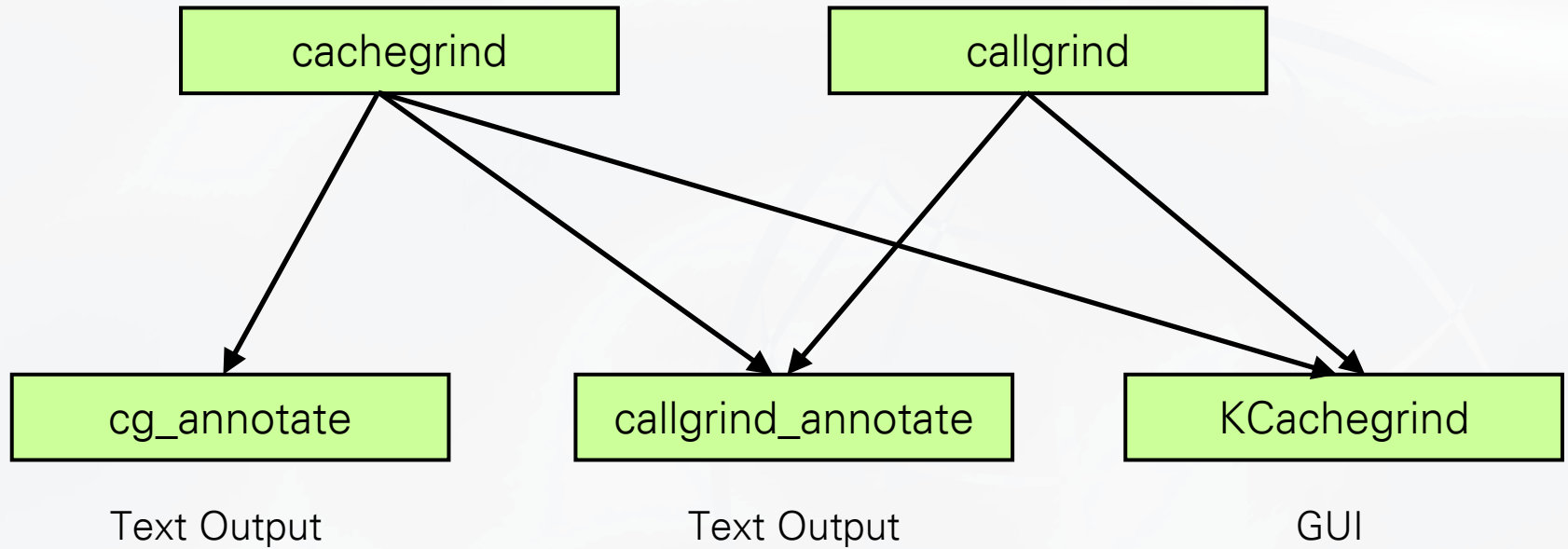
198 592

206 355

Aufrufkarte Profildaten Aufrufgraph Aufrufene Alle Aufrufenen Assembler

callgrind.out.30791 [1] - Gesamtkosten für Abschätzung CPU-Takte: 9 528 601

# Tool Chains



# Exercises

---

- `cd ~/callgrind`
- `make`

# Exercise 1 - loops-fast

```
1 #define N 1000
2
3 int main(int argc, char** argv)
4 {
5     double a[N][N];
6     int i,j;
7
8     for(i=0;i<N;i++)
9     {
10        for(j=0;j<N;j++)
11        {
12            a[i][j]=0.01;
13        }
14    }
15
16    printf("sum = %10.3f\n",array_sum(a));
17
18    return 0;
19 }
20
21 double array_sum(double a[N][N])
22 {
23     int i,j;
24     double s;
25     s=0;
26     for(i=0;i<N;i++)
27     for(j=0;j<N;j++)
28         s += a[i][j];
29
30     return s;
31 }
```

Fill 2D array

- Array size is 1000 x 1000 x 8 Byte = 8MB
- Phobos Cache:
  - 64kB L1i + 64kB L1d
  - 1MB L2
- Cache too small for array

Read the array

- If L2 cache was large enough (>8MB), no L2 miss would happen here!
- Because the upper loop loads the array into the cache

# Exercise 1 - loops-fast

---

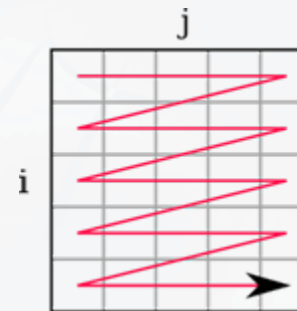
- Run the program with default cache settings:
  - `valgrind --tool=callgrind --simulate-cache=yes ./loops-fast`
  - 125.000 L1 and L2 misses in the write loop
  - 125.000 L1 and L2 misses in the read loop
- Run the program with custom cache settings, e.g. 16MB L2 cache:
  - `valgrind --tool=callgrind --simulate-cache=yes --L2=16777216,2,64 ./loops-fast`
  - Array fits in the L2 cache
  - No L2 misses in the read loop anymore
- View results with `callgrind_annotate`:
  - `callgrind_annotate --auto=yes ./callgrind.out.XXXX`
- View the results with `KCachegrind`:
  - `module load kcachegrind`
  - `kcachegrind ./callgrind.out.XXXX`

# Exercise 2 - loops-slow

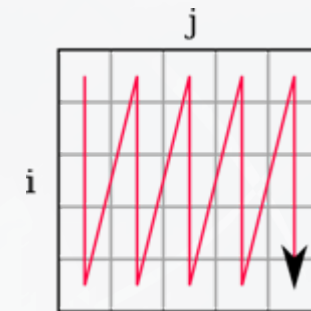
```
1 #define N 1000
2
3 int main(int argc, char** argv)
4 {
5     double a[N][N];
6     int i,j;
7
8     for(j=0;j<N;j++)
9     {
10        for(i=0;i<N;i++)
11        {
12            a[i][j]=0.01;
13        }
14    }
15
16    printf("sum = %10.3f\n",array_sum(a));
17
18    return 0;
19 }
20
21 double array_sum(double a[N][N])
22 {
23     int i,j;
24     double s;
25     s=0;
26     for(i=0;i<N;i++)
27         for(j=0;j<N;j++)
28             s += a[i][j];
29
30     return s;
31 }
```

Fill 2D array, loop has "bad" order

C Storage:  
row-major order



Access here:  
column-major order



- Array size is 1000 x 1000 x 8 Byte = 8MB
- Phobos Cache:
  - 64kB L1i + 64kB L1d
  - 1MB L2
- Cache too small for array

# Exercise 2 - loops-slow

```
1 #define N 1000
2
3 int main(int argc, char** argv)
4 {
5     double a[N][N];
6     int i, j;
7
8     for(j=0; j<N; j++)
9     {
10        for(i=0; i<N; i++)
11        {
12            a[i][j]=0.01;
13        }
14    }
15
16    printf("sum = %10.3f\n", array_sum(a));
17
18    return 0;
19 }
20
21 double array_sum(double a[N][N])
22 {
23     int i, j;
24     double s;
25     s=0;
26     for(i=0; i<N; i++)
27         for(j=0; j<N; j++)
28             s += a[i][j];
29
30     return s;
31 }
```

Fill 2D array, loop  
has "bad" order

- Cache misses will occur even if cache was large enough
  - Because array is not in cache
- But how many cache misses will occur?
  - Cache line size: 64 Byte = 8 double
  - 1.000.000 misses?
  - 125.000 misses?
- Try small L1 and L2 cache size (32768 Byte each) vs. default cache size setup
- Then try the example with "good" ordered loop with the small cache sizes and compare to "bad" ordered loop

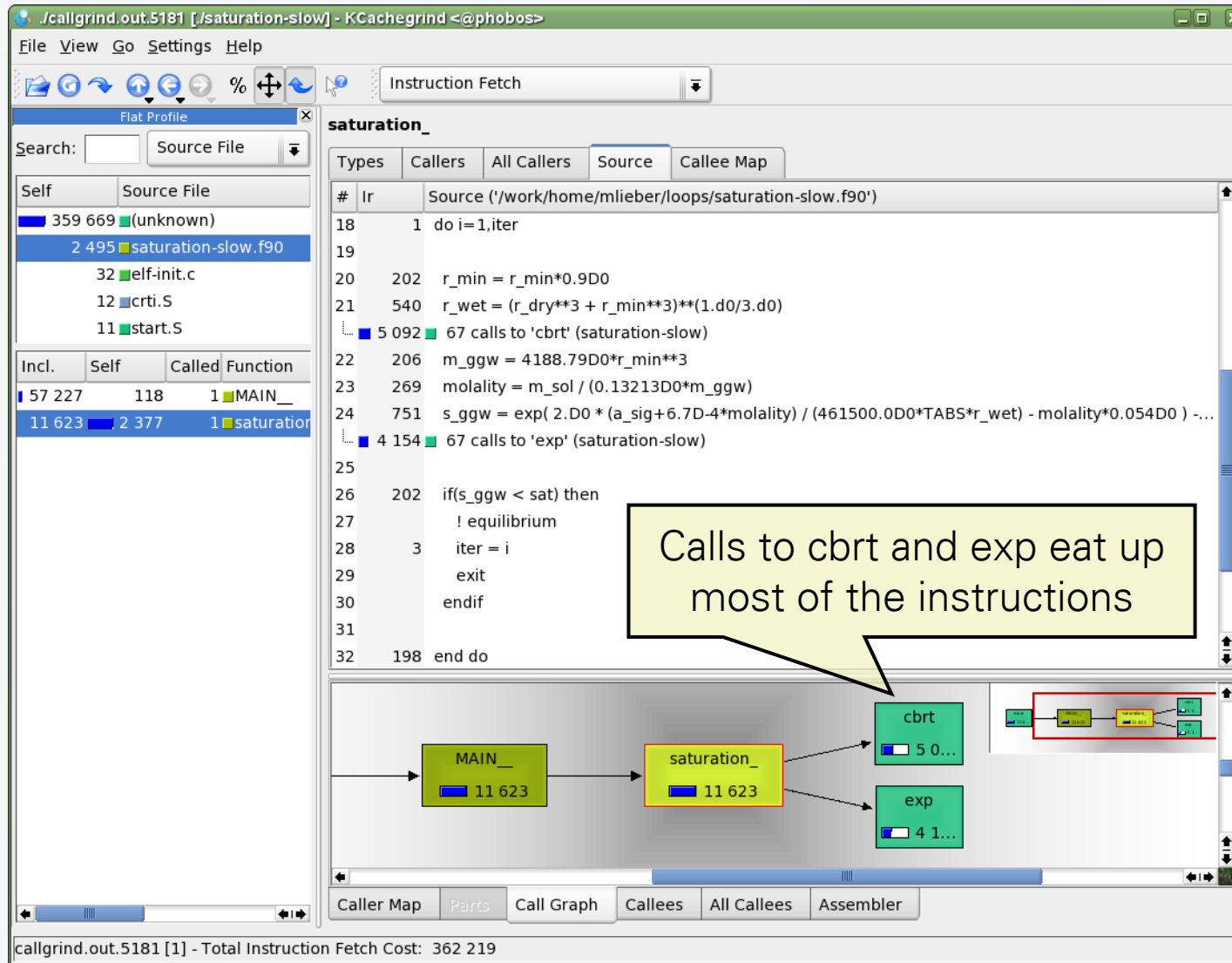


# Beyond Cache Misses

---

- Callgrind can also be used to find performance problems which are not related to CPU cache
  - What code lines eat up most instructions (CPU cycles, time)
  - What system/math/library functions are called and what do they cost?
- Recorded instructions can be a measure of computational costs in cache-friendly code
- KCachegrind's cycle estimation allows incorporation of cache misses in this measure

# Math function in KCachegrind: cbrt and exp are costly!



# Math function in KCachegrind: optimized version – no exp!

callgrind.out.6106 [1] - Total Instruction Fetch Cost: 348 022

Instructions dropped down from 11623 to 7199

```
19  sat_log = log(sat+1.0d0)
20
21  1 do i=1,iter
22
23  202  r_min = r_min*0.9D0
24  540  r_wet = (r_dry**3 + r_min**3)**(1.d0/3.d0)
25  5 092 67 calls to 'cbrt' (saturation-fast)
26  72  m_ggw = 4188.79D0*r_min**3
27  269  molality = m_sol / (0.13213D0*m_ggw)
28
29  616  s_ggw = 2.D0 * (a_sig+6.7D-4*molality) / (461500.0D0*TABS*r_wet) - molality*0.054D0
30
31  202  if(s_ggw < sat) then
32  ! equilibrium
33  3  iter = i
34  exit
35  endif
36
37  198  end do
```

## Exercise 3 - saturation

---

- Compare the slow and the fast version of the saturation example
  - Don't need to collect cache counters
  - **valgrind --tool=callgrind ./saturation-fast**
  - **valgrind --tool=callgrind ./saturation-slow**
- Display in KCachegrind

# Selected Callgrind Command-Line Options

---

- **--simulate-cache=[yes|no]** – enable cache simulation
- **--dump-instr=[yes|no]** – collect information at per-instruction granularity, only useful for assembler view in KCachegrind
- **--callgrind-out-file=<file>** - output file
- **--I1=<size>, <associativity>, <line size>** - specify L1 instruction cache
- **--D1=<size>, <associativity>, <line size>** - specify L1 data cache
- **--L2=<size>, <associativity>, <line size>** - specify L2 cache
  
- More features and options:
  - User Manual: <http://valgrind.org/docs/manual/cg-manual.html>
  - **valgrind --tool=callgrind --help**

# Summary

---

- Remember: Valgrind is based on simulation, no measurements!
  - Don't trust the results to be absolutely accurate
  - Large Overhead
- Whenever using Cachegrind / Callgrind:
  - Reduce problem size, but should still be representative
  - Large application: extract computational kernel routines
- Easy to use
- But not available for IA64 (Altix)
- Callgrind\_annotate is a good alternative when KCachegrind is not available (KCachegrind requires X11 and KDE libs)