Memory Checking and Single Processor Optimization with Valgrind

Rainer Keller

University of Stuttgart
High-Performance Computing-Center Stuttgart (HLRS)
www.hlrs.de

Outline

- Motivation
- Valgrind
  - Memcheck
  - Massif
  - Callgrind
  - Kcachegrind
  - Kcachegrind with RNAfold
  - Kcachegrind with Matrix Multiplication
  - Installation & General Usage
- Summary

Motivation – Performance Optimization

- You want the best performance possible for Your platform.
- Time-constraints of Your application.
- Before thinking about parallelizing your application for 2-4 processors: Optimize it and double performance instead,-]
- Or do both…

Valgrind – Overview

- An Open-Source Debugging & Profiling tool.
- Works with any dynamically & statically linked application.
- Emulates CPU, i.e. executes instructions on a synthetic x86.
- Currently it's only available for Linux/IA32
- Prevents Error-swamping by suppression-files.
- Has been used on many **large** Projects:
  KDE, Emacs, Gnome, Mozilla, OpenOffice.
- It's easily configurable to ease debugging & profiling through *skins*:
  - **Memcheck**: Complete Checking (every memory access)
  - Addrcheck: 2xFaster (no uninitialized memory check).
  - Cachegrind: A memory & cache profiler
  - **Callgrind**: A Cache & Call-tree profiler.
  - Helgrind: Find Races in multithreaded programs.
- How to use with MPIch: http://www.hlrs.de/people/keller

Valgrind – Usage

- Programs should be compiled with
  - Debugging support (to get position of bug in code)
  - Possibly without Optimization (for accuracy of position & less false positives):
    `gcc -O0 -g -o test test.c`
- Run the application as normal as parameter to valgrind:
  `valgrind ./test`
- Then start the MPI-Application as with TV as debugger:
  `mpirun -dbg=valgrind ./mpi_test`

Valgrind – Memcheck

- Checks for:
  - Use of uninitialized memory
  - Malloc Errors:
    - **Usage of free'd memory**
    - **Double free**
    - **Reading/writing past malloced memory**
    - **Lost memory pointers**
    - **Mismatched malloc/new & free/delete**
  - Stack write errors
  - Overlapping arguments to system functions like `memcpy`.

Valgrind – Example 1/3

```
#define SIZE 10

int main (int argc, char * argv[])
{
  int size;
  int rank;
  int * array;
  MPI_Status status;

  MPI_CHECK (MPI_Init (&argc, &argv));
  MPI_CHECK (MPI_Comm_rank (MPI_COMM_WORLD, &rank));
  MPI_CHECK (MPI_Comm_size (MPI_COMM_WORLD, &size));

  array = malloc (SIZE * sizeof (int));

  if (rank == 0)
    MPI_Recv (array, SIZE+1, MPI_INT, 1, 4711, MPI_COMM_WORLD, &status);
  else
    MPI_Send (array, SIZE+1, MPI_INT, 0, 4711, MPI_COMM_WORLD);

  printf ("(Rank:%d) array[0]:%d\n", rank, array[0]);
  MPI_CHECK (MPI_Finalize ());
  return 0;
}
rusraink@pcgpc9:~/C/MPI_TESTS >
```


Valgrind – Example 2/3

```
#define SIZE 10

int main (int argc, char * argv[])
{
  int size;
  int rank;
  int * array;
  MPI_Status status;

  array = malloc (SIZE * sizeof (int));

  if (rank == 0)
    MPI_Recv (array, SIZE+1, MPI_INT, 1, 4711, MPI_CO
  else
    MPI_Send (array, SIZE+1, MPI_INT, 0, 4711, MPI_CO

  printf ("(Rank:%d) array[0]:%d\n", rank, array[0]);
  MPI_CHECK (MPI_Finalize ());
  return 0;
}
rusraink@pcgpc9:~/C/MPI_TESTS >
```

Valgrind – Example 3/3

PID

- With Valgrind `mpirun -dbg=valgrind -np 2 ./mpi_murks`:

```
==11278== Invalid read of size 1
==11278==    at 0x4002321E: memcpy (../../memcheck/mac_replace_strmem.c:256)
==11278==    by 0x80690F6: MPID_SHMEM_Eagerb_send_short (mpich/../shmemshort.c:70)
.. 2 lines of calls to MPIch-functions deleted ...
==11278==    by 0x80492BA: MPI_Send (/usr/src/mpich/src/pt2pt/send.c:91)
==11278==    by 0x8048F28: main (mpi_murks.c:44)
==11278== Address 0x4158B0EF is 3 bytes after a block of size 40 alloc'd
==11278==    at 0x4002BBCE: malloc (../../coregrind/vg_replace_malloc.c:160)
==11278==    by 0x8048EB0: main (mpi_murks.c:39)
```

Buffer-Overrun by 4 Bytes in `MPI_Send`

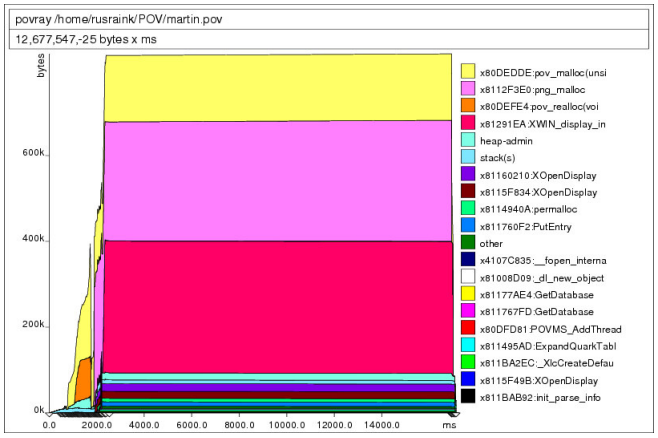
```
....

==11278== Conditional jump or move depends on uninitialised value(s)
==11278==    at 0x402985C4: _IO_vfprintf_internal (in /lib/libc-2.3.2.so)
==11278==    by 0x402A15BD: _IO_printf (in /lib/libc-2.3.2.so)
==11278==    by 0x8048F44: main (mpi_murks.c:46)
```

Printing of uninitialized variable

- It can not find:
  - May be run with 1 process: One pending Recv (→ use Marmot).
  - May be run with >2 processes: Unmatched Sends (→ use Marmot).

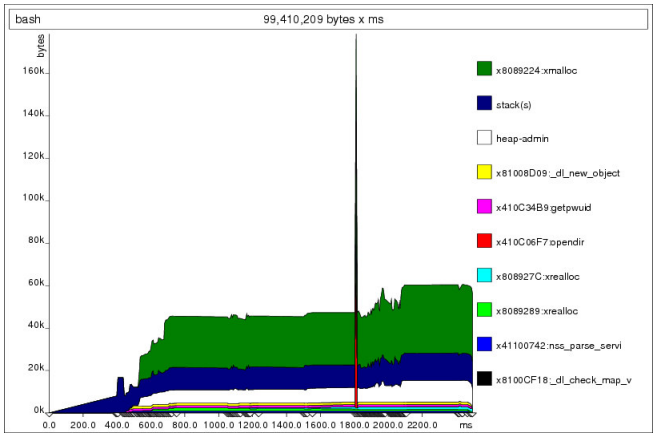
Valgrind – Massif 1/2

- The massif skin allows tracing of memory consumption over time:



Valgrind – Massif 2/2

- The massif skin allows tracing of memory consumption over time:



Valgrind – Callgrind 1/2

- The Callgrind skin (formerly Calltree skin):
  - Tracks memory accesses to check Cache-hit/misses.
  - Additionally records call-tree information.

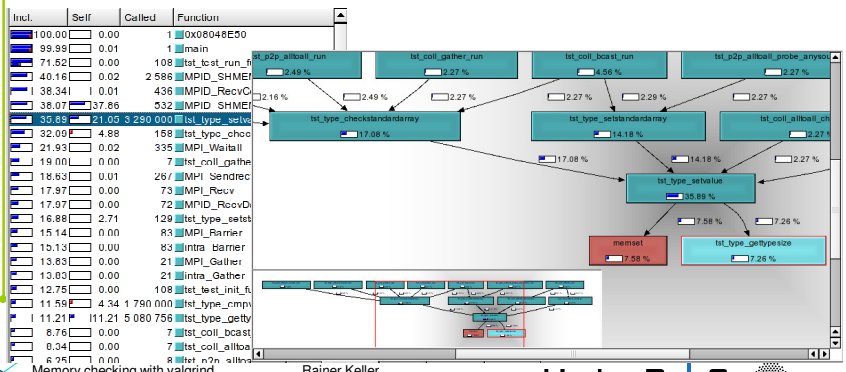
```
==11745== Calltree-0.9.6, a cache profiler for x86-linux.
==11745== Copyright (C) 2002, and GNU GPL'd, by N.Nethercote and J.Weider
==11745== Using valgrind-2.1.0, a program supervision framework for x86-l
==11745== Copyright (C) 2000-2003, and GNU GPL'd, by Julian Seward.
--11745-- warning: Pentium with 12 K micro-op instruction trace cache
--11745--          Simulating a 16 KB cache with 32 B lines
==11745== Estimated CPU clock rate is 1410 MHz
==11745== For more details, rerun with: -v
--11745--
```

- After the run, it reports overall program statistics:

```
==11810== D   refs:      497,790,574  (386,176,612 rd + 111,613,962 wr)
==11810== D1  misses:        863,493  (    369,495 rd +     493,998 wr)
==11810== L2d misses:        282,232  (     98,857 rd +     183,375 wr)
==11810== D1  miss rate:         0.1% (        0.0%   +         0.4%  )
==11810== L2d miss rate:         0.0% (        0.0%   +         0.1%  )
```

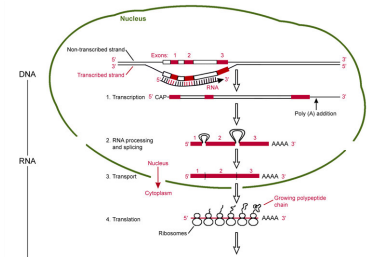
Valgrind – Callgrind 2/2

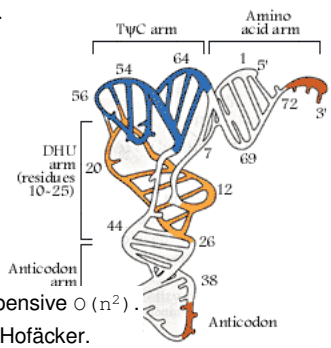
- Even more interesting: the output trace-file.
- With the help of kcachegrind, one may:
  - Investigate, where Instr/L1/L2-cache misses happened.
  - Which functions were called where & how often.



Valgrind – RNAfold 1/8

- RNAfold is a MPI-parallel application for computing the 2D-folding of a single-stranded RNA-sequence.
- The tertiary (3-D) structure defines the function of the RNA, computation is very expensive but may be predicted out of the secondary structure.
- The tightly coupled MPI-application RNAfold computes the secondary structure of minimal free energy of long RNA sequences.





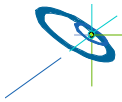
- Computationally $O(n^3)$ and communication expensive $O(n^2)$.
- Derived out of the Vienna-RNA package of Ivo Hofäcker.

Valgrind – RNAfold 2/8

- Running RNAfold with Valgrind/Callgrind for kcachegrind:
  `mpirun -np 4 -dbg=callgrind ./RNAfold test_1000.seq descr`
- This internally starts via rsh several processes:
  `valgrind -tool=callgrind -simulate-cache=yes -dump-instr=yes -collect-jumps=yes ./RNAfold test_1000.seq -p4pg Pixxx -p4wd /home/xxx`
- The advantage is, you may run several processes on **one** processor and emulate several processors; we we look at caching information.
- However, it runs very slow (2 MPI-processes, single-CPU machine):

N	No valgrind	With Valgrind	Factor
500	2,19	373,45	170
1000	8,97	1308,64	146
2000	46,66	7012,05	150

- This is due to:
  - Valgrind emulating every instructions and memory dereference, also MPIs
  - RNAfold being compiled with `-O0 -g`.



Valgrind – RNAfold 3/8

- The output for the 2000-base sequence run is:

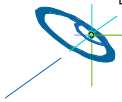
```
I    refs:      52,035,392,345
I1   misses:           323,136
L2i  misses:           239,455
I1   miss rate:            0.0%
L2i  miss rate:            0.0%
```

Instruction Cache information:
- Level-1 Cache misses
- Level-2 Cache misses
- Miss rate

- Data Cache information (L1 and L2 cache misses – read & write):

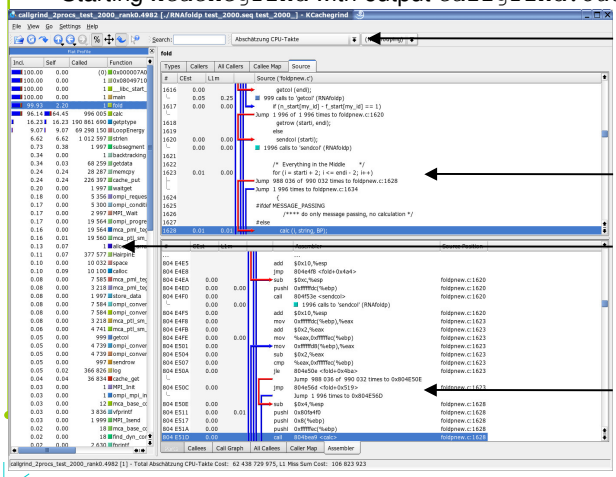
```
D    refs:      30,047,022,954  (22,966,284,972 rd + 7,080,737,982 wr)
D1   misses:       106,500,787  (   101,232,858 rd +     5,267,929 wr)
L2d  misses:        93,111,529  (    88,944,909 rd +     4,166,620 wr)
D1   miss rate:            0.3% (           0.4%   +           0.0%  )
L2d  miss rate:            0.3% (           0.3%   +           0.0%  )
```

```
L2 refs:          106,823,923  (   101,555,994 rd +     5,267,929 wr)
L2 misses:         93,350,984  (    89,184,364 rd +     4,166,620 wr)
L2 miss rate:            0.1% (           0.1%   +           0.0%  )
```



Valgrind – RNAfold 4/8

- Starting `kcachegrind` with output `callgrind.out.PID`:



Cost-function
- Instruction load
- L1 Cache misses

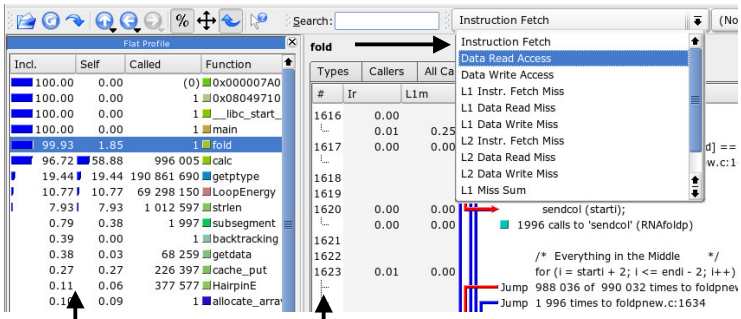
Source with:
- Line number
- 1st cost (here Instr)
- 2nd cost (D1mr)

Break down of
- Costs of function
- Times called
- Source/Object file

Output if
- Assembler (dump-instr
- Jump info (trace-jumps)
- Costs per instruction

Valgrind – RNAfold 5/8

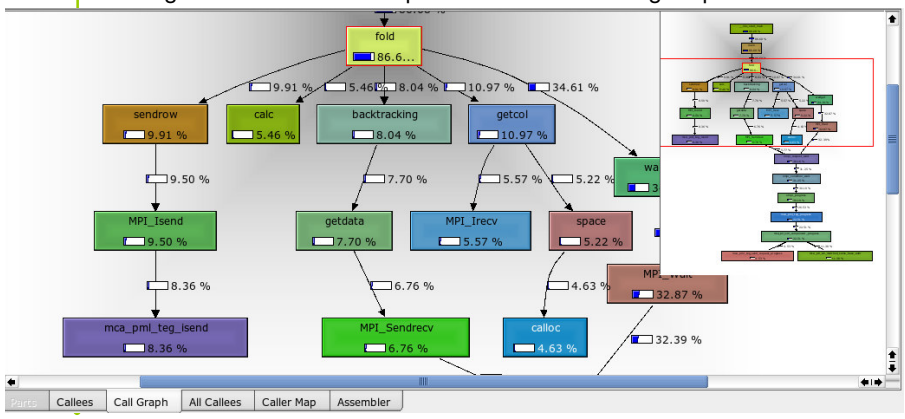
- The following cost functions may be analyzed:



- This (primary) cost function is shown per:
  - Line (Source View)
  - Function, aggregated over whole execution (Flat profile)
  - Assembler instruction (Assembler view – not shown here.

Valgrind – RNAfold 6/8

- To get an overview of the performance and calling sequence:



(Please note: Cost function chosen to see all possible functions called)

Valgrind – RNAfold 7/8

- Most important spots to improve for single-processor performance:

100.00	0.00	1	main
99.93	2.20	1	fold
96.14	64.45	996 005	calc
16.23	16.23	190 861 690	getptype
9.07	9.07	69 298 150	LoopEnergy
6.62	6.62	1 012 597	strlen
0.73	0.38	1 997	subsegment

- Most time spend in function `calc`.
- Functions `calc` and `LoopEnergy` need to be inlined.
- Can't help `strlen`, it's libc.

- Look at the biggest CPU-time consumer in `calc`:

```
944
945                 /* modular decomposition ----------------------------- */
946
947   7.39          for (decomp = INF, k = i + 1 + TURN; k <= j - 2 - TURN; k++)
                    Jump  996 005 times to foldpnew.c:951
948  42.72  77.33     decomp = MIN2 (decomp, fMLrow[i][k - i + 1] + fMLcol[j][k + 1]);
949
950
951   0.02   0.11    DMLrow[i][0] = decomp;      /*  --- store for use in ML decompositon */
952   0.01           new_fML = MIN2 (new_fML, decomp);
```

2nd cost function: Level-1 Cache miss sum

1st cost function: Estimated CPU-time

Valgrind – RNAfold 8/8

- Immediate things to do:

```
585                 PRIVATE INLINE char
586                 getptype (int ki, int kj, int kn)
587   0.99   0.00   {
588   0.33           int abtype = 0;
589   0.33   0.00    int comptype = 0;
590
591   1.32            if ((ki >= kn - TURN - 1))
 ...                Jump  467 593 909 of  467 593 917 times to foldpnew.c:596
592                     {
593   0.00               return abtype;
 ...                Jump  8 times to foldpnew.c:603
594                     }
595
596   5.28   0.39      abtype = pair[S[ki]][S[kj]];
```

Force the compiler to inline function `getptype`.

Hinting to the compiler, that jump is unlikely: `builtin_expect(x,0)`.

- Very intrusive things to optimize:
  - Compress pair table (instead of char table), 3 bits per base
  - Check layout of `ccol`, `crow`, `fMLrow`, `fMLcol` matrices…

Valgrind – Verification of Matrix multiplication

- Some clarifications with the C-code implementation:

```
#define MATRIX_TYPE double
#define MATRIX_IJ(m,i,j)   ((m)[(i)*MATRIX_SIZE+(j)])
```

- On ia32, double is 8 Byte floating point type.
- Use a macro to access matrix in 1-dim array in row-major order.
- Fortran uses Column-major, C uses Row-major Matrix-storage:

```
A=0
do I=1,1024
  do J=1,1024
    A(I,J) = B(I,J) ! Inefficient row-major access
  end do
end do
```

Valgrind – Verification of Matrix Multiplication

- The naïve Matrix Multiplication is a $O(n^3)$ loop:

```
for (i = 0; i < MATRIX_SIZE; i++)
    for (k = 0; k < MATRIX_SIZE; k++)
        for (j = 0; j < MATRIX_SIZE; j++)
            MATRIX_IJ (matrix_c, i, j) +=
             (MATRIX_IJ (matrix_a, i, k) *
              MATRIX_IJ (matrix_b, k, j));
```

- A simple test for different execution orders of the loops shows:

Matrix size	IJK	IKJ	JKI	JIK	KIJ	KJI
500	1,41	0,84	2,36	1,47	1,69	3,01
1000	12,69	7,12	21,36	12,55	13,33	28,31
1500	44,21	21,97	68,03	40,55	45,31	90,63
(icc) 1500	35,48	20,25	58,73	34,72	47,02	87,32

- The performance again depends largely on the usage of the cache.
- Optimal naïve solution may hardly be optimized by good compilers (I.e. the cache usage / memory bandwidth is the boundary).

Valgrind – Verification of Matrix Multiplication

- Running the resulting six loop variations proves the cache-effectiveness (or **total lack** of it!)

main

Types | Callers | All Callers | Callee Map | Source

#	L2m	L1m	Source ('matrix_multiplication.c')
182	16.50	30.65	MATRIX_IJ (matrix_c, i, j) += (MATRIX_IJ (matrix_a, i, k) * MATRIX_IJ (matrix_b, k, j));
141	16.50	30.65	MATRIX_IJ (matrix_c, i, j) += (MATRIX_IJ (matrix_a, i, k) * MATRIX_IJ (matrix_b, k, j));
115	16.50	17.27	MATRIX_IJ (matrix_c, i, j) += (MATRIX_IJ (matrix_a, i, k) * MATRIX_IJ (matrix_b, k, j));
155	16.50	17.24	MATRIX_IJ (matrix_c, i, j) += (MATRIX_IJ (matrix_a, i, k) * MATRIX_IJ (matrix_b, k, j));
169	16.50	1.92	MATRIX_IJ (matrix_c, i, j) += (MATRIX_IJ (matrix_a, i, k) * MATRIX_IJ (matrix_b, k, j));
128	16.50	1.92	MATRIX_IJ (matrix_c, i, j) += (MATRIX_IJ (matrix_a, i, k) * MATRIX_IJ (matrix_b, k, j));

Size:600x600
Work-size: 8,23 MB
Cache-Clean: 4MB

Valgrind-slowdown:
150x-170x

- Trace shows L1 and L2 misses (sorted by L1m), line numbers:

	IJK	IKJ	JKI	JIK	KIJ	KJI
Time (10^3)	12,69	7,12	21,36	12,55	13,33	28,31
Line	115	128	141	155	169	182

The innermost IKJ-loop in the ASM-viewer shows nicely the reference, multiplication and addition.

```
                 fld   %st(0)
16.47  1.92      fmull (%edx)                          matrix_multiplication.c:128
                 lea   (%ecx,%ebx,1),%eax
                 mov   0xffffffefc(%ebp),%edi
                 ...
                 add   $0x8,%edx
                 cmp   $0x257,%ecx
 0.03  0.00      faddl (%edi,%eax,8)                   matrix_multiplication.c:128
                 fstpl (%edi,%eax,8)
                 jle   8048886 <main+0x2e6>            matrix_multiplication.c:127
                 Jump  215 640 000 of  216 000 000 times...
                 fstp  %st(0)
```

Valgrind – Verification of Matrix Multiplication

- For optimal performance on cache-based systems,
  use a blocking approach with a blocksize, which fits into half cache:

```
for (kb = 0; kb < SIZE; kb += BLOCK_SIZE) {
  ke = MIN2 (kb + BLOCK_SIZE, SIZE);

  for (ib = 0; ib < SIZE; ib += BLOCK_SIZE) {
    ie = MIN2 (ib + BLOCK_SIZE, SIZE);

    for (jb = 0; jb < SIZE; jb += BLOCK_SIZE) {
      je = MIN2 (jb + BLOCK_SIZE, SIZE);

      for (i = ib; i < ie; i++)
        for (k = kb; k < ke; k++) {
          for (j = jb; j < je; j++)
            MATRIX_IJ (matrix_c, i, j) +=
              MATRIX_IJ (matrix_a, i, k) *
              MATRIX_IJ (matrix_b, k, j);        }}}}
```


Valgrind – Verification of Matrix Multiplication

- Comparison of simple loop to blocked version (IJK-loop order):

	L1 misses	L2 misses
Simple Loop	243 Mio	27 Mio
Blocked	29 Mio	0,4 Mio

	Blocked		Simple
Blocksize	IKJ	JKI	IKJ
16	12,22	18,03	21,97
32	12,37	20,44	
48	11,31	25,27	
64	11,21	29,24	
92	11,37	35,67	
128	11,31	40,55	
160	11,06	39,46	
192	11,05	40,21	

Callgrind – Usage

- In order to create a callgrind-output for Kcachegrind:

```
valgrind --tool=callgrind
  --base=cachegrind.out
  --simulate-cache=yes
  --dump-instr=yes
  --collect-jumps=yes    ./application
```

- Then open generated `cachegrind.out.PID`-file with kcachegrind
- If you do not specify `--base`, kcachegrind expects file-prefix `cachgrind.out.xxx`.

Installation of Valgrind, Callgrind & Kcachegrind

- If you want to download and install the tool, this is straightforward.
- Valgrind:
  - `configure`
  - `make && make install`
- Callgrind:
  - Since valgrind and callgrind uses the pkg-config tool, set `PKG_CONFIG_PATH=/usr/local/lib/pkgconfig`
  - `configure`
  - `make && make install`
- Kcachegrind:
  - Please specify: `./configure -prefix=XXX (/usr) -with-qt-dir=XXX (/usr/lib/qt-3.1)`
  - `make && make install`

Valgrind and Callgrind usage

- In order to use Callgrind & Valgrind:

```
valgrind -tool=callgrind -help
dump creation options:
  --base=<prefix>           Prefix for profile files [callgrind.out]
  --dump-instr=no|yes       Dump instruction address of costs? [no]
  --collect-jumps=no|yes    Collect jumps? [no]
  --collect-alloc=no|yes    Collect memory allocation info? [no]
  --collect-systime=no|yes  Collect system call time info? [no]

cost entity separation options:
  --separate-threads=no|yes Separate data per thread [no]
  --fn-skip=<function>      Ignore calls to/from function?

cache simulator options:
  --simulate-cache=no|yes   Do cache simulation [no]
  --simulate-wb=no|yes      Count write-back events [no]
  --simulate-hwpref=no|yes  Simulate hardware prefetch [no]
  --I1=<size>,<assoc>,<line_size>  set I1 cache manually
  --D1=<size>,<assoc>,<line_size>  set D1 cache manually
  --L2=<size>,<assoc>,<line_size>  set L2 cache manually
```


Valgrind – Deficiencies

- Valgrind cannot find of these Error-Classes:
  - Semantic Errors
  - Timing-critical errors
  - Uninitialized stack-memory not detected.
  - Problems with new instruction sets
    (e.g. SSE/SSE2 is supported, certain Opcode, 3dNow, is **not**).
    When using the Intel-Compiler: Use `-tpp5` for Pentium optimization, if you have unsupported Opcodes.

Valgrind – Summary

- Valgrind is a good framework for:
  - Buffer overrun detection
  - Memory leak detection
  - Uninitialized Memory detection
  - Erroneous system call usage
  - Pthread-error detection
- Due to versatile, modular architecture many good tools:
  - Memcheck
  - Massif
  - Callgrind
- And due to the way of checking of every memory reference:
  - For performance and cache usability.