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)
Matthias Lieber (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

This code is much better suited for Analysis Tools

![Diagram of Valgrind's dynamic recompilation process]

Machine Code → Valgrind Core → Byte Code → Valgrind Tool (e.g. Callgrind) → Instrumented Byte Code → Valgrind Core → Instrumented Machine Code
**Cache Profiling with 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

```
mieber@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
mieber@phobos:/loops> module load valgrind
Valgrind 3.4.1 loaded
Refer to http://valgrind.org/ for more info about the Valgrind tool suite.
mieber@phobos:/loops> valgrind --tool-callgrind --siminter-cache=yes ./loops-fast
--- Callgrind, a call-graph generating cache profiler.
--- Copyright (C) 2002-2008, and GNU GPL'd, by Josef Waidendorfer et al.
--- Using LibVEX rev 1884, a library for dynamic binary translation.
--- Copyright (C) 2004-2008, and GNU GPL'd, by OpenWorks LLP.
--- Using valgrind-3.4.1, a dynamic binary instrumentation framework.
--- Copyright (C) 2000-2008, and GNU GPL'd, by Julian Seward et al.
--- For more details, rerun with: -v
--- For interactive control, run 'callgrind_control -h'.
sum  = 100000.000
---
--- Events  :    Lr Dr Dw Ilmr D1mr D1mw I2mr D2mr D2mw
--- Collected : 8144314 1031520 1013097 783 126671 125566 781 126529 125546
---
--- I refs:       0.144,314
--- I1 misses: 783
--- L21 misses: 781
--- I1 miss rate: 0.0%
--- L21 miss rate: 0.0%
---
--- D refs: 2.044,617  (1.031,500 rd + 1.013,097 wr)
--- D1 misses: 252,237  ( 126,671 rd + 125,566 wr)
--- L2d misses: 252,075  ( 126,529 rd + 125,546 wr)
--- D1 miss rate: 12.3% ( 12.2% + 12.3% )
--- L2d miss rate: 12.3% ( 12.2% + 12.3% )
---
--- L2 refs: 253,020  ( 127,454 rd + 125,566 wr)
--- L2 misses: 252,856  ( 127,310 rd + 125,546 wr)
--- L2 miss rate: 2.4% ( 1.3% + 12.3% )
mieber@phobos:/loops> ```
Cache Profiling with Callgrind

Callgrind Example: callgrind_annotate

```
mlieder@phobos:/loops> ls -l
total 84
-rw------- 1 mlieder zih 22776 2009-06-08 18:11 callgrind.out.7827
-rwxr-xr-x 1 mlieder zih 14208 2009-06-08 18:10 loops-fast
-rw-r--r-- 1 mlieder zih  559 2009-06-08 14:39 loops-fast.c
-rw-r--r-- 1 mlieder zih  692 2009-06-08 18:10 loops-fast.o
-rwxr-xr-x 1 mlieder zih 14208 2009-06-08 18:10 loops-slow
-rw-r--r-- 1 mlieder zih  559 2009-06-08 14:39 loops-slow.c
-rw-r--r-- 1 mlieder zih  694 2009-06-08 18:10 loops-slow.o
-rw-r--r-- 1 mlieder zih  275 2009-06-08 14:39 Makefile
mlieder@phobos:/loops> callgrind_annotate callgrind.out.7827
```

Profile data file 'callgrind.out.7827' (creator: callgrind-3.4.1)

```
II cache: 65536 B, 64 B, 2-way associative
DI 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 Ilmr D1mr D1mu D2mr D2mu
Events shown: Ir Dr Dw Ilmr D1mr D1mu D2mr D2mu
Thresholds:
Include dirs:
User annotated:
Auto-annotation: off

```

```
8.144.317 1.031.520 1.013.097 783 126.671 125.566 781 126.529 125.546 PROGRAM TOTALS

```

```
<table>
<thead>
<tr>
<th>File:</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>loops-fast.c:main</td>
<td>/home/mlieder/loops/loops-fast.c:main</td>
</tr>
<tr>
<td>loops-fast.c:array_sum</td>
<td>/home/mlieder/loops/loops-fast.c:array_sum</td>
</tr>
<tr>
<td>loops-fast</td>
<td>/home/mlieder/loops/loops-fast</td>
</tr>
<tr>
<td>4</td>
<td>do_lookup_x [lib64/ld-2.3.3.so]</td>
</tr>
<tr>
<td>4</td>
<td>dl_relocate_object [lib64/ld-2.3.3.so]</td>
</tr>
</tbody>
</table>
```

mlieder@phobos:/loops>
Callgrind Example: callgrind_annotate with Source

```c
#define N 1000
double array_sum(double[N][N]):

double array_sum(double a[N][N])
{
    int i,j;
    double s;
    s=0;
    for(i=0;i<N;i++)
        for(j=0;j<N;j++)
            s += a[i][j];
    return s;
}

int main(int argc, char** argv)
{
    double a[N][N];
    int i,j;
    
    for(i=0;i<N;i++)
    {
        for(j=0;j<N;j++)
        {
            a[i][j]=0.01;
        }
    }

    /* this is just to prevent the compiler
       from optimizing the upper loop away */
    printf("sum = %0.3f\n",array_sum(a));
}
```

Callgrind Example: callgrind_annotate with Source

```
98 97 99 0 99 100 0 98 100 percentage of events annotated
```

mieber@phobos:"/loops" callgrind_annotate --auto=yes callgrind.out.7827
Cache Profiling with Callgrind

KCachegrind

```
main

250 007 loops-fast.c
2 843 (unknown)
2 start.s
2 self.init.c
2 crtb.s

main

# L2m
Source ('/home/mliebe/loops/loops-fast.c')

23
24 1 for(i=0;i<N;i++)
25 { for(j=0;j<N;j++)
27 { a[i][j]=0.01;
29 } }
30 }

32 /* this is just to prevent the compiler */
33 from optimizing the upper loop away */
34 3 printf("sum = %.10f
",array_sum(a));

35 ....
36 .... 1 call to 'array_sum' (loops-fast: loops-fast.c)
37 .... 261 1 call to 'printf' (libc.so.6)
38 .... 69 1 call to '_dl_runtime_resolve' (ld-2.3.3.so)
```

Caller Map

main

array_sum

250 007

125 001
KCachegrind
Tool Chains

- cachegrind
  - cg_annotate
    - Text Output
- callgrind
  - callgrind_annotate
    - Text Output
  - KCachegrind
    - GUI
Exercises

- cd ~/callgrind
- make
Exercise 1 - loops-fast

```c
#define N 1000

int main(int argc, char** argv)
{
    double a[N][N];
    int i,j;
    for(i=0;i<N;i++)
    {
        for(j=0;j<N;j++)
        {
            a[i][j]=0.01;
        }
    }

    printf("sum = %10.3f\n",array_sum(a));
    return 0;
}

double array_sum(double a[N][N])
{
    int i,j;
    double s;
    s=0;
    for(i=0;i<N;i++)
    {
        for(j=0;j<N;j++)
        {
            s += a[i][j];
        }
    }
    return s;
}
```

**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

```c
#define N 1000
int main(int argc, char** argv)
{
    double a[N][N];
    int i, j;
    for(j=0;j<N;j++)
    {
        for(i=0;i<N;i++)
        {
            a[i][j]=0.01;
        }
    }
    printf("sum = %10.3f\n",array_sum(a));
    return 0;
}

double array_sum(double a[N][N])
{
    int i,j;
    double s;
    s=0;
    for(i=0;i<N;i++)
    {
        for(j=0;j<N;j++)
        {
            s += a[i][j];
        }
    }
    return s;
}
```

- 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

```c
#define N 1000
int main(int argc, char** argv)
{
    double a[N][N];
    int i,j;
    for(j=0;j<N;j++)
    {
        for(i=0;i<N;i++)
        {
            a[i][j]=0.01;
        }
    }
    printf("sum = %10.3f\n",array_sum(a));
    return 0;
}
double array_sum(double a[N][N])
{
    int i,j;
    double s;
    s=0;
    for(i=0;i<N;i++)
    {
        for(j=0;j<N;j++)
        {
            s += a[i][j];
        }
    }
    return s;
}
```

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: \texttt{cbrt} and \texttt{exp} are costly!

Calls to \texttt{cbrt} and \texttt{exp} eat up most of the instructions.
Math function in KCachegrind: optimized version – no exp!

Instructions dropped down from 11623 to 7199
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:

- `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)