Valgrind Part 4: Cachegrind and Callgrind - accu.org

Overload Journal #111 - October 2012 + Programming Topics Author: Paul Floyd

When your application is slow, you need a profiler. Paul Floyd shows us how callgrind and cachegrind can help.

The good news is that you've read my previous two articles on Valgrind's memcheck tool, and now your application has no memory faults or leaks. The bad news is that your application is too slow and your customers are complaining. What can you do? There are plenty of options, like get faster hardware, better architecture/design of the application, parallelization and 'code optimization'. Before you do anything like that, you should profile your application. You can use the cachegrind component of Valgrind to do this.

Let's take a step back now for a very brief overview of profiling techniques. Profiling comes in several different flavours. It can be intrusive (as in you need to modify your code) or unintrusive (no modification needed). Intrusive profiling is usually the most efficient, if you know where to profile. The catch is that you usually don't know where to look. Profiling can use instrumented code, either instrumented at compile or link time or on the fly (which is the case for cachegrind and callgrind). Lastly, the profiler can be hardware based, for instance using the performance counters that are a part of Intel CPUs. as used by Intel Vtune. In addition to performing time based profiling, you can also perform memory profiling (for instance, using Valgrind's massif), I/O profiling and so on.

Profiling should give a clear picture as to whether there are any significant bottlenecks in your code. If you see that one function is taking up 60% of the time, then it's a prime candidate for optimization. On the other hand, if you see that no function uses more than a few percent of the time, then you're probably going to need to look at higher level approaches than code optimization (or else you will have to optimize a lot of code to make a big difference).

In general, profiling tools will tell you the time spent in each function, inclusive and exclusive of time spent in calls to other functions. They may also tell you the time spent on each line of code. Callgrind and Cachegrind generate output that has a lot in common, like 'time' spent per function and line. The main differences are that Callgrind has more information about the callstack whilst cachegrind gives more information about cache hit rates. When you're using these tools, you're likely to want to use the GUI that is available to browse the results, KCachegrind. This is not part of Valgrind; as the name implies, it is part of KDE.

Cachegrind

Let's start with a small example (see Listing 1). It's deliberately bad, and as already noted above, a classic case for using a better algorithm.

```
#include <iostream>
#include <cmath>

bool isPrime(int x)
{
  int limit = std::sqrt(x);
  for (int i = 2; i <= limit; ++i)
  {
    if (x % i == 0)
    {
      return false;
    }
  }
  return true;
}

int main()
{
  int primeCount = 0;
  for (int i = 0; i < 1000000; ++i)
  {
    if (isPrime(i))
    {
      ++primeCount;
    }
  }
}
```

Listing 1

Compiling and running this is easy, just use Valgrind with the `--tool=cachegrind` option.

```
$ g++ -g badprime.cpp -o badprime
$ valgrind --tool=cachegrind --log-file=cg.out
   ./badprime
```

I measured the time it took to run (in VirtualBox on a 2.8GHz Opteron that's about 5 years old). Without cachegrind it took 1.3s. Under cachegrind that rose to 12.6s.

So what do we have. Well, I told Valgrind to send the output to cg.out. Let's take a look at that (Figure 1).

```
==1842== Cachegrind, a cache and branch-prediction profiler
==1842== Copyright (C) 2002-2011, and GNU GPL'd, by Nicholas Nethercote et al.
==1842== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==1842== Command: ./badprime
==1842== Parent PID: 1758
==1842==
==1842==
==1842== I   refs:      922,466,825
==1842== I1  misses:          1,234
==1842== LLi misses:          1,183
==1842== I1  miss rate:        0.00%
==1842== LLi miss rate:        0.00%
==1842==
==1842== D   refs:      359,460,248  (349,345,693 rd   + 10,114,555 wr)
==1842== D1  misses:          9,112  (      7,557 rd   +      1,555 wr)
==1842== LLd misses:          6,316  (      5,119 rd   +      1,197 wr)
==1842== D1  miss rate:         0.0% (        0.0%     +        0.0%  )
==1842== LLd miss rate:         0.0% (        0.0%     +        0.0%  )
==1842==
==1842== LL refs:            10,346  (      8,791 rd   +      1,555 wr)
==1842== LL misses:           7,499  (      6,302 rd   +      1,197 wr)
```

```
==1842== LL miss rate:           0.0% (         0.0%     +         0.0%  )
```

Figure 1

Some of this looks a bit familiar. There’s the usual copyright header and the column of `== <pid> ==` on the left. The part that is specific to cachegrind is the summary of overall performance, counting instruction reads (I refs), 1st level instruction cache misses (I1 misses), last level cache instruction misses (Lli), data reads (D refs), 1st level data cache misses (D1 misses), last level cache data misses (Lld misses) and finally a summary of the last level cache accesses. The information for the data activity is split into read and write parts (no writes for instructions, thankfully!). Why ‘last level’ and not ‘second level’? Valgrind uses a virtual machine, VEX, to execute the application that is being profiled. This is an abstraction of a real CPU, and it uses 2 levels of cache. If your physical CPU has 3 levels of cache, cachegrind will simulate the third level of cache with its second level.

There’s a further option that you can add to get a bit more information, `--branch-sim=yes`. If I add that option, then (in addition to adding another second to the run time) there are a couple more lines in the output (Figure 2).

```
==2345==
==2345== Branches:      139,777,324  (139,773,671 cond +      3,653 ind)
==2345== Mispredicts:     1,072,599  (  1,072,250 cond +        349 ind)
==2345== Mispred rate:          0.7% (        0.7%     +        9.5%   )
```

Figure 2

That’s fairly straightforward, the number of branches, branches mispredicted (conditional and indirect branches). Examples of C or C++ code that produces indirect branch machine code are calls through pointers to functions and virtual function calls. Conditional branches are generated for `if` statements and the conditional ternary operator.

So far, nothing to get too excited about. In the directory where I ran cachegrind there is now a file cachegrind.out.2345 (where 2345 was the PID when it was executing, as you can see in the branch prediction snippet above). You can control the name of the file that cachegrind generates by using the `–cachegrind-out-file` option. Here’s a small extract. It’s not meant for human consumption (Figure 3).

```
desc: I1 cache:         65536 B, 64 B, 2-way associative
desc: D1 cache:         65536 B, 64 B, 2-way associative
desc: LL cache:         1048576 B, 64 B, 16-way associative
cmd: ./badprime
events: Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw Bc Bcm Bi Bim
fl=/build/buildd/eglibc-2.15/csu/../sysdeps/generic/dl-hash.h
fn=_init
44 1 0 0 0 0 0 0 0 0 0 0 0 0
45 1 0 0 0 0 0 0 0 0 0 0 0 0
46 13 0 0 4 1 0 0 0 0 4 3 0 0
49 16 0 0 0 0 0 0 0 0 0 0 0 0
50 8 0 0 0 0 0 0 0 0 0 0 0 0
```

```
63 8 0 0 0 0 0 0 0 0 0 0 0 0
68 1 0 0 0 0 0 0 0 0 0 0 0 0
```

Figure 3

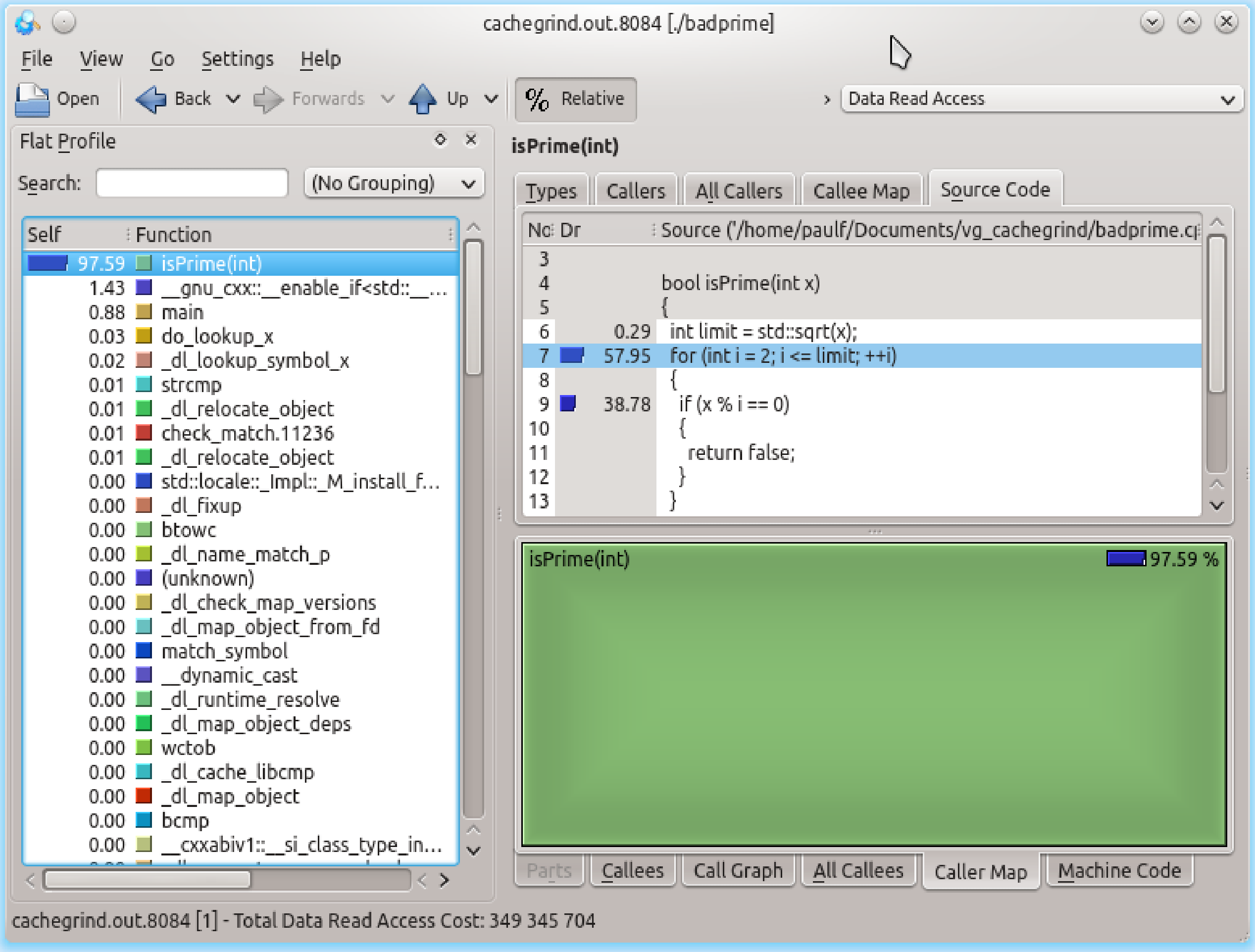
You can make this slightly more digestible by using `cg_annotate`. You just need to enter `cg_annotate cachegrind.out.<pid>` and the output will go to the terminal. Figure 4 is what I got for this example.

```
--------------------------------------------------------------------------------
I1 cache:         65536 B, 64 B, 2-way associative
D1 cache:         65536 B, 64 B, 2-way associative
LL cache:         1048576 B, 64 B, 16-way associative
Command:          ./badprime
Data file:        cachegrind.out.2345
Events recorded:  Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw Bc Bcm Bi Bim
Events shown:     Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw Bc Bcm Bi Bim
Event sort order: Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw Bc Bcm Bi Bim
Thresholds:       0.1 100 100 100 100 100 100 100 100 100 100 100 100
Include dirs:
User annotated:
Auto-annotation:  off

--------------------------------------------------------------------------------
         Ir  I1mr  ILmr          Dr  D1mr  DLmr         Dw  D1mw  DLmw          Bc
--------------------------------------------------------------------------
922,466,826 1,234 1,183 349,345,693 7,557 5,119 10,114,555 1,555 1,197 139,773,671 1

--------------------------------------------------------------------------------
         Ir I1mr ILmr          Dr  D1mr  DLmr        Dw  D1mw DLmw          Bc     B
--------------------------------------------------------------------------------
895,017,739    1    0 340,937,515     0     0 5,000,000     0    0 135,559,306 968,5
 16,000,000    0    0   5,000,000     0     0 4,000,000     0    0   2,000,000
 10,078,513    2    1   3,078,503     0     0 1,000,003     0    0   2,000,001  94,0
```

Figure 4

Still it isn't very easy to read. You can see the same sort of notation as we saw for the overall summary (I – instruction, D – data and B – branch). I've cut the full paths to the file:function part. You can filter and sort the output with the `–threshold` and `–sort` options. You can also generate annotated source either with `-auto=yes` or on a file by file basis by passing the (fully qualified) filename as an argument to `cg_annotate`. I won't show an example here as it is rather long. Basically it shows the same information as in the `cg_annotate` output, but on a line by line basis. Lastly for `cg_annotate`, if you are using it for code coverage metrics, then you can combine multiple runs using `cg_merge`.

As already mentioned, you can use Kcachegrind. This is a fairly standard GUI application for browsing the performance results. It isn't part of Valgrind, rather it is a component of KDE so you may need to install it separately. If you want to be able to see the funky graphics, you'll need to have GraphViz installed. Figure 5 is a screen shot showing, on the left, percent of total time per function in descending order. The top 3, as expected, are the awful `isPrime`, `sqrt` and `main` with 99.9% between them. On the top right I've selected the Source Code tab, and we can see three lines with percent of time next to them. In this small example, the other tabs aren't very interesting.


cachegrind.out.8084 [./badprime]
File View Go Settings Help
Open Back Forwards Up % Relative
Data Read Access
Flat Profile
Search: (No Grouping)
isPrime(int)
Types Callers All Callers Callee Map Source Code
isPrime(int) 97.59 %
Parts Callees Call Graph All Callees Caller Map Machine Code

Self	Function
97.59	isPrime(int)
1.43	__gnu_cxx::__enable_if<std::__...
0.88	main
0.03	do_lookup_x
0.02	_dl_lookup_symbol_x
0.01	strcmp
0.01	_dl_relocate_object
0.01	check_match.11236
0.01	_dl_relocate_object
0.00	std::locale::_Impl::_M_install_f...
0.00	_dl_fixup
0.00	btowc
0.00	_dl_name_match_p
0.00	(unknown)
0.00	_dl_check_map_versions
0.00	_dl_map_object_from_fd
0.00	match_symbol
0.00	__dynamic_cast
0.00	_dl_runtime_resolve
0.00	_dl_map_object_deps
0.00	wctob
0.00	_dl_cache_libcmp
0.00	_dl_map_object
0.00	bcmp
0.00	__cxxabiv1::__si_class_type_in...

```
No  Dr     Source ('/home/paulf/Documents/vg_cachegrind/badprime.c
3
4          bool isPrime(int x)
5          {
6    0.29    int limit = std::sqrt(x);
7   57.95    for (int i = 2; i <= limit; ++i)
8            {
9   38.78      if (x % i == 0)
10             {
11               return false;
12             }
13           }
```

cachegrind.out.8084 [1] - Total Data Read Access Cost: 349 345 704

Figure 5

Most of the navigating can be done by clicking the list of functions in the left pane. You can search for functions and also filter by the following groups – none, ELF object, source file, C++ class and function cycle. The two right panes have similar displays, roughly for displaying information about callers and callees (that is, functions called by the current function). Clicking on the **% Relative** toggle switches between showing absolute units (e.g., instruction fetches) and the percentage of the total. The drop down box on the top right allows you to display cache hit/miss rates, instruction fetches and cycle estimates.

Callgrind

For an example using callgrind, I downloaded and built a debug version of Icarus Verilog [Icarus] and compiled and simulated a Fibonacci calculator [Fibonacci] with small modifications to make the simulation last longer and exit on completion).

The commands that I used were:

```
iverilog -o fib fib*.v
```

for the compilation, and

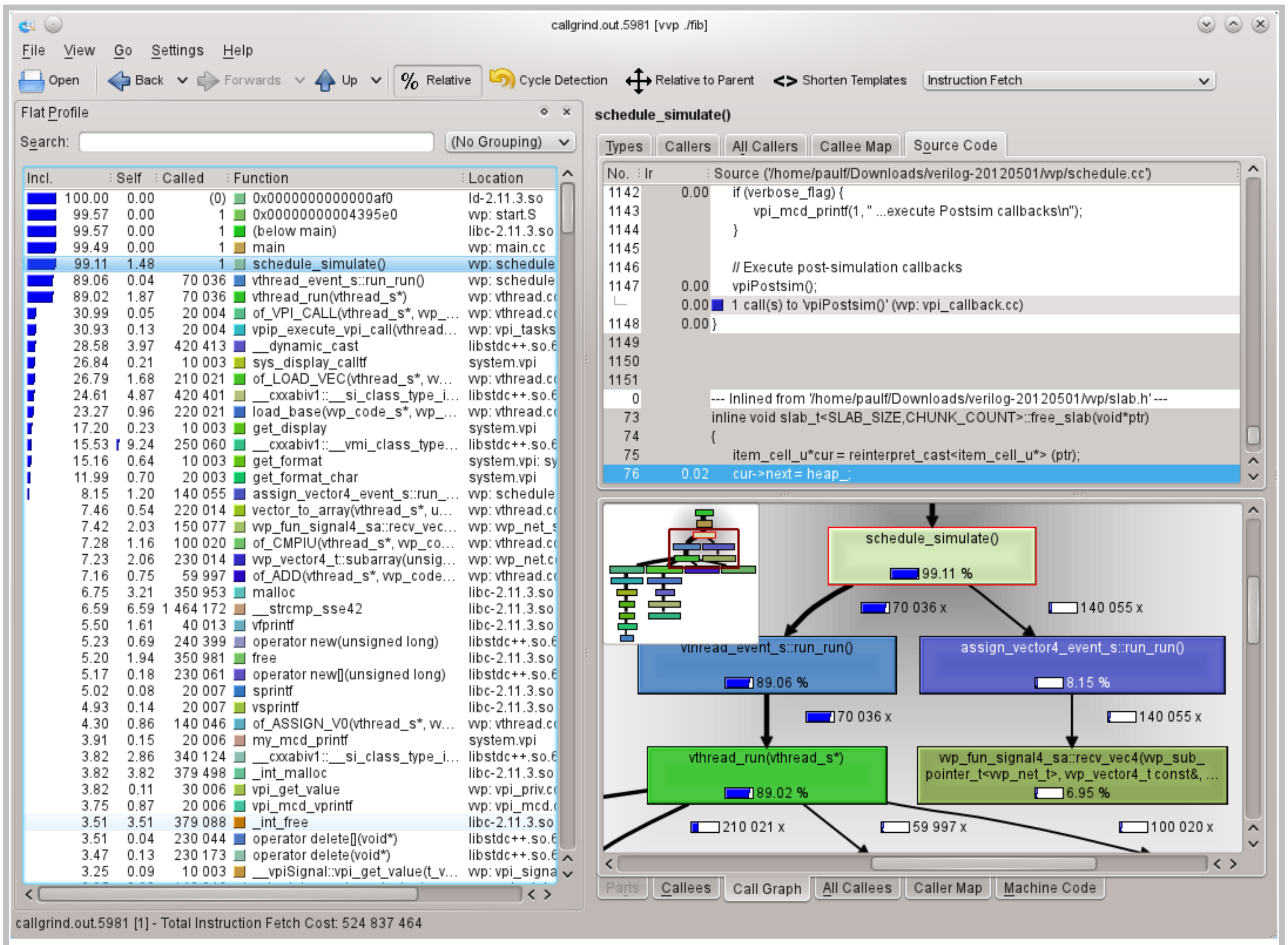
```
valgrind --tool=callgrind
--log-file=callgrind.log vvp ./fib
```

to profile the simulation.

Finally

```
kcachegrind callgrind.out.5981
```

to view the results (see Figure 6).



Incl.	Self	Called	Function	Location
100.00	0.00	(0)	0x0000000000000af0	ld-2.11.3.so
99.57	0.00	1	0x00000000004395e0	vvp: start.S
99.57	0.00	1	(below main)	libc-2.11.3.so
99.49	0.00	1	main	vvp: main.cc
99.11	1.48	1	schedule_simulate()	vvp: schedule
89.06	0.04	70 036	vthread_event_s::run_run()	vvp: schedule
89.02	1.87	70 036	vthread_run(vthread_s*)	vvp: vthread.c
30.99	0.05	20 004	of_VPI_CALL(vthread_s*, vvp_…	vvp: vthread.c
30.93	0.13	20 004	vpip_execute_vpi_call(vthread…	vvp: vpi_tasks
28.58	3.97	420 413	__dynamic_cast	libstdc++.so.6
26.84	0.21	10 003	sys_display_calltf	system.vpi
26.79	1.68	210 021	of_LOAD_VEC(vthread_s*, vv…	vvp: vthread.c
24.61	4.87	420 401	__cxxabiv1::__si_class_type_i…	libstdc++.so.6
23.27	0.96	220 021	load_base(vvp_code_s*, vvp_…	vvp: vthread.c
17.20	0.23	10 003	get_display	system.vpi
15.53	9.24	250 060	__cxxabiv1::__vmi_class_type…	libstdc++.so.6
15.16	0.64	10 003	get_format	system.vpi: sy
11.99	0.70	20 003	get_format_char	system.vpi
8.15	1.20	140 055	assign_vector4_event_s::run_…	vvp: schedule
7.46	0.54	220 014	vector_to_array(vthread_s*, u…	vvp: vthread.c
7.42	2.03	150 077	vvp_fun_signal4_sa::recv_vec…	vvp: vvp_net_s
7.28	1.16	100 020	of_CMPIU(vthread_s*, vvp_co…	vvp: vthread.c
7.23	2.06	230 014	vvp_vector4_t::subarray(unsig…	vvp: vvp_net.c
7.16	0.75	59 997	of_ADD(vthread_s*, vvp_code…	vvp: vthread.c
6.75	3.21	350 953	malloc	libc-2.11.3.so
6.59	6.59	1 464 172	__strcmp_sse42	libc-2.11.3.so
5.50	1.61	40 013	vfprintf	libc-2.11.3.so
5.23	0.69	240 399	operator new(unsigned long)	libstdc++.so.6
5.20	1.94	350 981	free	libc-2.11.3.so
5.17	0.18	230 061	operator new[](unsigned long)	libstdc++.so.6
5.02	0.08	20 007	sprintf	libc-2.11.3.so
4.93	0.14	20 007	vsprintf	libc-2.11.3.so
4.30	0.86	140 046	of_ASSIGN_V0(vthread_s*, vv…	vvp: vthread.c
3.91	0.15	20 006	my_mcd_printf	system.vpi
3.82	2.86	340 124	__cxxabiv1::__si_class_type_i…	libstdc++.so.6
3.82	3.82	379 498	_int_malloc	libc-2.11.3.so
3.82	0.11	30 006	vpi_get_value	vvp: vpi_priv.c
3.75	0.87	20 006	vpi_mcd_vprintf	vvp: vpi_mcd.c
3.51	3.51	379 088	_int_free	libc-2.11.3.so
3.51	0.04	230 044	operator delete[](void*)	libstdc++.so.6
3.47	0.13	230 173	operator delete(void*)	libstdc++.so.6
3.25	0.09	10 003	__vpiSignal::vpi_get_value(t_v…	vvp: vpi_signa

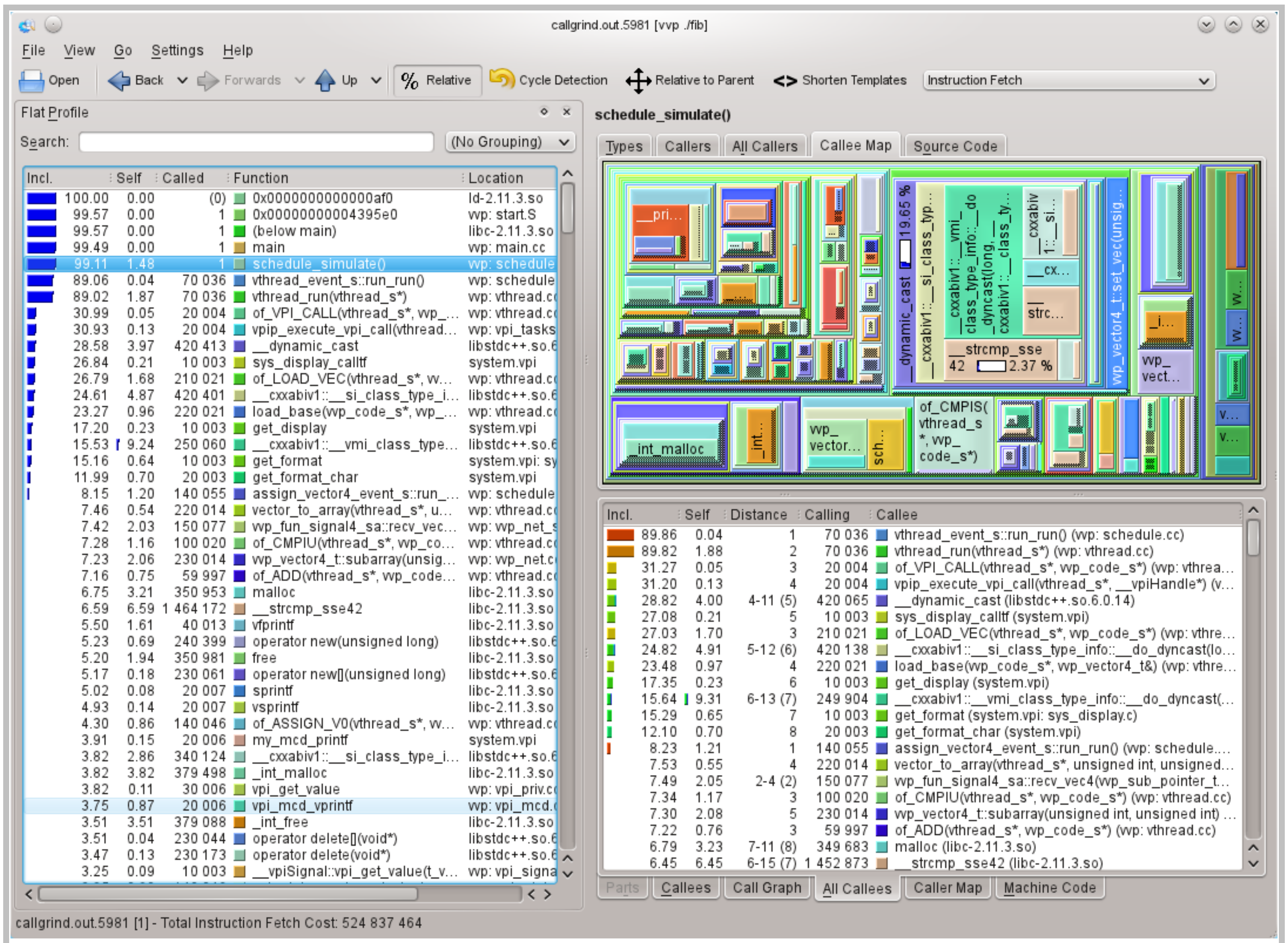
```
No.  Ir    Source ('/home/paulf/Downloads/verilog-20120501/vvp/schedule.cc')
1142 0.00      if (verbose_flag) {
1143                vpi_mcd_printf(1, " ...execute Postsim callbacks\n");
1144           }
1145
1146           // Execute post-simulation callbacks
1147 0.00      vpiPostsim();
     0.00      1 call(s) to 'vpiPostsim()' (vvp: vpi_callback.cc)
1148 0.00  }
1149
1150
1151
   0       --- Inlined from '/home/paulf/Downloads/verilog-20120501/vvp/slab.h' ---
  73       inline void slab_t<SLAB_SIZE,CHUNK_COUNT>::free_slab(void*ptr)
  74       {
  75           item_cell_u*cur = reinterpret_cast<item_cell_u*> (ptr);
  76 0.02      cur->next = heap_;
```

callgrind.out.5981 [1] - Total Instruction Fetch Cost: 524 837 464

Figure 6

This time, there isn't all of the cache hit/miss rate information, but instead there is far more on the interrelations between functions and time. You can navigate through the functions by double clicking in the call graph.

In the last snapshot, I show the funky callee map (which I expect will look better in the PDF version of the magazine in colour than in the black and white print magazine). The areas of the rectangles is proportional to the time spent in the function. Very pretty, but when it's as cluttered as this example, it's not much use. Both the map and the list are dynamic, and you can click in one and the area or list item will be highlighted in the other. You can also double click to jump to a different function. When you are navigating like this, you can use the back/forwards buttons on the toolbar to navigate in a similar fashion to a web browser, or navigate up the call stack (see Figure 7).


callgrind.out.5981 [vvp ./fib]
File View Go Settings Help
Open Back Forwards Up Relative Cycle Detection Relative to Parent Shorten Templates Instruction Fetch
Flat Profile
Search:
(No Grouping)
schedule_simulate()
Types Callers All Callers Callee Map Source Code
Parts Callees Call Graph All Callees Caller Map Machine Code

```
Incl.   Self   Called   Function                              Location
100.00  0.00       (0)  0x0000000000000af0                    ld-2.11.3.so
 99.57  0.00         1  0x00000000004395e0                    vvp: start.S
 99.57  0.00         1  (below main)                          libc-2.11.3.so
 99.49  0.00         1  main                                  vvp: main.cc
 99.11  1.48         1  schedule_simulate()                   vvp: schedule
 89.06  0.04    70 036  vthread_event_s::run_run()            vvp: schedule
 89.02  1.87    70 036  vthread_run(vthread_s*)               vvp: vthread.c
 30.99  0.05    20 004  of_VPI_CALL(vthread_s*, vvp_…         vvp: vthread.c
 30.93  0.13    20 004  vpip_execute_vpi_call(vthread…        vvp: vpi_tasks
 28.58  3.97   420 413  __dynamic_cast                        libstdc++.so.6
 26.84  0.21    10 003  sys_display_calltf                    system.vpi
 26.79  1.68   210 021  of_LOAD_VEC(vthread_s*, vv…           vvp: vthread.c
 24.61  4.87   420 401  __cxxabiv1::__si_class_type_i…        libstdc++.so.6
 23.27  0.96   220 021  load_base(vvp_code_s*, vvp_…          vvp: vthread.c
 17.20  0.23    10 003  get_display                           system.vpi
 15.53  9.24   250 060  __cxxabiv1::__vmi_class_type…         libstdc++.so.6
 15.16  0.64    10 003  get_format                            system.vpi: sy
 11.99  0.70    20 003  get_format_char                       system.vpi
  8.15  1.20   140 055  assign_vector4_event_s::run_…         vvp: schedule
  7.46  0.54   220 014  vector_to_array(vthread_s*, u…        vvp: vthread.c
  7.42  2.03   150 077  vvp_fun_signal4_sa::recv_vec…         vvp: vvp_net_s
  7.28  1.16   100 020  of_CMPIU(vthread_s*, vvp_co…          vvp: vthread.c
  7.23  2.06   230 014  vvp_vector4_t::subarray(unsig…        vvp: vvp_net.c
  7.16  0.75    59 997  of_ADD(vthread_s*, vvp_code…          vvp: vthread.c
  6.75  3.21   350 953  malloc                                libc-2.11.3.so
  6.59  6.59 1 464 172  __strcmp_sse42                        libc-2.11.3.so
  5.50  1.61    40 013  vfprintf                              libc-2.11.3.so
  5.23  0.69   240 399  operator new(unsigned long)           libstdc++.so.6
  5.20  1.94   350 981  free                                  libc-2.11.3.so
  5.17  0.18   230 061  operator new[](unsigned long)         libstdc++.so.6
  5.02  0.08    20 007  sprintf                               libc-2.11.3.so
  4.93  0.14    20 007  vsprintf                              libc-2.11.3.so
  4.30  0.86   140 046  of_ASSIGN_V0(vthread_s*, vv…          vvp: vthread.c
  3.91  0.15    20 006  my_mcd_printf                         system.vpi
  3.82  2.86   340 124  __cxxabiv1::__si_class_type_i…        libstdc++.so.6
  3.82  3.82   379 498  _int_malloc                           libc-2.11.3.so
  3.82  0.11    30 006  vpi_get_value                         vvp: vpi_priv.c
  3.75  0.87    20 006  vpi_mcd_vprintf                       vvp: vpi_mcd.
  3.51  3.51   379 088  _int_free                             libc-2.11.3.so
  3.51  0.04   230 044  operator delete[](void*)              libstdc++.so.6
  3.47  0.13   230 173  operator delete(void*)                libstdc++.so.6
  3.25  0.09    10 003  __vpiSignal::vpi_get_value(t_v…       vvp: vpi_signa
```

```
Incl.  Self  Distance   Calling  Callee
89.86  0.04         1    70 036  vthread_event_s::run_run() (vvp: schedule.cc)
89.82  1.88         2    70 036  vthread_run(vthread_s*) (vvp: vthread.cc)
31.27  0.05         3    20 004  of_VPI_CALL(vthread_s*, vvp_code_s*) (vvp: vthrea…
31.20  0.13         4    20 004  vpip_execute_vpi_call(vthread_s*, __vpiHandle*) (v…
28.82  4.00  4-11 (5)   420 065  __dynamic_cast (libstdc++.so.6.0.14)
27.08  0.21         5    10 003  sys_display_calltf (system.vpi)
27.03  1.70         3   210 021  of_LOAD_VEC(vthread_s*, vvp_code_s*) (vvp: vthre…
24.82  4.91  5-12 (6)   420 138  __cxxabiv1::__si_class_type_info::__do_dyncast(lo…
23.48  0.97         4   220 021  load_base(vvp_code_s*, vvp_vector4_t&) (vvp: vthre…
17.35  0.23         6    10 003  get_display (system.vpi)
15.64  9.31  6-13 (7)   249 904  __cxxabiv1::__vmi_class_type_info::__do_dyncast(…
15.29  0.65         7    10 003  get_format (system.vpi: sys_display.c)
12.10  0.70         8    20 003  get_format_char (system.vpi)
 8.23  1.21         1   140 055  assign_vector4_event_s::run_run() (vvp: schedule.…
 7.53  0.55         4   220 014  vector_to_array(vthread_s*, unsigned int, unsigned…
 7.49  2.05  2-4 (2)    150 077  vvp_fun_signal4_sa::recv_vec4(vvp_sub_pointer_t…
 7.34  1.17         3   100 020  of_CMPIU(vthread_s*, vvp_code_s*) (vvp: vthread.cc)
 7.30  2.08         5   230 014  vvp_vector4_t::subarray(unsigned int, unsigned int) …
 7.22  0.76         3    59 997  of_ADD(vthread_s*, vvp_code_s*) (vvp: vthread.cc)
 6.79  3.23  7-11 (8)   349 683  malloc (libc-2.11.3.so)
 6.45  6.45  6-15 (7) 1 452 873  __strcmp_sse42 (libc-2.11.3.so)
```

callgrind.out.5981 [1] - Total Instruction Fetch Cost: 524 837 464

Figure 7

Callgrind has `callgrind_annotate` that is a text processor for its output files. This is the callgrind equivalent of `cg_annotate`, and since the output is very similar, I won't show it here.

Practical difficulties

One problem that I've experienced with callgrind was with differences between the floating point results with and without callgrind. In this case, I encountered numerical instability with callgrind that I didn't get when running the application natively. This is worse for 32bit x86, where by default code gets generated to use the venerable x87 floating point coprocessor. This uses 10byte (80bit) precision internally. Usually this means that if you have an expression that involves several operations on doubles, the operations will be done at 80bit precision and the result converted back to double precision (8byte, 64bit, required by the C and C++ standards). Valgrind performs the calculations at 64bit precision, so some of the intermediate precision is lost. Even with x64, which by default uses 64bit MMX/SSE for floating point calculations and so shouldn't have any internal truncation, I've still seen examples where the results are slightly different.

Another problem with big real-world programs is speed. Like all of the Valgrind tools, there is a big overhead in performing the measurement. You can mitigate this by controlling when callgrind performs data collection. The controlling can be done using command line options, using `callgrind_control --instr=<on|off>` or lastly by attaching gdb and using the monitor commands `instrumentation on|off`. You can statically control callgrind using Valgrind macros defined in callgrind.h (like those for memcheck that I described in my previous article on 'Advanced

Memcheck' in *Overload 110*). You will probably need to do some profiling of the entire application to get an idea of where you want to concentrate your efforts.

In part 5 of this series, I'll be covering Massif, Valgrind's heap memory profiler.

References

[Fibonacci] http://www2.engr.arizona.edu/~slysecky/resources/verilog_tutorial.html (http://www2.engr.arizona.edu/~slysecky/resources/verilog_tutorial.html)

[Icarus] http://iverilog.icarus.com (http://iverilog.icarus.com)