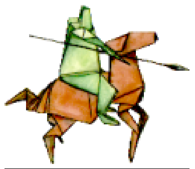

5. Cachegrind: a cache and branch-prediction profiler

Table of Contents

To use this tool, you must specify `--tool=cachegrind` on the Valgrind command line.

5.1. Overview

Cachegrind simulates how your program interacts with a machine's cache hierarchy and (optionally) branch predictor. It simulates a machine with independent first-level instruction and data caches (I1 and D1), backed by a unified second-level cache (L2). This exactly matches the configuration of many modern machines.

However, some modern machines have three or four levels of cache. For these machines (in the cases where Cachegrind can auto-detect the cache configuration) Cachegrind simulates the first-level and last-level caches. The reason for this choice is that the last-level cache has the most influence on runtime, as it masks accesses to main memory. Furthermore, the L1 caches often have low associativity, so simulating them can detect cases where the code interacts badly with this cache (eg. traversing a matrix column-wise with the row length being a power of 2).

Therefore, Cachegrind always refers to the I1, D1 and LL (last-level) caches.

Cachegrind gathers the following statistics (abbreviations used for each statistic is given in parentheses):

- I cache reads (`Ir`, which equals the number of instructions executed), I1 cache read misses (`I1mr`) and LL cache instruction read misses (`ILmr`).
- D cache reads (`Dr`, which equals the number of memory reads), D1 cache read misses (`D1mr`), and LL cache data read misses (`DLmr`).
- D cache writes (`Dw`, which equals the number of memory writes), D1 cache write misses (`D1mw`), and LL cache data write misses (`DLmw`).
- Conditional branches executed (`Bc`) and conditional branches mispredicted (`Bcm`).
- Indirect branches executed (`Bi`) and indirect branches mispredicted (`Bim`).

Note that D1 total accesses is given by `D1mr` + `D1mw`, and that LL total accesses is given by `ILmr` + `DLmr` + `DLmw`.

These statistics are presented for the entire program and for each function in the program. You can also annotate each line of source code in the program with the counts that were caused directly by it.

On a modern machine, an L1 miss will typically cost around 10 cycles, an LL miss can cost as much as 200 cycles, and a mispredicted branch costs in the region of 10 to 30 cycles. Detailed cache and branch profiling can be very useful for understanding how your program interacts with the machine and thus how to make it faster.

Also, since one instruction cache read is performed per instruction executed, you can find out how many instructions are executed per line, which can be useful for traditional profiling.

5.2. Using Cachegrind, cg_annotate and cg_merge

First off, as for normal Valgrind use, you probably want to compile with debugging info (the -[illegible]n). But by contrast with normal Valgrind use, you probably do want to turn optimisation on, since you should profile your program as it will be normally run.

Then, you need to run Cachegrind itself to gather the profiling information, and then run cg_annotate to get a detailed presentation of that information. As an optional intermediate step, you can use cg_merge to sum together the outputs of multiple Cachegrind runs into a single file which you then use as the input for cg_annotate. Alternatively, you can use cg_diff to difference the outputs of two Cachegrind runs into a single file which you then use as the input for cg_annotate.

5.2.1. Running Cachegrind

To run Cachegrind on a program `prog`, run:

```
valgrind --tool=cachegrind prog
```

The program will execute (slowly). Upon completion, summary statistics that look like this will be printed:

```
==31751== I   refs:      27,742,716
==31751== I1  misses:           276
==31751== LLi misses:           275
==31751== I1  miss rate:        0.0%
==31751== LLi miss rate:        0.0%
==31751==
==31751== D   refs:      15,430,290  (10,955,517 rd + 4,474,773 wr)
==31751== D1  misses:        41,185  (    21,905 rd +    19,280 wr)
==31751== LLd misses:        23,085  (     3,987 rd +    19,098 wr)
==31751== D1  miss rate:        0.2% (       0.1%   +       0.4%)
==31751== LLd miss rate:        0.1% (       0.0%   +       0.4%)
==31751==
==31751== LL misses:         23,360  (     4,262 rd +    19,098 wr)
==31751== LL miss rate:         0.0% (       0.0%   +       0.4%)
```

Cache accesses for instruction fetches are summarised first, giving the number of fetches made (this is the number of instructions executed, which can be useful to know in its own right), the number of I1 misses, and the number of LL instruction (`LLi`) misses.

Cache accesses for data follow. The information is similar to that of the instruction fetches, except that the values are also shown split between reads and writes (note each row's `rd` and `wr` values add up to the row's total).

Combined instruction and data figures for the LL cache follow that. Note that the LL miss rate is computed relative to the total number of memory accesses, not the number of L1 misses. I.e. it is `(ILmr + DLmr + DLmw) / (Ir + Dr + Dw)` not `(ILmr + DLmr + DLmw) / (I1mr + D1mr + D1mw)`

Branch prediction statistics are not collected by default. To do so, add the option `--branch-sim=yes`.

5.2.2. Output File

As well as printing summary information, Cachegrind also writes more detailed profiling information to a file. By default this file is named `cachegrind.out.<pid>` (where `<pid>` is the program's process ID), but its name can be changed with the `--cachegrind-out-file` option. This file is human-readable, but is intended to be interpreted by the accompanying program cg_annotate, described in the next section.

The default `.<pid>` suffix on the output file name serves two purposes. Firstly, it means you don't have to rename old log files that you don't want to overwrite. Secondly, and more importantly, it allows correct profiling with the `--trace-children=yes` option of programs that spawn child processes.

The output file can be big, many megabytes for large applications built with full debugging information.

5.2.3. Running cg_annotate

Before using cg_annotate, it is worth widening your window to be at least 120-characters wide if possible, as the output lines can be quite long.

To get a function-by-function summary, run:

```
cg_annotate <filename>
```

on a Cachegrind output file.

5.2.4. The Output Preamble

The first part of the output looks like this:

```
--------------------------------------------------------------------------------
I1 cache:              65536 B, 64 B, 2-way associative
D1 cache:              65536 B, 64 B, 2-way associative
LL cache:              262144 B, 64 B, 8-way associative
Command:               concord vg_to_ucode.c
```

```
Events recorded:       Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
Events shown:          Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
Event sort order:      Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
Threshold:             99%
Chosen for annotation:
Auto-annotation:       off
```

This is a summary of the annotation options:

- I1 cache, D1 cache, LL cache: cache configuration. So you know the configuration with which these results were obtained.
- Command: the command line invocation of the program under examination.
- Events recorded: which events were recorded.
- Events shown: the events shown, which is a subset of the events gathered. This can be adjusted with the `--show` option.
- Event sort order: the sort order in which functions are shown. For example, in this case the functions are sorted from highest `Ir` counts to lowest. If two functions have identical `Ir` counts, they will then be sorted by `I1mr` counts, and so on. This order can be adjusted with the `--sort` option.

  Note that this dictates the order the functions appear. It is *not* the order in which the columns appear; that is dictated by the "events shown" line (and can be changed with the `--show` option).
- Threshold: cg_annotate by default omits functions that cause very low counts to avoid drowning you in information. In this case, cg_annotate shows summaries the functions that account for 99% of the `Ir` counts; `Ir` is chosen as the threshold event since it is the primary sort event. The threshold can be adjusted with the `--threshold` option.
- Chosen for annotation: names of files specified manually for annotation; in this case none.
- Auto-annotation: whether auto-annotation was requested via the `--auto=yes` option. In this case no.

5.2.5. The Global and Function-level Counts

Then follows summary statistics for the whole program:

```
--------------------------------------------------------------------------------
Ir         I1mr ILmr Dr         D1mr   DLmr  Dw        D1mw   DLmw
--------------------------------------------------------------------------------
27,742,716  276  275 10,955,517 21,905 3,987 4,474,773 19,280 19,098  PROGRAM TOTALS
```

These are similar to the summary provided when Cachegrind finishes running.

Then comes function-by-function statistics:

```
--------------------------------------------------------------------------------
Ir        I1mr ILmr Dr        D1mr  DLmr  Dw        D1mw   DLmw    file:function
--------------------------------------------------------------------------------
8,821,482    5    5 2,242,702 1,621    73 1,794,230      0      0  getc.c:_IO_getc
5,222,023    4    4 2,276,334    16    12   875,959      1      1  concord.c:get_word
2,649,248    2    2 1,344,810 7,326 1,385         .      .      .  vg_main.c:strcmp
2,521,927    2    2   591,215     0     0   179,398      0      0  concord.c:hash
2,242,740    2    2 1,046,612   568    22   448,548      0      0  ctype.c:tolower
1,496,937    4    4   630,874 9,000 1,400   279,388      0      0  concord.c:insert
  897,991   51   51   897,831    95    30        62      1      1  ???:???
  598,068    1    1   299,034     0     0   149,517      0      0  ../sysdeps/generic/lockfile.c:__flockfile
  598,068    0    0   299,034     0     0   149,517      0      0  ../sysdeps/generic/lockfile.c:__funlockfile
  598,024    4    4   213,580    35    16   149,506      0      0  vg_clientmalloc.c:malloc
  446,587    1    1   215,973 2,167   430   129,948 14,057 13,957  concord.c:add_existing
  341,760    2    2   128,160     0     0   128,160      0      0  vg_clientmalloc.c:vg_trap_here_WRAPPER
  320,782    4    4   150,711   276     0    56,027     53     53  concord.c:init_hash_table
  298,998    1    1   106,785     0     0    64,071      1      1  concord.c:create
  149,518    0    0   149,516     0     0         1      0      0  ???:tolower@@GLIBC_2.0
  149,518    0    0   149,516     0     0         1      0      0  ???:fgetc@@GLIBC_2.0
   95,983    4    4    38,031     0     0    34,409  3,152  3,150  concord.c:new_word_node
   85,440    0    0    42,720     0     0    21,360      0      0  vg_clientmalloc.c:vg_bogus_epilogue
```

Each function is identified by a `file_name:function_name` pair. If a column contains only a dot it means the function never performs that event (e.g. the third row shows that `strcmp()` contains no instructions that write to memory). The name `???` is used if the file name and/or function name could not be determined from debugging information. If most of the entries have the form `???:???` the program probably wasn't compiled with `-g`.

It is worth noting that functions will come both from the profiled program (e.g. `concord.c`) and from libraries (e.g. `getc.c`)

5.2.6. Line-by-line Counts

By default, all source code annotation is also shown. (Filenames to be annotated can also be specified manually as arguments to cg_annotate, but this is rarely needed.) For example, the output from running `cg_annotate <filename>` for our example produces the same output as above followed by an annotated version of `concord.c`, a section of which looks like:

```
--------------------------------------------------------------------------------
-- Auto-annotated source: concord.c
```

```
--------------------------------------------------------------------------------
Ir        I1mr ILmr Dr      D1mr  DLmr  Dw      D1mw   DLmw

        .    .    .       .     .     .       .      .      .  void init_hash_table(char *file_name, Word_Node *table[])
        3    1    1       .     .     .       1      0      0  {
        .    .    .       .     .     .       .      .      .      FILE *file_ptr;
        .    .    .       .     .     .       .      .      .      Word_Info *data;
        1    0    0       .     .     .       1      1      1      int line = 1, i;
        .    .    .       .     .     .       .      .      .
        5    0    0       .     .     .       3      0      0      data = (Word_Info *) create(sizeof(Word_Info));
        .    .    .       .     .     .       .      .      .
    4,991    0    0   1,995     0     0     998      0      0      for (i = 0; i < TABLE_SIZE; i++)
    3,988    1    1   1,994     0     0     997     53     52          table[i] = NULL;
        .    .    .       .     .     .       .      .      .
        .    .    .       .     .     .       .      .      .      /* Open file, check it. */
        6    0    0       1     0     0       4      0      0      file_ptr = fopen(file_name, "r");
        2    0    0       1     0     0       .      .      .      if (!(file_ptr)) {
        .    .    .       .     .     .       .      .      .          fprintf(stderr, "Couldn't open '%s'.\n", file_name);
        1    1    1       .     .     .       .      .      .          exit(EXIT_FAILURE);
        .    .    .       .     .     .       .      .      .      }
        .    .    .       .     .     .       .      .      .
  165,062    1    1  73,360     0     0  91,700      0      0      while ((line = get_word(data, line, file_ptr)) != EOF)
  146,712    0    0  73,356     0     0  73,356      0      0          insert(data->;word, data->line, table);
        .    .    .       .     .     .       .      .      .
        4    0    0       1     0     0       2      0      0      free(data);
        4    0    0       1     0     0       2      0      0      fclose(file_ptr);
        3    0    0       2     0     0       .      .      .  }
```

(Although column widths are automatically minimised, a wide terminal is clearly useful.)

Each source file is clearly marked (`User-annotated source`) as having been chosen manually for annotation. If the file was found in one of the directories specified with the `-I`/`--include` option, the directory and file are both given.

Each line is annotated with its event counts. Events not applicable for a line are represented by a dot. This is useful for distinguishing between an event which cannot happen, and one which can but did not.

Sometimes only a small section of a source file is executed. To minimise uninteresting output, Cachegrind only shows annotated lines and lines within a small distance of annotated lines. Gaps are marked with the line numbers so you know which part of a file the shown code comes from, eg:

```
(figures and code for line 704)
-- line 704 ----------------------------------------
-- line 878 ----------------------------------------
(figures and code for line 878)
```

The amount of context to show around annotated lines is controlled by the `--context` option.

Automatic annotation is enabled by default. cg_annotate will automatically annotate every source file it can find that is mentioned in the function-by-function summary. Therefore, the files chosen for auto-annotation are affected by the `--sort` and `--threshold` options. Each source file is clearly marked (`Auto-annotated source`) as being chosen automatically. Any files that could not be found are mentioned at the end of the output, eg:

```
------------------------------------------------------------------
The following files chosen for auto-annotation could not be found:
------------------------------------------------------------------
  getc.c
  ctype.c
  ../sysdeps/generic/lockfile.c
```

This is quite common for library files, since libraries are usually compiled with debugging information, but the source files are often not present on a system. If a file is chosen for annotation both manually and automatically, it is marked as `User-annotated source`. Use the `-I`/`--include` option to tell Valgrind where to look for source files if the filenames found from the debugging information aren't specific enough.

Beware that cg_annotate can take some time to digest large `cachegrind.out.<pid>` files, e.g. 30 seconds or more. Also beware that auto-annotation can produce a lot of output if your program is large!

5.2.7. Annotating Assembly Code Programs

Valgrind can annotate assembly code programs too, or annotate the assembly code generated for your C program. Sometimes this is useful for understanding what is really happening when an interesting line of C code is translated into multiple instructions.

To do this, you just need to assemble your `.s` files with assembly-level debug information. You can use compile with the `-S` to compile C/C++ programs to assembly code, and then assemble the assembly code files with `-g` to achieve this. You can then profile and annotate the assembly code source files in the same way as C/C++ source files.

5.2.8. Forking Programs

If your program forks, the child will inherit all the profiling data that has been gathered for the parent.

If the output file format string (controlled by `--cachegrind-out-file`) does not contain `%p`, then the outputs from the parent and child will be intermingled in a single output file, which will almost certainly make it unreadable by cg_annotate.

5.2.9. cg_annotate Warnings

There are a couple of situations in which cg_annotate issues warnings.

- If a source file is more recent than the `cachegrind.out.<pid>` file. This is because the information in `cachegrind.out.<pid>` is only recorded with line numbers, so if the line numbers change at all in the source (e.g. lines added, deleted, swapped), any annotations will be incorrect.
- If information is recorded about line numbers past the end of a file. This can be caused by the above problem, i.e. shortening the source file while using an old `cachegrind.out.<pid>` file. If this happens, the figures for the bogus lines are printed anyway (clearly marked as bogus) in case they are important.

5.2.10. Unusual Annotation Cases

Some odd things that can occur during annotation:

- If annotating at the assembler level, you might see something like this:

```
      1    0    0  .    .    .  .    .    .          leal -12(%ebp),%eax
      1    0    0  .    .    .  1    0    0          movl %eax,84(%ebx)
      2    0    0  0    0    0  1    0    0          movl $1,-20(%ebp)
      .    .    .  .    .    .  .    .    .          .align 4,0x90
      1    0    0  .    .    .  .    .    .          movl $.LnrB,%eax
      1    0    0  .    .    .  1    0    0          movl %eax,-16(%ebp)
```

  How can the third instruction be executed twice when the others are executed only once? As it turns out, it isn't. Here's a dump of the executable, using `objdump -d`:

```
      8048f25:       8d 45 f4                lea    0xfffffff4(%ebp),%eax
      8048f28:       89 43 54                mov    %eax,0x54(%ebx)
      8048f2b:       c7 45 ec 01 00 00 00    movl   $0x1,0xffffffec(%ebp)
      8048f32:       89 f6                   mov    %esi,%esi
      8048f34:       b8 08 8b 07 08          mov    $0x8078b08,%eax
      8048f39:       89 45 f0                mov    %eax,0xfffffff0(%ebp)
```

  Notice the extra `mov %esi,%esi` instruction. Where did this come from? The GNU assembler inserted it to serve as the two bytes of padding needed to align the `movl $.LnrB,%eax` instruction on a four-byte boundary, but pretended it didn't exist when adding debug information. Thus when Valgrind reads the debug info it thinks that the `movl $0x1,0xffffffec(%ebp)` instruction covers the address range 0x8048f2b--0x804833 by itself, and attributes the counts for the `mov %esi,%esi` to it.

- Sometimes, the same filename might be represented with a relative name and with an absolute name in different parts of the debug info, eg: `/home/user/proj/proj.h` and `../proj.h`. In this case, if you use auto-annotation, the file will be annotated twice with the counts split between the two.
- If you compile some files with `-g` and some without, some events that take place in a file without debug info could be attributed to the last line of a file with debug info (whichever one gets placed before the non-debug-info file in the executable).

This list looks long, but these cases should be fairly rare.

5.2.11. Merging Profiles with cg_merge

cg_merge is a simple program which reads multiple profile files, as created by Cachegrind, merges them together, and writes the results into another file in the same format. You can then examine the merged results using `cg_annotate <filename>`, as described above. The merging functionality might be useful if you want to aggregate costs over multiple runs of the same program, or from a single parallel run with multiple instances of the same program.

cg_merge is invoked as follows:

```
cg_merge -o outputfile file1 file2 file3 ...
```

It reads and checks `file1`, then read and checks `file2` and merges it into the running totals, then the same with `file3`, etc. The final results are written to `outputfile`, or to standard out if no output file is specified.

Costs are summed on a per-function, per-line and per-instruction basis. Because of this, the order in which the input files does not matter, although you should take care to only mention each file once, since any file mentioned twice will be added in twice.

cg_merge does not attempt to check that the input files come from runs of the same executable. It will happily merge together profile files from completely unrelated programs. It does however check that the `Events:` lines of all the inputs are identical, so as to ensure that the addition of costs makes sense. For example, it would be nonsensical for it to add a number indicating D1 read references to a number from a different file indicating LL write misses.

A number of other syntax and sanity checks are done whilst reading the inputs. cg_merge will stop and attempt to print a helpful error message if any of the input files fail these checks.

5.2.12. Differencing Profiles with cg_diff

cg_diff is a simple program which reads two profile files, as created by Cachegrind, finds the difference between them, and writes the results into another file in the same format. You can then examine the merged results using `cg_annotate <filename>`, as described above. This is very useful if you want to measure how a change to a program affected its performance.

cg_diff is invoked as follows:

```
cg_diff file1 file2
```

It reads and checks `file1`, then read and checks `file2`, then computes the difference (effectively `file1` - `file2`). The final results are written to standard output.

Costs are summed on a per-function basis. Per-line costs are not summed, because doing so is too difficult. For example, consider differencing two profiles, one from a single-file program A, and one from the same program A where a single blank line was inserted at the top of the file. Every single per-line count has changed. In comparison, the per-function counts have not changed. The per-function count differences are still very useful for determining differences between programs. Note that because the result is the difference of two profiles, many of the counts will be negative; this indicates that the counts for the relevant function are fewer in the second version than those in the first version.

cg_diff does not attempt to check that the input files come from runs of the same executable. It will happily merge together profile files from completely unrelated programs. It does however check that the `Events:` lines of all the inputs are identical, so as to ensure that the addition of costs makes sense. For example, it would be nonsensical for it to add a number indicating D1 read references to a number from a different file indicating LL write misses.

A number of other syntax and sanity checks are done whilst reading the inputs. cg_diff will stop and attempt to print a helpful error message if any of the input files fail these checks.

Sometimes you will want to compare Cachegrind profiles of two versions of a program that you have sitting side-by-side. For example, you might have `version1/prog.c` and `version2/prog.c`, where the second is slightly different to the first. A straight comparison of the two will not be useful -- because functions are qualified with filenames, a function `f` will be listed as `version1/prog.c:f` for the first version but `version2/prog.c:f` for the second version.

When this happens, you can use the `--mod-filename` option. Its argument is a Perl search-and-replace expression that will be applied to all the filenames in both Cachegrind output files. It can be used to remove minor differences in filenames. For example, the option `--mod-filename='s/version[0-9]/versionN/'` will suffice for this case.

Similarly, sometimes compilers auto-generate certain functions and give them randomized names. For example, GCC sometimes auto-generates functions with names like `T.1234`, and the suffixes vary from build to build. You can use the `--mod-funcname` option to remove small differences like these; it works in the same way as `--mod-filename`.

5.3. Cachegrind Command-line Options

Cachegrind-specific options are:

`--I1=<size>,<associativity>,<line size>`

Specify the size, associativity and line size of the level 1 instruction cache.

`--D1=<size>,<associativity>,<line size>`

Specify the size, associativity and line size of the level 1 data cache.

`--LL=<size>,<associativity>,<line size>`

Specify the size, associativity and line size of the last-level cache.

`--cache-sim=no|yes [yes]`

Enables or disables collection of cache access and miss counts.

`--branch-sim=no|yes [no]`

Enables or disables collection of branch instruction and misprediction counts. By default this is disabled as it slows Cachegrind down by approximately 25%. Note that you cannot specify `--cache-sim=no` and `--branch-sim=no` together, as that would leave Cachegrind with no information to collect.

`--cachegrind-out-file=<file>`

Write the profile data to `file` rather than to the default output file, `cachegrind.out.<pid>`. The `%p` and `%q` format specifiers can be used to embed the process ID and/or the contents of an environment variable in the name, as is the case for the core option `--log-file`.

5.4. cg_annotate Command-line Options

`-h --help`

Show the help message.

`--version`

Show the version number.

`--show=A,B,C [default: all, using order in cachegrind.out.<pid>]`

Specifies which events to show (and the column order). Default is to use all present in the `cachegrind.out.<pid>` file (and use the order in the file). Useful if you want to concentrate on, for example, I cache misses (`--show=I1mr,ILmr`), or data read misses (`--show=D1mr,DLmr`), or LL data misses (`--show=DLmr,DLmw`). Best used in conjunction with `--sort`.

`--sort=A,B,C [default: order in cachegrind.out.<pid>]`

Specifies the events upon which the sorting of the function-by-function entries will be based.

`--threshold=X [default: 0.1%]`

Sets the threshold for the function-by-function summary. A function is shown if it accounts for more than X% of the counts for the primary sort event. If auto-annotating, also affects which files are annotated.

Note: thresholds can be set for more than one of the events by appending any events for the `--sort` option with a colon and a number (no spaces, though). E.g. if you want to see each function that covers more than 1% of LL read misses or 1% of LL write misses, use this option:

```
--sort=DLmr:1,DLmw:1
```

`--show-percs=<no|yes> [default: yes]`

When enabled, a percentage is printed next to all event counts. This helps gauge the relative importance of each function and line.

`--auto=<no|yes> [default: yes]`

When enabled, automatically annotates every file that is mentioned in the function-by-function summary that can be found. Also gives a list of those that couldn't be found.

`--context=N [default: 8]`

Print N lines of context before and after each annotated line. Avoids printing large sections of source files that were not executed. Use a large number (e.g. 100000) to show all source lines.

`-I<dir> --include=<dir> [default: none]`

Adds a directory to the list in which to search for files. Multiple `-I`/`--include` options can be given to add multiple directories.

5.5. cg_merge Command-line Options

`-o outfile`

Write the profile data to `outfile` rather than to standard output.

5.6. cg_diff Command-line Options

`-h --help`

Show the help message.

`--version`

Show the version number.

`--mod-filename=<expr> [default: none]`

Specifies a Perl search-and-replace expression that is applied to all filenames. Useful for removing minor differences in paths between two different versions of a program that are sitting in different directories.

`--mod-funcname=<expr> [default: none]`

Like `--mod-filename`, but for filenames. Useful for removing minor differences in randomized names of auto-generated functions generated by some compilers.

5.7. Acting on Cachegrind's Information

Cachegrind gives you lots of information, but acting on that information isn't always easy. Here are some rules of thumb that we have found to be useful.

First of all, the global hit/miss counts and miss rates are not that useful. If you have multiple programs or multiple runs of a program, comparing the numbers might identify if any are outliers and worthy of closer investigation. Otherwise, they're not enough to act on.

The function-by-function counts are more useful to look at, as they pinpoint which functions are causing large numbers of counts. However, beware that inlining can make these counts misleading. If a function `f` is always inlined, counts will be attributed to the functions it is inlined into, rather than itself. However, if you look at the line-by-line annotations for `f` you'll see the counts that belong to `f`. (This is hard to avoid, it's how the debug info is structured.) So it's worth looking for large numbers in the line-by-line annotations.

The line-by-line source code annotations are much more useful. In our experience, the best place to start is by looking at the `Ir` numbers. They simply measure how many instructions were executed for each line, and don't include any cache information, but they can still be very useful for identifying bottlenecks.

After that, we have found that LL misses are typically a much bigger source of slow-downs than L1 misses. So it's worth looking for any snippets of code with high `DLmr` or `DLmw` counts. (You can use `--show=DLmr --sort=DLmr` with cg_annotate to focus just on `DLmr` counts, for example.) If you find any, it's still not always easy to work out how to improve things. You need to have a reasonable understanding of how caches work, the principles of locality, and your program's data access patterns. Improving things may require redesigning a data structure, for example.

Looking at the `Bcm` and `Bim` misses can also be helpful. In particular, `Bim` misses are often caused by `switch` statements, and in some cases these `switch` statements can be replaced with table-driven code. For example, you might replace code like this:

```
enum E { A, B, C };
enum E e;
int i;
...
switch (e)
{
    case A: i += 1; break;
    case B: i += 2; break;
    case C: i += 3; break;
}
```

with code like this:

```
enum E { A, B, C };
enum E e;
int table[] = { 1, 2, 3 };
int i;
...
i += table[e];
```

This is obviously a contrived example, but the basic principle applies in a wide variety of situations.

In short, Cachegrind can tell you where some of the bottlenecks in your code are, but it can't tell you how to fix them. You have to work that out for yourself. But at least you have the information!

5.8. Simulation Details

This section talks about details you don't need to know about in order to use Cachegrind, but may be of interest to some people.

5.8.1. Cache Simulation Specifics

Specific characteristics of the cache simulation are as follows:

- Write-allocate: when a write miss occurs, the block written to is brought into the D1 cache. Most modern caches have this property.
- Bit-selection hash function: the set of line(s) in the cache to which a memory block maps is chosen by the middle bits M--(M+N-1) of the byte address, where:

  line size = 2^M bytes

  (cache size / line size / associativity) = 2^N bytes
- Inclusive LL cache: the LL cache typically replicates all the entries of the L1 caches, because fetching into L1 involves fetching into LL first (this does not guarantee strict inclusiveness, as lines evicted from LL still could reside in L1). This is standard on Pentium chips, but AMD Opterons, Athlons and Durons use an exclusive LL cache that only holds blocks evicted from L1. Ditto most modern VIA CPUs.

The cache configuration simulated (cache size, associativity and line size) is determined automatically using the x86 CPUID instruction. If you have a machine that (a) doesn't support the CPUID instruction, or (b) supports it in an early incarnation that doesn't give any cache information, then Cachegrind will fall back to using a default configuration (that of a model 3/4 Athlon). Cachegrind will tell you if this happens. You can manually specify one, two or all three levels (I1/D1/LL) of the cache from the command line using the `--I1`, `--D1` and `--LL` options. For cache parameters to be valid for simulation, the number of sets (with associativity being the number of cache lines in each set) has to be a power of two.

On PowerPC platforms Cachegrind cannot automatically determine the cache configuration, so you will need to specify it with the `--I1`, `--D1` and `--LL` options.

Other noteworthy behaviour:

- References that straddle two cache lines are treated as follows:

  If both blocks hit --> counted as one hit

  If one block hits, the other misses --> counted as one miss.

  If both blocks miss --> counted as one miss (not two)
- Instructions that modify a memory location (e.g. `inc` and `dec`) are counted as doing just a read, i.e. a single data reference. This may seem strange, but since the write can never cause a miss (the read guarantees the block is in the cache) it's not very interesting.

  Thus it measures not the number of times the data cache is accessed, but the number of times a data cache miss could occur.

If you are interested in simulating a cache with different properties, it is not particularly hard to write your own cache simulator, or to modify the existing ones in `cg_sim.c`. We'd be interested to hear from anyone who does.

5.8.2. Branch Simulation Specifics

Cachegrind simulates branch predictors intended to be typical of mainstream desktop/server processors of around 2004.

Conditional branches are predicted using an array of 16384 2-bit saturating counters. The array index used for a branch instruction is computed partly from the low-order bits of the branch instruction's address and partly using the taken/not-taken behaviour of the last few conditional branches. As a result the predictions for any specific branch depend both on its own history and the behaviour of previous branches. This is a standard technique for improving prediction accuracy.

For indirect branches (that is, jumps to unknown destinations) Cachegrind uses a simple branch target address predictor. Targets are predicted using an array of 512 entries indexed by the low order 9 bits of the branch instruction's address. Each branch is predicted to jump to the same address it did last time. Any other behaviour causes a mispredict.

More recent processors have better branch predictors, in particular better indirect branch predictors. Cachegrind's predictor design is deliberately conservative so as to be representative of the large installed base of processors which pre-date widespread deployment of more sophisticated indirect branch predictors. In particular, late model Pentium 4s (Prescott), Pentium M, Core and Core 2 have more sophisticated indirect branch predictors than modelled by Cachegrind.

Cachegrind does not simulate a return stack predictor. It assumes that processors perfectly predict function return addresses, an assumption which is probably close to being true.

See Hennessy and Patterson's classic text "Computer Architecture: A Quantitative Approach", 4th edition (2007), Section 2.3 (pages 80-89) for background on modern branch predictors.

5.8.3. Accuracy

Valgrind's cache profiling has a number of shortcomings:

- It doesn't account for kernel activity -- the effect of system calls on the cache and branch predictor contents is ignored.

- It doesn't account for other process activity. This is probably desirable when considering a single program.
- It doesn't account for virtual-to-physical address mappings. Hence the simulation is not a true representation of what's happening in the cache. Most caches and branch predictors are physically indexed, but Cachegrind simulates caches using virtual addresses.
- It doesn't account for cache misses not visible at the instruction level, e.g. those arising from TLB misses, or speculative execution.
- Valgrind will schedule threads differently from how they would be when running natively. This could warp the results for threaded programs.
- The x86/amd64 instructions `bts`, `btr` and `btc` will incorrectly be counted as doing a data read if both the arguments are registers, eg:

  ```
      btsl %eax, %edx
  ```

  This should only happen rarely.
- x86/amd64 FPU instructions with data sizes of 28 and 108 bytes (e.g. `fsave`) are treated as though they only access 16 bytes. These instructions seem to be rare so hopefully this won't affect accuracy much.

Another thing worth noting is that results are very sensitive. Changing the size of the executable being profiled, or the sizes of any of the shared libraries it uses, or even the length of their file names, can perturb the results. Variations will be small, but don't expect perfectly repeatable results if your program changes at all.

More recent GNU/Linux distributions do address space randomisation, in which identical runs of the same program have their shared libraries loaded at different locations, as a security measure. This also perturbs the results.

While these factors mean you shouldn't trust the results to be super-accurate, they should be close enough to be useful.

5.9. Implementation Details

This section talks about details you don't need to know about in order to use Cachegrind, but may be of interest to some people.

5.9.1. How Cachegrind Works

The best reference for understanding how Cachegrind works is chapter 3 of "Dynamic Binary Analysis and Instrumentation", by Nicholas Nethercote. It is available on the Valgrind publications page.

5.9.2. Cachegrind Output File Format

The file format is fairly straightforward, basically giving the cost centre for every line, grouped by files and functions. It's also totally generic and self-describing, in the sense that it can be used for any events that can be counted on a line-by-line basis, not just cache and branch predictor events. For example, earlier versions of Cachegrind didn't have a branch predictor simulation. When this was added, the file format didn't need to change at all. So the format (and consequently, cg_annotate) could be used by other tools.

The file format:

```
file         ::= desc_line* cmd_line events_line data_line+ summary_line
desc_line    ::= "desc:" ws? non_nl_string
cmd_line     ::= "cmd:" ws? cmd
events_line  ::= "events:" ws? (event ws)+
data_line    ::= file_line | fn_line | count_line
file_line    ::= "fl=" filename
fn_line      ::= "fn=" fn_name
count_line   ::= line_num ws? (count ws)+
summary_line ::= "summary:" ws? (count ws)+
count        ::= num | "."
```

Where:

- `non_nl_string` is any string not containing a newline.
- `cmd` is a string holding the command line of the profiled program.
- `event` is a string containing no whitespace.
- `filename` and `fn_name` are strings.
- `num` and `line_num` are decimal numbers.
- `ws` is whitespace.

The contents of the "desc:" lines are printed out at the top of the summary. This is a generic way of providing simulation specific information, e.g. for giving the cache configuration for cache simulation.

More than one line of info can be presented for each file/fn/line number. In such cases, the counts for the named events will be accumulated.

Counts can be "." to represent zero. This makes the files easier for humans to read.

The number of counts in each `line` and the `summary_line` should not exceed the number of events in the `event_line`. If the number in each `line` is less, cg_annotate treats those missing as though they were a "." entry. This saves space.

A `file_line` changes the current file name. A `fn_line` changes the current function name. A `count_line` contains counts that pertain to the current filename/fn_name. A "fn=" `file_line` and a `fn_line` must appear before any `count_line`s to give the context of the first `count_line`s.

Each `file_line` will normally be immediately followed by a `fn_line`. But it doesn't have to be.

The summary line is redundant, because it just holds the total counts for each event. But this serves as a useful sanity check of the data; if the totals for each event don't match the summary line, something has gone wrong.

<< 4. Memcheck: a memory error detector

Up

Home

6. Callgrind: a call-graph generating cache and branch prediction profiler >>