Appendix to Practice 4: Cachegrind basics

To execute a code using the cachegrind tool, we must type the following command

```
valgrind --tool=cachegrind ./executable
```

When executing Valgrind, the option `-v` can be used to show additional details such as the configuration of the simulated caches, which by default mimics the cache hierarchy in the computer used for running the simulation. Valgrind only accounts for 2 levels of cache, and configures them according to the level 1 cache and last level (LL) cache in the system (usually level 3 in most of the modern processors). [1] When interpreting the miss rates for the LL cache, keep in mind that the results shown by Cachegrind **are not** local miss rates for LL caches, that is, the percentage of LL accesses that produce a miss, but rather the global miss rates, i.e., the percentage of all memory accesses that result in a miss at the LL level. To calculate the local miss rates of LL you need to manually divide the number of LL misses by the total number of L1 misses (I1 misses + D1 misses). Additionally, Cachegrind does always print the number of accesses and misses using commas as thousands separators, which makes it difficult to copy-and-paste its output. One option to avoid this problem is to filter the output by Cachegrind to remove these commas. This can be done using the following command:

```
  valgrind --tool=cachegrind ./executable  2>&1 | sed -e 's/,//g'
```

The configuration of the level-1 data cache can be controlled using the option

```
--D1=<tamaño>,<asociatividad>,<tamaño de línea>
```

where all sizes are given in bytes. The tool restricts the combinations that can be used to realistic ones. For instance, cache line sizes must be a power of two and larger than 16. Similarly, the LL size is controlled using the option

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

Cachegrind stores detailed information about the simulations in files that can be parsed using the `cg_annotate` command. By default, these files are named `cachegrind.out.<pid>` (where `pid` is the process identifier of the simulation), but the user can change this name using `--cachegrind-out-file=<fichero>`. The output of `cg_annotate` is quite wide, so it might be convenient to resize the terminal to a width of at least 120 characters. If, for instance, the `pid` of the last ran simulation was 13195, we can read this file by running

```
  cg_annotate cachegrind.out.13195
```

The output is documented in Cachegrind's manual. It basically shows the configuration of the simulated cache hierarchy, the simulated events, and the global statistics for the whole program, as well as for each individual function. Functions are sorted in decreasing order by `Ir`, i.e., the number of instructions read (and therefore executed); and include the instructions explicitly written in our code plus those in called libraries. For instance, it will show functions which have been invoked by the operating system during the load and start of our program, or standard library functions that we have explicitly called in our code. Each function is labeled as `file:function_name`. Each of these fields can be replaced by `???` if `cg_annotate` is not capable of inferring it from the available information. As such, you will see that, in our example program, the list of functions is headed by `main`, but instead of showing up as `traspuesta.c:main` it appears as `???:main`. The reason is that we must compile our codes including the `-g` option so that debugging information is included for `cg_annotate` to find. You can find more information about this compilation option in the compiler's manual (man gcc). Another advantage of compiling with `-g` is that Cachegrind can then report information at the line-of-code level. To enable this behavior, execute `cg_annotate` with the `--auto=yes` option. If we execute

[1] Some Valgrind versions do not correctly recognize the last level cache in some processors. The actual size of this cache, along with other processor features, can be found in the `/proc/cpuinfo` file.

```
gcc -g -O3 -o traspuesta traspuesta.c
valgrind --tool=cachegrind ./traspuesta
cg_annotate --auto=yes  cachegrind.out.13618
```

and assuming that the `pid` of the simulation was 13168, below the list of statistics for each function we will find a list for the statistics of each LOC in the executable, including the `main` function.

Note that, thanks to `cg_annotate`, we can estimate the misses incurred by each line of code independently. We can even see the miss count for each reference if we structure our code so that it emits a single memory reference per line, which can be done using code transformations such as

$$\texttt{a[i] = b[i] + c[i];} \Longrightarrow \begin{cases} \texttt{br = b[i];} \\ \texttt{cr = c[i];} \\ \texttt{a[i] = br + cr;} \end{cases}$$

The downside of this transformation is that we need to introduce additional, temporary variables. If we employ some degree of optimization the compiler will lower these scalar variables to CPU registers, avoiding any impact in the statistics provided by Cachegrind. Unfortunately, enabling compiler optimizations might trigger deeper code transformations, preventing `cg_annotate` from providing statistics for each line in the original code. If this were to happen, we might want to disable compiler optimization. In this case, the added temporary variables will be accessed through the cache, polluting the obtained statistics. However, we can safely assume that all the accesses to this variables will be hits in the D1 cache.