we watch modifications to `a` as follows:

```
$ gdb a.out
(gdb) watch a
Watchpoint 1: a
(gdb) r
Hardware watchpoint 1: a

Old value = 0
New value = 1
main () at watch.c:7
7         return a;
(gdb)
```

`thread` *n*

To switch to a different thread, identified by the integer *n*, the command `thread` *n* is used.

6.6 `operf`

To make it easier to understand and improve the performance of complete systems including OS kernels, or individual programs, modern processors can be told to count for instance the number of executed instructions, the number of clock cycles, the number of the different types of cache misses (i.e. access type and cache level — not true or false sharing miss). The generic term is that the processor counts different events. Instead of just printing how many of a counted event occurred after the program completed, statistics can be produced for each instruction address or summarized for source code lines. It works as follows. When a counter has reached a certain number of events, the processor raises a **performance monitor exception** and then two items of information are saved: (1) which counted event happened, and (2) the instruction address of the program which triggered the exception (but not e.g. which variable caused a cache miss). With this information, good statistics about the behavior of the program can be collected while only disturbing the measurements marginally (such as less than one percent). Raising an exception after fewer counted events provides finer granularity but disturbs the measurements and is not recommended. The predefined defaults should usually not be changed.

With this information we can measure which functions and instructions are executed most frequently, by counting clock cycles, or what actually is happening in the hardware unit we are interested in. The command `operf` is part of **OProfile** for Linux, and similar programs exist for other operating systems as well. Older versions needed root access to the computer to do profiling. With

recent versions, only an initial command needs to be done as root (each time your computer is started and you want to do profiling). Although it may sound not so pleasant, but you need to lower the Linux security level so that all or a certain group of users can use this profiler. On Linux there is a special file `/proc/sys/kernel/perf_event_paranoid`. This file does not actually exist on a harddisk. Instead, reading it will ask the Linux kernel for the value of a certain variable (in the kernel), and writing to it, will set that variable. There are numerous similar files for other purposes. Every user can read the file but writing to it must be done by a so called superuser. Depending on the security requirements of your machine, you either might want to profile only as root (which has its own dangers, since running programs as root usually is a bad idea, unless you trust it completely), create a Unix group for certain users and let members of that group use the profiler, or let all users profile their own commands. On Ubuntu Linux, to install and then select the last option, type the following

```
sudo bash
apt install oprofile
echo 1 > /proc/sys/kernel/perf_event_paranoid
```

Each processor has its own set of performance counters and typing the command

```
ophelp
```

will list them. The most important is `CYCLES`. We will make measurements on the program `intopt`, whose pseudo code can be found in Chapter B. This program is intentionally *not* optimized except for avoiding doing excessive amounts of heap memory allocations (at Lund University, making a fast implementation of this pseudo code is an assignment in the course EDAG01 Efficient C). We will profile intopt when it finds an optimal solution to an integer program with with 15 variables and constraints. When a program is first profiled, it is usually most convenient to compile it without optimization. This gives a first overview. We first issue the command

```
operf -e CYCLES:100000:0:0:1 intopt -y 15 1
```

The meaning of this is that a the program counter will be saved every 100,000 clock cycle, a so called unit mask is zero (which we can ignore), kernel code will not be profiled (the last zero), and that user code will be profiled. The program to profile with any arguments is then given (which in this case is the size and a seed).

The statistics is saved in the directory `oprofile_data`. There are two programs with which we can access this data.

```
opreport -t 0.7 -l
```

prints all functions in which at least 0.7 % of the cycles where sampled. The essential parts of the output are:

```
CPU: ppc64 POWER8, speed 3491 MHz (estimated)
Counted CYCLES events (Cycles)
samples  %        symbol name
36404    87.3207  pivot
2544      6.1022  xsimplex
844       2.0245  select_nonbasic
673       1.6143  xinitial
368       0.8827  extend
```

Redoing the measurement with the `-O3` option to gcc, we instead get:

```
12079    86.3279  intopt                  pivot
1216      8.6907  intopt                  xinitial
242       1.7296  intopt                  xsimplex.constprop.0
145       1.0363  libc-2.27.so            _int_malloc
```

As we can see, `xsimplex` has been cloned and specialized with constant propagation. The goal is to find the most time-consuming function, which in both cases is `pivot`. To see which source line in `pivot` gets most samples, we can use

```
opannotate -s
```

which prints the source code file and the number of samples taken for each line. With optimization, the line number information is no longer accurate and therefore we show the output from the unoptimized run. Most samples were taken in the following loop

```
  603  1.4464: for (i = 0; i < m; i += 1) {
  104  0.2495:  if (i == row)
   55  0.1319:   continue;
 1721  4.1281:  for (j = 0; j < n; j += 1)
  743  1.7822:   if (j != col)
27868 66.8458:    a[i][j] = a[i][j] - a[i][col] * a[row][j] / a[row][col];
             : }
```

which is not very surprising if we look at the pseudo code in Chapter B. The command

```
opannotate -s -a
```

prints samples at the level of dispatch groups (see Section 3.4, on page 158). A trick to easier find a piece of code in an assembler listing is to put something easy to find in the source file, such as an `x ^= 1234` provided it is easy to find an xor instruction. To prevent the compiler from optimizing away this statement, `x` should be a global volatile variable. On Power, search for `xori.*1234` in an editor.

6.7 gprof

The profiler gprof is more than 30 years old and still one of the most important UNIX tools. It was developed at Berkeley by BSD kernel developers Kessel and McKusick, and Prof. S. Graham [28].

In contrast to operf it relies on the compiler to generate special instrumentation code, through which exact counts of functions calls are produced. The compiler switch is -pg. The profiled program ends with writing the profile data to a file gmon.out, and therefore it must terminate by returning from main or by calling exit. Let us look at the output from profiling intopt, introduced in Section 6.6, on page 220. Having compiled with -pg and run intopt, we can give the command below, where -T stands for "traditional" (i.e. BSD style):

```
gprof -T intopt
```

which produces the output:

```
  %   cumulative     self              self   total
time     seconds  seconds   calls  ms/call ms/call name
88.3        1.05     1.05   148711    0.01    0.01 pivot
 5.9        1.12     0.07     7038    0.01    0.16 xsimplex
 2.5        1.15     0.03   148796    0.00    0.00 select_nonbasic
```

There is more output from gprof shown below. The output shows which functions take most of the time, but not which source code line. The number of calls to each function is exact.

As we can see, the relative times are somewhat different than what was reported by operf. It's important to have in mind that we don't use these profilers to collect the exact execution time of a piece of code (which can better be done as shown in Section 13.26.1.2, on page 555), but rather to collect information about how we should proceed to improve the code. Both profilers are extremely valuable for that purpose.

In addition to the above output, gprof also shows, for each function f, which other function called f, i.e. the callers of f, and which functions f calls, i.e. the callees of f. For example, intopt calls succ which calls the functions below.

```
                0.00        1.19    3518/3518        intopt [1]
[4]    100.0    0.00        1.19    3518         succ [4]
                0.00        1.18    3518/3519        simplex [6]
                0.01        0.00    3518/3518        extend [14]
                0.00        0.00    2444/2445        integer [23]
                0.00        0.00    2428/2429        branch [24]
                0.00        0.00    1758/3519        free_node [20]
                0.00        0.00      16/16          bound [26]
```

The first column is the time spent in a function and the second column includes called functions. The third column is important. It says intopt made all

3518 calls to `succ`, and, for instance, that `succ` made 3518 of the total of 3519 calls to `simplex` (`intopt` also calls `simplex`). The number to the right is an index to help finding functions.

6.8 gcov test coverage

The tool `gcov` prints the number of times each source code line was executed, and is used for understanding the behavior of the program both for tuning and finding out to what extent our test cases execute each source code line at least once.

To use `gcov`, use gcc and compile with the switches `-fprofile-arcs` and `-ftest-coverage`, and execute the program normally. In this section we will continue to use to use the intopt program. After having compiled and run the program, we use `gcov` as follows:

```
gcov simplex.c
```

which produces the file `simplex.c.gcov`:

```
 4130292: 478: for (i = 0; i < m; i += 1) {
 3981581: 479:  if (i == row)
  148711: 480:   continue;
64570580: 481:  for (j = 0; j < n; j += 1)
60737710: 482:   if (j != col)
56904840: 483:    a[i][j] = a[i][j] - a[i][col] * a[row][j] / a[row][col];
       -: 484: }
```

The number in the first column is the number of times that line was executed, and the second column shows the line number. We can also get branch frequencies by using the `-b` switch.

```
gcov -b intopt.c
```

In the file `intopt.c.gcov` we find for the function `is_integer`:

```
    33933:  327:        r = lround(x);
        -:  328:
    33933:  329:        if (fabs(r-x) < intopt_eps) {
branch  0 taken 88% (fallthrough)
branch  1 taken 12%
    29743:  330:                *xp = r;
    29743:  331:                return 1;
        -:  332:        } else
     4190:  333:                return 0;
        -:  334:
```

We can see that the input to this function most of the time is (or is close to) an integer. Branch frequencies can be very useful when trying to understand how a