Unix Profiling Tools for C
gprof and gcov applied to Conway's Game of Life

CS 510, Summer 2011

Author: Charles L. Capps

Last edited: July 31, 2011

Due: July 18th, 2011

1 Purpose

The purpose of this paper is to analyse several common Unix profiling and coverage tools for the C language. We examine `gprof` for profiling and `gcov` for line-by-line analysis and statement coverage. The `time` command will be used to measure time spent in user code, time spent in system code, and the total time. The study of these tools is necessarily intertwined with the `gcc` compiler.

Performance will be measured for different levels of `gcc` optimisation with examples drawn from the author's implementation of Conway's game of life. Several optimisations of the code will be implemented, and the change in performance will be measured. The resultant data from `gprof` and `gcov` will be examined for every combination of `gcc` optimization and code improvement.

In addition, optimisations made by `gcc` are inferred. For example, when a function is "inlined" by `gcc` it will no longer appear in the `gprof` results. This highlights the importance of running `gprof` and `gcov` *without compiler optimisations*. We demonstrate other issues one encounters when trying to profile optimised code.

2 Method: How this is accomplished

Each run of the Game of Life consists of 1024 evolutions starting from a fixed input of size `42x89`. The input is given in the Appendix. The input has a "Gosper glider gun" and a pulsar. The Gosper glider gun is the first example of a structure in the Game of Life that has unbounded growth, discovered by Gosper in 1970. It makes gliders, which eventually wrap around and destroy the glider gun.

Four versions of the most performance-critical function (`get_neighbors_torus`) are presented. This function counts the living neighbors of a cell by wrapping the left side of the rectangle to the right side, and the top to the bottom, like a torus. It's the most performance-critical function, because it's called for every cell, for every evolution.

```
(1) (1024 evolutions)x(42 rows)x(89 columns) = 3,827,712 calls to get_neighbors_torus
```

Version 1 of `get_neighbors_torus` is the naive initial implementation. Version 2 makes the easy optimisation of defining constants for the bounds of the loops instead of referring to `game->rows` inside a struct. We'd expect this to eliminate some memory accesses. Version 3 improves the structure of the `if-else` statements inside the loop that counts living neighbors and eliminates checking that the input row and column are within bounds. Version 4 is more daring: in version 4 the `new_game` function is modified to `malloc` the required memory for a game of life in a single block (instead of allocating the row pointers and the space for each row separately). The hypothesis is that version 4 will reduce cache misses by improving spatial locality. Loop unrolling will also be explored later.

Standard statistical analysis is applied. A bash script is used to run each combination of `gcc` optimisation and code version 100 times. Each final data point is given by the average of 100 runs, with error bars equal to the standard deviation. The distribution of times is assumed to be Gaussian, but some data is fit to a Gaussian to verify this assumption.

The results are graphed with error bars using Libre Office.

3 Tools

3.1 Justification for tools used

As mentioned above, the tools used in this project include `gcc`, `gprof`, `gcov`, and `time`. The `gcc` compiler (GNU compiler collection) was first released in 1987 prior to the creation of the Linux kernel. It's become an extremely widespread compiler system. GCC is used on all unix-based operating systems, such as Mac OS X, Linux, and Free BSD. It can even target several videogame consoles.

Originally only supporting the C language, GCC has been extended to support C++, Objective-C, Java (`gcj` tool), Go, and some older languages such as Fortran and Ada. For more information see [4].

The profiling tool, `gprof`, was released in 1982. According to Wikipedia ([5]), it was the first tool to give a complete call-graph analysis. A call-graph gives information beyond simply how much time was spent in each function. A call-graph indicates the *callers* and *callees* of each function, and how many times the function was called by each of its callers. This is extremely useful, since a programmer can determine not only *which* function was called frequently, but also *where* it was called the most. This paper will demonstrate the usefulness of this feature.

Aside from being mature and widely used tools, the author is simply interested in free software and developing software with the GNU suite of tools. There are alternatives, mentioned below.

3.2 Alternative tools

One of the most notable alternative tools is `valgrind`. Many of the features of `valgrind` overlap with the features of `gprof` and `gcov`. For example, the `cachegrind` tool included with `valgrind` will annotate code with branch probabilities and the number of times each line was executed. `gcov` provides the same features. `cachegrind` will also output the number of times each function was called, much like `gprof`. The `cachegrind` manual can be found here: `http://valgrind.org/docs/manual/cg-manual.html` `.cachegrind` can also annotate assembly code; and it can determine cache miss rate. Time permitting, `cachegrind` will also be studied.

As an alternative to `time`, the author could have used `get_time_of_day` to measure elapsed time in microseconds, or `clock_gettime` to measure time in nanoseconds. These functions may be examined if time permits, but the primary purpose of this report is to study *command-line tools*. If we give the CPU plenty of work to do, the `time` command is more than sufficiently precise. Also, it has the advantage that it measures the time spent in kernel code (system time) and the total time including context switches. Arguably, the total time is the ultimate measure of performance!

Another interesting option is OProfile ([7]). According to the project homepage, it can profile all code running, including kernel code, hardware interrupt handlers, and applications. It's interesting that OProfile can sample code as it's running. In contrast, `gprof` requires you to run a program from start to completion– only then you can see profiling results.

From my survey of the tools available, it appears that the GNU tools are by far the most commonly used for Unix-based operating systems. In fact, many of the other profiling tools available are modifications of a particular version of `gcc`, `gprof` or `gcov`. This website ([8]), by a Linux afficionado in Manchester, UK, has a good list of available profilers for Linux.

4 Techniques

4.1 Discussion of techniques used

The techniques used for versions 1-3 of the code are pretty basic. Using constant loop bounds instead of accessing a struct every time reduces memory accesses. Using if-else constructs where possible to avoid checking a condition clearly reduces computation. The interest is mainly in seeing how much of an impact these changes have on the performance, and studying how we can use these tools to get useful information.

More interesting are versions 4 and 5. Version 4 uses a single call to `malloc` to allocate all the data in a game of life matrix and the pointers to each row. One then has to tie the row pointers to the proper location in memory. The usual way to do this is to allocate each row separately; there is a clear explanation of this common process on the C-faq website ([2]). See Figure 1.

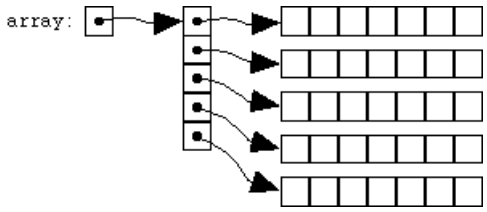


Figure 1: Standard way to dynamically allocate 2d array with malloc.

This process has two drawbacks. A call to malloc is made for each row of the matrix – malloc is an expensive operation. Also, there is no guarantee the memory will be allocated contiguosly. Allocating memory likely to be used at the same time can increase cache hits and improve performance. We will demonstrate this with time data and cache hit/miss data.

Figure 2 shows how we chose to allocate memory with a single call to `malloc`.

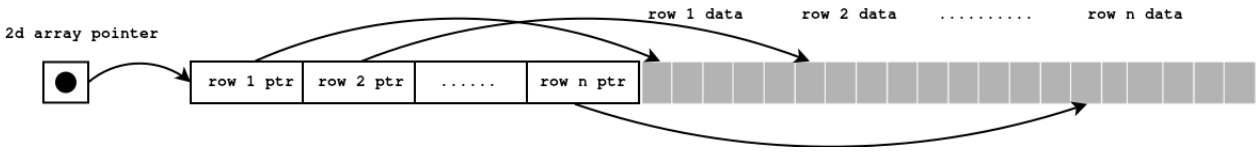


Figure 2: Using a single call to malloc to allocate a 2d array.

4.2 Where are these techniques used

Using a single call to `malloc` is a fairly common technique to improve performance. A good explanation of the technique was found on Lawrence University's website for the class *Introduction to Scientific Programming* ([3]).

The technique taught at Lawrence University uses a slightly different approach, but is essentially the same. A struct is used for the 2d array. Two calls to `malloc` are made – one to allocate the row pointers, and one to allocate all the data in a single call. It still avoids calling `malloc` for each row, and the data is still contiguous in memory. However, the row pointers will likely be stored in a different place on the heap than the data.

5 Results

5.1 Data from unix time command

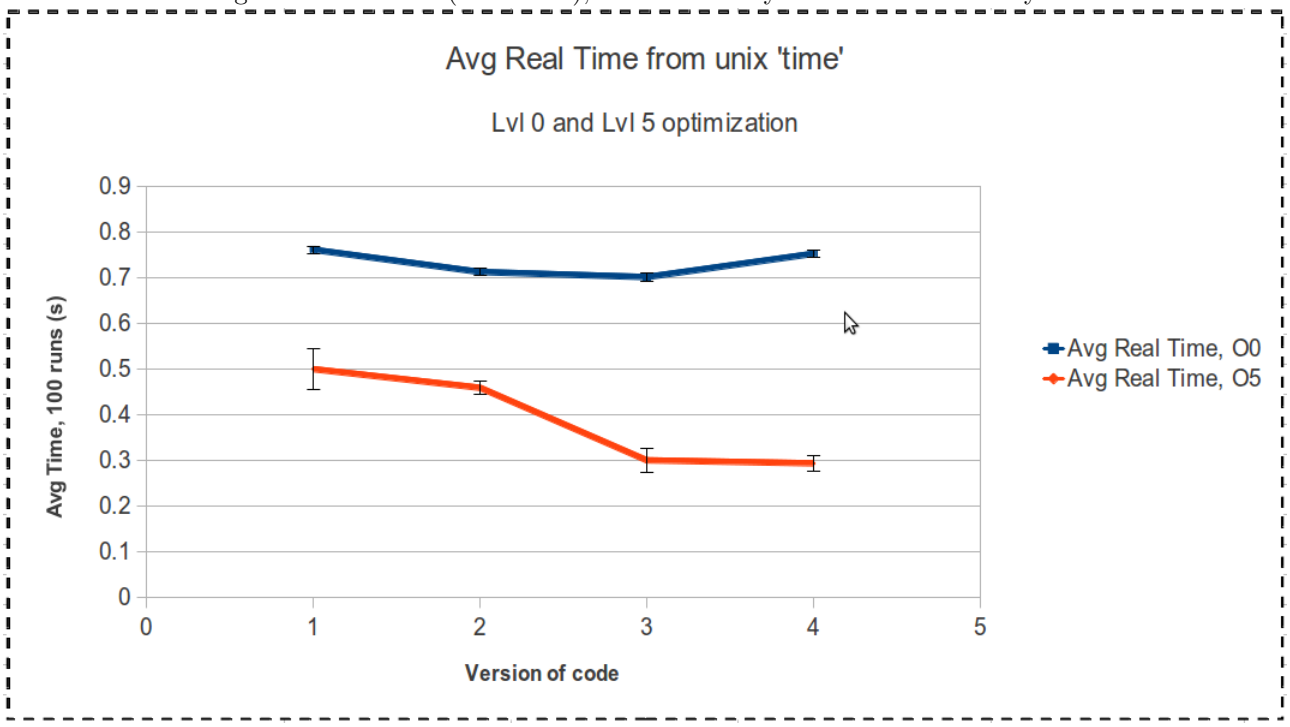
Following is a graph of the 'real time' (total time) measured by the unix `time` command (see Figure 3). As mentioned above, each datapoint represents 100 runs of the game of life; each run is 1024 evolutions starting from the input given in the appendix.

A few things are interesting to note. Each version of the code improves performance, except for version 4 with level 0 optimisation. With level 5 optimisation, there is a very slight improvement. From this graph one could infer that the strategy of allocating all the rows of a board contiguosly in memory doesn't significantly

improve performance, if at all (the error bars of the version 4 datapoint overlap with the error bars of the version 3 datapoint with level 5 optimisation).

It's also interesting that the O5 optimisation data has significantly higher standard deviations. Apparently some optimisations that `gcc` implements cause the performance to be more nondeterministic (the performance varies more from run to run).

Figure 3: Real time (total time), as measured by the unix 'time' utility.



Next is the graph of the system time for both levels of optimisation (Figure 4). Here it appears that the different versions of the code improve performance for level 0 optimisation. Performance is relatively constant for level 5 optimisation. However, this would be an invalid inference. The standard deviations are so high that we can't infer anything about how the different versions of the code affect the system performance.

It's unfortunate that no conclusion can be reached about the effect of the code improvements on system performance. The experiment could be repeated with more runs or a greater amount of work per run to reduce this error.

The last data from the `time` command is given in Figure 5 below. This gives the time spent in user code. This data supports the hypothesis that allocating all memory for the game of life contiguously improves performance by increasing spatial locality. In each successive version, the level 0 optimisation performance gets better. For level 5 optimisation, the difference between version 3 and version 4 of the code is negligible.

One can argue that the level 5 optimisation probably makes an improvement similar to what version 4 of the code accomplishes. Therefore, the difference between version 4 and version 5 with O5 optimisation is small. With level 0 optimisation a significant improvement occurs between version 3 and version 4. In fact, the improvement is greater than the improvement between version 2 and version 3!

Therefore, the system time (or context switching) must account for the lack of improvement seen in the 'real time' data. Since we are really interested in the time it takes to run the user code, it appears the version 4 code improvement is useful after all. Again, the experiment could be repeated with more runs or a larger amount of work per run so that the effect of context switching and system code is less important.

Figure 4: System time, as measured by the unix 'time' utility.

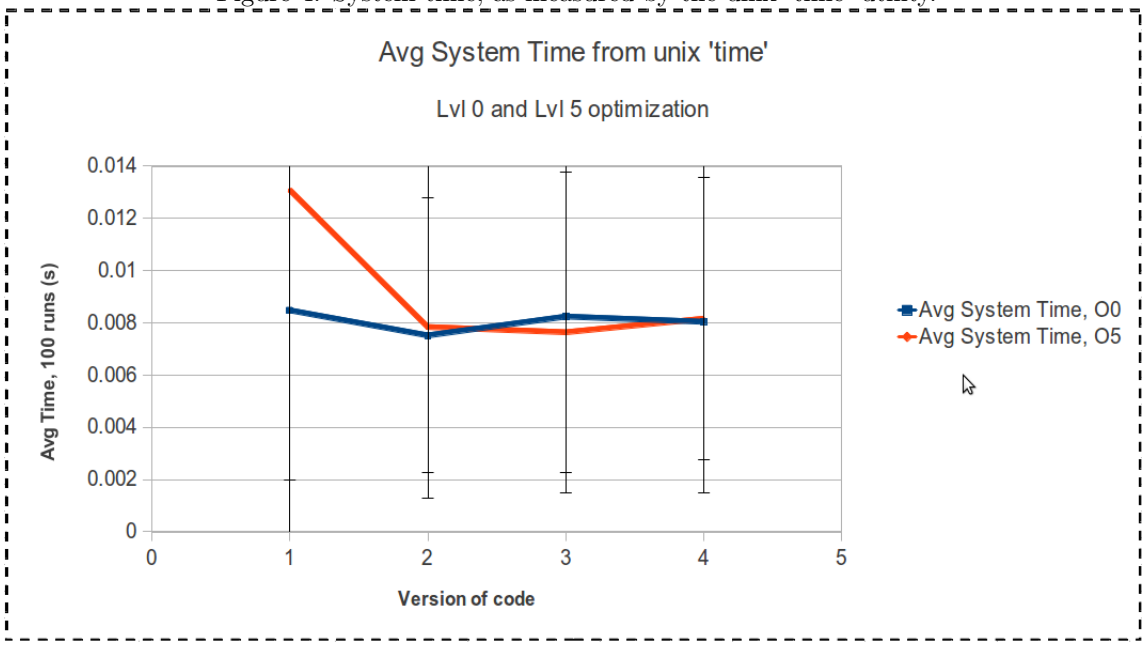
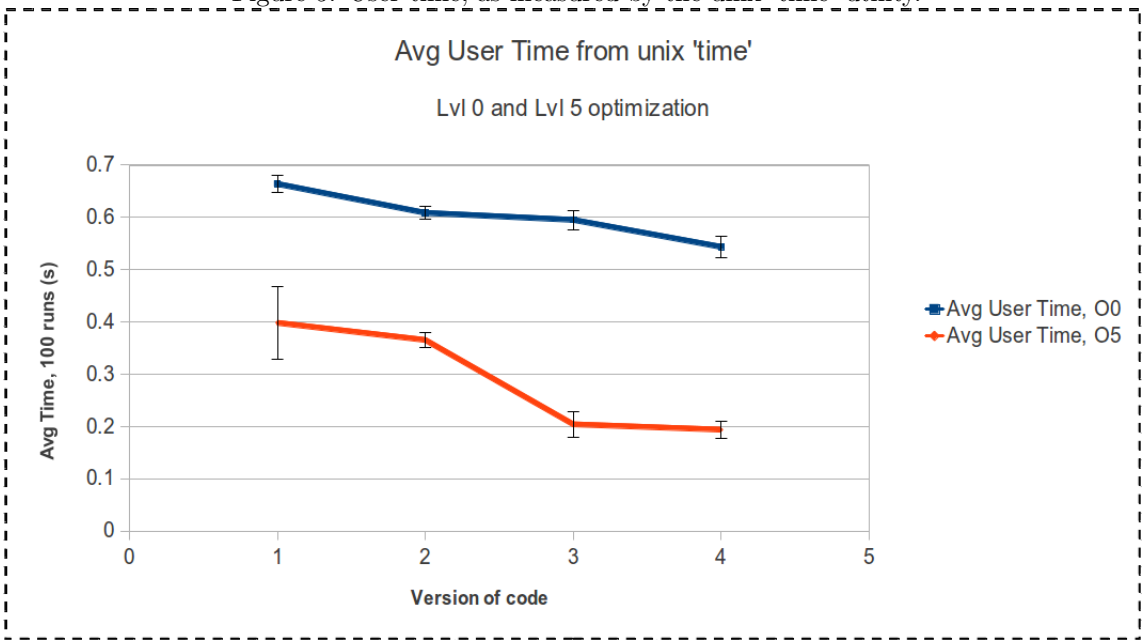


Figure 5: User time, as measured by the unix 'time' utility.



5.2 Profiling Results

The `gprof` results are more difficult to analyse, at least in terms of performance. We could write a script to profile each version of the code many times. This could be done in a more detailed study.

Following is a sample flat-profile result. This is from version 4 of the code with no optimisation:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 92.59      0.50     0.50  3827712     0.13     0.13  neighbors_torus
  7.41      0.54     0.04     1024    39.06   527.34  evolve_torus
  0.00      0.54     0.00     1025     0.00     0.00  new_game
  0.00      0.54     0.00     1024     0.00     0.00  clone_game
  0.00      0.54     0.00     1024     0.00     0.00  free_game
  0.00      0.54     0.00        1     0.00     0.00  game_from_file
  0.00      0.54     0.00        1     0.00     0.00  print_game
```

Without optimisation, the total us/call goes from .14, .15, .12, to .13 in the 4 different versions of the code. The standard deviation is probably quite high, since `gprof` uses sampling. Also, it only gives 2 digits of precision, as opposed to the 3 digits given by `time`. `gprof` is more useful for telling you which functions take up the most time, but it doesn't give precise performance metrics. All of the reuslts make sense: `neighbors_torus` takes up 92.59% of the time, as expected. This is followed by `evolve_torus` at 7.41%, since that fucntion has to check every cell to see if it is alive or dead in the next generation.

Below is the profiling result for level 5 optimisation, version 4 of the code:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
100.00      0.11     0.11     1024   107.42   107.42  evolve_torus
  0.00      0.11     0.00     1024     0.00     0.00  free_game
  0.00      0.11     0.00        1     0.00     0.00  game_from_file
  0.00      0.11     0.00        1     0.00     0.00  new_game
  0.00      0.11     0.00        1     0.00     0.00  print_game
```

Interestingly, the function `neighbors_torus` disappears with version 3 and 4 of the code! It appears that all of the work was done in the `evolve_torus` function. We can infer that the function was inlined as part of an optimisation. We examined the assembly code by using the command `gcc -S main.c game_v4.c`. In the assembly code file `game_v4.s`, the function `evolve_torus` never makes any calls to `neighbors_torus`, demonstrating that it was inlined.

Next we give some results from `gcov`. We see how useful `gcov` is for finding "hotspots" in your code, and also for simply learning facts about how your program behaved during runtime.

Below is the gcov output for the `evolve_torus` function, version 1 of the code with no optimisation. The numbers on the left give the number of times a line of code was executed.

This function does one iteration of the game of life with wrapping like a torus. It's interesting to see how we gained net living cells by the end of the process. Based on the number of calls to the line that turns a living cell into a dead cell, and the number of calls to the line that turns a dead cell into a living cell, we can infer that 53812 living cells died. 53999 cells came back to life. 187 net cells came to life! We used `vim` to count the number of O's before and after the 1024 evolutions. The input started with 46 O's and ended up with 233 O's. 233 - 46 = 187 cells that came to life, verifying the `gcov` results.

```
    1024:   92:game* evolve_torus(game* g) {
       -:   93:
    1024:   94:    game* next = clone_game(g); //Clone game for evolution
    1024:   94-block  0
       -:   95:
       -:   96:    int i, j, neighbor_cnt;
   44032:   97:    for (i = 0; i < g->rows; i++) {
   43008:   97-block  0
   44032:   97-block  1
 3870720:   98:        for (j = 0; j < g->cols; j++) {
   43008:   98-block  0
 3827712:   98-block  1
 3870720:   98-block  2
 3827712:   99:            neighbor_cnt = neighbors_torus(g, i, j);
 3827712:   99-block  0
 3827712:  100:            if (g->board[i][j] == ALIVE) { //ALIVE case
  131135:  101:                if (neighbor_cnt == 2 || neighbor_cnt ==3) {
  131135:  101-block  0
       -:  102:                    ; //Do nothing, stays alive
       -:  103:                }
       -:  104:                else {
   53812:  105:                    next->board[i][j] = DEAD; //Else it dies
   53812:  105-block  0
       -:  106:                }
       -:  107:            }
       -:  108:            else { //DEAD case
 3696577:  109:                if (neighbor_cnt == 3) { //If it has 3 living neighbors, it comes back
 3696577:  109-block  0
   53999:  110:                    next->board[i][j] = ALIVE;
   53999:  110-block  0
       -:  111:                }
       -:  112:            }
       -:  113:        }
       -:  114:    }
       -:  115:
    1024:  116:    return next;
    1024:  116-block  0
       -:  117:}
```

We also examined the `gcov` results for the most performance-critical function, `neighbors_torus`.

Below is a sample of the `gcov` results for the `neighbors_torus` function, with level 5 optimisation. This is the most performance critical function. Branch probabilities are turned on. We can see that the "`if` $(i == r \mid\mid j == c)$" statement fell through 11% of the time. This is exactly what we expect, since 11% is close to 1/9 and the center cell takes up 1/9 of the 3x3 square around a cell. This at least gives a sanity check that these probabilities are valid.

gcov output for loop inside neighbors_torus function:

```
        -:  158:    int i, j, i_mod, j_mod;
 15310848:  159:    for (i = r - 1; i <= r + 1; i++) {
branch  0 taken 75%
branch  1 taken 25% (fallthrough)
 45932544:  160:        for (j = c - 1; j <= c + 1; j++) { //Calculate neighbors with modulo arithmetic
branch  0 taken 75%
branch  1 taken 25% (fallthrough)
 34449408:  161:            if (i == r && j == c) {
branch  0 taken 11% (fallthrough)
branch  1 taken 89%
  3827712:  162:                continue; //Do not count the center cell in calculation
        -:  163:            }
```

Below is some of the same annotated code with gcc optimization turned off completely (-O0). It's interesting that without optimization, `gcov` considers "i==r" and "j==c" as different branches.

The branch "i==r" is taken 33% of the time and "j==c" is taken 33% of the time. This is the result we expect, since the middle row takes up 1/3 of the rows, and the middle column takes up 1/3 of the columns. To get the percentage that this if-statement is evaluated true, you have to take the product of the probabilities, 33%x33% = 11%. This is the same result from above, but *we get more information with optimization turned off.*

```
 34449408:  161:            if (i == r && j == c) {
branch  0 taken 33% (fallthrough)
branch  1 taken 67%
branch  2 taken 33% (fallthrough)
branch  3 taken 67%
```

Also notice from the figure above that the single line called the most is line 160. It's called 45,932,544 times! This is a clearly a hotspot in our code, and any optimisation would greatly improve performance.

6 Conclusion

The GNU suite of compilers and profiling tools provide a powerful set of features for analysing your code's performance. The unix `time` command is also great for accurately measuring the time your program spends in user space, kernel space, and the total time including context-switching.

Using constants in loops, rather than accessing a memory location every time, improves performance noticably. This is true with or without compiler optimisation. This is probably due to memory aliasing, i.e. the compiler can't assume the value in memory is constant even with optimisation, so it has to access it every time. With a declared constant, however, a single value stored in a register can be used.

Using a single call to `malloc` to allocate a contiguous 2-d array gives significant performance improvement with *user time*. In terms of total time, performance appears to be worse without optimisation, and only slightly better with optimisation. We hypothesize this is due to the large standard deviation in system time. With more runs or a more complex program, this technique might prove to be more useful.

`gcov` and `gprof` can be used to find which functions are called the most, and even which line of code is executed the most. These tools are also great for sanity-checking your code, e.g. we found out how many cells came to life without needing to examine the program's output.

It is incredibly important to first use these tools without any optimisation turned on! Optimisation can wash out a lot of information. For example, the `neighbors_torus` function was inlined (as seen by examining the assembly code), so it didn't show up in `gprof`. After optimisation, `gcov` gives less detailed information about branch probabilities.

For a future project, we could explore in many directions. It would be interesting to examine when loop-unrolling is beneficial. The principle is that loop-unrolling breaks a contingency chain so that the CPU can immediately start more operations in parallel. Loop-unrolling also reduces the total number of operations, since the increment operation (e.g. `i++`) is performed once every n iterations (with degree-n loop unrolling) rather than every iteration.

The main disadvantage of loop-unrolling is that it increases the size of the code. This could lead to more cache misses. An interesting paper on the subject by Nan Li and Julian Shun of Carnegie Mellon University was found ([1]). They use machine learning to predict the optimal degree of loop-unrolling.

In conclusion, many aspects of the GNU tools remain to be explored. Other techniques for improving performance–such as loop-unrolling–could be analysed in a future project. An important principle when using any profiling tool is to *not* use any compiler optimisations. When optimisations are turned on, important information about your code's performance is lost.

References

[1] Li, Nan and Shun, Julian. "Exposing Instruction-Level and Task Parallelism in Loops using Supervised Classication" http://www.cs.cmu.edu/~jshun/15745/finalreport_sigplan.pdf

[2] Website, "Dynamically Allocating Multidimensional Arrays"
http://c-faq.com/~scs/cclass/int/sx9b.html

[3] Website, "Matrices"
http://www.lawrence.edu/fast/greggj/CMSC110/matrices/matrices.html

[4] Wikipedia article, "GNU Compiler Collection"
http://en.wikipedia.org/wiki/GNU_Compiler_Collection

[5] Wikipedia article, "Profiling (Computer Programming)"
http://en.wikipedia.org/wiki/Gprof

[6] Website, "gcov–A Test Coverage Program"
http://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[7] Website, "OProfile: Overview"
http://oprofile.sourceforge.net/about/

[8] Website, "Available Profiling Packages"
http://www.movementarian.org/linux-profiling/profilers.html

7 Appendix: Game of Life Input