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 18, 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.

```
(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 author's hope is that this will reduce cache misses by increasing 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 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!

4 Justification for tools used

5 Discussion of techniques used

6 Where are these techniques used

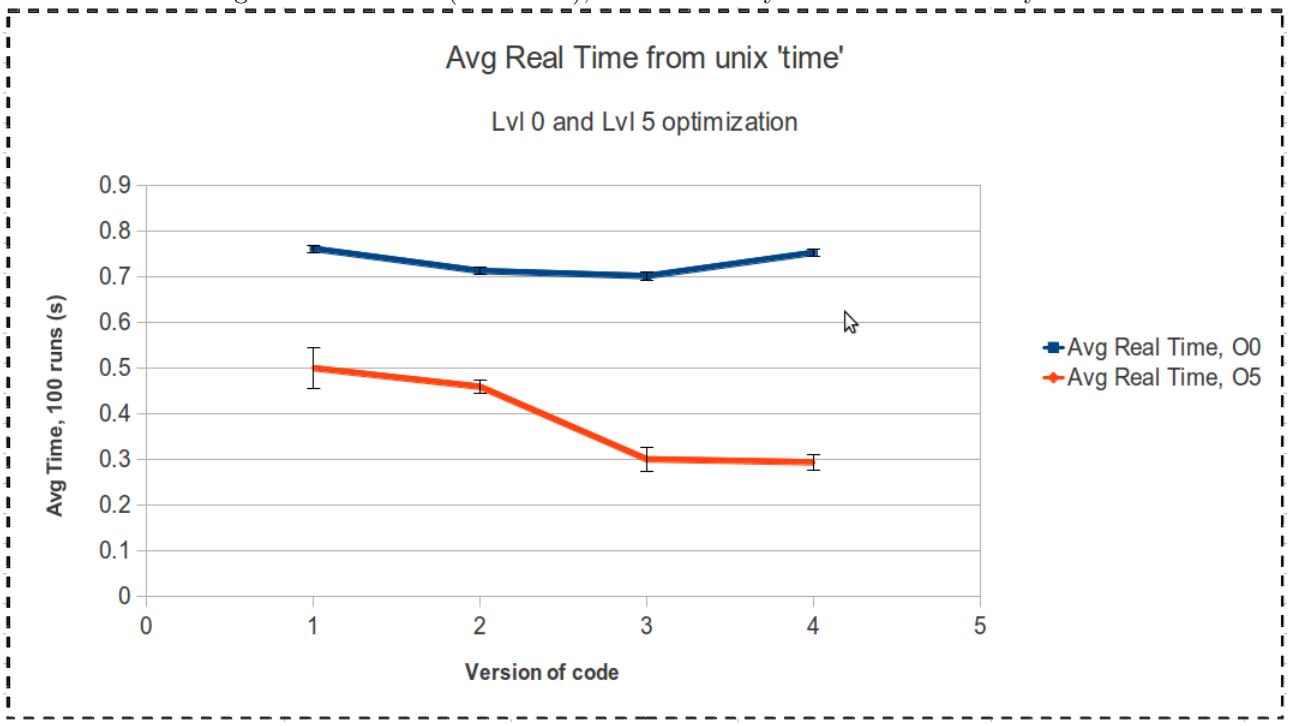
7 Results

Following is a graph of the 'real time' (total time) measured by the unix `time` command (see Figure 1). 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 1: 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 2). 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 3 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 2: System time, as measured by the unix 'time' utility.

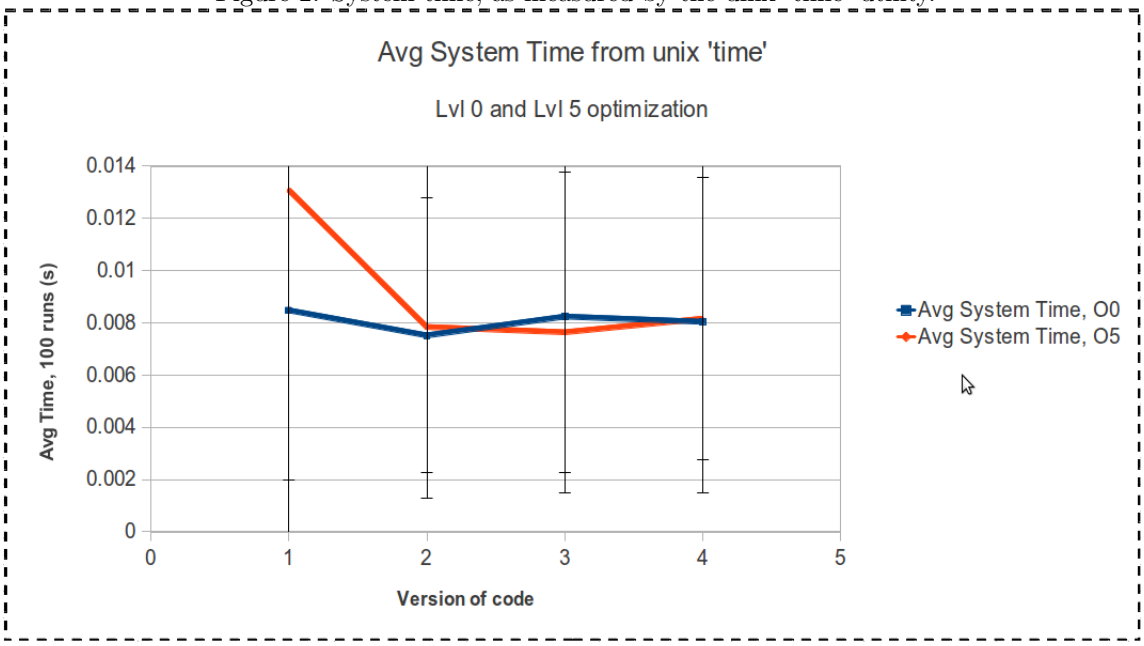
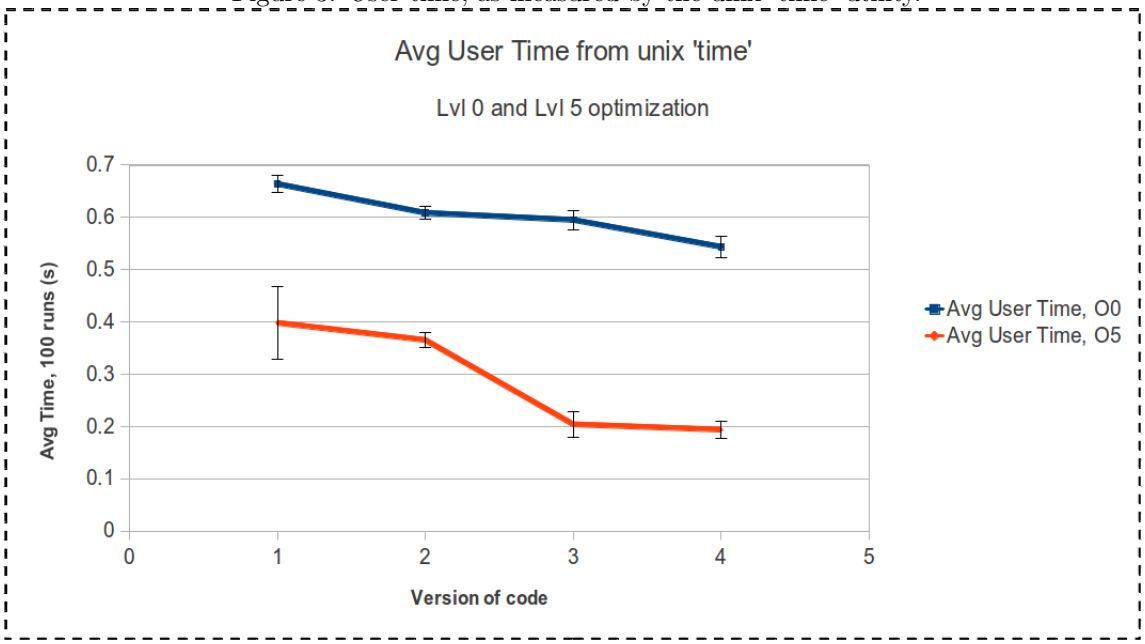


Figure 3: User time, as measured by the unix 'time' utility.



11 Appendix: Game of Life Input