|  |
| --- |
|  |
| CS 1632 – DELIVERABLE 5 |
| Performance Testing Conway’s Game of Life - GameOfLife |

|  |
| --- |
|  |

Github URL: https://github.com/swanc12/SlowLifeGUI.git

Colin Swan – CRS110@pitt.edu

April 5th, 2016

# Summary

For this assignment I started off by doing exploratory testing on the GameOfLife application. Immediately I noticed something off with the “Write” button, as it seemed to take a long time to save the state of the panel. Beyond that there was not anything obviously off about the application’s performance, at least not on the computer I tested on. The panels seemed to turn on and off at a reasonable rate, and not so fast that I could not follow what was going on.

I then used Java Visual VM to profile the application. The biggest hotspot was in the RunContinuous method, though upon digging into the method it turned out that the method itself wasn’t the biggest cause of slowdowns. It turned out that the convertToInt method did a lot of unnecessary work that slowed down the method calling it (getNumNeighbors). The method simply returns its input after a sequence of computationally intensive loops, so I decided to deprecate use of the method all together, as it was only used in getNumNeighbors. This was enough to drastically improve the performance of the getNumNeighbors method, which in turn improved the RunContinuous method further up the stack.

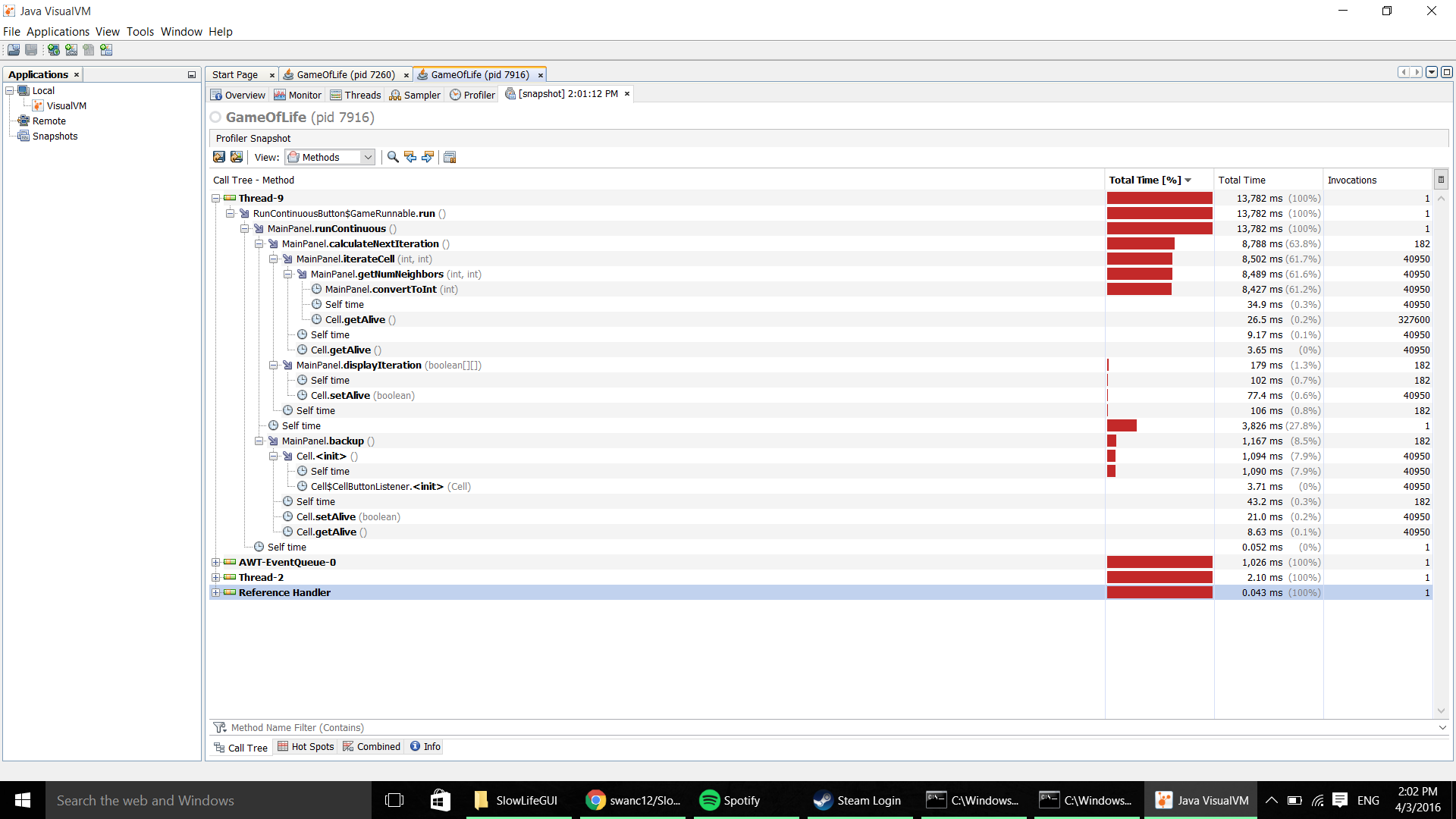
The RunContinuous method also had some superfluous code, containing a for loop that changed the \_r value. The \_r value was then returned to its original value after the for loop, and the new \_r value calculated in the for loop was never used, so the whole loop just served as a resource drain. Because of this I chose to remove the loop altogether.

These improvements provided the most obvious benefits in the profiler, as after that the only hotspot was RunContinuous itself, which was simply a result of it being called so many times. The next place I chose to look was the write method, as I had noticed slowdowns there earlier even though the profiler did not seem to notice. Tracing through the code led me to the toString method for the MainPanel class, which had some redundant code but nothing that seemed to drastically impact performance. The Cell class did contain a resource drain in its toString method however, calling a series of loops that served no purpose. Once that issue was fixed the write method worked seemingly instantaneously. I was only able to come up with two pinning tests for the Cell’s toString method though, so I also created tests for the MainPanel’s toString method, since that was where to cell’s method was primarily called. I also removed the redundant if…else… statement in the MainPanel’s toString method, even though it should not have affected performance much.

Beyond that I made two minor changes: I added a getBackupCells method to the MainPanel object, which I used for testing purposes, and I added a Boolean parameter to the runContinuous method, also for testing purposes. I explored running the runContinuous method in a thread and then calling the “stop” method, but this did not seem to work consistently, so I opted for the simpler parameter approach. The method is only called in one location so this seemed like a reasonable modification to make.

The only other minor hotspot left after the improvements was in the backup method, seemingly caused by the Cell initializations. I did not see anything majorly wrong with the method so I left it be, but one possible improvement would be to save a matrix of Booleans instead of cells, with each Boolean representing an alive or dead cell in the panel. This could be used to set the state of the already existing cells, instead of creating a whole new set of cells.

Performance Initial



Performance after modifications

