# Introduction to Test Code

The test code can concurrently runs 30,000.000 (30 million) GOL instances. This is accomplished by using an algorithm which is described as “staged msg pipelining” to implement concurrency. This is a small subset of the algorithm I mentioned in our original interview and which I believe will inform the clean up of your SBC board running Android. The full “staged message lock-free pipeline” algorithm provides software which is lock-free, massively scalable, fully deterministic with guaranteed latency, massively resource limited and secured through determinism.

Hopefully, this POC of just the massive savings of resources feature will generate enough interest so I can get into the next round of interviews. My study of your product base found several areas where such high performance event driven software is applicable. It is certainly applicable to the work I am being considered for.

The code does not implement several of the features mentioned in the test instructions.

The code is at the following github link.

# ZSpace Action Items

The code is submitted for you to generate a decision on my candidacy. Also, another consideration because this initial delivery does not contain many of the features mentioned in the instructions, you may request to finalize all the features. However, in your decision, please don’t be biased by the many shortcuts I had to take to get the POC in your hands quickly. A list of shortcuts is below.

# Current Status of the Test Code

This GOL C/S code has about 14 hours in it. It will take about 2 more hours to add all the features per the test instructions. It is about 8 objects in more than 1000 loc. The TGOL code was completed in about 2 hours.

The code is a POC for optimum management of 4 core systems or larger. The performance should make it clear it does not use traditional pier thread models, or even pooled thread models. It is a unique paradigm where all flow is controlled through staged messages.

The code is compiled to the V140 Toolset in MSVC 2015

The code requires the Microsoft KM-TEST Loopback driver

The code requires at least a 2 core machine

The configuration x64 Debug is the only one tested.

In order to manipulate the throughput, several defines are listed in xplatform.h.

# Known Problems

Problem 1: The Windows overlapped socket mechanism does not work as advertised. Also, the Winsock loopback system does not strictly adhere to standard Berkeley socket functionality. Clearly the test environment should be VMs, which I did not have access to. In this time frame, work arounds had to be develop. For one example, I had to experiment with normalizing timeouts in the wait on multiple object section. Part of the workaround involved the installation of the Microsoft KM-TEST Loopback Driver. This is available in every Microsoft Windows OS.

Problem 2: The Microsoft tools I am using artificially limit array size both at compile time and load time. Without this limitation I could accommodate more instances by several factors, close to 100M (100 Million) .

Problem 3: An intermittent bug hangs system when the number of instances is greater than 50,000.

# ShortCuts Taken to Save Time

The code contains several shortcuts which are clearly bad practice but were put in for expediency. Without these, the project, which I took the liberty to expand, is quite large. These several shortcuts allowed the work to be done in a reasonable time. I list the shortcuts here so that they are not interpreted as sloppy coding but a conscious decision.

## Concurrent Reads

The ability to simultaneously read (not write) a variable from different cores is assumed. If not, the reads are simply replaced with an assembly language read instruction prefixed with the “lock” qualifier which is the actual “lock” prefix instruction on x86, x64 or the memory barrier/ cache flush instructions on ARM ( dmb , dsb and isb ). This is related to the “board level cache coherent” question we discussed in the initial interview.

## Concurrency Protection

Standard protections from concurrent access to resources is by running both the client and server functions of a particular resource on the same thread with that thread bound by hard processor affinity and set to the same real-time priority. In other words, the scheduler provides critical sectioning.

I also use knowledge of this limited first version to skip many checks.

## Error Checks

Many times I do not check for errors. Also, several function signatures have return codes when no error can take place.

## Memory Allocation

All memory is statically allocated. It should be dynamically allocated which allows placement with respect to NUMA issues.

## Affinity Calculations

I do not take into account the full CPU core architecture. I simply hard code affinity masks to my own test system.

## Const Objects

Many more variables could be const.

## Extended Error

Rarely is there a query for extended error info

## Limited Deallocation

In a few places I do not deallocate system resources.

## Multiple Entries

Many function parameters can many different values, but the current code only deals with a few case.

## No use of \_\_int128

I could have implemented several tricks to generate numbers higher than 30M. For example, I could have used \_\_int128\_t. The current 30M limitation could be increased by several factors except for several limitations imposed by the compiler I am using.

## No ROR, ROL

Some assembly instruction were optimal for this game, specifically x64 ROR and ROL. However, I did not use them.

## Void ptr

Void ptr instead of typing is used in a couple of places.

# Advanced Features

Some aspects of the code are advanced. To insure my work is appreciated, I list them here.

## NUMA aware

The use of hard processor affinity and preexisting memory structures is a good way to take advantage of the dominant hardware architecture.

## Massive Resource Savings

The unique pipeline thread model with staged messaging allows for massive resource savings. In fact, if it were not for a few artificial limitations of the Microsoft 64 bit compiler I am using, I could demonstrate this point more emphatically by increase the number of simultaneous instanced by several factors.

## ROL

I would like to point out that I was able to take the time to identify certain instructions which could significantly help this particular software. For example, I identified that the ROL and ROR ARM/x64 assembly instructions could speed up some functions by a large percentage.

## Backed by Math Model

This code and the more powerful full staged message pipeline algorithm, is entirely backed by a mathematical model. This is important in applying your product to any real world problems. In fact, several important software credentials (DO178) require this theoretical proof of concept.

## No Copies

The code completely minimizes copies.

## Memory Management

Structures to encapsulate all the memory required by the runtime modules are created. This is an important step in the creation of optimal real-time software which does no memory allocations. (In fact, I do no memory allocations in this code.)

Such pre-existing memory structures are a function of speed/consume/produce analysis, but that analysis was skipped. However, the point is that the code is ready to move onto the next step of actual real-time implementation.

### FYI: Speed and feed

My colleague and an examiner of my lock-free staged message pipeline algorithms and advisory consultant , Dr Shin, Founding Director of the Real-Time Computer Laboratory at the University of Michigan in Ann Arbor once lectured on an experiment to determine what aspect of coding slowed productivity the most. It was the requirement to manage your own memory.

## Object Oriented Design with UML

I have included my initial StarUML documents. They contain both sequence diagrams and class hierarchy.

## Coding Style

In terms of programming style, please note that I adhere for a 1 exit point philosophy. This is very helpful in debugging and maintaining code quality in long life software.

## Hierarchical Modules

The class hierarchy is used to assign a level of primitiveness to the software modules. Some modules only provide primitive services and never call higher level functions. This helps minimize circular include files and is helpful in large program creation. It is also helpful in maintaining code quality in long life software.

## Simplicity

This code is as simple as can be considering it simultaneously runs 30,000,000+ (thirty million) game instances. This is accomplished by using a proprietary algorithm addressing a rather new field of computer science called rea-time software. About 20 years ago, there were only 2 universities in the nation which even had a course in real-time software: University of Michigan and MIT. Luckily, I attended both schools.

This area of high performance real-time event driven software is my specialty. Over the years I have created and refined my proprietary algorithms loosely called staged message pipeline processing. These algorithms have been vetted by my colleague Dr. Shin, Founding Director, Real-Time Computing Laboratory, University of Michigan, Ann Arbor.

A very small percentage of programmers might not find this approach simple and intuitive. However, this is the singular optimal approach to the problem. Hence, if the benefits of massive reduction of resource usage, massive scalability, lock-free runtime , full determinism with guaranteed latency and security through determinism are required, this method must be considered.

## More Thorough

Hopefully the above code characteristics prove the thorough nature of my engineering style. Although this might take a little longer, I believe this is a good characteristic for the high level positions I am applying for: Architect, Principle, Etc.

# Availability Going Forward

Because of preexisting obligations, I will not have more time for this process until the afternoon of November 14, and hopefully after November 26.