Tsunami

A visualisation tool for state machines over time

*James A. McCombe*

*Last Updated: May 30th, 2014*

Tsunami transforms this …

TsunamiStartTimeline("MyTimeline", "Test.vcd", (1 \* 1024 \* 1024));

{

uint32\_t i, q;

for (i = 0; i < 32; i ++) {

/\* Basic value dumping \*/

TsunamiSetValue(i, "MyTimeline", "root.basic.value");

/\* Pulse values \*/

if (i & 0x1)

TsunamiPulseValue(1, "MyTimeline", "root.pulse.value");

/\* Formatted-name value dumping \*/

q = 0;

TsunamiSetValue((i \* 0.25), "MyTimeline", "root.format\_test.signal[%i]", q);

q = 1;

TsunamiSetValue((i \* 0.5), "MyTimeline", "root.format\_test.signal[%i]", q);

/\* Advance timeline \*/

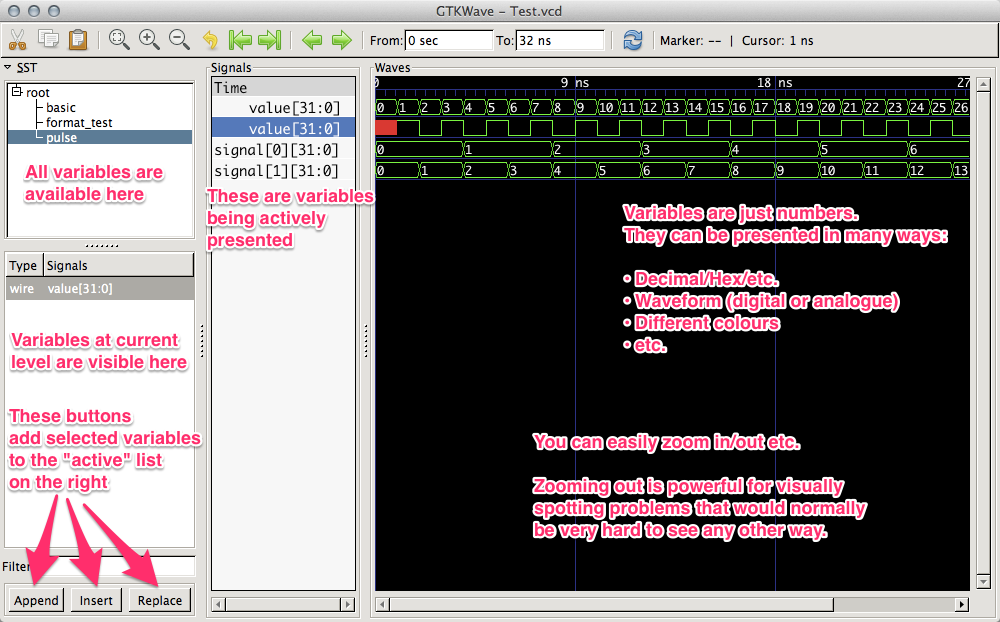
TsunamiAdvanceTimeline("MyTimeline");

}

}

TsunamiFinishTimeline("MyTimeline");

… into this:



Problem Overview

When I write production and/or performance critical software, it typically involves at least the following areas often revisited many times over the course of the effort:

1. Figuring out what I want to build
2. Implementation (writing the code)
3. Compilation (fixing typos, syntax issues, etc.)
4. Basic debugging (ensuring valid pointers, basic logic functionality, etc.)
5. Tuning / optimisation (ensuring intended operation under real workloads and identifying bottlenecks).

**Tsunami is largely intended to assist with Tuning and Optimisation and potentially for debugging certain problems.**

I conceived of Tsunami several years ago when I started doing some hobby hardware architecture work and wanted to write software models to simulate the way a hardware pipeline would behave. I learned the tools that exist for hardware designers and wanted to make these powerful tools available to software people. This approach might not be for everyone and other people may have their own things that work for them already. I’m just sharing what worked for me.

Most software people think about their programs in a serial manner, i.e. instructions execute in sequential order: this is born out of the way todays instruction sets are designed and thus most software languages. The standard development tools and debuggers like gdb, Visual Studio, Xcode, etc. are all designed to facilitate this model. While this is effective for basic debugging and catching hard logical errors in the program, it can be challenging to find subtle behavior bugs or performance related issues with complex state machines that only exhibit problems over longer periods of time. Furthermore, when the program has multiple concurrent threads, this program becomes further magnified.

Here are a few examples of typical problems I’ve run into:

* **Application software** where internal state is changing constantly based on one or more users interacting with the software. If the developer was trying to debug why performance was being lost or other issues, they ideally want a way to dump the state of all relevant internal variables over time.
* **A game engine.** Lots of state changing quickly over time.
* **An audio/video codec.** It may be important to ensure that audio, video and user interaction are all being processed efficiently by the pipeline. In the case where stalls are observed, it would be useful to dump internal state of perhaps many variables to identify the cause of the issue, perhaps an under-flowing queue, etc.
* **Driver development.** It may be useful to dump internal state every time an interrupt occurs or a user→kernel space system call is issued. This could help identify the issues that led up to a failure or performance problem.
* **Any internal state structure that changes frequently.**

Beyond using normal software debugging programs, I had previously employed several schemes to tackle this type of problem, the most common being:

* **“printf()” style debugging**: Just dump out values ad-hoc to the console or a file. Not necessarily actually a printf(), but something conceptually similar for whatever platform is being developed for.
* **Custom built logging/visualisation tools**: This also usually involves some kind of custom viewer that allows values to be observed in relation to other values over time.

While printf() style debugging is a low mental investment to insert into your program, it quickly falls apart when you are trying to track more than a couple of variables at a time since it is hard to “see” the relationships between the values especially when many iterations need to elapse before the problems occur. In addition, for behaviours which occur at a high frequency and are dependent on real time, writing text to the standard output (console) or a file handle can affect the performance and behavior of the program you are trying to debug.

The approach of writing custom logging/visualisation tools can be highly effective, but it is a more serious time investment to develop the tools and even then, its often hard to get the data presented in a meaningful way. Furthermore, it tends to be so application specific that it ends up being hard to adapt to new problems which arise in different parts of the system in the future. It also has the potential to affect the performance of the application being measured unless careful consideration is made to how the data is logged internally.

Having spent many years employing both methods described, I wanted to find a way that had the simplicity of integration and quick turnaround of printf() style debugging combined with sophisticated visualisation, ease of inspection of the captured data and high performance capture.

In a project I worked on several years ago, I had the experience of architecting some hardware and working very closely with experienced hardware engineers in pursuit of an optimal implementation. Due the asynchronous nature of hardware modules, hardware engineers think about the behaviour of their logical state machines over time. As such, their debugging tools are surprisingly visual and are designed to present complex data in a way that makes it easy to see internal logic state over time; multiple state machines can all be seen over different time periods all on screen at once: very dense and valuable information. In hardware, a “timeslice” is usually one cycle of the “clock” driving the chip (i.e. 100Mhz clock means all the various state machines in the chip can change up to 100 million times per second). They use a graphical tool called a “wave viewer” to observe these variables (or signals as they call them) changing over time.

In our software world, we also often have clear “timeslices”; for example, each frame of our game engine, an iteration through the loops in our code, a mutexed critical section or perhaps a user→kernel system call. What we lack are good tools to view them over time.

The purpose of Tsunami is to make it extremely simple for software developers to track internal variables in their code over time and then allow this to be visualised using graphical “wave viewers” which are typically used by hardware engineers.

Tsunami Design Goals

Tsunami was designed with the following goals:

* Simplest possible integration into varied types of software
* Minimalistic API with no exposure of internal object tracking, allocation, etc.
* High internal book keeping performance. Tsunami should have the smallest possible impact on the performance of the application it is measuring.
* Re-use of existing standard hardware data file format. VCD (Value Change Dump) has been around for many years and allows any hardware wave viewer to see the output from Tsunami.

Tsunami Concepts / Glossary

Timeline:

Asshole:

How Tsunami works

Tsunami logs change deltas to variables that you ask it to monitor in your program. In order to make it quick and easy to make variables, you refer to a variable in Tsunami with a c-string name that can optionally represent a hierarchy, i.e. “root.state\_machine\_a.index”. If the name hasn’t been seen before, Tsunami will create a record for it internally and automatically. To make it faster, Tsunami caches a c-string to internal variable lookup using a local static variable is placed in your code directly.

Tsunami’s internal log is a circular buffer containing a fixed data structure representing a value change delta or a jump in the timeline. It effectively run-length encodes variables. As a circular buffer, this means that it effectively maintains a sliding time window of your program state. You get to control the allocation size of the buffer. This works pretty well since you can run your program and quit it at any time and Tsunami will have recorded the most recent sequence of events up to the buffer size you allocated.

Tsunami Problems and limitations

Tsunami is far from perfect. It has a number of limitations that will probably get fixed gradually depending on whether they present a real problem:

1. Tsunami timelines are thread safe on Linux/Mac OS X. I didn’t get around to making them thread safe on Windows threads.
2. Tsunami calls within a single timeline are never thread safe. You need to put access to Tsunami within your critical section for multi-threaded apps. Its pretty light-weight.
3. Tsunami values are 32-bits and no wider.
4. The log entry format in Tsunami is not as compact as it could be. It needs to be at least 32-bits but it currently 96Bytes on 32-bit systems and 128Bytes on 64-bit systems. I’d like to get it to 64bits on all systems.