Quantum as Lockless Programming

By Tomas Arce, July 4 2010

I was first introduced to multicore computer programming about 6 years ago. For an engine programmer it was a long time coming. When they explain how a lockless system worked it was all in terms of the compare-and-swap instruction. This is, of course, how it works at the assembly level, but it made it hard to think about what was going on at the algorithm level. I realize that this was because no one had taught me a way to think about it, rather just how it works. But if I was going to master this dark art of programming I would need an intuitive way of thinking about the problem space.

If you are a geek like me, then you like anything that gives you an understanding on how the universe actually works. This inevitably leads to a basic understanding of some physics, which leads to some understanding of quantum mechanics. Here is where I found my solution to thinking about multicore programming. It was the concept of the wave particle duality. I love this concept because it defines so clearly our universe, even if our humble human brain can only take so much of it. So I created a metaphor that allowed me to explain to others how to think about the problem space as well as have some common language to describe what was going on.

First lets understand our universe a little bit. What the hell is the wave particle duality? Well, most people learn what an electron is in high school. YAY! For public education! What we all learn is that a particle, which looks like a ball, orbits around the nucleus of the atom, which is also made out of balls. This is also a metaphor because electrons are not balls and while you can say that they orbit the nucleus, I would not say that very loudly. The main reason I say this is because it is an oversimplification on really what is going on and therefore misleading. Here is where the wave/particle stuff starts to matter.

When you try to read the position of an electron it turns out that it could, in fact, be anywhere in the universe. WTF? You may say loudly… Do you mean it's in my pocket and in my eye at the same time, God Style? Lets say yes for now. But let's also say that this is because it lives in a special place called the wave world (I like to call this space the quantum world). It is called the wave world because everything in it is made of probability waves. In this world nothing is clearly defined and everything that can happen could be happening in it. Strange universe we live in, right? Well… not so fast, if it was this confusing it would be impossible to make sense out of anything. So the universe also provides a place for our tiny minds to think. This is why when you look at it or measure it, it feels like it makes sense. For instance, when you try to answer the question of where is the electron and you look for it (measure where an electron is), the electron decides that it is in a single place in the

universe and you were lucky enough to measure it. Not always mind you but most times. We call this collapsing the wave of possibilities; I call it your local reality.

So far we have two spaces that we can think where an electron is, the quantum world where it could be anywhere and the local reality which is where you have clearly identified that it is in a particular place. So this is neat and enlightening but how does this help us think in multicore programming? Well you are about to enter the Digital Quantum. Lets start with something simple like a counter.

Let's say we have a global variable and we want to count it up. Remember the counter should go up by one every time it's set. So our program will look like this (don't mind the variable names for now):

```
u32 QuantumVariable=0;

void main( void )
{
    while(1) QuantumVariable += 1;
}
```

No problem so far right? But now Lets say that we have a huge computer with 1 million hardware threads and 10 million software threads with different priorities (just to make sure that time is variant). So our cool counter function is not going to work any more… why?

It has to do with how the assembly looks like. Our function in assembly looks like this.

```
void MyThread( void )
{
    00. infinity:

    01. mov r1, QuantumVariable
    02. add r1, r1, 1
    03. mov QuantumVariable, r1

    04. jump infinity
}
```

Well between 01 and 02 the value of QuantumVariable could have changed, right? Because some other thread may have just set the value of the QuantumVariable. If so it does not make any sense to add 1 because our counter will be wrong. You can also phrase this as our thread reality does not match the quantum reality any more. Furthermore even if we were lucky and the add executed with all the correct values when we try to set the global variable again someone may beat us and the value we will be trying to set will be incorrect (just because the quantum reality and our local reality did not match). Lets rewrite our C++ function so it matches more closely with the assembly function.

```
void MyThread( void )
{
    while(1)
    {
        u32 LocalReality = QuantumVariable;
        LocalReality += 1;
        QuantumVariable = LocalReality;
    }
}
```

But this is where God Intel decided to give us a hand and created the compare-and-exchange (CMPXCHG) atomic introduction. You can think like this instruction takes only one cycle so in a way only one core can be right. This instruction works like the following function:

```
bool cas( u32& QuantumVar, u32 LocalReality, u32 NewValue )
{
    if( QuantumVar == LocalReality )
    {
        QuantumVar = NewValue;
        return true;
    }
    return false;
}
```

The actual code in C++ for the Windows version of the cas32 looks like this:

```
inline bool cas32( u32&  QuantumVar,
                   u32   LocalReality,
                   u32   NewValue)
{
    return bool(_InterlockedCompareExchange(
                    (volatile long *)&QuantumVar,
                                     NewValue,
                                     LocalReality) == LocalReality);
}
```

So now with this super cool function we can solve our problem once and for all because there is a certainty that reality of the quantum variable and our local reality will match. So we can solve our problem like this:

```
void MyThread( void )
{
    while(1)
    {
        u32 LocalReality = QuantumVariable;
        u32 NewValue = LocalReality  +1;
        cas( QuantumVariable, LocalReality, NewValue );
    }
}
```

Note that if the cas fails we don't care… we will try again soon enough thanks to our infinite loop. But at least our counter is going up by one every time. One thing that you may notice is all the wasted computation that happens for all other threads that failed. But still if the goal is to count up as fast as possible this is it in a multicore computer.

Now lets see if the names of the variables make sense…

Lets say that I start running the program and I ask you what is the value of the QuantumVariable inside one of the threads. You will say it is undefined… because it could be changing all the time. Really? So what about if we start the program just a second ago and I ask you if the variable is closer to 1 or 10000000000. Well you may answer that you are still not sure because depends if the thread we are using to measure is a high priority or a low priority. In other words time passes different in different threads (relativity anyone? ☺) If it is a low priority it may not even be running yet. But then in a very smart move you say you can tell me with a certain probability distribution if I really want to know… This to me sounds a lot like trying to know where an electron is. In other words it certainly looks like a quantum world to me.

What about the word *reality*, does it make sense? Well it makes sense because it is a way to name a particular vision of the world in a specific universal time tick. Then this reality becomes a unit. Once we have a unit of something we can then copy it, compare it, modify it, etc. So it is an easy way to express this whole idea in a single word. What about the word local? Well that one is simple enough it is local to our thread (our context), so yes it does make sense.

Finally what about the cas? By using the cas we can be certain that we are dealing with the right set of realities and therefore we can build a consistency in the quantum world.

Now armed with this analogy we can talk freely about whether some variables are in the quantum space or they are in some local space.

The next question is can we have two-quantum variables entangle? Well funny you should ask this question… ☺

In quantum physics, to entangle two particles means that when you collapse the wave function in one of the particles the other should auto collapse instantaneously to have the same but inverse value. Could we do the same thing? Well, let's try it:

```
u32 QuantumParticleA=0;
u32 QuantumParticleB=~QuantumParticleA;

void mythread( void )
{
    while(1)
    {
        // Read both in 1 atomic operation there by
        // automatically entangling the two variables
        // We can say that we have a consistent reality
        // in both of them.
        u64 LocalReality = *((u64*)&QuantumParticleA);
        u32 ParticleA = u32(LocalReality&0xffffffff);
        u32 ParticleB = u32(LocalReality>>32);

        // Make sure that entangle particles have inverse values
        ASSERT( ParticleA == ~ParticleB );

        // Randomize the quantum world a little…
        // Note that we don't need to use cas to set the value
        // since the assembly instruction move64 is atomic and
        // will set both values at the same time, and since we
        // don't care what they are is all good.
        *((u64*)&QuantumParticleA) = ~LocalReality;
    }
}
```

But Digital quantum is not just like our universe since it is not as constrained. The variables could have any value; they don't have to be the inverse of the other. By using this entanglement trick we can do things like stacks and queues, which are very useful. Let's take a look at a stack and see what the code would look like:

```
// we define a quantum pointer like this
struct qtpointer
{
    qtpointer*    pNext;                    // 32bit pointer
    u32           Guard;
}

// head of our stack
qtpointer g_qtHead;

void Push(qtpointer & Node )
{
    while(1)
    {
        // Get our local reality from the quantum world.
        // The compiler should read this as a u64 if it
        // does not we must force it using type casting since it
        // needs to have the two values entangle.
        qtpointer LocalHead = g_qtHead;

        // Set our node value.
        // Note that once we have our reality we never touch
```

```
        // the quantum world again.
        Node.pNext = LocalHead.pNext;
        Node.Guard = 0;

        // Set the new future head
        qtpointer NewHeadValue;
        NewHeadValue.pNext = &Node;
        NewHeadValue.Guard = 1 + LocalHead.Guard;

        // set it back to the quantum world
        // Note that this cas works exactly as the previous one
        // except with 64bit values
        if( cas64((u64*)&g_qtHead,
                *((u64*)&LocalHead),
                *((u64*)&NewHeadValue)) ) break;
    }
}
```

So I guess the first question that you may have is why the need to entangle the pNext and the Guard? Well as you know the quantum world is a world of probabilities and crazy crap to put it nicely. SO… imagine this case… Lets imagine that we did not use the Guard and just the pNext. Then it could happen that right after we read our local reality our thread falls sleep. Someone could pull out the node that our head was pointing and, after a while, someone could put the same old node back in. Now our thread wakes up and we continue as if nothing has happened but in reality the node that used to be in the head has been in and out a few times already. Without our guard counter we never would have known that this was the case. This is known as the ABA problem. You rightly will say so what? There is nothing wrong with a particular node going in and out as long as the head has the right value when we set it. This is true for this case but it won't be true for every case. Sometimes it does matter such as in the case of a queue. But since the queue is harder to explain I am just giving you the heads up.

Note that even with the guard counter there is a very small chance (less or equal than 1 in 2^32 in this case since Guard is 32 bits) that we could be confused and the guard could have rolled back to the exact same number, but not only that the pHead will also have to be the exact same thing. In other words the code is not bulletproof nor fully deterministic. Deterministic behavior ends here in a horrible death. We change our horse to a probabilistic world, which actually it always was but we like to fool ourselves thinking we could write bulletproof code.

There is one more trick that we can do with entanglement. This trick is about providing consistency across two qtpointer variables. Why would you need this? Well this has to do with the queue again… stupid queue! OK let's look at how to push an entry into a lockless queue.

```
qtpointer g_qtHead     = {NULL,0};       // Pointer to the head of the queue
qtpointer g_qtTail     = {&g_qtHead, 0}; // Pointer to the last node in the
                                         // queue which we lie and say is our
// VOLATILE?                             // head.
```

```
void push( qtpointer& Node )
{
    // Prepare our node pointer
    // Note that we don't set the Node Guard to an initial value we simply
    // incremented from the get go. This is the right thing to reset the guard.
    Node.pNext = NULL;
    Node.Guard++;

    while(1)
    {

        // Get a local reality for both variables. Note that each variable is
        // really like two particles that are entangle with pNext and Guard.
        const qtpointer lrTail = g_qtTail;
        const qtpointer lrNext = *lrTail.pNext;

        // If our local tail is already different from the qtTail then
        // there is no point on trying to do anything.
        // Because lrNext will have information that we don't care about.
        // ** By doing this we ensure that at least lrTail and lrNext are in
        // the same reality since qtTail always changes before the node next
        // note this should be a u64 compare.
        if( lrTail != g_qtTail ) continue;

        // Is the qtTail pointing already to last node?
        // If not then we have to move the tail forward.
        if( lrNext.pNext != NULL )
        {
            // Set the new tail pointing and the next node
            qtpointer NewTail;
            NewTail.pNext = lrNext.pNext;
            NewTail.Guard = lrTail.Guard + 1;

           // We don't care if we are successful at this point
            cas64( (u64*)&g_qtTail,
                 *((u64*)&lrTail),
                 *((u64*)&NewTail\));

            // Lets try again and hope next time the qtTail is at the end of the list
            continue;
        }

        // Okay lets try to update the last node to point to our node
        qtpointer NewNext;
        NewNext.pNext = &Node;
        NewNext.Guard = rtNext.Guard + 1;

        // ok lets set the last node next to our node in the quantum world
        if( cas64( (u64*)g_qtTail.pNext,
                 *((u64*)&lrNext),
                 *((u64*)&NewNext)) )
        {
            // Since we are successful at setting our node into the link-list
            // we must now move the tail to point to our node

            // Set the new tail pointing to our node
            qtpointer NewTail;
            NewTail.pNext = &Node;
            NewTail.Guard = lrTail.Guard + 1;

            // We don't care if we are successful at this point
            cas64( (u64*)&g_qtTail,
                 *((u64*)&lrTail),
                 *((u64*)&NewTail\));

            // We are done
            break;
        }
    } // WHILE
}
```

The previous code simply creates a link-list of nodes, where the tail "tries" to point always to the last inserted node. Why do we say it "tries"? Because, again, we must assume that the execution of our function can be stopped at any time as well as some other thread beating us to insert new nodes. Like I said, the quantum world is a crazy world and it is very hard to think about it. By doing a simple queue you can really see how stupid humans really are. But let's not sit and cry in a corner. Instead let's try to understand a few more things. Note that the code is trying to do two things simultaneously, but unrelated. The first job is to link our node into the end of the link-list, which is the queue. The second is to simply update the tail to point to the last node.

The work that it takes to update the tail may be actually distributed across all threads. In other words, any thread that tries to call this function will help move the tail first; if there are others working on it that is fine. This is unlike the case of inserting the node itself where each thread really is competing with each other. Note that we can fail to update the tail to the last node after the function is finished, but it won't matter since it is a collaborative effort. However, inserting our node must have been successful.

Actually, a sort of scary thing is that the "last node" in the queue can be pulled out of the link list before we actually call the cas to link our node. Hell, the user may even free the memory for that last node. So we may be touching (as in reading only) memory that at some point may have been freed. This may be OK as long as the Guard from the node in the quantum world will change to something else that does not match our local guard. This is again a reminder that it is not deterministic programming.

The last note I want to make is that unlike regular programming where you can sort of write one function independently of another, in lockless programming that is impossible. Every function is connected with any other function that deals with the same quantum world variables. In other words, this push function won't work if someone cannot create the right push function. Trust me it is not an easy thing to do. In fact, as good as this code looks like it may not really work once you add the pull function.

It does take a new type of programmer to deal with this craziness. This type of programming is not only difficult to implement and debug, but it is equally difficult to just think about it. We have talked about realities, different spaces and entanglement, all with the hope that we can have a good metaphor to help us think though this chaos. However, it is just a metaphor and definitely is not a 1 to 1 mapping so it has its limitations. Just like an object means different things in different contexts maybe it is a good thing to borrow some of the language from physics and apply them in computer science, after all isn't this what optimization is all about?