Lock-free algorithms: The opportunistic cache

devblogs.microsoft.com/oldnewthing/20110414-00

April 14, 2011

Raymond Chen

Suppose profiling reveals that a specific calculation is responsible for a significant portion of your CPU time, and instrumentation says that most of the time, it's just being asked to calculate the same thing over and over. A simple one-level cache would do the trick here.

```
BOOL IsPrime(int n)
{
 static int nLast = 1;
 static BOOL fLastIsPrime = FALSE;
 // if it's the same value as last time, then
 // use the answer we cached from last time
 if (n == nLast) return fLastIsPrime;
 // calculate and cache the new result
 nLast = n;
 fLastIsPrime = slow_IsPrime(n);
 return fLastIsPrime;
}
```

Of course, this isn't thread-safe, because if one thread is pre-empted inside the call to `slow_IsPrime`, then another thread will see values for `nLast` and `fLastIsPrime` that do not correspond to each other.

One solution would be to put a critical section around this code, but this introduces an artificial bottleneck: If the most recent cached result is `nLast = 5`, `fLastIsPrime = TRUE`, and if two threads both try to see whether 5 is prime, you don't want those two threads to serialize against each other.

Another solution is to use slim reader-writer locks and acquire in shared mode when checking the cache and in exclusive mode when updating the cache. But let's try a lock-free solution.

We're going to combine two different techniques. First, we use the change counter technique we saw last time when we investigated try/commit/(try again), but we also combine it with a lock that is manipulated with a try/commit/abandon pattern.

```
#define IsLocked(l) ((l) & 1)
BOOL IsPrime(int n)
{
 static int nLast = 1;
 static BOOL fLastIsPrime = FALSE;
 static LONG lCounter = 0;
 // see if we can get it from the cache
 LONG lCounterStart = InterlockedReadAcquire(&lCounter, -1);
 if (!IsLocked(lCounterStart) && n == nLast) {
  BOOL fResult = fLastIsPrime;
  if (InterlockedReadRelease(&lCounter, -1) == lCounterStart)
   return fResult;
 }
 // calculate it the slow way
 BOOL fIsPrime = slow_IsPrime(n);
 // update the cache if we can
 lCounterStart = lCounter;
 if (!IsLocked(lCounterStart) &&
     InterlockedCompareExchangeAcquire(&lCounter,
              lCounterStart+1, lCounterStart) == lCounterStart) {
  nLast = n;
  fLastIsPrime = fIsPrime;
  InterlockedCompareExchangeRelease(&lCounter,
              lCounterStart+2, lCounterStart+1);
 }
 return fIsPrime;
}
```

The `lCounter` consists of a LOCK bit as the bottom bit and a change counter in the remaining bits. (Choosing the bottom bit as the LOCK bit makes the operations of clearing the lock and incrementing the counter very simple.)

There are two parts to this function, the part that reads the cache and the part that updates the cache.

To read the cache, we first read the counter with acquire semantics, so that the reads of `nLast` and `fLastIsPrime` will not take place until after we get the counter. If the counter says that the cache is not locked, then we go ahead and fetch the last value and the last result. If the last value in the cache matches the value we're calculating, then we go ahead and use the last result. But as a final check, we make sure that the counter hasn't changed while we were busy looking at the protected variables. If it has, then it means that we may have read inconsistent values and cannot trust the cached result.

If we have a cache miss or we couldn't access the cache, we go ahead and calculate the result the slow way.

Next, we try to update the cache. This time, instead of just looking to see whether the cache is locked, we try to lock it ourselves by setting the bottom bit. (If the lock fails, then we skip the cache update and just return the value we calculated.) Once the lock is taken, we update the protected variables, then atomically release the lock and increment the counter. (This is where putting the lock in the bottom bit comes in handy: You can increment the counter by adding 2 and not worry about a carry out of the counter bits turning into an accidental lock bit.) We use Release semantics so that the values of the protected values are committed to memory before lock bit clears.

Note that in both halves of the function, if the cache is locked, we just proceed as if there were no cache at all. The theory here is that it's better just to say "Oh, the heck with it, I'll just do it myself" than to line up and wait to access the cache. Continuing instead of waiting avoids problems like priority inversion, but it also means that you get some spurious cache misses. Fortunately, since it's just a cache, an occasional spurious miss is not the end of the world.

You could do something similar with the `TryEnterCriticalSection` function provided you're running Windows NT 4.0 or higher:

```
BOOL IsPrime(int n)
{
 static int nLast = 1;
 static BOOL fLastIsPrime = FALSE;
 BOOL fHaveAnswer = FALSE;
 BOOL fIsPrime;
 // see if we can get it from the cache
 if (TryEnterCriticalSection(&g_cs)) {
  if (n == nLast) {
   fHaveAnswer = TRUE;
   fIsPrime = fLastIsPrime;
  }
  LeaveCriticalSection(&g_cs);
 }
 if (fHaveAnswer) return fIsPrime;
 // calculate it the slow way
 fIsPrime = slow_IsPrime(n);
 // update the cache if we can
 if (TryEnterCriticalSection(&g_cs)) {
  nLast = n;
  fLastIsPrime = fIsPrime;
  LeaveCriticalSection(&g_cs);
 }
 return fIsPrime;
}
```

This does have the disadvantage that multiple readers will lock each other out, so we can switch to a slim reader/writer lock provided we're running on Window 7 or higher:

```
BOOL IsPrime(int n)
{
 static int nLast = 1;
 static BOOL fLastIsPrime = FALSE;
 BOOL fHaveAnswer = FALSE;
 BOOL fIsPrime;
 // see if we can get it from the cache
 if (TryAcquireSRWLockShared(&g_lock)) {
  if (n == nLast) {
   fHaveAnswer = TRUE;
   fIsPrime = fLastIsPrime;
  }
  ReleaseSRWLockShared(&g_lock);
 }
 if (fHaveAnswer) return fIsPrime;
 // calculate it the slow way
 fIsPrime = slow_IsPrime(n);
 // update the cache if we can
 if (TryAcquireSRWLockExclusive(&g_lock)) {
  nLast = n;
  fLastIsPrime = fIsPrime;
  LeaveSRWLockExclusive(&g_lock);
 }
 return fIsPrime;
}
```

This still has the problem that readers can lock out a cache update. If the function is hot (and if it weren't, why would you switch to a lock-free algorithm?), and the usage pattern shifts (say, instead of checking whether 13 is prime over and over, it starts checking whether 17 is prime over and over), everybody will be so busy reading the cache to see if the cached value is 17 that nobody will get a chance to update the cache to actually *be* 17!

Exercise: What constraints must be imposed on the protected variables for this technique to be successful?

Raymond Chen

Follow