Why does a single integer assignment statement consume all of my CPU?

devblogs.microsoft.com/oldnewthing/20150116-00

January 16, 2015

Raymond Chen

Here's a C++ class inspired by actual events. (Yes, the certificate on that Web site is broken.) It is somebody's attempt to create a generic value type, similar to `VARIANT`.

```
class Value
{
public:
 Value(Type type) : m_type(V_UNDEFINED) { }

 Type GetType() const { return m_type; }
 void SetType(Type type) { m_type = type; }

 int32_t GetInt32() const
 {
  assert(GetType() == V_INT32);
  return *reinterpret_cast<const int32_t *>(m_data);
 }

 void SetInt32(int32_t value)
 {
  assert(GetType() == V_INT32);
  *reinterpret_cast<int32_t *>(m_data) = value;
 }

 // GetChar, SetChar, GetInt64, SetInt64, etc.

private:
 char m_data[sizeof(int64_t)];
 char m_type;
};

...

Value CalculateTheValue()
{
 int32_t total;
 // ... a bunch of computation ...

 Value result;
 result.SetType(V_INT32);
 result.SetInt32(total);
 return result;
}
```

Profiling showed that over 80% of the time spent by `CalculateTheValue` was inside the `SetInt32` method call, in particular on the line

```
  *reinterpret_cast<int32_t *>(m_data) = value;
```

Why does it take so much time to store an integer to memory, dwarfing the actual computation to calculate that integer?

Alignment.

Observe that the underlying data for the `Value` class is declared as a bunch of `char` s. Since a `char` is just a byte, it has no alignment restrictions. On the other hand, data types like `int32_t` typically do have alignment restrictions. For example, accessing a 32-bit value is usually more efficient if the value is stored in memory starting at a multiple of 4.

How much more efficient depends on the processor and the data type.

Of the processors that allow unaligned memory access, the penalty can be zero, or only 10% or maybe 100%.

Many processor architectures are less forgiving of misaligned data access and raise an alignment exception if you break the rules. When such an exception occurs, the operating system might choose to terminate the application. Or the operating system may choose to emulate the instruction and fix up the misaligned access. The program runs much slower, but at least it still runs. (In Windows, the decision how to respond to the alignment exception depends on whether the process asked for alignment faults to be forgiven. See `SEM_NOALIGNMENTFAULTEXCEPT` .)

It appears that the original program is in the last case: An alignment exception occurred, and the operating system handled it by manually reading the four bytes from `m_data[0]` through `m_data[4]` and assembling them into a 32-bit value, then resuming execution of the original program.

Dispatching the exception, parsing out the faulting instruction, emulating it, then resuming execution. That is all very slow. Probably several thousand instruction cycles. This can easily dwarf the actual computation performed by `CalculateTheValue` .

Okay, but why is the `result` variable unaligned?

Since, as we noted a while back, the way the `Value` class is defined requires only byte alignment, the compiler is not constrained to align it in any particular way. If there were a `int16_t` local variable in the `CalculateTheValue` function, the compiler might choose to arrange its stack frame like this:

- Start at an aligned address *X*.
- Put `int32_t total` at *X*+0 through *X*+3.
- Put `int16_t whatever` at *X*+4 through *X*+5.
- Put `Value result` at *X*+6 through *X*+22.

Since *X* is a multiple of 4, *X*+6 is not a multiple of 4, so the `m_data` member is misaligned and incurs an alignment fault at every access.

What's more, since the `Value` class has an odd number of total bytes, if you create an array of `Value` s, you are guaranteed that three quarters of the elements will be misaligned.

The solution is to fix the declaration of the `Value` class so that the alignment requirements are made visible to the compiler. Instead of jamming all the data into a byte blob, use a discriminated union. That is, after all, what you are trying to emulate in the first place.

```
class Value
{
public:
 Value(Type type) : m_type(V_UNDEFINED) { }

 Type GetType() const { return m_type; }
 void SetType(Type type) { m_type = type; }

 int32_t GetInt32() const
 {
  assert(GetType() == V_INT32);
  return m_data.m_int32;
 }

 void SetInt32(int32_t value)
 {
  assert(GetType() == V_INT32);
  m_data.m_int32 = value;
 }

 // GetChar, SetChar, GetInt64, SetInt64, etc.

private:
 union
 {
  char    m_char;
  int32_t m_int32;
  int64_t m_int64;
  // etc.
 } m_data;
 char m_type;
};
```

Exercise: One guess as to the cause of the problem is that the assignment statement is incurring paging. Explain why this is almost certainly not the reason.

Bonus chatter: I'm ignoring RVO here. If you are smart enough to understand RVO, you should also be smart enough to see that RVO does not affect the underlying analysis. It just shifts the address calculation to the caller.

Raymond Chen

Follow