"Box pruning revisited" – an optimization project by Pierre Terdiman – 2017

Part 4 – sentinels

So we started optimizing the code, first modifying the two inner 'while' loops. That was a random place to start with, just the first thing that came to mind. But it was a good one, because as it turns out we're not done with these two lines yet. Beyond the compiler-related issues, we can also optimize the algorithm itself here.

The two loops now look like this:

```
while(RunningAddress<LastSorted && PosList[*RunningAddress++]<MinLimit);

while(RunningAddress2<LastSorted && PosList[Index1 = *RunningAddress2++]<=MaxLimit)
```

In both cases, the first comparisons ("RunningAddress<LastSorted") are only here to ensure we don't read past the end of the buffers. But they have no actual value for the algorithm itself. It's scaffolding to support the actual "meat" of the algorithm. You would never mention that part in pseudocode to explain what the algorithm does. In other words: we could omit the first comparisons and provided it wouldn't make the code crash, we'd still get the correct results out of the function. This wouldn't be the same for the second comparisons: without them the algorithm would collapse entirely.

Identifying these "useless" bits is a classical part of optimization for me: I remember using this strategy a lot on the Atari ST to identify which instructions I could target first. That's half of the battle. That gives you a goal: how do I get rid of them?

Once that goal is clearly expressed and identified, solutions come to mind more naturally and easily. In our case here, a classical solution is to use "sentinel" values, stored within *PosList*, to ensure that the second comparisons exit the loop when we reach the end of the buffers. This is a fairly standard strategy and I already explained it in the SAP document. So if you are not familiar with it, please take a moment to read about it there (it's in Appendix B, page 24).

The implementation for the box-pruning function is straightforward. We allocate one more entry for the sentinel:

```
// Allocate some temporary data
float* PosList = new float[nb+1];   // MODIFIED: Allocate one more entry for the sentinel
```

Then we fill up that buffer as usual for the first 'nb' entries. But we write one extra value (the sentinel) at the end of the buffer:

```
PosList[nb] = FLT_MAX;  // NEW: Write sentinel value
```

Afterwards it's as easy as it gets, just remove the first comparisons in the inner loops:

```
while(PosList[*RunningAddress++]<MinLimit); // MODIFIED

while(PosList[Index1 = *RunningAddress2++]<=MaxLimit)   // MODIFIED
```

The first comparisons can be removed because we guarantee that we will eventually fetch the sentinel value from *PosList*, and that value will always be greater than *MinLimit* and *MaxLimit*.

Thus this will be enough to exit the loop.

Now, as far as the disassembly is concerned, the changes are a bit hard to follow. The code got re-arranged (check out the position of "call operator delete" in versions 3 and 4 for example) and in its default form the disassembly is a bit obscure. Nonetheless, using the NOPS macro around the while instruction reveals that our change had the desired effect: one of the two comparisons clearly vanished.

```
00A92D98  nop
00A92D99  nop
00A92D9A  nop
00A92D9B  nop
00A92D9C  nop
//CPUID
        while(PosList[Index1 = *RunningAddress2++]<=MaxLimit)   // MODIFIED
00A92D9D  mov         edi,dword ptr [ebx]
00A92D9F  mov         edx,dword ptr [esp+10h]
00A92DA3  comiss      xmm1,dword ptr [edx+edi*4]
00A92DA7  jb          CompleteBoxPruning+24Ah (0A92E6Ah)
00A92DAD  mov         ebx,dword ptr [esp+34h]
00A92DB1  lea         ebp,[nb]
        {
//CPUID
NOPS
00A92DB4  nop
00A92DB5  nop
00A92DB6  nop
00A92DB7  nop
00A92DB8  nop
```

More importantly, the benefits were not just theoretical. We also see the performance increase in practice:

Office PC:

Complete test (brute force): found 11811 intersections in 815456 K-cycles.
78703 K-cycles.
77667 K-cycles.
78142 K-cycles.
86118 K-cycles.
77456 K-cycles.
77916 K-cycles.
77156 K-cycles.
78100 K-cycles.
81364 K-cycles.
77259 K-cycles.
86848 K-cycles.
79394 K-cycles.
81877 K-cycles.
80809 K-cycles.
84473 K-cycles.
81347 K-cycles.
Complete test (box pruning): found 11811 intersections in 77156 K-cycles.

Home PC:

Complete test (brute force): found 11811 intersections in 781900 K-cycles.
82811 K-cycles.
81905 K-cycles.
82147 K-cycles.
81923 K-cycles.
82156 K-cycles.
81878 K-cycles.
82289 K-cycles.
82125 K-cycles.
82474 K-cycles.
82050 K-cycles.
81945 K-cycles.
82257 K-cycles.
81902 K-cycles.
81834 K-cycles.
82587 K-cycles.
82360 K-cycles.
Complete test (box pruning): found 11811 intersections in 81834 K-cycles.

The gains are summarized here:

Home PC	Timings (K-Cycles)	Delta (K-Cycles)	Speedup	Overall X factor
(Version1)	(101662)			
Version2 - base	98822	0	0%	1.0
Version3	93138	~5600	~5%	~1.06
Version4	81834	~11000	~12%	~1.20

Office PC	Timings (K-Cycles)	Delta (K-Cycles)	Speedup	Overall X factor
(Version1)	(96203)			
Version2 - base	92885	0	0%	1.0
Version3	88352	~4500	~5%	~1.05
Version4	77156	~11000	~12%	~1.20

Note that this optimization will only work if the input bounding boxes do not already contain our sentinel value - otherwise there would be no way to distinguish between the sentinel value and a regular value.

This is the first incidence of a rather common trend while optimizing code: the loss of genericity. Sometimes to make the code run faster, you need to accept some compromises or limitations compared to a fully "generic" implementation.

In this case, the limitation enforced by the optimization is that input bounding boxes cannot contain FLT_MAX. This is not a big problem in practice: usually only planes have infinite bounding boxes, and you can always make them use FLT_MAX/2 instead of FLT_MAX if needed. So this limitation is easy to accept. Sometimes they are a lot more debatable.

What we learnt:

Identify “useless” parts of the code that don’t contribute to the results, then eliminate them.

A less generic function can often run faster.

And that’s already it for this part, short and sweet.