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

Part 13 – code alignment

Last time we rewrote the main loop in assembly, and found a way to make it run faster.

Knowing what we know now, can we rewrite the C++ version and get the same performance gains? Can we make it generate what we want? That is what we will explore today.

In the assembly version our main new trick was to use the same offset to index both box arrays, and then reuse that same offset again to lazy-evaluate one of the box indices when an overlap occurs.

In the C++ version this looks like a rather nebulous affair since, well, there was only one index in the first place:

```
while(BoxListX[Index1].mMinX<=MaxLimit)
{
    SIMD_OVERLAP_TEST(BoxListYZ[Index1])
        pairs.Add(RIndex0).Add(Remap[Index1]);

    Index1++;
}
```

It's hard to see how to "improve" this.

But as I often say, you should not focus on what the C++ code looks like. You should see through it, and think in terms of generated assembly. In that respect, that code can be improved.

We know from last time that this single "Index1++" does in fact generate three separate "adds" used for three different things. To avoid this, take a leap of faith: start from the assembly and translate it back to C++ in the most direct way.

These lines from version 12:

```
mov         edx, BoxListYZ              // edx = BoxListYZ
mov         edi, RunningAddress         // edi = Index1
mov         eax, edi                    // eax = Index1
shl         eax, 4                      // eax = Index1 * 16
add         edx, eax                    // edx = &BoxListYZ[Index1]
```

Now become:

```
const char* const CurrentBoxListYZ = (const char*)&BoxListYZ[RunningAddress];
```

These:

```
mov         eax, BoxListX                // eax = BoxListX
lea         esi, dword ptr [eax+edi*8]   // esi = BoxListX + Index1*8 = &BoxListX[Index1] (*8 because sizeof(SIMD_AABB_X)==8)
```

Now become:

```
const char* const CurrentBoxListX = (const char*)&BoxListX[RunningAddress];
```

The single offset:

```
        xor             ecx, ecx
```

Becomes:

```
        udword          Offset = 0;
```

The loop:

```
    comiss      xmm1, xmmword ptr [esi]     // [esi] = BoxListX[Index1].mMinX, compared to MaxLimit
    jb          ExitLoop
    ...
EnterLoop:
    add         ecx, 8
    comiss      xmm1, xmmword ptr [esi+ecx]     // [esi] = BoxListX[Index1].mMinX, compared to MaxLimit
    jae         EnterLoop
```

Is transformed back into a regular while loop, using the single offset:

```
while(*(const float*)(CurrentBoxListX + Offset)<=MaxLimit)
{
    ...
    Offset += 8;
}
```

The SIMD overlap test:

```
    movaps      xmm3, xmm2
    cmpltps     xmm3, xmmword ptr [edx+ecx*2]
    movmskps    eax, xmm3
    cmp         eax, 0Ch
    jne         NoOverlap
    ...
NoOverlap:;
```

Now uses *_mm_cmplt_ps* in the C++ code as well, with swapped arguments, and our single offset again:

```
const float* box = (const float*)(CurrentBoxListYZ + Offset*2);
if(_mm_movemask_ps(_mm_cmplt_ps(b, _mm_load_ps(box)))==12)
```

And finally the part that outputs the pair:

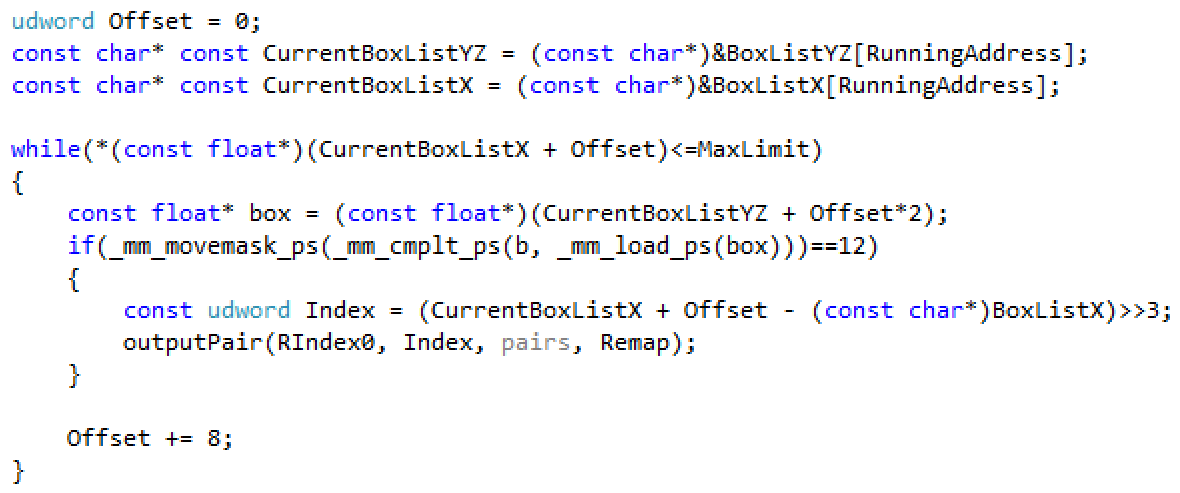
```
movaps      SavedXMM1, xmm1
movaps      SavedXMM2, xmm2
pushad
    // Recompute Index1
    add         ecx, esi
    sub         ecx, BoxListX
    shr         ecx, 3

    push        Remap
    push        pairs
    push        ecx
    push        RIndex0
    call        outputPair;
    add         esp, 16
popad
movaps      xmm1, SavedXMM1
movaps      xmm2, SavedXMM2
```

Is reproduced in C++ this way:

```
const udword Index = (CurrentBoxListX + Offset - (const char*)BoxListX)>>3;
outputPair(RIndex0, Index, pairs, Remap);
```

...with "*outputPair*" the same non-inlined function as in the assembly code.

The resulting C++ code is thus:



```
udword Offset = 0;
const char* const CurrentBoxListYZ = (const char*)&BoxListYZ[RunningAddress];
const char* const CurrentBoxListX = (const char*)&BoxListX[RunningAddress];

while(*(const float*)(CurrentBoxListX + Offset)<=MaxLimit)
{
    const float* box = (const float*)(CurrentBoxListYZ + Offset*2);
    if(_mm_movemask_ps(_mm_cmplt_ps(b, _mm_load_ps(box)))==12)
    {
        const udword Index = (CurrentBoxListX + Offset - (const char*)BoxListX)>>3;
        outputPair(RIndex0, Index, pairs, Remap);
    }

    Offset += 8;
}
```

It looks a bit strange, and I suppose some would say it's not "safe", but….

Does it make any difference?

Home PC:

```
Complete test (brute force): found 11811 intersections in 781701 K-cycles.
 14495 K-cycles.
 13701 K-cycles.
 13611 K-cycles.
 13597 K-cycles.
 13611 K-cycles.
 13600 K-cycles.
 13597 K-cycles.
 14091 K-cycles.
 13718 K-cycles.
 13599 K-cycles.
 13728 K-cycles.
 13635 K-cycles.
 13596 K-cycles.
 13598 K-cycles.
 13638 K-cycles.
 13628 K-cycles.
Complete test (box pruning): found 11715 intersections in 13596 K-cycles.
```

Office PC:

```
Complete test (brute force): found 11811 intersections in 807508 K-cycles.
 13324 K-cycles.
 12650 K-cycles.
 14015 K-cycles.
 12647 K-cycles.
 12646 K-cycles.
 13743 K-cycles.
 12648 K-cycles.
 12647 K-cycles.
 13057 K-cycles.
 13643 K-cycles.
 12684 K-cycles.
 12648 K-cycles.
 13095 K-cycles.
 13330 K-cycles.
 12645 K-cycles.
 12648 K-cycles.
Complete test (box pruning): found 11715 intersections in 12645 K-cycles.
```

Not really.

Let's recap where we stand now:

On the home PC we see that version 13 is perhaps a tiny bit better than version 11 now, but not as good as the assembly version:

C++ (version 11, unsafe)	14372
Assembly (version 12, unsafe)	11731
C++ (version 13, unsafe)	13596

But on the office PC, it's a wash:

C++ (version 11, unsafe)	12570
Assembly (version 12, unsafe)	10014
C++ (version 13, unsafe)	12645

Was it all for nothing then?

Wait, wait. Don’t forget to check the disassembly:

```
00C93020  cmpltps     xmm1,xmmword ptr [ecx+esi*2]
00C93025  movmskps    eax,xmm1
00C93028  cmp         eax,0Ch
00C9302B  jne         CompleteBoxPruning+292h (0C93052h)

00C9302D  push        dword ptr [esp+20h]
00C93031  mov         eax,edi
00C93033  push        dword ptr [pairs]
00C93036  sub         eax,edx
00C93038  add         eax,esi
00C9303A  sar         eax,3
00C9303D  push        eax
00C9303E  push        dword ptr [esp+3Ch]
00C93042  call        outputPair (0C92580h)
00C93047  mov         edx,dword ptr [esp+24h]
00C9304B  mov         ecx,dword ptr [esp+44h]
00C9304F  add         esp,10h

00C93052  movss       xmm0,dword ptr [esp+38h]
00C93058  movaps      xmm1,xmmword ptr [esp+40h]
00C9305D  add         esi,8
00C93060  comiss      xmm0,dword ptr [edi+esi]
00C93064  jae         CompleteBoxPruning+260h (0C93020h)
```

Well at least it does look quite nicer and tighter than before.

The three blocks have been color-coded like in previous versions. The main codepath is only **9** instructions (against **13** in version 11 and **8** in version 12).

There is a single add (00C9305D) in the third block, instead of three before: *that's what we wanted*. In that respect, the strategy worked: we did trick the compiler into generating a single add!

But unfortunately both *xmm0* and *xmm1* are constantly reloaded from the stack in the third block, and that is probably what kills it. Ignore these two instructions, and the code is pretty much what we wanted.

The assembly version (our target code) was like this:

```
movaps          xmm3, xmm2
cmpltps         xmm3, xmmword ptr [edx+ecx*2]
movmskps        eax, xmm3
```

And if we move one line from the third block we see that we just got:

```
00C93058  movaps      xmm1,xmmword ptr [esp+40h]
00C93020  cmpltps     xmm1,xmmword ptr [ecx+esi*2]
00C93025  movmskps    eax,xmm1
```

So close. But no cigar.

Now what? How do you tell the compiler to stop spilling *xmm* registers to the stack?

Well: I don't know. There is probably no way.

But you can try the same strategy as before: when you're stuck, try random changes. Mutate the code. In particular, you can replace this:

```
if(_mm_movemask_ps(_mm_cmplt_ps(b, _mm_load_ps(box)))==12)
```

With that:

```
if(_mm_movemask_ps(_mm_cmpnle_ps(_mm_load_ps(box), b))==12)
```

It looks like a rather innocent change, but amazingly, it generates the following assembly:

```
00AE2FA0  movaps     xmm0,xmmword ptr [ecx+esi*2]
00AE2FA4  cmpnleps   xmm0,xmm1
00AE2FA8  movmskps   eax,xmm0
00AE2FAB  cmp        eax,0Ch
00AE2FAE  jne        CompleteBoxPruning+29Ah (0AE2FDAh)

00AE2FB0  push       dword ptr [esp+20h]
00AE2FB4  mov        eax,edi
00AE2FB6  push       dword ptr [pairs]
00AE2FB9  sub        eax,edx
00AE2FBB  add        eax,esi
00AE2FBD  sar        eax,3
00AE2FC0  push       eax
00AE2FC1  push       dword ptr [esp+3Ch]
00AE2FC5  call       outputPair (0AE2500h)
00AE2FCA  movaps     xmm1,xmmword ptr [esp+50h]
00AE2FCF  mov        edx,dword ptr [esp+24h]
00AE2FD3  mov        ecx,dword ptr [esp+44h]
00AE2FD7  add        esp,10h

00AE2FDA  movss      xmm0,dword ptr [esp+38h]
00AE2FE0  add        esi,8
00AE2FE3  comiss     xmm0,dword ptr [edi+esi]
00AE2FE7  jae        CompleteBoxPruning+260h (0AE2FA0h)
```

One of the loads from the stack vanished: it moved from 00C93058 in the previous version to 00AE2FCA now, i.e. it moved from the frequent codepath to the infrequent one. The arguments swap allowed the compiler to do so, because *xmm1* is not destroyed by the comparison anymore. The resulting code is now almost exactly the same as our assembly version: the only difference is that *xmm0* is reloaded from the stack in the third block (00AE2FDA), while it should instead be kept in a constant *xmm* register.

This doesn't look like much, but this innocent change produces a measurable performance gain compared to the version just before. Our revisited C++ version, while still not as neat as the assembly version, is now clearly faster than what we got in version 11:

Home PC (unsafe version):

```
Complete test (brute force): found 11811 intersections in 782053 K-cycles.
 12972 K-cycles.
 12162 K-cycles.
 12152 K-cycles.
 12152 K-cycles.
 12422 K-cycles.
 12207 K-cycles.
 12154 K-cycles.
 12151 K-cycles.
 12174 K-cycles.
 12153 K-cycles.
 12148 K-cycles.
 12398 K-cycles.
 12184 K-cycles.
 12152 K-cycles.
 12150 K-cycles.
 13039 K-cycles.
Complete test (box pruning): found 11725 intersections in 12148 K-cycles.
```

Office PC (unsafe version):

```
Complete test (brute force): found 11811 intersections in 815031 K-cycles.
 11409 K-cycles.
 10029 K-cycles.
 11243 K-cycles.
 10726 K-cycles.
 10055 K-cycles.
 10784 K-cycles.
 10588 K-cycles.
 10548 K-cycles.
 10290 K-cycles.
 10029 K-cycles.
 10408 K-cycles.
 10464 K-cycles.
 10030 K-cycles.
 10475 K-cycles.
 10028 K-cycles.
 10028 K-cycles.
Complete test (box pruning): found 11725 intersections in 10028 K-cycles.
```

The gains are summarized here:

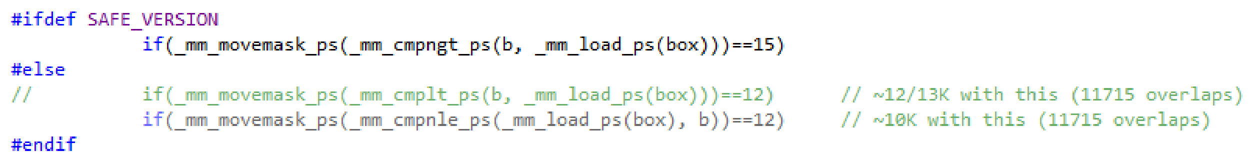
Home PC	Timings (K-Cycles)	Delta (K-Cycles)	Speedup	Overall X factor
(Version1)	(101662)			
Version2 - base	98822	0	0%	1.0
Version3	93138	~5600	~5%	~1.06
Version4	81834	~11000	~12%	~1.20
Version5	78140	~3600	~4%	~1.26
Version6a	60579	~17000	~22%	~1.63
Version6b	41605	~18000	~31%	~2.37
(Version7)	(40906)	-	-	-
(Version8)	(31383)	(~10000)	(~24%)	(~3.14)
Version9a	34486	~7100	~17%	~2.86
Version9b - unsafe	32477	~2000	~5%	~3.04
Version9b - safe	32565	~1900	~5%	~3.03
Version9c - unsafe	16223	~16000	~50%	~6.09
Version9c - safe	14802	~17000	~54%	~6.67
(Version10)	(16667)	-	-	-
Version11 - unsafe	14372	~1800	~11%	~6.87
Version11 - safe	14512	~200	~2%	~6.80
(Version12 - 1st)	(14309)	-	-	~6.90
(Version12 – 2nd)	(11731)	(~2600)	(~18%)	(~8.42)
Version13 - unsafe	12148	~2200	~15%	~8.13

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
Version5	73778	~3300	~4%	~1.25
Version6a	58451	~15000	~20%	~1.58
Version6b	45634	~12000	~21%	~2.03
(Version7)	(43987)	-	-	-
(Version8)	(29083)	(~16000)	(~36%)	(~3.19)
Version9a	31864	~13000	~30%	~2.91
Version9b - unsafe	15097	~16000	~52%	~6.15
Version9b - safe	15116	~16000	~52%	~6.14
Version9c - unsafe	12707	~2300	~15%	~7.30
Version9c - safe	12562	~2500	~16%	~7.39
(Version10)	(15648)	-	-	-
Version11 - unsafe	12570	~100	~1%	~7.38
Version11 - safe	12611	-	-	~7.36
(Version12 – 1st)	12917	-	-	~7.19
(Version12 – 2nd)	(10014)	(~2500)	(~20%)	(~9.27)
Version13 - unsafe	10028	~2500	~20%	~9.26

The gains are computed between version 13 and version 11. Version 12 is now ignored: we can do pretty much the same without the need for assembly.

But doing the assembly version allowed us to find this hidden optimization. And I had to present the assembly version first, to explain where this was coming from. If I would have switched from version 11 to version 13 directly, it would have looked like black magic.

Now, we only got an “unsafe” version 13, since it just got converted from version 12, for which we only had an “unsafe” version. Fine. So we complete the C++ code as we did before e.g. in version 11, for the safe version. The only change in the inner loop is this:



```
#ifdef SAFE_VERSION
            if(_mm_movemask_ps(_mm_cmpngt_ps(b, _mm_load_ps(box)))==15)
#else
//          if(_mm_movemask_ps(_mm_cmplt_ps(b, _mm_load_ps(box)))==12)      // ~12/13K with this (11715 overlaps)
            if(_mm_movemask_ps(_mm_cmpnle_ps(_mm_load_ps(box), b))==12)     // ~10K with this (11715 overlaps)
#endif
```

This generates the proper number of pairs, and the disassembly is exactly the same as for the unsafe version, except *cmpnleps* is replaced with *cmpnltps*. But it is otherwise the exact same assembly, same instructions, no extra spilling to the stack, all the same.

And yet…

Office PC (safe version):

```
Complete test (brute force): found 11811 intersections in 805609 K-cycles.
 12524 K-cycles.
 11874 K-cycles.
 11726 K-cycles.
 11726 K-cycles.
 11941 K-cycles.
 11884 K-cycles.
 12073 K-cycles.
 11725 K-cycles.
 11756 K-cycles.
 12752 K-cycles.
 12267 K-cycles.
 12534 K-cycles.
 12274 K-cycles.
 12588 K-cycles.
 11726 K-cycles.
 12184 K-cycles.
Complete test (box pruning): found 11811 intersections in 11725 K-cycles.
```

Madness. *Why is this one slower*?

I came back and forth between the assembly and the C++, looking for something I could have overlooked. And then I saw it.

The start of the most inner loop was not aligned on a 16-bytes boundary.

That…. was weird. I could have sworn that the compiler was always aligning the loops. I mean that's where all these countless *nops* and *lea esp,[esp]* instructions come from. I see them all the time in the disassembly. That's what you're supposed to do, right? I had this rule in mind but was suddenly unsure where it was coming from. A quick bit of googling revealed it was still a perfectly valid rule, see for example the Intel optimization manual, chapter 3.4.1.5, it says right there:

All branch targets should be 16-byte aligned.

But the compiler apparently does not do that. Or at least: not always. I have no idea why. But that was my new theory anyway: the safe version is slower because the loop is not aligned.

Unfortunately I don't know how to tell the compiler to align a loop in C++. So just for testing, I added back the same line as in the assembly version before the loop:

```
            _asm   align  16
```

"You'll never believe what happened next!"

Office PC (safe version):

```
Complete test (brute force): found 11811 intersections in 846230 K-cycles.
 11174 K-cycles.
 10626 K-cycles.
 10375 K-cycles.
 10549 K-cycles.
 10053 K-cycles.
 10340 K-cycles.
 10727 K-cycles.
 10709 K-cycles.
 10536 K-cycles.
 10643 K-cycles.
 10578 K-cycles.
 10379 K-cycles.
 11036 K-cycles.
 10793 K-cycles.
 10462 K-cycles.
 10993 K-cycles.
Complete test (box pruning): found 11811 intersections in 10053 K-cycles.
```

At that point I was starting to sweat.

Because provided the analysis is correct and this is not just because of something entirely different that I cannot see, then:

- I have no idea how to fix this without using at least one line of assembly.
- I have no idea if this random alignment affected the previous versions, whose timings may all be questionable now.

In fact, we had a bit of an unsolved mystery at the end of version 9c, where for some reason the safe & unsafe versions at home showed a measurable performance difference. Rings a bell? That might have been because of this. The safe version adds two instructions in the code flow before the loop starts, so that's enough to change its alignment…

With the enforced loop alignment, both safe and unsafe versions show the same timings both at home and in the office. Having to use one line of assembly is a bit unfortunate since I was trying to convert back everything to something more portable. That being said, this issue is obviously very compiler-dependent, so maybe that extra line doesn't need porting – other compilers on other platforms might do the right thing immediately. Who knows?

Not a rhetorical question: seriously, who knows about that stuff? Email me and tell me.

In any case, after adding that line the timings become:

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
Version5	78140	~3600	~4%	~1.26
Version6a	60579	~17000	~22%	~1.63
Version6b	41605	~18000	~31%	~2.37
(Version7)	(40906)	-	-	-
(Version8)	(31383)	(~10000)	(~24%)	(~3.14)
Version9a	34486	~7100	~17%	~2.86
Version9b - unsafe	32477	~2000	~5%	~3.04
Version9b - safe	32565	~1900	~5%	~3.03
Version9c - unsafe	16223	~16000	~50%	~6.09
Version9c - safe	14802	~17000	~54%	~6.67
(Version10)	(16667)	-	-	-
Version11 - unsafe	14372	~1800	~11%	~6.87
Version11 - safe	14512	~200	~2%	~6.80
(Version12 - 1st)	(14309)	-	-	~6.90
(Version12 – 2nd)	(11731)	(~2600)	(~18%)	(~8.42)
Version13 - unsafe	12296	~2000	~14%	~8.03
Version13 - safe	12236	~2200	~15%	~8.07

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
Version5	73778	~3300	~4%	~1.25
Version6a	58451	~15000	~20%	~1.58
Version6b	45634	~12000	~21%	~2.03
(Version7)	(43987)	-	-	-
(Version8)	(29083)	(~16000)	(~36%)	(~3.19)
Version9a	31864	~13000	~30%	~2.91
Version9b - unsafe	15097	~16000	~52%	~6.15
Version9b - safe	15116	~16000	~52%	~6.14
Version9c - unsafe	12707	~2300	~15%	~7.30
Version9c - safe	12562	~2500	~16%	~7.39
(Version10)	(15648)	-	-	-
Version11 - unsafe	12570	~100	~1%	~7.38
Version11 - safe	12611	-	-	~7.36
(Version12 – 1st)	12917	-	-	~7.19
(Version12 – 2nd)	(10014)	(~2500)	(~20%)	(~9.27)
Version13 - unsafe	10168	~2400	~19%	~9.13
Version13 - safe	10053	~2500	~20%	~9.23

I will freely admit that for this post, I have no big confidence in my analysis. Forcing the compiler to align the loop *seems* to fix the issue and provide consistent results.

If nothing else, this whole post shows that one can sometimes *tame the compiler* and make it generate something closer to the ideal. Even though it’s difficult. And fragile. And not easily maintainable. And results will vary from compiler to compiler. Yeah, ok, fine: it’s as much luck as engineering at this point. All of this is true.

Still, converting the assembly back to C++ remains a fascinating experiment, and the results are there. You may not like the way the C++ code looks now, but this version is almost as fast as the hand-written assembly version.

Pheew.

Optimization: that's a 24/7 job. Just look at how much time we can spend on one simple function…

What we learnt:

The assembly version showed us the way. While technically it still “wins”, the compiler-generated version stroke back and came very close in terms of performance. The generated code is almost the same as our hand-written assembly.

We reached a point where a single extra *movaps* in the loop has a measurable cost.

The compiler does not seem to always align loops, so some of them are effectively randomly aligned. Any extra instruction before the loop in the code flow can change the alignment and thus affect performance.

Thus there is no such thing as an "innocent change".

Are we there yet, Papa Smurf?

No, I’m afraid we are not.