OPEN CASCADE 学习笔记
——改进 EXTREMA

著: **Roman Lygin**

译：**George Feng**

这是一篇关于开源三维建模软件 OPEN CASCADE 内核的博文：ROMAN LYGIN 是 OPEN CASCADE 的前程序开发员和项目经理，曾经写过许多关于该开源软件开发包的深入文章，可以在他的博客(HTTP://OPENCASCADE.BLOGSPOT.COM)上面找到这些文章。

序

在 **Open Cascade** 的论坛上知道了 **Roman Lygin** 在他的博客上写了 **Open Cascade notes** 系列文章，考虑到 **Open Cascade** 的学习资料并不多，于是从他的博客上下载了其中绝大部分文章，将其翻译过来以方便大家学习交流。如果大家发现文中翻译有错误或不足之处，望不吝赐教，可以发到我的邮箱 **fenghongkui@sina.com.cn**，十分感谢。

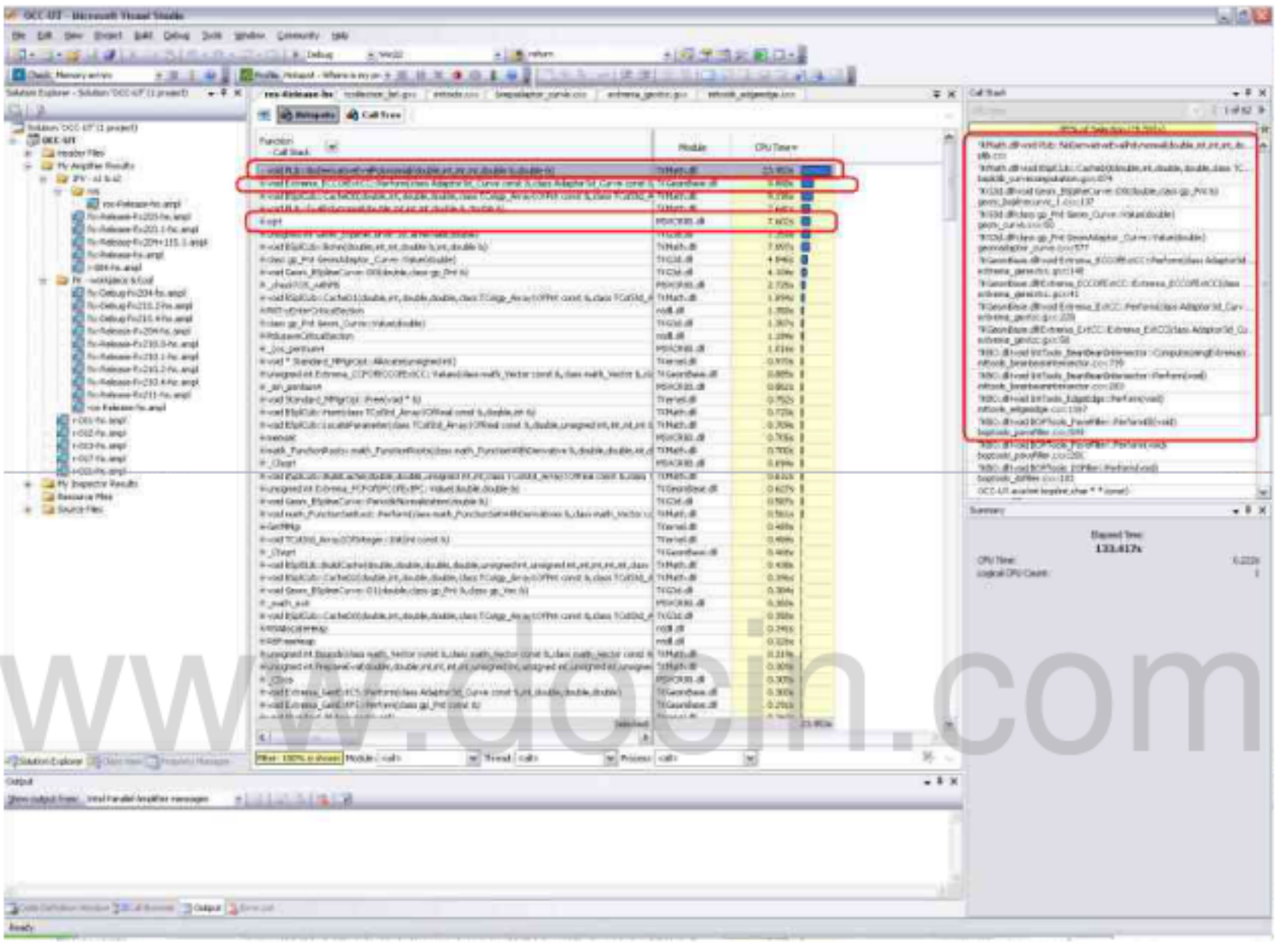
2012 年 **11** 月 **22** 日星期四

第 1 节　Extrema 的性能问题

一直看我博客的朋友可能还会记得提高布尔运算速度这篇文章(参考 Why are Boolean Operations so slooo...ooow?)，在这篇文章中我说过 Pawel K 提供了一个模型文件，之前的方法对这个模型文件的处理速度几乎没有任何提升。所以我决心解决掉这个问题，并花费更多的时间在这方面。非常感谢 Pawl 提供的这个模型。这个模型中的曲线显得非常特别(例如，小的椭圆弧段，具有 300 多个 poles 和 40 个 knots 的长 B 样条曲线)，通过检测这个模型我发现了一个广泛存在的问题。

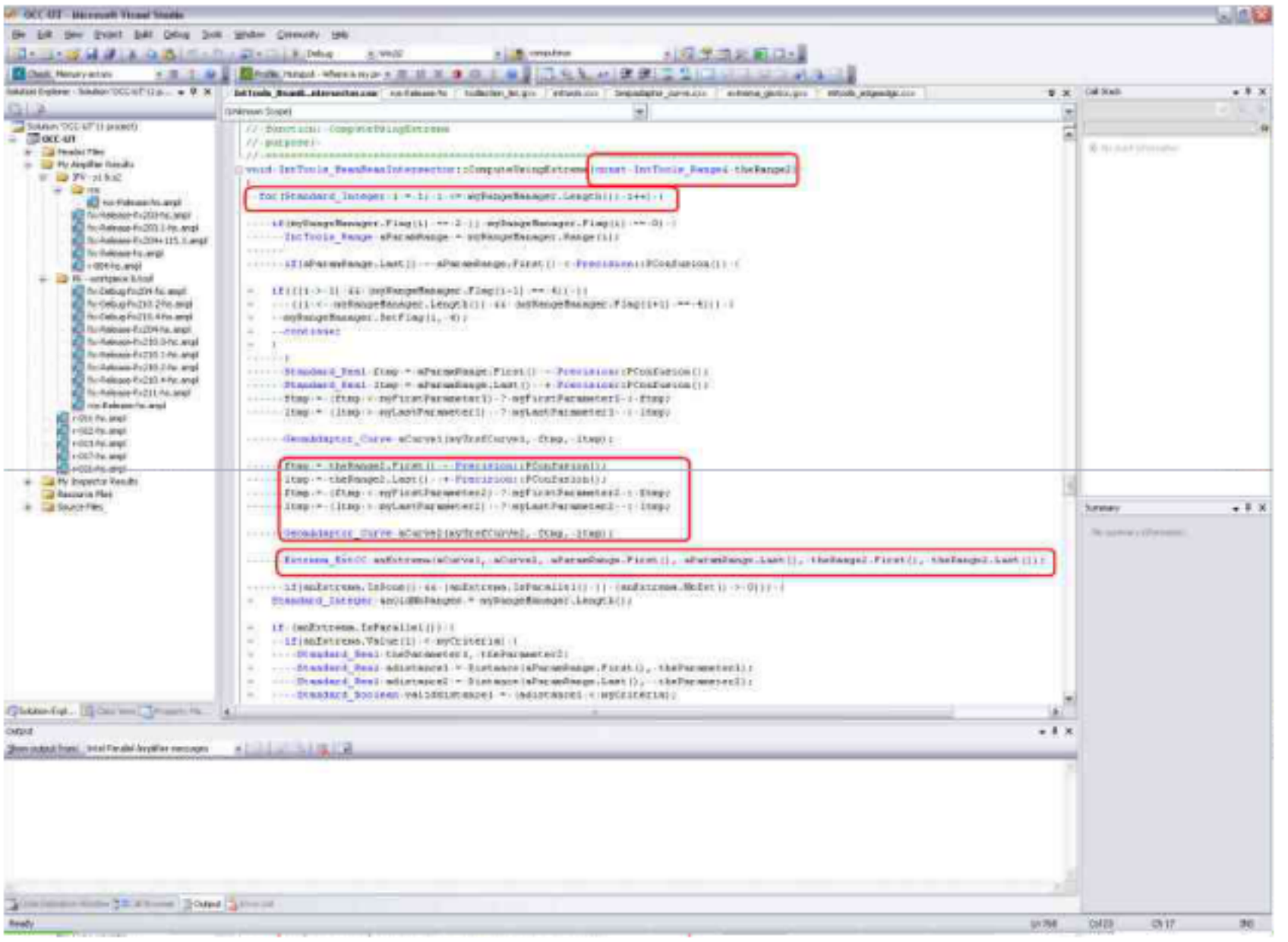
分界线：速度提升了 2.3x 倍速(从 113 秒到 49 秒)。还算不错，但是不是重大突破，花费的时间仍然是 10 秒的量级。其他方面(不是很清楚)还可以进一步优化，但是可能需要花费我更多的时间，我还没有打算去做它(更重要的是我已经花费了我的一半的假期)。
但是更重要的是，这个修改为其他 OCC 算法的速度提升打开了一扇大门，因为修改的是核心部分——极值求解 (Extrema)。今天我们讲的是怎么修改它... ...

像通常一样，我使用 Intel Parallel Amplifier (顺便说一下，我们刚刚发布了一个内部版本，公共的 beta 版应该在一月份的时候发布)检测热点区域。



它们与我之前处理的例子完全不同，它们与求边与边相交有关系(看图上右边的堆栈)。

查看函数 IntTools_BeanBeanIntersector::ComputeUsingExtrema()就明白到底发生了什么。



可能相交(例如，假如它们的包围盒相交)的每一对边都需要分析。每一条边(edge)都被分割成很多小段(例如，椭圆分成了 40 段)，然后需要分析这些小段与其他边(edge)的小段是否相交。虽然在 ComputeUsingExtrema()内部，第 2 个段没有发生变化，Extrema 对象(用于计算最小距离)的生成不依赖于任何已经计算的东西，每次都直接使用第 2 段初始化函数(And though inside ComputeUsingExtrema() the 2nd range does not change, the Extrema object (used to calculate the smallest distance) is created from scratch and initialized with it every time)。使用调试器进入 Extrema，我发现了 B 样条曲线被分割成了 C2 连续的小段(在 Pawel 的模型中大约是 20 段)。沿着每一小段曲线，有一些采样点(默认是 32 个点)。距离是临时计算的，可能相交的边传入 math_Function*作进一步的精确计算。在这个函数内部，采样点有时候会增大到 2x 倍(只是为了增加采样的可靠性)。如此，由于没有重用之前计算的结果，需要重新进行大量的计算。

优化的主要思想就很清楚了——缓存并重用。这需要修改 Extrema API(因为它不支持这么做)，好在我最终完成了这个工作。当前还仅仅局限在 3D 曲线上，同样的改进也可以用在其他方面，在 OCC 人确认这样的修改之后没有问题之后，我才准备继续做这样的改进。

但是在重新设计 API 之前，我准备先做一些其他改进，以消除一些小的瓶颈，否则重新设计 API 带来的大性能提升会将这些小瓶颈改善带来的小性能提升淹没掉。

(待续... ...).

POSTED BY ROMAN LYGIN AT 13:59, 2008-12-29

第 2 节　开始修改 Extrema

(接上节... ...)

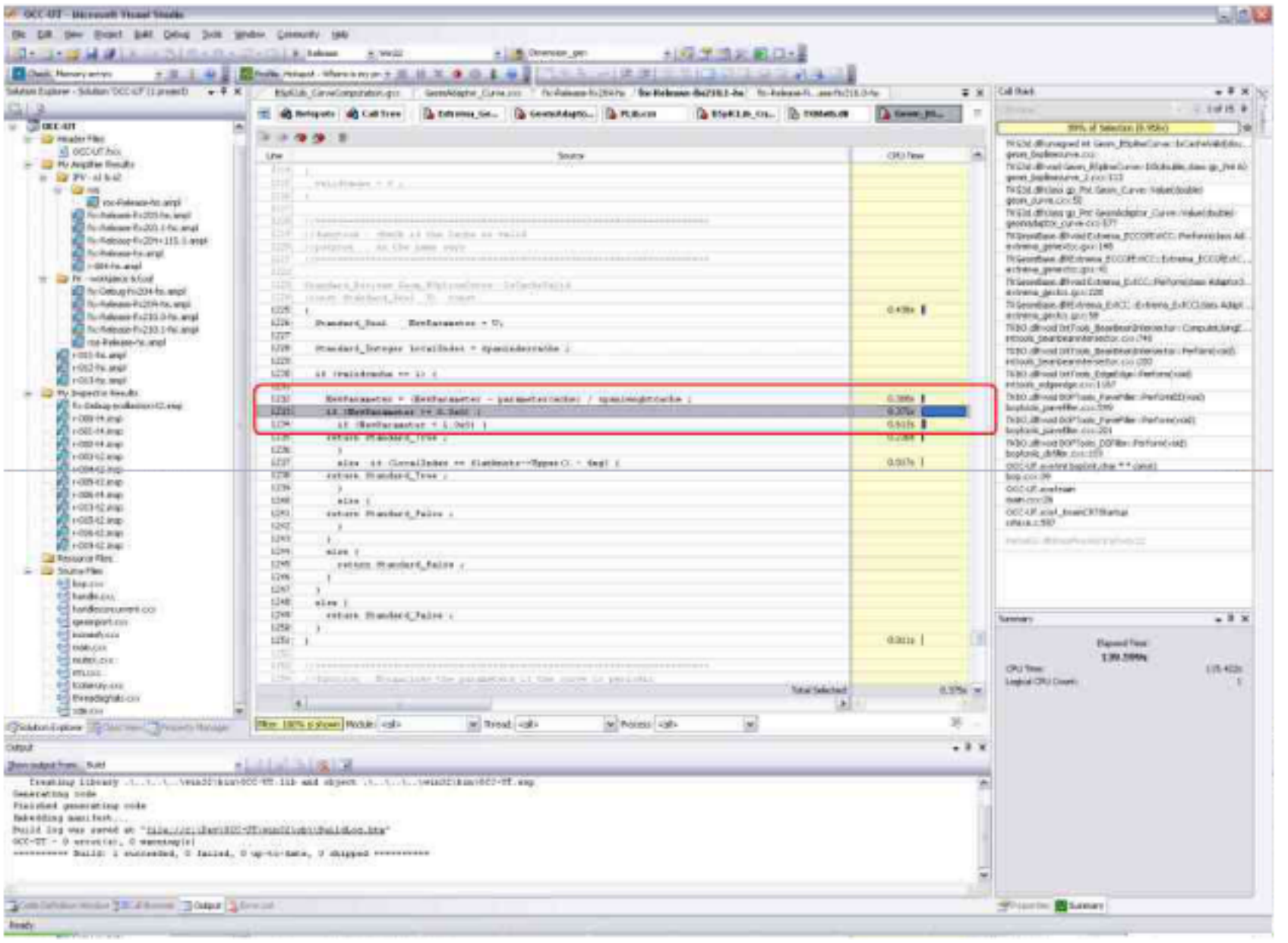
sqrt()

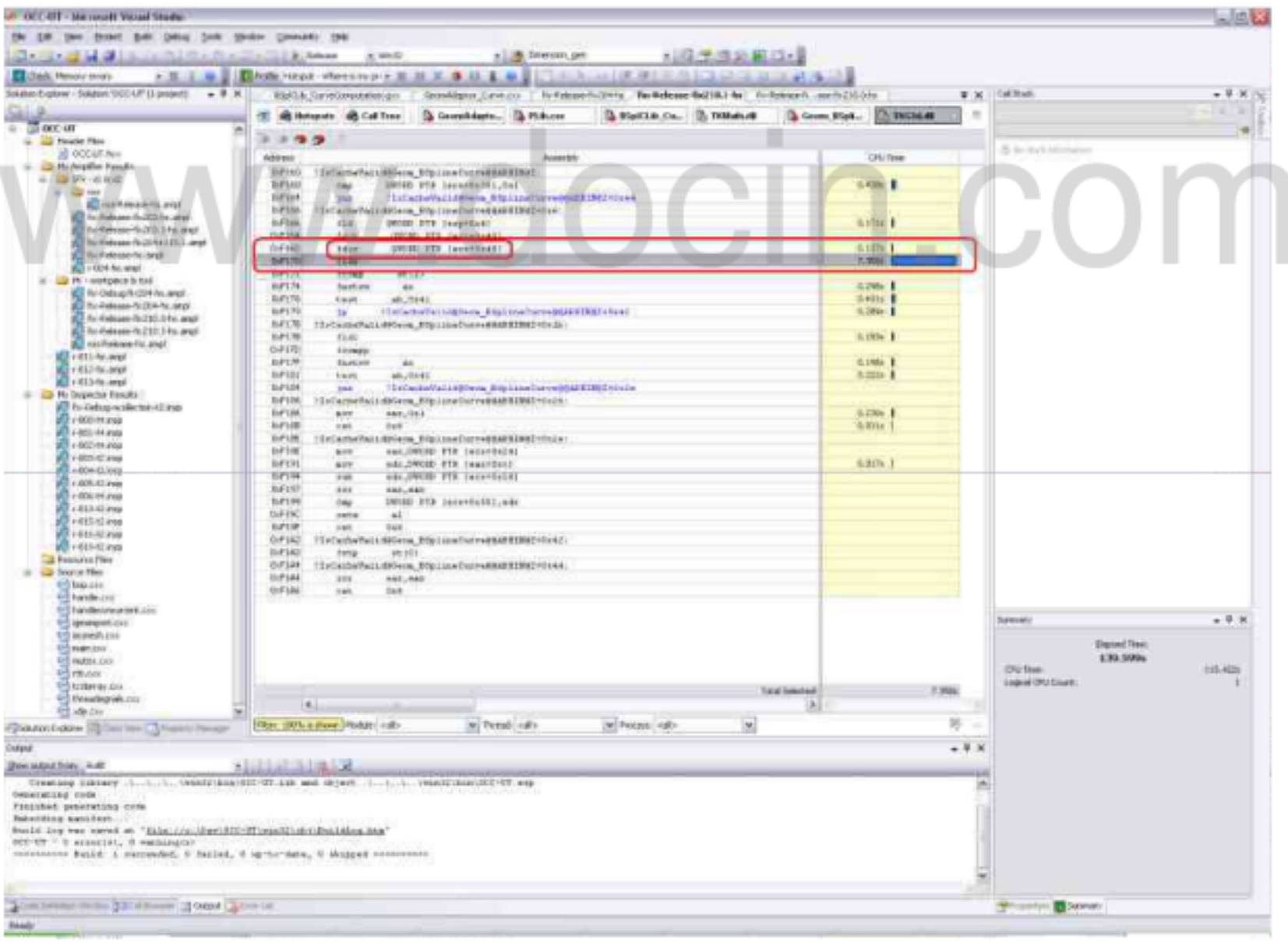
第一个小的瓶颈位置是 sqrt(), 它是调用最多的前 5 个函数之一。由于这是一个系统函数，直到我按了 Amplifier 工具条上的一个按钮之后，我才开始注意到它(实际上 Amplifier 会记录调用系统函数的时间)。

通过回溯堆栈我找到了原因。Extrema 在很多地方使用 gp_Pnt::Distance()(有时候使用 gp_Vec::Magnitude())计算精确距离判断是否一个点比另一个点近。这么做浪费了 6.5%的 CPU 时间。不用做其他修改，只要将距离的计算替换成平方距离就可以了。这对你应该也有启发，尽量使用 SquareDistance()或者 SquareModulus()而不是 sqrt()。这样修改以后，我节省了 5-6%的 CPU 时间(我并没有将所有的这种调用都改掉)。

IsCacheValid()

另一个改进与 Geom_BSplineCurve 有关。从前面的讨论可以看出沿着 B-样条曲线上的点有大量的计算，下面我们会更深入的研究。Geom_BSplineCurve::IsCacheValid()用于判断是否有局部缓存用于快速计算。它占用了 6.25%的 CPU 时间，看一下这个函数的实现代码以及生成的二进制程序就能知道这是为什么了。





原因——通过除以 spinlenghtcache 从而将范围单位化到[0, 1)的范围。实际上可以很容易避免这么做，并且我还消除了很多的分支判断，从而提高了可读性(或许性能也提高，因为分支判断也非常费时)。实现方式的对比如下：

```
Standard_Boolean Geom_BSplineCurve::IsCacheValid
(const Standard_Real U) const
{
  Standard_Real    NewParameter = U;

  Standard_Integer LocalIndex = spanindexcache;

  if (validcache == 1) {

    NewParameter = (NewParameter - parametercache) / spanlenghtcache;
    if (NewParameter >= 0.0e0) {
      if (NewParameter < 1.0e0) {
        return Standard_True;
      }
      else if (LocalIndex == flatknots->Upper() - deg) {
        return Standard_True;
      }
      else {
        return Standard_False;
      }
    }
    else {
      return Standard_False;
    }
  }
  else {
    return Standard_False;
  }
}
```

```
Standard_Boolean Geom_BSplineCurve::IsCacheValid
(const Standard_Real U) const
{
  Standard_Real aDelta = U - parametercache;

  return (validcache &&
          (aDelta >= 0.0e0) &&
          ((aDelta < spanlenghtcache) || (spanindexcache == flatknots->Upper() - deg)));
}
```

这样修改以后，这个函数花费的 CPU 时间下降了大约 60%。即使减法比除法开销小，但是由于浮点运算非常费时，这个函数花费的 CPU 时间仍然较多。

计算 B 样条曲线上的点

通过回溯(unwinding)堆栈，可以定位出来函数 PLib::NoDerivativeEvalPolynomial()是由 Geom_BSplineCurve::Do()调用的，Geom_BSplineCurve::Do()又是由 Extrema 调用(是否还记得上面提到的在内部大量调用 Extrema？)。我添加了一个计数器统计调用次数，达到了 2.723 亿次！哇！

由于这个方法本身不能做任何改进，我决定向上回溯，修改 Extrema。

Extrema

直到现在关于曲线与曲线方面的 Extrem 架构都不够灵活。经常会将先前计算的结果冲掉，重新进行所有的计算。另外，点与曲面、曲线与曲面都有一些缓冲，所以我就有些奇怪为什么就单单忽略了曲线与曲线的计算结果缓冲。

不管怎样，我扩展了这个 API，从而能够设置每一条曲线和它的域(ranges)，并在之后进行计算。为了利用这些计算结果，我又修改了函数 IntTools_BeanBeanIntersector::ComputeUsingExtrema()，为参数 theRange2 只赋值一次。这只是第一步，然后我还将 Geom_BSplineCurve::Do()/ PLib::PLib::NoDerivativeEvalPolynomial()的调用次数减少到了 860 万次。

(待续... ...)

POSTED BY ROMAN LYGIN AT 17:16, 2008-12-29

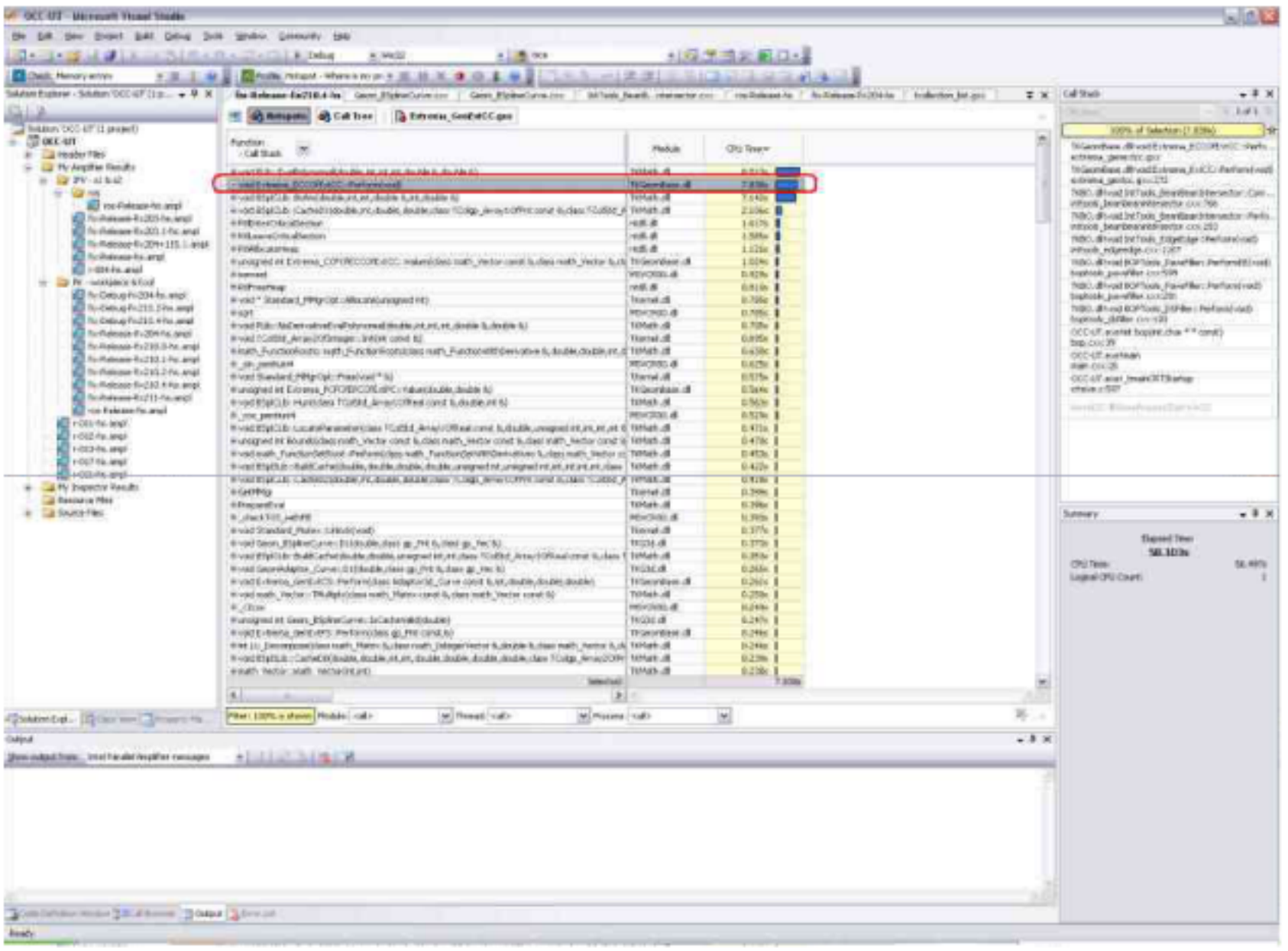
第 3 节　继续缓存结果

(接上节... ...)

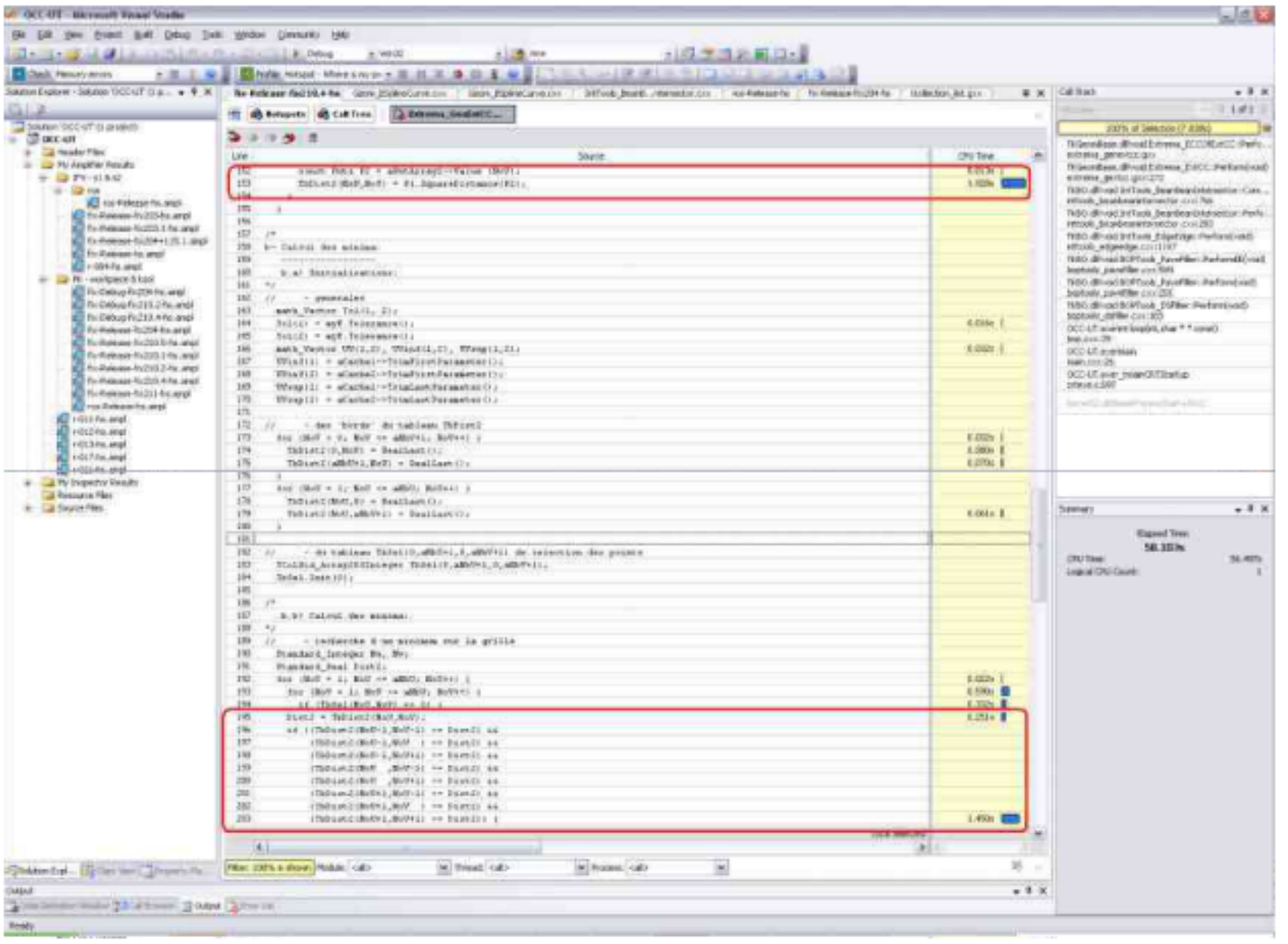
然而，这并没有完全解决缓存结果的问题。正如我之前提到的，B 样条曲线被分割成 C2 连续的小段，然后分别处理。例如，在一条 B 样条曲线上，有 20 个 C2 连续的小段(intervals)，缓冲只能在其段内起作用，但是每次另外一条边的子域调用时，所有的数据都要重新计算(but every time a new subrange of the another edge's range was called, all data were recalculated.)。所以，我扩展了缓存，使其能够支持缓存列表，当每次从第一条边中取出一小段后，第二条边的小段中的点都从缓存中读取(So, I extended caching capabilities to support lists of caches and every time a new subrange of the 1st edge's range is used, points along subranges of the 2nd edge's range are retrieved from the cache.)。这样修改之后，Do()只调用了 47.5 万次，这相对于原来 2.732 亿次减少为 570 分之一。CPU 时间减少了，速度提高了 2+x 倍速。是的，我做到了！

* 新目标*

好了，接下来做什么？看下图中的新的热点区域，可以看到Extrema_ECCOfExtCC::Perform()在头三个之中。



那么在它的内部又有什么呢？看他的头三个热点区域(第 1 和第 2 如下图所示，第 3 个与第 2 个相似)都与访问 double 数组 TbDist2 有关(其中保存了两条曲线上的采样点之间的平方距离(square distances))。



我怀疑这是由于数据延迟(data latency)造成的，例如，缓存失准(cache misses)，例如当引用的数据在 CPU 处理器的缓存中不存在时，就从 RAM 中装载，这比较耗费时间。当从内存中读取数据时，一些内存中临近的数据也被载入处理器中(处理器假定你也需要其中的某些数据，并提前将这些数据载入到缓存中)。对于 TbDist2 来说，这么做并不正确，因为对于每一个元素[i, j]需要检查 8 个临近数据([i-1,j-1], [i-1,j]...[i+1,j+1])。大部分数据在内存中并不相邻接，因此就会有很多次读取数据失准。而且由于调用的次数在百万量级上(正如你猜想的那样经过上面的修改之后这并没有什么改变)，从而引起一个热点区域(a hotspot)。我不知道怎样才能解决这个问题，我可能不得不与 Intel 的同事一起讨论一下这个问题，看看这种情况下能不能做些改进。或许唯一可能的解决方案在于上游(upstream)，即降低边被分割成小段的数量。今天，经过用简短的电子邮件与 OCC 团队联系，对于这个问题能不能解决和怎么解决我还没有任何线索。

另外两个热点区域——PLib::EvalPolynomial(), BSplCLib::CacheD1()——都与 B-Spline 曲线的一阶导数的调用有关。当计算两条曲线之间的最小距离的函数的根时(when finding a root of function of minimizing a distance between two curves)需要调用 Geom_BSplineCurve::D1()。BSplCLib::Bohm()调用了 240 万次，PLib::EvalPolynomial()调用了 3000 万次。我不知道如何缓存它们的计算

结果(假如你知道，请告诉我)，可能解决的方法也是往上游查找。

* 下一步*
好了，我目前就研究到这个程度了，坦白说我准备就到此为止。我将修改发给了 OCC 等待它们确认修改没有引起数据库退化(I sent modifications to OCC for validation on the regression database)。假如修改效果很好，我准备对于二维情况做相似的修改，从而使它们保持一致，也可能对 Extrema 做一些其他的改进，从而使得将来的维护更容易。

当然，也可能在 BOPs 中试试多线程，我现在还没有准备这么做，这需要深入理解算法，以及可能的并发性。我们走着瞧吧... ....

假如有人想试试做这些修改的话，我同时会将这些修改发布在 SourceForge 上。像以前一样，欢迎大家给我反馈意见，特别是你发现了性能提升或者降低。

(结束)

顺便说一下
第一篇文章之后，我又收到了几个测试案例，而且他还做了很多修改提升 BOP 的性能。这是 OCC 的客户做的，在这注册的名字叫 Solar Angel。酷！我很高兴受到这些修改，而且几年之后居然又跟他联系上了！