# 1近期工作

没法从微处理器层面优化，一方面是因为算法优化层面使得代码特别的难读；另一方面OpenSSL的主要运算在性能上已经优化得十分完美，已经不满足微处理器层面优化的原则了，这也是因为OpenSSL是开源软件的缘故，微处理器层面的优化也已经被考虑进去了。所以我只做了三个方向的微处理器优化对比分析实验，即原始版本和除去优化后的两个版本代码的性能对比。

## 1.1微架构优化总结

主要可以从以下几个方面来考虑优化：

寄存器重命名（Register Renaming ）技术，cache失效，数据和代码的对齐问题，第一次运行消耗，分支预测成功率，取指令问题，寄存器读延迟，过长的依赖环。

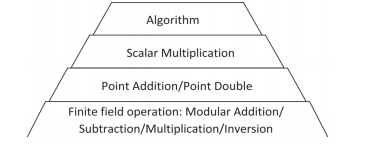
微处理层面的优化原则如下：

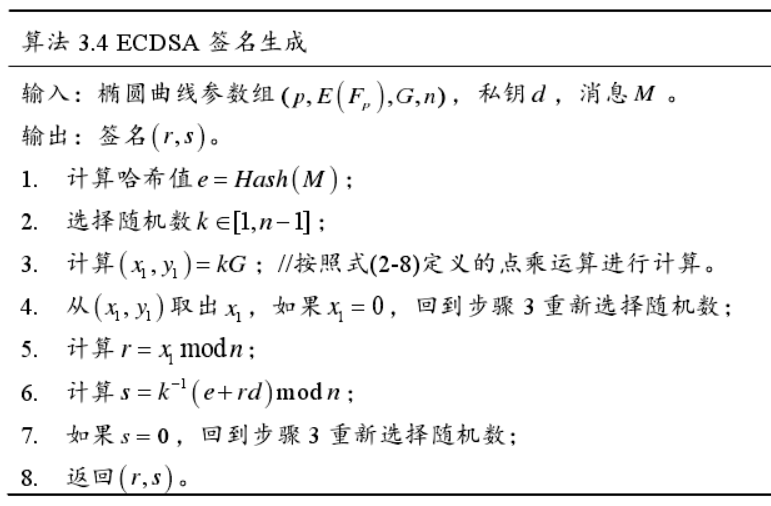
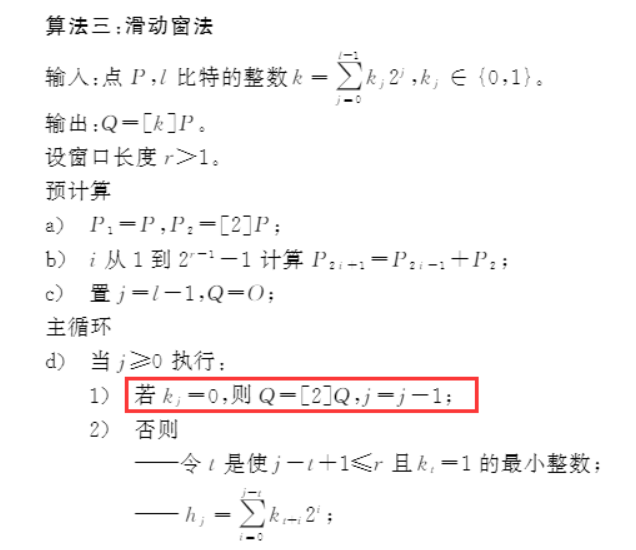
在把你的代码转为汇编的之前，确认你的算法是足够优化的。 首先，通过优化算法的方式来改进代码所获得的性能上的提升要远比代码通过汇编进行优化的性能提升要高的多。第二，你必须找到你的程序里最关键的部分。 在性能优化方面永远注意80-20原则，即20%的程序消耗了80%的运行时间，因而我们要改进效率，最主要是考虑改进那20%的代码。不要优化程序中开销不大的那80%，这是劳而无功的。

## 1.2难点

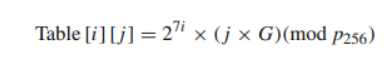
1. ECDSA签名算法主要过程是选择随机数，通过随机数与生成元参数简述现有 openssl 最快实现的性能最大提升的点 (主要是 wNAF 的优点减少了 k 的二进制展开中 1 的个数，预计算除去了每次处理一个 bit 时的平方运算，以及一个数值比较小的点乘操作，点乘的标量越大运算量就越大)

在ECDSA和ECC算法中，标量乘法的实现占用大量时间，在传感器设备中，标量乘法需要80%以上的运行时间[8]。



Hierarchy of operations in ECC algorithm  

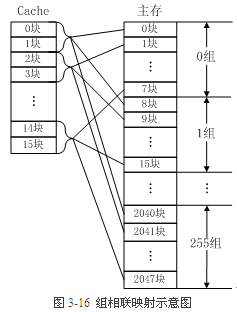
1. wNAF 算法的窗口为 7 ，一个占用内存空间 0.15 M 的预计算表如下：



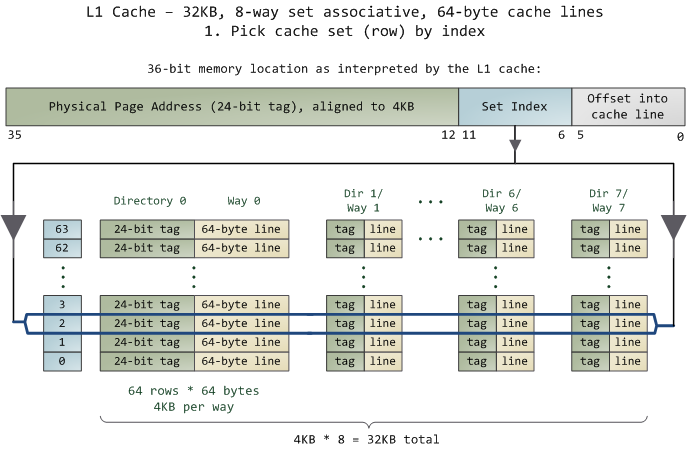
将 256 位的 k 分成 37 份，实现标量乘法的时候，从高到低遍历这 37 份，每一次的遍历只需要查表和进行简单的标量加法即可

1. 结果发现点乘优化的效果太好 (主要还是归功于预计算表) 本来点乘运算占整个签名算法的比重大概是 80%，现在已经只有 10% 不到，固继续优化点乘算法 (从硬件和软件) 已经优势不大，而且其他代码的代码重用率并不高，所以该签名算法效果已经很好，继续优化的意义并不大

## 1.3 cache



将主存分割成了许多4KB大小的物理内存页，而cache中一路的大小也是4KB，每一页都含有4KB / 64 bytes == 64条缓存线。在一个4KB的页中，第0到63字节是第一条缓存线，第64到127字节是第二条缓存线，以此类推。每一页都重复着这种划分，所以第0页第3条缓存线与第1页第3条缓存线是不同的，这两条缓存线被存入到cache中不能在同一路，但是第0页第3条缓存线与第1页第4条缓存线可以存入到cache中的同一路上。在优化过程中，不同页的同一条缓存线不能超过cache的路数。比如现代Intel的L1 cache是8路组相联，那么你一段代码中处理的数据不能超过8页。



## 1.4第一次和重复运行

一片代码往往在第一次运行时比重复运行消耗更多的时间。 原因见下:

1. 从 RAM 读入代码到cache花去了比运行它更多的时间。

2. 代码操作的所有数据都必须加载到cache， 这比执行那些操作更花时间。 当代码重复运行的时候， 数据几乎都在 cache 里。

3. 跳转指令在第一次运行的时候并不在分支目的缓存(branch target buffer，简称BTB)里， 因此一般都不能正确的预测。 见第22章。

4. PPlain 上， 代码的解码是个瓶颈。 如果花掉一个时钟周期去检测指令长度， 那么就不可能在一个时钟周期解码两条指令， 因为处理器不知道第二条指令从那里开 始的。 PPlain 通过记住上次运行后保存在cache里的每条指令的长度来解决这个问题。 这样做的结果是， PPlain上第一次执行时，指令如果不是只有1个字节长的话就不会配对执行。 PMMX， PPro， PII 和 PIII 在第一次解码却没有这个问题。

因为这四个原因，在循环内部的一段代码第一次运行通常比随后的运行花去更多的时间。

如果你使用了一个很大的循环而不能放入代码cache，将导致效率下降，因为它们不能在 cache 运行。 因此你应该重新组织一下循环使cache能放下它们。

如果你有非常多的跳转，调用，分支在循环里，就会反复的产生分支目的缓存失败。

同样的，如果循环反复操作一个对数据cache而言太大的数据结构，也会一直得到数据cache不命中的惩罚。

## 1.5对齐

对齐主要有4byte对齐、8byte对齐、32byte对齐、64byte对齐、4K对齐。4byte对齐和8byte对齐很正常，分别对应32位处理器的数据总线是32位，64位处理器的数据总线是64位。而后面4个对齐是与cache有关的，分别对应了64bytes大小的cache块和32bytes大小cache块。

拿32bytes和4K对齐来说明，cache 行总是在物理地址的 32 字节对齐。每次你读数据未命中，处理器将从内存读出一整条 cache 行，当你从一个可以被 32 整除的地址读出一个字节， 下 31 字节的读写就不会有多余的消耗。 因此在内存中，你可以把相关数据项放在对齐的32字节块里（集中访问）来获得好处。 例如，如果你有一个循环要操作两个数组，你就可以将两个数组穿插成一个结构数组， 让一起使用的数据的物理位置也在一起。所以一个中等大小的数据（比如20bytes）要放在被32整除的地址上，也就是我们所说的32bytes对齐，这样处理器将该数据从内存放入cache只需要1次就可以完成。

对于4K对齐而言，如果你的代码的关键部分要操作很大的数据结构或者随机数据地址，你可能会想保存所有常用的变量(计数器，指针，控制变量等) 在一个单独的最大为 4k 的连续块里面， 这样你就有一个完整的空闲cache行集来访问随机数据。 因为你通常总是需要栈空间来为子程序保存参数和返回地址， 最好的做法是复制所有的常用静态数据到堆栈（把它们复制成动态变量），如果它们被改变，就在关键循环外再复制回去。

## 1.6期间的几个实验

针对OpenSSL中nist-prime256曲线的ECDSA算法进行了优化分析实验，分别从cache、对齐、寄存器重命名三个方面进行了研究，主要是研读了mul\_point (汇编实现)代码，基本的大数加减乘函数。并从这些代码中找到了上面提到的三个方面的优化点，然后将这些优化分别去掉，看看对ECDSA签名执行的速度是否有影响，影响多大。

## 1.7实验环境和配置

| **配置** | **参数** |
| --- | --- |
| cpu型号 | Intel Core i7-9750H |
| 微架构 | coffee lake |
| openssl 版本 | OpenSSL 3.0.0-beta1-dev |
| 内存 | 24G |
| 编译器 | gcc |
| 操作系统 | ubuntu 20.04 |

## 1.8 cache性能对比实验

### 实验思路

选取的函数是大整数的加减乘三个函数，在签名过程中会频繁调用这些函数，所以他们的优化至关重要。拿 vli\_add 即大数加法函数为例，函数中有一个for循环处理两个数组：

for(i=0; i<NUM\_ECC\_DIGITS; ++i)

{

uint64\_t l\_sum = p\_left[i] + p\_right[i] + l\_carry;

if(l\_sum != p\_left[i])

{

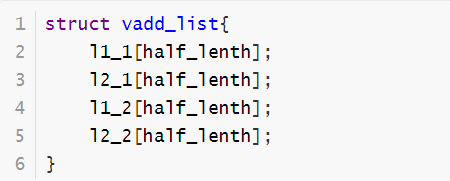
l\_carry = (l\_sum < p\_left[i]);

}

p\_result[i] = l\_sum;

}

这是OpenSSL中非asm优化的版本，而asm版本中将这两个数组穿插成一个结构数组， 让一起使用的数据的物理位置也在一起，两个数组的结构可等价于下面的代码：



这样在64位处理器中，取一个64位的数，就相当于取了两个大数的一部分，与优化之前少访问了一次内存！

### 实验结果

本次实验分别将优化的签名算法版本和去掉优化的签名算法版本分别跑10s，然后取平均值结果如下（单位为时钟周期）：

|  |  |
| --- | --- |
| cache优化版本的点乘算法开销 | 除去优化的点乘算法开销 |
| 25516 | 27545 |

## 1.9对齐性能对比实验

### 实验思路

正如1.2所说的ECDSA算法优化程度取决于点乘算法进行了多少次点加，而在OpenSSL中的ECDSA只需要37次（一般二进制展开平均需要400次点加），而为此的代价是预存了0.15M大小的预算表和比原来C代码长的多的汇编代码，这都增加了内存上的开销。可能这点内存的开销对现代微处理普遍8G的内存不算什么，但是比较影响Cache的命中率，因为这个预计算表的大小已经大大超过了32byte的一级缓存。所以为了缓解这个问题，OpenSSL在预算表部分数据和代码中使用 .align伪指令进行代码和数据的对齐，关于对齐是如何提高性能的，1.5节已经做出了详细的解释。

现在我们只需要将预计算表部分的数据和代码中的 .align 伪指令去掉得到一个未对齐的签名算法版本，然后与原始代码的性能进行对比即可

### 实验结果

本次实验的结果简单求平均值很难区分两者的差别，因为对齐和未对齐只是在性能开销上稳定性强弱会有差别，所以我在一个小时取了三个阶段，每个阶段分别对两种版本的签名算法跑了3个历程，每次一个历程运行了200次签名算法并且取平均值 （单位为时钟周期）

|  |  |  |  |
| --- | --- | --- | --- |
| 时间段 | 历程 | 对齐了的点乘算法开销 | 未对齐的点乘算法开销 |
| 前20min | 第一次 | 22564 | 21564 |
| 第二次 | 21003 | 32128 |
| 第三次 | 19692 | 41233 |
| 中20min | 第一次 | 20435 | 21806 |
| 第二次 | 22342 | 21564 |
| 第三次 | 20098 | 21329 |
| 后20min | 第一次 | 19851 | 32665 |
| 第二次 | 22686 | 19965 |
| 第三次 | 21451 | 21564 |

可以看到未对齐的点乘算法波动比较明显

## 1.10寄存器重命名性能对比实验

### 实验思路

观察到ECDSA签名算法的底层实现函数中有很多消除不同代码依赖的代码：

MOV EAX， [MEM1]

IMUL EAX， 6

MOV [MEM2]，EAX

MOV EAX， [MEM3]

INC EAX

MOV [MEM4]，EAX

这里的最后三条指令是独立于开始的三条指令的，因为它们不需要前面三条指令的结果。

为了优化它，在早期的处理器中，你必须在后三条指令中不用EAX寄存器，并且调整指令的顺序使得六条指令两两配对。在每次你写EAX寄存器的时候，它分派一个新的临时寄存器。 因此，MOVEAX，[MEM3]相对前面的指令独立了。 在乱序执行之后，有可能在较慢的指令IMUL完成之前，MOV [MEM4]，EAX已经完成了。寄存器重命名是完全自动的。

每当一条指令写一个永久性的寄存器时，一个新的临时寄存器被当作它的化身般分派。 一条对一个寄存器既读又写的指令也将引发寄存器重命名。

为了对比采用现代处理器的重命名技术提升了多少性能，我们将上面类似的代码改成如下：

MOV EAX， [MEM1]

MOV [MEM2]，EBX

IMUL EAX， 6

MOV EBX， [MEM3]

INC EAX

MOV [MEM4]，EAX

即将前面4行指令的顺序替换，并将 line2 和 line4 的寄存器由EAX换成EBX

### 实验结果

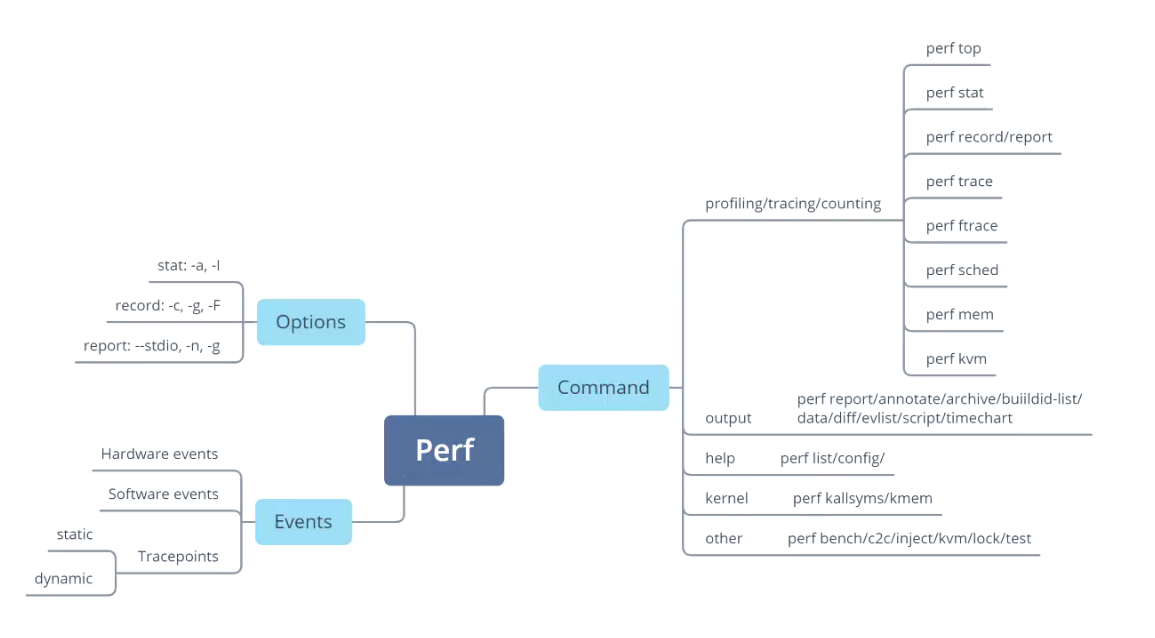
本次实验分别将使用寄存器重命名的签名算法版本和未利用寄存器重命名的签名算法版本分别跑10s，然后取平均值结果如下（单位为时钟周期）：

|  |  |
| --- | --- |
| cache优化版本的点乘算法开销 | 除去优化的点乘算法开销 |
| 23864 | 23532 |

并没没观察到明显的结果，可能是代码关联性没有考虑太周到，依赖链还是过长了，需要对相关代码进行更深入的研读和分析

# 2接下来的工作

接下来希望可以自己实现一遍ECDSA算法，因为有很多部分的代码因为高度优化的缘故阅读十分困难。仍然还有几个可以通过微架构层面优化的地方，比如提高分支预测的概率，减少过长的依赖环，但是在我看来应该不会有进展；因为OpenSSL是个开源库，基本上从各个层面已经实现了高度的优化，最近的三个优化实验，我也只是通过去掉OpenSSL代码中的优化来进行的对比分析实验。所以接下来可能只是学习微处理器的优化技巧，并不会真正的优化ECDSA签名，这些可以给以后实现高效的SM2签名算法提供技术支持。



# 3参考资料

1. 深入理解 CPU 的分支预测https://blog.csdn.net/hanzefeng/article/details/82893317
2. Spectre原理详解及分支毒化的实现https://zhuanlan.zhihu.com/p/114680178
3. Intel® 64 and IA-32 Architectures Optimization Reference Manual
4. Agner Fog, Technical University of Denmark, Optimizing subroutines in assembly language
5. Agner Fog, Technical University of Denmark, The microarchitecture of Intel, AMD and VIA CPUs, An optimization guide for assembly programmers and compiler makers
6. Agner Fog. Copenhagen University College of Engineering, Test programs for measuring clock cycles and
7. [cache组相联https://www.cnblogs.com/east1203/p/11572500.html](https://www.cnblogs.com/east1203/p/11572500.html)
8. Al-Zubaidie M, Zhang Z, Zhang J. Efficient and secure ECDSA algorithm and its applications: a survey[J]. arXiv preprint arXiv:1902.10313, 2019.