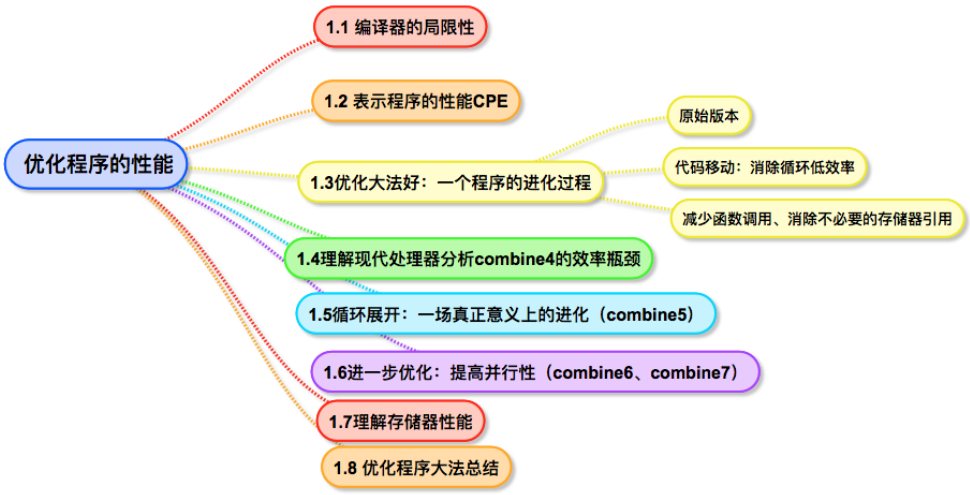
《深入理解计算机系统》| 优化程序的性能

唐鱼的学习探索 关注

0.394 2018.02.09 05:09:26 字数 5,436 阅读 1,574



编写运行的快的程序有三个因素：①选择合适的算法和数据结构；②理解编译器的能力，使用有效的方式让编译器能进行优化；③对于运算量特别大的程序，可能还需要进行任务分解。在这一过程中可能还需要对程序的可读性和运行速度进行权衡。

在阅读这一章节的过程中花费了大量的时间对我自己的自动办公软件进行了优化，算是学以致用。选择合适的算法和数据结构不在本章的讲解内容中，我们从编译器的能力和局限性讲起着重介绍几种提高程序运行速度的方法

1.1 编译器的局限性

编译器遵循的一个优化程序的原则是：安全优化，也就是说为了保证程序的正常运行（优化后的版本与未优化的版本有一致的行为，这不是废话吗）编译器一般都是很保守的。来看一个例子：

```
1   void twiddle1(int *xp, int *yp)
2   {
3       *xp += *yp;
4       *xp += *yp;
5   }
6
7   void twiddle2(int *xp, int *yp)
8   {
9       *xp += 2* *yp;
10  }
```

安全优化：存储器别名

从上面的例子不能看出，一般情况下twiddle2要求三次存储器引用（读*xp，读*yp，写*xp）而中twiddle1要求六次存储器引用（2次读*xp，2次读*yp，2次写*xp）。所以twiddle2的效率要高于twiddle1，但是如果考虑到xp等于yp，指向存储器中的同一个位置的时候，我们用twiddle2来优化twiddle1的版本就会造成程序的运行结果的不同。比如当xp = yp = 2的时候，f（twiddle1） = 8；而f（twiddle2） = 6 这就是我们说的编译器的局限性。

1.2 表示程序的性能CPE

在继续介绍优化大法的时候，我们对提高程序的性能做一个量化的参考，在以后的章节中好对比我们的优化后版本的执行效率。

CPE：每元素周期（Cycles Per Element），使用时钟周期，度量每隔周期执行了多少条指令。通常当一个标有“4GHz”的处理器，这表示的是处理器时钟运行频率4X10的9次方Hz每秒，那么一个时钟周期就是时钟频率的倒数，为0.25纳秒。

我们来看一个计算集合值的两个函数，我们假设有集合a = {1,2,4,5,7,9,10,12,16}集合p为集合a的前置和也就是p={1, 1+2, 1+2+4, 1+2+4+5,1+2+4+5+7，.....1+2+4+5+7+9+10+12+16}

我们有两种计算前置和p的方式，psum1和psum2：

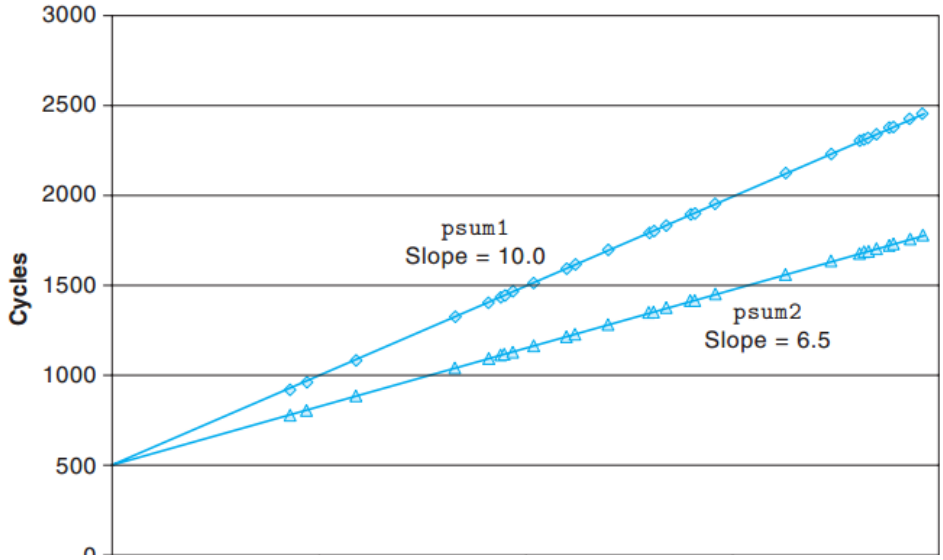
```
/* Compute prefix sum of vector a */
void psum1(float a[], float p[], long int n)
{
    long int i;
    p[0] = a[0];
    for (i = 1; i < n; i++)
        p[i] = p[i-1] + a[i];
}

void psum2(float a[], float p[], long int n)
{
    long int i;
    p[0] = a[0];
    for (i = 1; i < n-1; i+=2) {
        float mid_val = p[i-1] + a[i];
        p[i]    = mid_val;
        p[i+1]  = mid_val + a[i+1];
    }
    /* For odd n, finish remaining element */
    if (i < n)
        p[i] = p[i-1] + a[i];
}
```

循环展开技术

psum1是我们通常用到的版本，看起来也比较顺眼，psum2是我们以后要详细讲解的循环展开技术，核心的思想就是每次循环计算两个元素p[i]和p[i+1]从而减少了循环的次数。这个内容我们以后讲解。

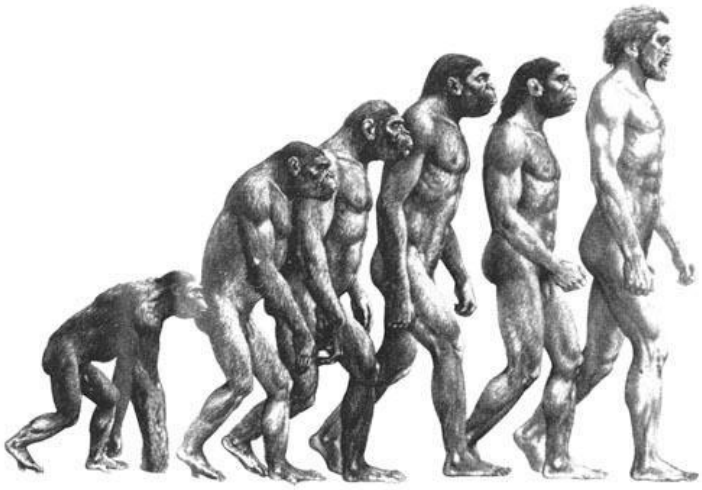
来看一下两个函数的性能对比，数据说话：



x轴表示处理的元素，y轴表示周期

我们可以很明显的看出来，当处理的数据量小的时候，两个版本的区别不大，但当周期在1000以上的时候，能处理元素的个数就明显不同了而且这种趋势越拉越大。

1.3优化大法好：一个程序的进化过程



智人的进化过程

从大约7万年前的认知革命开始，智人的进化经历了漫长的过程，终于实现了从动物到“上帝”的转变，我们将从一个简单的程序示例讲起带领大家一步步实现这个过程，当然不会花费上万年的时间。

① 原始版本：程序示例

```
1   /* Implementation with maximum use of data abstraction */
2   void combine1(vec_ptr v, data_t *dest)
3   {
4       long int i;
5
```

计算一个向量的集合

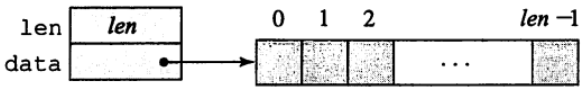
有必要解释一个combine1函数的作用：计算一个向量的集合

我们的向量有如下数据结构：

code/opt/vec.h

```
1   /* Create abstract data type for vector */
2   typedef struct {
3       long int len;
4       data_t *data;
5   } vec_rec, *vec_ptr;
```

code/opt/vec.h



向量由头信息加上指定长度的数组表示

我们定义typedef int data_t，方便我们用data_t表示不同的int、float、doubule数据。

我们使用： #define IDENT 0 和 #define OP +来对不同的运算进行求值，其中OP代表运算符号，而IDENT代表不同的初始值。

好了，作为一个起点，我们来看看我们的黑猩猩版本：combine1的效率：

函数	页码	方 法	整数		浮点数		
			+	*	+	F*	D*
combine1	331	抽象的未优化的	20.02	29.21	27.40	27.90	27.36
combine1	331	抽象的 -O1	12.00	12.00	12.00	12.01	13.00

未优化版本的效率

的确有点儿惨不忍睹，我们能为他做些什么呢？开始来进行 一些改进吧

② 代码移动：消除循环的低效率

改进循环的效率：将vec_length移除循环外

一个看上去无足轻重的代码片段可能隐藏有渐近低效率，上面combine2只是将求得向量长度的vec_length移除了循环外，因为向量的长度不会随着循环的进行而改变。我们来看看性能的改变：

函 数	页码	方 法	整数		浮点数		
			+	*	+	F*	D*
combine1	331	抽象的 -O1	12.00	12.00	12.00	12.01	13.00
combine2	333	移动 vec_length	8.03	8.09	10.09	11.09	12.08

③ 减少函数的调用

分析：combine2的代码可以看出，在循环的过程中每次都会调用get_vec_element来访问向量的元素，对于数组的引用，检查边界是合理的，但分析我们向量的数据结构不难看出，不进行边界检查我们也能够进行合法的访问：

code/opt/vec.c

```
1  data_t *get_vec_start(vec_ptr v)
2  {
3      return v->data;
4  }
```

code/opt/vec.c

```
1  /* Direct access to vector data */
2  void combine3(vec_ptr v, data_t *dest)
3  {
4      long int i;
5      long int length = vec_length(v);
6      data_t *data = get_vec_start(v);
7
8      *dest = IDENT;
9      for (i = 0; i < length; i++) {
10         *dest = *dest OP data[i];
11     }
12 }
```

消除循环中的函数调用

就像《葵花宝典》开篇就讲到的内容，欲练此功挥刀自宫，当我们在进行循环体内的调用函数优化的时候，必然会损害一些程序的模块性，慎用！

④ 消除不必要的存储器引用

分析：combine3中每次合并计算会将值累计在指针dest指定的位置上，我们来看看汇编代码：

```
   combine3: data_t = float, OP = *
   i in %rdx, data in %rax, dest in %rbp
1   .L498:                              loop:
2     movss   (%rbp), %xmm0               Read product from dest
3     mulss   (%rax,%rdx,4), %xmm0        Multiply product by data[i]
4     movss   %xmm0, (%rbp)               Store product at dest
```

rbp保存dest的值

从以上汇编代码中我们看出，dest的值存放在rbp中，每次循环，要先读rbp到xmm0，计算后的

结果又会重新写入到rbp中去，这样写很浪费。我们能够消除这样不合理的引用：

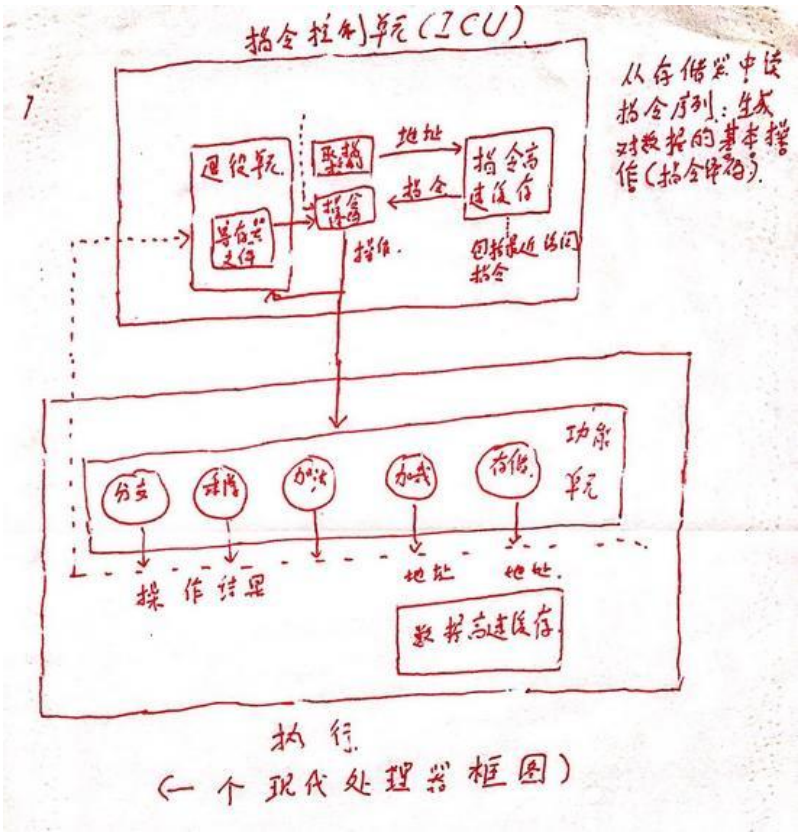
```
1   /* Accumulate result in local variable */
2   void combine4(vec_ptr v, data_t *dest)
3   {
4       long int i;
5       long int length = vec_length(v);
6       data_t *data = get_vec_start(v);
7       data_t acc = IDENT;
8
9       for (i = 0; i < length; i++) {
10          acc = acc OP data[i];
11      }
12      *dest = acc;
13  }
```

临时局部变量acc存放中间结果

我们使用局部变量acc保存累计计算的结果，这样就消除了每次循环都要对存储器进行取值和写回，使得程序性能有显著的提高。

1.4 理解现代处理器分析combine4的效率瓶颈

到目前为止的优化都不依赖于机器的特性，我们将学习现代处理器的一些知识，比如：关键路径、延迟界限以达到处理器级别的优化。



上图是一个简易的处理器框架图，在实际的处理中，处理器同时对多条指令求值（指令级并行），同时又呈现出一种简单的顺序执行的现象。 整个框架分为两个大部分：指令控制单元（ICU）和指令执行单元（EU），前者负责从存储器中读出指令序列，生成对数据的基本操

作。后者负责执行。我们分别进行讲解：

指令控制单元（ICU）： ICU从指令高速缓存（包含最近访问的指令）中读取指令，通常在ICU当前执行指令很早之前就开始取指，译码并发送到EU单元执行。当遇到了分支指令的时候：处理器采用分支预测，**投机执行，**在未确定该执行哪些操作的时候就对不同的分支目标地址进行取值和译码甚至执行。如果预测错误就回到最初的位置。使用投机执行技术求得的值不会存放在寄存器和数据存储器中，寄存器中的退役单元控制着寄存器的更新，只有当所有的分支都确定是正确的时候，指令才会退役，所有对寄存器的更新才会实际执行，否则清空该指令。

指令译码：将实际的指令转化为一组基本的操作 addl %eax，4（%edx）转为：①从存储器中加载一个值到处理器中；②将加载的值加上eax；③重新写回到存储器中

指令执行单元（EU）： 接受指令译码传来的一组操作，然后分配到功能单元中，这些功能单元包括：分支、乘除、加法、加载和写存储器。其中对存储器的访问，通过加载和存储功能单元对数据高速缓存的访问来实现。

！寄存器重命名机制的实现方式：

先来看看什么是寄存器重命名：

考虑下述代码片段在乱序执行CPU上的运行：

1. R1=M[1024]
2. R1=R1+2
3. M[1032]=R1
4. R1=M[2048]
5. R1=R1+4
6. M[2056]=R1

图1

指令4,5,6在功能上并不依赖于1,2,3的执行，但是必须要等待1,2,3完成之后才能执行4.

通过改变一下寄存器的名字可以解除限制：

1. R1=M[1024]	4. R2=M[2048]
2. R1=R1+2	5. R2=R2+4
3. M[1032]=R1	6. M[2056]=R2

图2：将R1重命名为R2

实现方式：当一条更新R1为R2指令译码时，将[r（R1），t（R2）]的对应关系加入到一张表中，随后当图1指令4需要再次访问到R1的时候，发送到执行单元的值会将R2作为操作数源的值，而当M[2048]完成赋值任务以后，会形成（v，t）的结果，指明标记的结果M[2048]。所有等待R2的值都会使用v作为源值转发。这样做的好处就是值可以从一个操作直接转发到另一个操作，而不是写到寄存器文件再读出来。

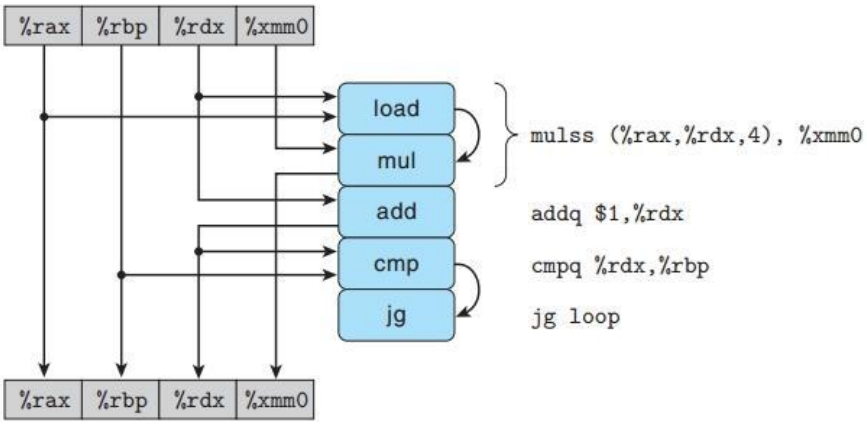
我们学习了一些基础的知识，我们重新来分析一下combine4的一些特性：

combine4：瓶颈在循环部分

```
combine4: data_t = float, OP = *
i in %rdx, data in %rax, limit in %rbp, acc in %xmm0
1   .L488:                          loop:
2     mulss   (%rax,%rdx,4), %xmm0    Multiply acc by data[i]
3     addq    $1, %rdx                Increment i
4     cmpq    %rdx, %rbp              Compare limit:i
5     jg      .L488                   If >, goto loop
```

已float的乘法为例

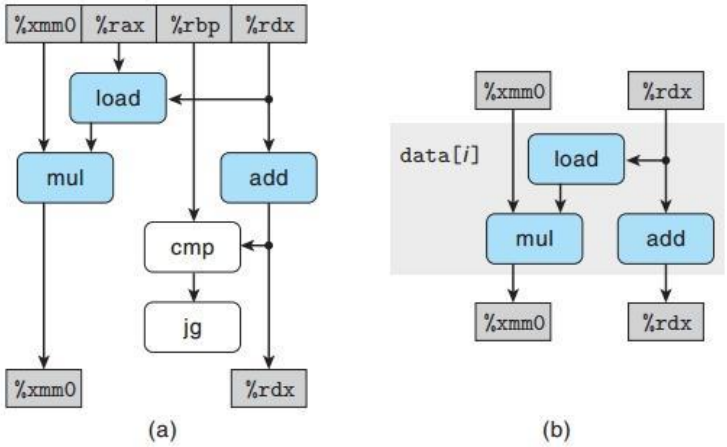
我们所熟悉的指令译码器，会将以上这4条指令扩展为5个步骤：



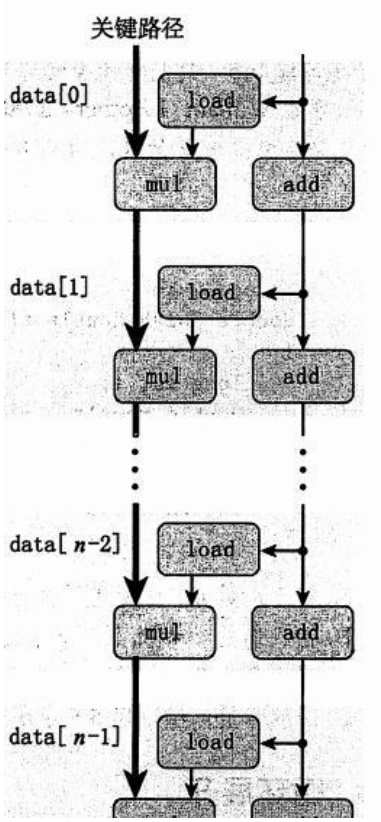
combine4的循环代码图形化表示

我们简单的来理解一下mulss指令的两个操作：load指令加载rax、rdx的值并将计算的结果直接传入到mul指令中，与xmm0进行乘法运算并将结果写入到xmm0中。

我们重新画一下上图的内容，使得结果看起来更清晰一些：



图b中我们删除了白色区域（无相关项）和没有修改寄存器的部分，只留下了循环执行过程中对xmm0和rdx迭代进行的一系列操作。



关键路径

终结一下：我们可以看出，两大关键链条分别是：mul对acc的操作，和add对i的操作，而左边的mul链条会成为关键路径。通过对处理器结构的分析，我们接下来不难看出，要再一步进行优化，就只有对关键路径进行优化了。（继续讲解循环展开技术）

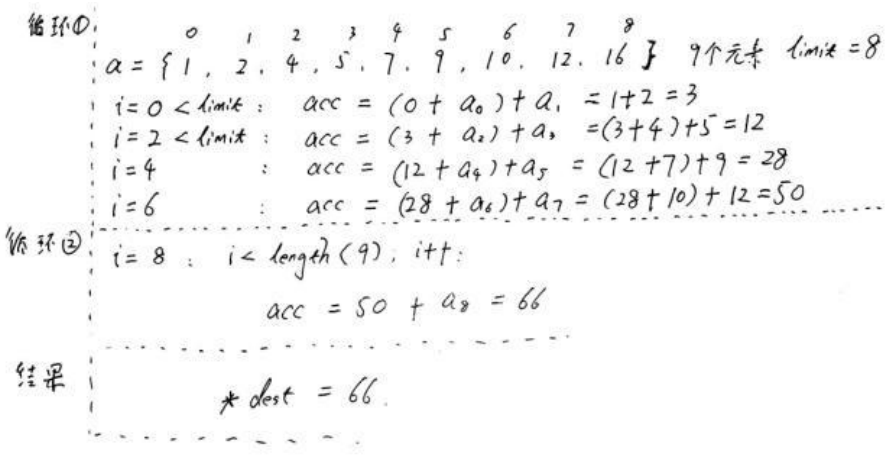
1.5 循环展开：一场真正意义上的进化（combine5）

```
1   /* Unroll loop by 2 */
2   void combine5(vec_ptr v, data_t *dest)
3   {
4       long int i;
5       long int length = vec_length(v);
6       long int limit = length-1;
7       data_t *data = get_vec_start(v);
8       data_t acc = IDENT;
9
10      /* Combine 2 elements at a time */
11      for (i = 0; i < limit; i+=2) {
12          acc = (acc OP data[i]) OP data[i+1];
13      }
14
15      /* Finish any remaining elements */
16      for (; i < length; i++) {
17          acc = acc OP data[i];
18      }
19      *dest = acc;
20  }
```

循环展开能减少循环开销的影响

还是以我们之前讲到的向量为例：

我们假设有集合a = {1,2,4,5,7,9,10,12,16}使用combine函数进行求和运算：我们模拟计算机的执行顺序，一步步在草图上分析combine5代码实现的功能。



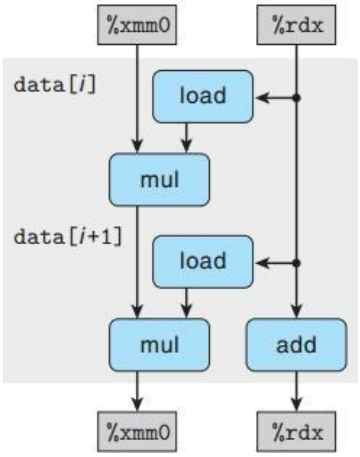
k=2循环展开两次，字太丑见谅

总结：我们之所以将limit定义为length-1，是向量的长度不一定是2的倍数，如上图中的9，为了不至于在第一次循环中越界访问，我们将limit设置为length-1.虽然我们用到了两次循环，但循环展开大大缩短的关键路径，提高了效率。我们将这个思想归纳为循环展开k次，k < length + 1 ，我们上面讲的内容就是k=2次，一次计算2个元素的和。我们看看效率的提升：

函 数	页码	方 法	整数		浮点数		
			+	*	+	F*	D*
combine4	338	无展开	2.00	3.00	3.00	4.00	5.00
combine5	349	展开 2 次	2.00	1.50	3.00	4.00	5.00
		展开 3 次	1.00	1.00	3.00	4.00	5.00
延迟界限			1.00	3.00	3.00	4.00	5.00
吞吐量界限			1.00	1.00	1.00	1.00	1.00

浮点运算无变化

注：为什么浮点运算的没有性能的提升？虽然展开了两次循环，但是必须要顺序的执行，所以没有性能的提升。



两次运算展开后是顺序执行的

1.6 进一步优化：提高并行性（combine6、combine7）

分析：我们将累积变量放在一个单独的acc中，在前面的计算完成前，不能计算新的acc值

```
1  /* Unroll loop by 2, 2-way parallelism */
2  void combine6(vec_ptr v, data_t *dest)
3  {
4      long int i;
5      long int length = vec_length(v);
6      long int limit = length-1;
7      data_t *data = get_vec_start(v);
8      data_t acc0 = IDENT;
9      data_t acc1 = IDENT;
10
11     /* Combine 2 elements at a time */
12     for (i = 0; i < limit; i+=2) {
13         acc0 = acc0 OP data[i];
14         acc1 = acc1 OP data[i+1];
15     }
16
17     /* Finish any remaining elements */
18     for (; i < length; i++) {
19         acc0 = acc0 OP data[i];
20     }
21     *dest = acc0 OP acc1;
22 }
```

两次循环展开，使用两路并行

性能对比图：

函 数	页码	方 法	整数		浮点数		
			+	*	+	F*	D*
combine4	338	累积在临时变量中	2.00	3.00	3.00	4.00	5.00
combine5	349	展开 2 次	2.00	1.50	3.00	4.00	5.00
combine6	352	2 次展开，2 路并行	1.50	1.50	1.50	2.00	2.50
延迟界限			1.00	3.00	3.00	4.00	5.00
吞吐量界限			1.00	1.00	1.00	1.00	1.00

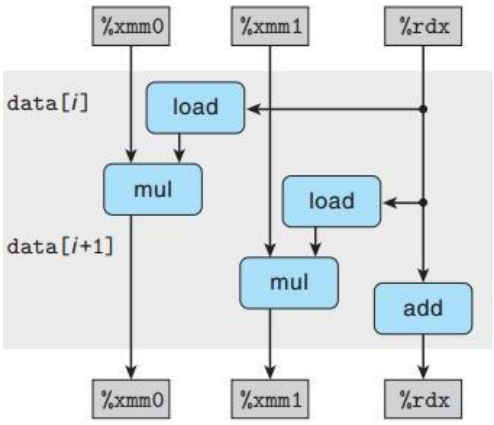
怎样理解combine6带来的性能提升：

我们看到唯一不同的地方在与循环①（标号12）处代码中，加入两个累积变量：acc0和acc1，这样做有什么好处呢？

> acc = (acc OP data[i]) OP data[i+1]; 转变为：
>
> acc0 = acc0 OP data[i]; 和 acc1 = acc1 OP data[i+1];

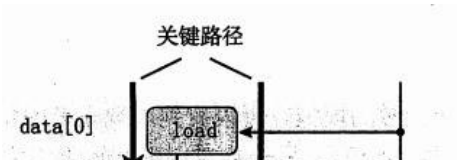
带入分析图（combine6）

我们再来看看图形化的数据分析：



引入新变量acc0和acc1分配到不同寄存器寄存器xmm0和xmm1

这样一来关键路径就成了两路并行，效率大大提升了：



cimbine6的关键路径

还有没有其他方法能打破顺序相关而提高效率？来看看combine7变种：

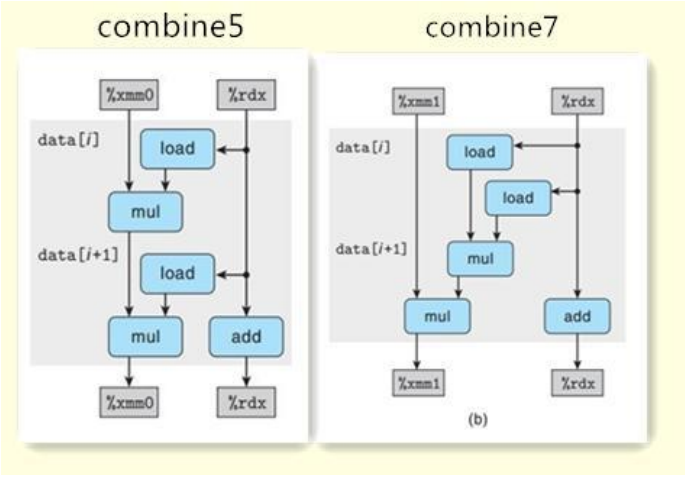
```
1   /* Change associativity of combining operation */
2   void combine7(vec_ptr v, data_t *dest)
3   {
4       long int i;
5       long int length = vec_length(v);
6       long int limit = length-1;
7       data_t *data = get_vec_start(v);
8       data_t acc = IDENT;
9
10      /* Combine 2 elements at a time */
11      for (i = 0; i < limit; i+=2) {
12          acc = acc OP (data[i] OP data[i+1]);
13      }
14
15      /* Finish any remaining elements */
16      for (; i < length; i++) {
17          acc = acc OP data[i];
18      }
19      *dest = acc;
20  }
```

combine7重新结合变换

标号12的语句中与combine5相比，只是结合方式发生了变化，将：

acc = (acc OP data[i]) OP data[i+1] **变成了** acc = acc OP （data[i] OP data[i+1])

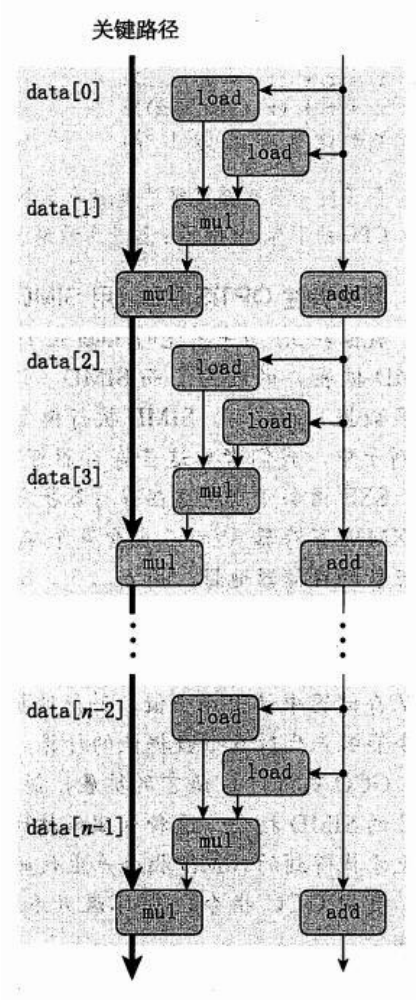
我们来对比一下combine5和combine7两个版本的数据流图：



数据流图对比

在combine7的版本中，第一个mul通过两个load指令将i和i+1的乘机计算出来，然后交给第二个mul将乘积累积到xmm1（acc）中。就不像combine5中的load mul顺序执行，必须等到第一个

load mul执行完成以后才能进行第二次load和mul操作。我们将combine7的模型复制几次就能看的关键路径变成了n/2个操作，这就带来了性能的提升：



combine7：我们看到只有一条关键路径，而且包含n/2个操作

总结：到目前为止，我们已经完成了从combine1到combine7的进化，带来了至少10倍以上的效率提高，我们发现**循环展开**、**并行累积值**在多个变量中，是可靠的提高程序性能的方法。那还有那些限制因素呢制约着程序的性能呢?

一些限制因素：

① **寄存器溢出：**当我们的并行度超过了可用的寄存器数量，编译器就会将结果溢出到栈中，性能就会急剧下降，笔记访问存储器的时间要长很多。

② **避免分支预测和预测错误处罚：**1> 不要过分关心可预测的分支，因为带来的性能差异很小；2> 书写适合用条件传送实现的代码。

1.7理解存储器性能

分析：为什么要理解存储器的性能?

当我们要处理的数据小于1000个元素的向量，数据量不会超过8000个字节，这些内容都会存放在多个高速缓存存储器中，已方便我们快速访问。接下来我们会研究在高速缓存中的加载和存储操作对性能的影响，充分利用高速缓存来编写高效率的代码

加载的性能：

我们在对链表的访问中，可以看出加载函数对性能的影响，举个例子：

加载操作的延迟

我们来看看ls = ls->next这句的汇编代码：

```
   len in %eax, ls in %rdi
1   .L11:                   loop:
2     addl    $1, %eax        Increment len
3     movq    (%rdi), %rdi    ls = ls->next
4     testq   %rdi, %rdi      Test ls
5     jne     .L11            If nonnull, goto loop
```

movq是这个循环的关键瓶颈

在标号3中，使用movq指令，加载值到rdi寄存器中，而加载操作又依赖于rdi来计算加载的位置，也就是说，必须要等到前一次加载完成才能进行下一次循环。这个函数的CPE等于4也就是说加载的延迟为4.

存储的性能：

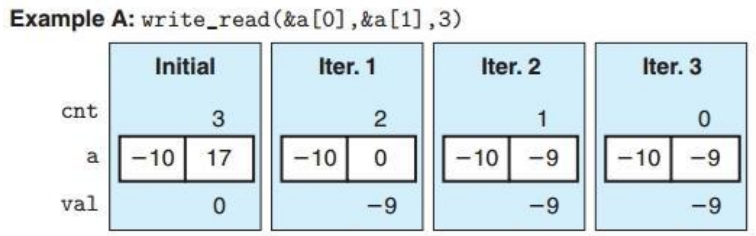
分析：从理论上来讲，存储操作并不影响任何寄存器的值，不会产生任何数据相关。而只有加载操作是受存储的影响的，因为只有加载操作读取的是有存储器写操作的值。

写读的相关性：

```
1   /* Write to dest, read from src */
2   void write_read(int *src, int *dest, int n)
3   {
4       int cnt = n;
5       int val = 0;
6
7       while (cnt--) {
8           *dest = val;
9           val = (*src)+1;
10      }
11  }
```

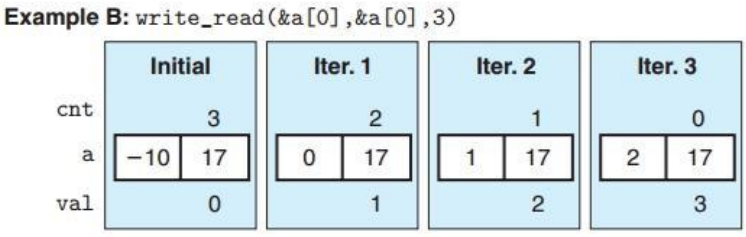
写读相关

为了讨论写和读的相关性，我们来看看上述代码的两种不同的情况：假设a[0]=-10,a[1]=17:



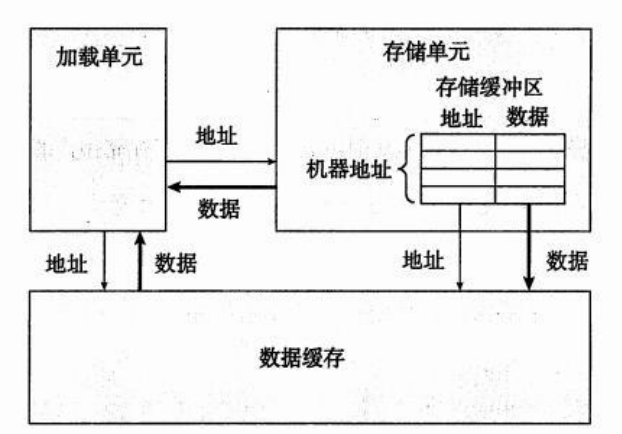
互不相干的情况CPE=2

举例A中可以看出，初始化的条件下a{-10，17}，val = 0， cnt = 3；当程序开始执行循环的时候，写操作：*dest = val 而读操作：val = （*src）+1访问的分别是不同位置，a[0]和a[1]。就是我们前面说过的数据不相关。



读写相关CPE=6

而举例B的情况就完全不一样了，dest和src操作的是同一块位置a[0]，一个存储器读的结果依赖于一个最近的存储器的写。为什么读写相关以后程序的性能就降低了？我们来看看加载和存储单元的内部构造：



加载和存储单元的细节

在上图的内部构造中，我们发现存储单元多了一个存储缓冲区，这样做有一个好处就是，当一些列存储操作开始执行的时候，不需要等待高速缓存更新完成就能够开始执行了。而当一条加载操作发生的时候，加载单元必须先检查存储缓冲区，看看有没有相匹配的条目，如果匹配成功就从存储缓冲区中取出数据作为加载的结果。

内循环数据流图

上图的内容有点儿乱，我们来说明一下：

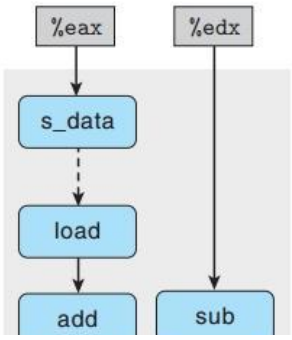
①指令movl被译码成两个操作：s_addr和s_data其中前者负责计算存储器的地址，并在存储缓冲区中创建一个条目，设置地址字段；后者负责设置该条目的数据字段；

②s_addr同s_data右边的箭头表示，设置数据字段必须要等到计算地址阶段完成才能进行。此外，第二条movl指令被译码成了load指令，这条加载指令必须要检查所有的地址，包括正在读写的地址，所以s_addr与这条load指令也有相关性；还有一条虚线相关性，连接s_data和load，这表示如果两个地址相同，那么load必须要等到s_data将数据段设置到存储缓冲区。我们将上图修改一下，大家容易理解：

读存数据相关流图

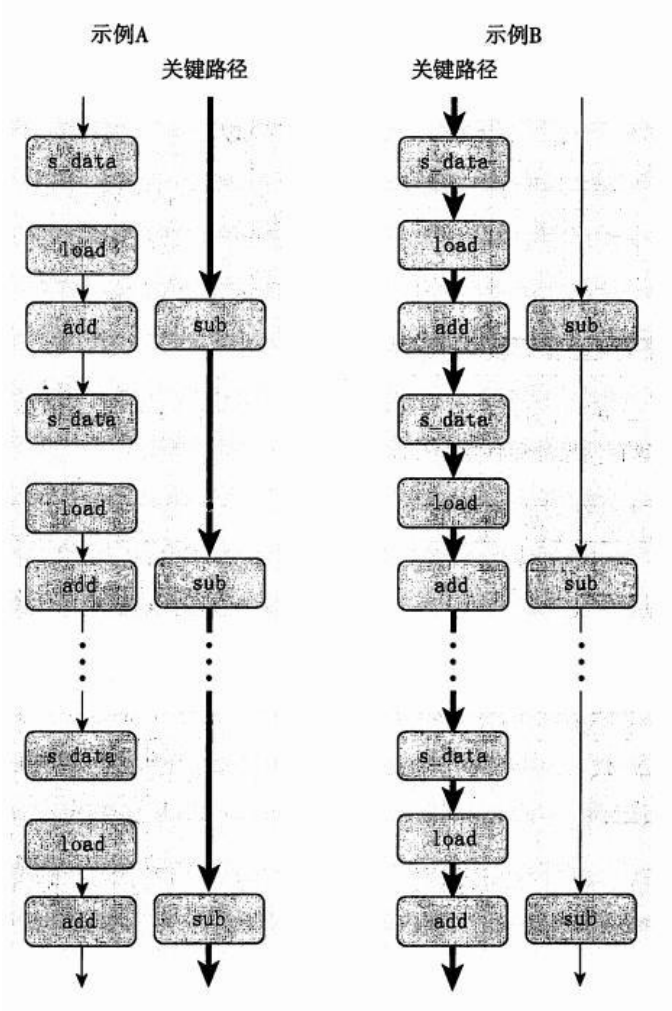
> 标号①表示：存储地址必须在数据被存储之前计算出来；
>
> 标号②表示：load操作将它的地址与所有未完成操作的地址进行比较；
>
> 标号③表示：数据相关，当访问相同位置时出现

我们剔除不影响数据操作的关联后形成如下图：



读写相关

我们清楚的看到了两条的关键路径，其中左边表示的是：存储加载相关；右边表示的是增加数据值相关。将以上图复制几次，并同不相关的数据进行比较，我们能看到读写相关对CPE的影响了：



左边地址不同，右边相同

总结：对存储器的操作，只有当加载和存储地址都被计算出来了以后才能确定其对性能的影响。

到目前为止，我们基本上上讲完了所有的优化程序性能的方法，套路如下：

① 高级设计：选择适当的算法和数据结构，要提高警惕，避免渐近低效率；

② 基本编码原则：

*消除连续的函数调用，有可能的时候将计算移动到循环体外；

*消除不必要的存储器引用，引入临时变量保存中间值。

③ 低级优化：

*展开循环，降低开销；

*提高并行，使用多个累积变量或者重新结合，用良好的风格重新条件操作。

在实际的优化程序的过程中，我们不可能像之前讲到的简单程序那样，快速的分析出一个程序片段的性能瓶颈，毕竟在真实的项目中源码的量相当大，这时候我们有必要运用一些软件来分析程序的性能瓶颈，Unix系统中有一个GPROF可以实现相关分析，在此不做讲解了。

> 备注：（Amdahl定律）Gene Amdahl曾经发现过一个很有意思的现象，最后以Amdahl命名了这个定律，大意就是：当我们选择提高某个系统的效率的时候，被改进的这一部分效率，对整体的影响，在于被改进部分到底有多重要（听起来像废话）。我们来举个例子，加入一个软件的耗时分为Told=（1+6+3）=10我们将6这个部分的效率提高了3倍，变成了Tnew=（1+2+3）=6。整个系统的加速还是不大。这就是Amdahl定律要告诉我们的主要观点，要想获得整体性能的提升，我们必须要提高很大一部分系统的速度。单靠一个方面是不行的。

唐鱼的学习探索 如果我像一般人一样读那么多书，我就跟他们一样愚蠢了。
总资产29 (约2.81元) 共写了10.4W字 获得530个赞 共463个粉丝
关注

写下你的评论...

全部评论 2 只看作者 按时间倒序 按时间正序

倪修一
2楼 2018.12.21 16:26

java语言需要学习这么复杂的底层基础吗?还是说只是C语言需要

赞 回复

唐鱼的学习探索 作者
2018.12.26 08:52

推荐看这个：https://zhuanlan.zhihu.com/p/19959253

回复

添加新评论

被以下专题收入，发现更多相似内容

操作系统 《深入理解计算...

推荐阅读

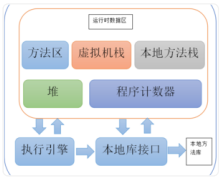
更多精彩内容 >

Android - 收藏集

Android 自定义View的各种姿势1 Activity的显示之ViewRootImpl详解 Activity...

passiontim 阅读 135,747 评论 17 赞 578

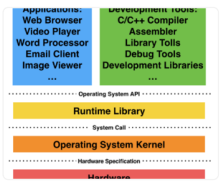
《深入理解Java虚拟机》笔记_第一遍



《深入理解Java虚拟机》笔记_第一遍 先取看完这本书（JVM）后必须掌握的部分。 第一部分 走近 Java 从传...

xiaogmail 阅读 2,592 评论 1 赞 29

iOS 程序员的自我修养 — 读《程序员的自我修养-链接、装载...

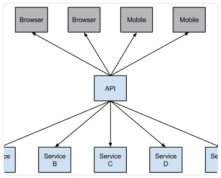


2016年国庆假期终于把此书过完，整理笔记和体会于此。 关于书名 书名源于俄罗斯的演员斯坦尼斯拉夫斯基创作的《演员...

李剑飞的简书 阅读 4,249 评论 1 赞 53

Spring Cloud

Spring Cloud为开发人员提供了快速构建分布式系统中一些常见模式的工具（例如配置管理，服务发现，断路器，智...



卡卡罗2017 阅读 84,405 评论 14 赞 122

我很快乐

每天我都会觉得自己很快乐，不为什么，只是因为我在最正当的年纪做了自己最喜欢做的事情，所以每一天都是新的一天，每...

糖酪雅 阅读 20 评论 0 赞 0