优化程序性能

Introduction to Computer Systems
Oct. 31st

叶炜宁

Today

- **编译器的优化能力**
- 编译器优化的局限性
- 提高程序性能的方法

编译器的优化能力

- Code Motion
  - 减少一些相同结果的重复的计算。尤其是将循环中相同的计算移到循环外面。

```
void set_row(double *a, double *b,
   long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}
```

```
    long j;
    long ni = n*i;
    double *rowp = a+ni;
    for (j = 0; j < n; j++)
        *rowp++ = b[j];
```

编译器的优化能力

- Reduction in Strength
  - 将一些比较慢的运算符替换成快的。
  - 比如用左移右移和加法去替换乘法和除法。
- Share Common Subexpressions
  - 重复使用一部分表达式。

```
/* Sum neighbors of i,j */
up =    val[(i-1)*n + j  ];
down =  val[(i+1)*n + j  ];
left =  val[i*n     + j-1];
right = val[i*n     + j+1];
sum = up + down + left + right;
```

```
long inj = i*n + j;
up =    val[inj - n];
down =  val[inj + n];
left =  val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

Today

- 编译器的优化能力
- **编译器优化的局限性**
- 提高程序性能的方法

编译器优化的局限性

- 编译器必须很小心的对程序只使用安全的优化，对于程序可能遇到的所有情况，要让优化后的程序和未优化的程序有一样的行为。
- 有两个妨碍优化的因素：内存别名使用，函数调用。

内存别名使用

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

在以上程序中，编译器并不会将twiddle1优化成twiddle2，如果xp和yp指向内存中同一个位置，那么两个过程跑出来的结果是不一样的。编译器会假设xp和yp可能会相等，因此不做优化。

函数调用

```
1  long f();
2
3  long func1() {
4      return f() + f() + f() + f();
5  }
6
7  long func2() {
8      return 4*f();
9  }
```

```
1  long counter = 0;
2
3  long f() {
4      return counter++;
5  }
```

以上左边程序，看上去func1会被优化成func2，但考虑 f( )是右边情形，即对一个全局变量做了修改，执行func1和func2是完全不同的。而编译器并不会去判断函数有没有副作用，而是假设最糟的情况，保持调用不变。

内联函数

- 包含函数调用的代码可以用内联函数替换的方法优化。
- 内联函数的作用是将函数调用替换为函数体，即把整个函数体放在调用处。
- 这样的好处是减少了函数调用的开销，并且也可以对展开的代码做进一步的优化。

Today

- 编译器的优化能力
- 编译器优化的局限性
- **提高程序性能的方法**

提高程序性能的方法

- 表示程序性能
  - 每元素的周期数，CPE
  - 我们用这个作为一种我们程序性能的度量。
- 与机器无关
  - 消除循环的低效率
  - 减少过程调用
  - 消除不必要的内存引用
- 与机器有关
  - 理解现代处理器
  - 循环展开
  - 提高并行性

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

```
1   /* Implementation with maximum use of data abstraction */
2   void combine1(vec_ptr v, data_t *dest)
3   {
4       long i;
5
6       *dest = IDENT;
7       for (i = 0; i < vec_length(v); i++) {
8           data_t val;
9           get_vec_element(v, i, &val);
10          *dest = *dest OP val;
11      }
12  }
```

函数	方法	整数		浮点数	
		+	*	+	*
combine1	抽象的未优化的	22.68	20.02	19.98	20.18
combine1	抽象的-O1	10.12	10.12	10.17	11.14

消除循环的低效率

```
1   /* Move call to vec_length out of loop */
2   void combine2(vec_ptr v, data_t *dest)
3   {
4       long i;
5       long length = vec_length(v);
6
7       *dest = IDENT;
8       for (i = 0; i < length; i++) {
9           data_t val;
10          get_vec_element(v, i, &val);
11          *dest = *dest OP val;
12      }
13  }
```

由于编译器不会优化函数调用，我们需要手动将vec_length移到循环外面。

函数	方法	整数		浮点数	
		+	*	+	*
combine1	抽象的-O1	10.12	10.12	10.17	11.14
combine2	移动 vec_length	7.02	9.03	9.02	11.03

减少过程调用

```
1   /* Direct access to vector data */
2   void combine3(vec_ptr v, data_t *dest)
3   {
4       long i;
5       long 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  }
```

combine3将每一次调用get_vec_element变成了调用一次get_vec_start，但我们发现结果并不理想，主要是因为循环内的其他操作形成了瓶颈，限制性能超过了调用get_vec_element。

函数	方法	整数		浮点数	
		+	*	+	*
combine2	移动 vec_length	7.02	9.03	9.02	11.03
combine3	直接数据访问	7.17	9.02	9.02	11.03

消除不必要的内存引用

```
1   /* Accumulate result in local variable */
2   void combine4(vec_ptr v, data_t *dest)
3   {
4       long i;
5       long 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  }
```

对于combine3，每一次OP都要访问dest地址的内存一次并且写入一次，非常慢，combine4把结果先放在临时变量中最后再写入内存。

函数	方法	整数		浮点数	
		+	*	+	*
combine3	直接数据访问	7.17	9.02	9.02	11.03
combine4	累积在临时变量中	1.27	3.01	3.01	5.01

理解现代处理器

- 超标量处理器
  - 可以在每个时钟周期执行多个操作。
  - 乱序：指令执行的顺序不一定要与在机器级程序中的顺序一致。
  - 乱序处理器需要更大更复杂的硬件，但可以更好地达到更高的指令级并行度。
- Haswell CPU
  - 8个功能单元
  - 4个单元能执行整数操作（加法，位级操作和移位等）
  - 2个单元能执行加载操作
  - 2个单元能执行浮点乘法
  - 浮点加法和整数乘除都只有1个单元

功能单元的性能

- 刻画运算的数值
  - 延迟：表示完成运算所需要的总时间
  - 发射时间：表示两个连续的同类型的运算之间需要的最小时钟周期。发射时间为1，表示可以被完全流水线化。
  - 容量：能够执行该运算的功能单元的数量
  - 延迟界限，当一系列操作必须按照严格顺序执行时，就会遇到延迟界限。
  - 吞吐量界限，这是程序性能的终极限制。

运算	整数			浮点数		
	延迟	发射	容量	延迟	发射	容量
加法	1	1	4	3	1	1
乘法	3	1	1	5	1	2
除法	3 ~ 30	3 ~ 30	1	3 ~ 15	3 ~ 15	1

界限	整数		浮点数	
	+	*	+	*
延迟	1.00	3.00	3.00	5.00
吞吐量	0.50	1.00	1.00	0.50

循环展开

- **循环展开从两方面改进性能**
  - 第一，减少了不直接有助于程序结果的操作的数量，例如循环索引计算和条件分支。
  - 第二，它提供了一些方法，可以进一步变化代码，减少整个计算中关键路径上的操作数量。

```
1   /* 2 x 1 loop unrolling */
2   void combine5(vec_ptr v, data_t *dest)
3   {
4       long i;
5       long length = vec_length(v);
6       long 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  }
```

循环展开后，CPE均接近于延迟界限，这得益于减少了循环开销操作，此时，整数加法的一个周期的延迟成为了限制性能的因素。

函数	方法	整数		浮点数	
		+	*	+	*
combine4	无展开	1.27	3.01	3.01	5.01
combine5	2×1 展开	1.01	3.01	3.01	5.01
	3×1 展开	1.01	3.01	3.01	5.01
延迟界限		1.00	3.00	3.00	5.00
吞吐量界限		0.50	1.00	1.00	0.50

提高并行性

- 加法和乘法的功能单元是完全流水线化的，理论上可以每个时钟周期一个新操作，并且有些操作可以被多个功能单位执行。
- 但之前我们都将值放在一个单独的变量acc中，在前面的计算完成之前，都不能计算acc的新值。我们要打破这种顺序相关。

2*2循环展开

```
1   /* 2 x 2 loop unrolling */
2   void combine6(vec_ptr v, data_t *dest)
3   {
4       long i;
5       long length = vec_length(v);
6       long 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  }
```

对于可结合可交换的合并运算，我们可将其分成两个部分，奇数项和偶数项分开计算，分别存在acc1和acc0中。这样计算acc1和acc0是独立的，我们可以两路并行。因此CPE理论可以达到延迟的一半，唯一例外是整数加法，主要还是太多的循环开销。

函数	方法	整数		浮点数	
		+	*	+	*
combine4	在临时变量中累积	1.27	3.01	3.01	5.01
combine5	2×1 展开	1.01	3.01	3.01	5.01
combine6	2×2 展开	0.81	1.51	1.51	2.51
延迟界限		1.00	3.00	3.00	5.00
吞吐量界限		0.50	1.00	1.00	0.50

2*1a循环展开

```
1   /* 2 x 1a loop unrolling */
2   void combine7(vec_ptr v, data_t *dest)
3   {
4       long i;
5       long length = vec_length(v);
6       long 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  }
```

在执行当前这步时，我们可以提前执行下一步括号内的项，这样CPE还是可以达到延迟的一半。

函数	方法	整数		浮点数	
		+	*	+	*
combine4	累积在临时变量中	1.27	3.01	3.01	5.01
combine5	2×1 展开	1.01	3.01	3.01	5.01
combine6	2×2 展开	0.81	1.51	1.51	2.51
combine7	2×1a 展开	1.01	1.51	1.51	2.51
延迟界限		1.00	3.00	3.00	5.00
吞吐量界限		0.50	1.00	1.00	0.50

一些限制因素

- **寄存器溢出**
  - 如果我们的并行度p超过了可用的寄存器数量，编译器会将某些临时值存放到内存中，通常是在运行时堆栈上分配空间。这样CPE反而会变差。
- **分支预测**
  - 一般采用的行为是，向上的分支一般是循环，所以预测采用。向下的分支一般是if语句，预测不采用。
  - 书写适合用条件传送实现的代码。如果编译器能够产生使用条件数据传送而不是使用条件控制转移的代码，可以极大地提高程序的性能。