定浮点数据算术及其优化

石学林　张兆庆　武成岗
（中国科学院计算技术研究所　中国科学院研究生院　北京100080）

摘　要　定点算法对于商业计算非常重要，但由于成本和功耗的限制，某些嵌入式芯片尚不能提供浮点部件。针对这些情况，我们必须寻求支持整数ALU的十进制定浮点算数的解决方案。本文提出一种新的基于十进制编码的长整数方法以进行小数运算。实验表明我们的数据模型和算法其性能优于Java BigDecimal，在实际商业应用中获得了20%加速。

关键词　十进制算法，定浮点优化，BigDecimal

Optimizing Decimal Fixed and Floating Point Arithmetic for Computers

SHI Xue-Lin　ZHANG Zhao-Qing　WU Cheng-Gang

(Institute of Computing Technology, Chinese Academy of Sciences, Graduate School of the Chinese Academy of Sciences, Beijing 100080)

Abstract　Fixed point arithmetic is very important for business computing. And also some embedded chips do not provide float unit because of the limit of cost and power. Under both these conditions, we must find a solution to support decimal fixed or float point arithmetic on Integer ALU. This paper presents a new approach based on decimal encoded long integer to do decimal arithmetic. Experiments show that our data model and algorithms is superior to Java BigDecimal in performance. We get a speedup of 20% in real commercial applications.

Keywords　Decimal arithmetic, Fixed-point, Floating-point optimization, BigDecimal

1　引言

早期的计算机系统采用各种各样的数据格式和算法实现了浮点部件以支持科学计算程序，比如20世纪50年代的Perkins[1]，60年代的Jones和Wymore[2]，70年代的Chen[3]等。但是这些采用不同字长和不同数据范围的浮点数据的计算机系统却很难互联互通，同时也给程序的移植增加了不必要的复杂性。随着IEEE754[4]和IEEE854[5]二进制浮点标准的发布，今天大多数通用计算机系统都实现了标准的浮点部件，但是这些浮点数据通常采用了$(-1)^{sign} * coefficient * 2^{exponent}$的方式，比如IEEE754规定的单精度格式为符号位占一个比特，系数占23比特，指数占8比特，这种数据格式的特点使它并不能很好地支持定点计算。最近的文献包括Cowlishaw等[6,7]提出的一个十进制浮点规范。根据统计Tsang[8]，大约有55%的数据库表字段采用了定点数据类型，同时还有43.7%的整数数据类型同样可以存储为定点数据类型[9]。大多数程序设计语言都通过基本数据类型或用户定义类型的方式实现了定点数据类型，比如Ada，PL/I，SQL NUMERIC，COBOL，C#，Java BigDecimal。由于缺乏硬件的直接支持，这些数据类型通常都通过软件库的方式提供支持。同时在嵌入式芯片领域，受到成本和功耗制约一般都不提供浮点部件，在这些系统中如果需要使用小数，也需要通过软件形式加以支持。但是各种定浮点编码格式以及算法的多样性使得正确选择适合的数据模型支持不同应用要求变得困难。

本文通过对小数定浮点特性的深入分析，指出了在各种应用要求限制下的选择策略，并且对多精度的定浮点运算给出了以十进制编码系数的数据模型及其详细的算法。同时我们将此数据模型和算法运用在Cobol2Java翻译系统中，将实际的事务处理程序性能提高20%左右。本文第2部分首先对影响定浮点数据实现方便性和性能的各种参数进行了分析，然后在第3部分详细地描述了十进制编码的数据模型及其算法，第4部分给出了实验结果，最后进行了总结。

2　设计空间选择分析

2.1　数据宽度

每一个有限的有理数都可表示为：$V=(-1)^{sign} * M * B^E$，我们把数据宽度定义为用来对整系数M进行编码的比特数。我们知道对一个a位整数，b位小数的十进制有限有理数，根据香农的信息编码理论，在不丢失任何有效数字的情况下，至少需要n位二进制位表示，其中$n=ceil((a+b) * \log_2 10)$，$ceil(x)$表示超过$x$的最小整数。

在大多数缺乏浮点部件支持的计算机系统中，如果需要进行小数计算，那么首先必须以一定的格式将小数数据编码成整数，然后用整数ALU部件完成计算。因此如果我们能够把整系数M直接二进制编码到ALU部件的寄存器，那么每一个小数类型的算术运算都有可能只需要一个对应的整数指令完成，从而实现非常高效的计算。如果整系数可以用机器字长编码并且可以直接用ALU部件进行计算，那么称这种数据模型为单精度的，否则称为多精度数据模型。

对于多精度数据来说，由于其数据宽度超过机器字长，那么隐含着需要进行长整数算术操作。通常程序语言对这类长整数的算术操作的支持都是不能满足需求的。比如C语言所能支持的最长的整数为long long类型。一般来说在32位平台上，long long类型仅有64位二进制数据宽度，也就是说超过64位宽度的数据操作是不能直接用两个long long类型的操作直接实现的。更不幸的是即使两个操作数都是long long类

石学林　博士研究生，研究方向为高级编译技术。张兆庆　研究员，主要研究领域为高级编译技术。武成岗　博士，主要研究领域为高级编译技术，二进制翻译技术。

型，不同的编译器根据目标机的硬件情况对 long long 类型所能支持的操作都是极其有限而不完整的，在这种情况下采用一个数组来表示所有有效数字将是不可避免的。

2.2 定浮点特性

顾名思义，所谓定点数据指的是数据模型中指数部分是一个固定不变的整数，也就是说小数点的位置是固定不动的。而浮点数据恰好相反，其指数是一个可变的整数，因此意味着小数点的位置可以左右移动。由于我们总是可以固定整数部分的长度，而将可变部分划分至小数部分，因此只要在数据模型中，数据的小数部分是一个固定长度的序列，那么整个数据都可以有一个确定的长度。换句话说，定点数据通常都可以有一个确定的长度，而浮点数据的长度则是不确定的。在IEEE754标准中的单精度浮点类型，因为它采用了规格化数据，在数据宽度超过其规定的整系数长度时采用了舍入处理，因而其数据长度可以保持不变。很明显这样的处理丢失了数据的有效数字，因此也成为运算结果失真的一个重要原因，但是固定长度的数据模型却是硬件实现的一个基本条件。在软件实现中，我们可以采用的存储在长度上比定长的寄存器灵活得多，可以采用按需分配存储的方案，也就是说按数据实际有效数字的长度分配存储空间，这样在提高空间性能的同时也改善了时间性能。

2.3 分辨率要求

另一方面我们也可将一个有限有理数表示为：

$$\begin{aligned}V &= a_{n-1}B^{n-1}+a_{n-2}B^{n-2}+\cdots+a_1B+a_0+a_{-1}B^{-1}+\cdots+a_{-m}B^{-m}\\ &= a_{n-1}B^{n-1}+a_{n-2}B^{n-2}+\cdots+a_1B+a_0+B^{-m}(a_{-1}B^{m-1}+\cdots+a_{-m})\\ &= a_{n-1}B^{n-1}+a_{n-2}B^{n-2}+\cdots+a_1B+a_0+B^{-m}M\end{aligned}$$

其中 $B^i(i\in\{-m,-m+1,\cdots,n-1\})$为各数据位所对应的权重，$B^{-m}M$ 为数据模型的小数部分，它所能表达的数据集合构成了一个从零开始的由$0,1/B^m,2/B^m,\cdots,B^m-1/B^m$ 等 B^m 个元素的等差数列，其等差因子为$1/B^m$。这个等差因子标识了一个数据模型所能表示的数据最小分辨率。任何小于这个因子的小数其他部分，由于受到编码长度的限制都不可能成为有效数字编码到整系数中，也就是说这个数据模型不能区分比这个因子更小的数值，因而将其视为零。比如一个具有8位十进制小数的数据模型，其分辨率为10^{-8}，也就是说小数点后的第8位以后的数据都将被视为零。同样一个8位二进制小数的数据模型的分辨率为2^{-8}，也就是说1/256，十进制表示为0.00390625。不幸的是在实践中，人们普遍习惯于使用十进制，比如我们经常见到的“保留到小数点后第4位”等类似的需求，实质上要求数据模型的分辨率是0.0001。如上例中8位二进制小数的十进制分辨率只有0.01，因为其等差因子为0.00390625，并不能表达0.001这样的数据。说分辨率为0.01，并不总是意味着我们可以精确地表达0.01，从下文的分析可以看出，这与数据模型的编码方案有着密切的关系。

2.4 编码方案选择

一种直观的编码方法，就是将数据的整数和小数分别编码为两个整数，我们称为两段式编码。首先考虑两段采用不同进制进行编码，比如整数部分采用二进制编码，小数部分采用十进制编码或相反。如果进行运算的数据模型是单精度数据模型，可以考虑采用 ALU 部件直接进行算术运算，但是通常 ALU 部件都是二进制编码，因此运算前需要将非二进制编码部分转换成相对应的二进制整数才能进行运算，从而增加不必要的开销。对于多精度数据模型，会造成不必要的乘法困难。因此实践中，一般不采用这种混合进制的编码。再次考虑全部采用十进制编码，对于单精度数据模型，这是一种无必要的数据编码，因为不能和硬件编码方式一致，导致无谓的进制转换，从而增加时间开销，造成性能下降。对于多精度数据模型，我们发现它和一段式编码完全相同，在下一节将详细讨论。最后如果全部采用二进制编码，对于单精度模型，在满足分辨率要求的情况下，可以实现一定精度的定点运算。比如一个要求有四位小数的数据模型，可考虑将四位小数编码到14位二进制序列中。在进行算术运算时，对小数部分和整数部分分别进行。

另一种编码方法就是将所有的有效数字看作是一个整数，另外记住小数点位置即可。对于定点数据模型，由于所有的数据都具有相同的小数点位置，因此可以省略。由公式可以看出由于二进制小数编码中，小数位的权重都是1/2的幂次，这使得十进制的小数在这种编码中不能得到精确的表示。由数学知识可知，如果需要精确的表示一个十进制小数，需要一个以1/2为等比因子的无限级数表示，这意味着需要无限个二进制位存储这个小数部分，也就是说需要无限的存储空间，同时也意味着操作这种数据也需要无限长的时间。这种时空开销在实践中是不可行的，因此实际的二进制编码都作了不同程度的近似，比如 IEEE754。因此在需要作精确计算时，我们都需要采用10的幂次作为权重。这样每一个有限的有理数就可以表示为：$V=(-1)^{sign}*M*10^{-E}$。那么在多精度下，整系数 M 应该用二进制编码还是用十进制编码呢？由2.2节定浮点特性可知，在保证运算结果不失真的情况下，浮点编码的时空性能优于定点编码，因此这里仅限于讨论浮点数据。考虑两个参加加减法的操作数，由于都是浮点数据，因此如果两个操作数的指数不同，也就是小数点位置不同，依据算术运算规则需要首先对齐小数点，相当于将指数较小的那个操作数的整系数乘以$10^{|e1-e2|}$这个指数因子。这一点在十进制编码中，可以轻松通过移位操作实现，首先分配相应的存储空间，然后从左至右以四个比特为单位移到相应位置即可。但是对于二进制编码，就只能通过多精度乘法操作来实现了。如果这个指数因子超过单精度表示范围，那么将这个指数因子转换成对应的长整数将是不可避免的。一个避免动态转换的方法是预先转换好部分数据放在表格中备查，但是这样对于支持任意精度的计算是不合适的。

对于多精度除法操作，一般都采用类似于手工算术的长除法一次得到商的若干位，因为不可能一次得到商的所有数据位。这种算法采用猜商的算法，首先得到一个部分商，然后与除数做乘法，再和部分被除数做减法，取被除数剩下若干个有效数字组成一个新的被除数继续作除法。在被除数的所有有效数字用尽后，如果继续要求除到一定位的小数，需要在余数后补零组成新的被除数继续作除法，这个过程一直持续到得到指定的小数位截止。如果采用二进制编码，那么每次得到的部分商都是一个二进制编码的整数，因此我们并不能直观得到所对应的十进制整数的位数，需要用额外的算法才能知道当前所对应的十进制位数。而这些算法通常都是较为昂贵的，而且比较复杂，不利于实现。反观十进制编码，我们可以看到对于每次得到的部分商，其十进制位数都是一目了然的，便于我们控制除法的迭代过程。

对于十进制编码的空间利用率，n 个十进制有效数字需要 $f(n)=(n+1)/2$个字节，相对应的二进制编码则需要 $g(n)=(ceil(n\log_2 10)+8)/8$个字节，其中 $ceil(x)$表示超过 x 的最小整数。可以证明在 $n<=10$时，$f(n)=g(n)$，$n>10$时，$f(n)>g(n)$。但是即使 n 趋向无穷时，$f(n)/g(n)$都不超过$4/\log_2 10=1.204$。也就是说十进制编码至多比二进制编码浪费

20.4%左右的空间。考虑实践中，比如 Cobol 标准要求36位有效数字，其中18位整数和18位小数，这时的 $f(36)=18, g(36)=16$，也就是说空间占用仅比二进制编码多12.5%。另外系统中实际参加运算的操作数还有相当部分都不会有10个有效数字以上，因此实际浪费的空间比12.5%还要低。

3 多精度数据模型及其算术优化

3.1 数据存储模型

运用面向对象的语言来描述数据模型是一个很好的方式，如图1所示。

```
-12.345 = -1 * 12345 * 10^-3

class FixNum{
   int      signum;          -1
   byte[]   coefficient;     0x012345
   int      scale;           3
}
```

图1 数据模型示意图

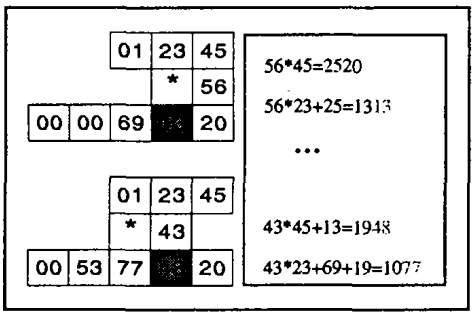

图2 乘法示意图

每一个多精度数据可以用三个属性来描述，分别为符号、数据位序列和小数点位置。其中符号用一个表示有符号的整数表示，称为符号数。整个数据的值大于零时取值+1，小于零时取值-1，数据值为零时，符号数也取值为零。数据位序列在存储中用一个字节数组表示，每一个数据位单独编码，按照大尾存储方式逐个排放在此数组中。小数点的位置用一个无符号的整数表示，存储从右边算起的小数点位置。比如一个十进制的数据-0127.73可表示为{-1,0x012773,2}，从数据位序列可以看出的存储可以看出，整个数据需要的存储空间为 (5+1)/2=3，具有最低权重的数字"3"放在整个数组的最右边的四个比特，类似于大尾存储方式。在接下来的四节中，我们详细讨论了多精度十进制数据模型的各种运算，因为在十进制数据模型中，对于运算结果的小数点位置的确定是非常明显的，为了降低算法描述的复杂性，我们都忽略了对结果小数点位置的确定。

3.2 移位

移位操作是整个算术系统中经常需要用到的操作之一，比如在加减法操作之间需要对齐时，就是一个左移操作。同样在除法操作中我们也需要用到左移操作，因为每次只能操作部分被除数，再经过一次迭代后，需要将余数左移，然后从被除数中去取出若干十进制位拼装到部分被除数中，继续下一次迭代。在十进制编码中，移位操作是相当方便的，如果是移动偶数个十进制位，只需进行字节拷贝即可。如果移动奇数个十进制位，也只需要在每次迭代中，将前一次迭代的字节的前四个比特和当前字节的后四个比特组成一个新字节存储。

3.3 加减法

由基本的算术特性知道，无论参与加减运算的两个操作数的符号是什么，总可以通过比较其绝对值的大小，然后通过无符号的加减法得到正确的运算结果。因此这里仅考虑无符号的加减运算。由运算规则知，当两个数据的指数或者说小数点的位置不同时，首先需要对齐小数点，这可由上面的移位操作实现。对齐后从右至左逐字节地进行加减操作。值得注意的是，每次进行操作时，首先需要将一个字节表示的两个十进制位转换成一个二进制编码的整数，这是因为 ALU 部件的加减法是二进制编码的。另外加法操作后，需要分离出可能的进位，这一点可将加法结果简单除十进制100得到，余数需要再次转换成十进制编码。同样减法操作时，如果需要借位，则被减数字节加上十进制100后再和减数字节做差，最后仍然需要将结果转换成十进制编码。

3.4 乘法和幂操作

首先确定乘法结果的存储空间大小，由乘法算术规则可知，m 位十进制数据和一个 n 位十进制数据相乘，其结果至多有 $m+n$ 位十进制数字。因为十进制编码和十进制有效数字的简单映射关系，可以简单地给乘法结果分配空间为两个操作数存储空间之和。整个乘法算法主体由一个倒置的两重循环(循环索引从后向前)构成，乘法结果的第 $i+j+1$ 个字节的值是一个累加的过程，并且每一次计算的结果由三部分之和构成，第一部分由乘数的第 i 个字节和被乘数的第 j 个字节的乘积构成，第二部分是内层循环中上一次迭代的进位，第三部分是外层循环迭代中已经存储在第 $i+j+1$ 个字节的值。同样在最终的和中需要分离出可能的进位，余数才是所得结果。如上图2表示了对1.2345和43.56做乘法的过程，当 i 和 j 同时为1时，参加乘法的两个字节分别为23和43。

对于幂运算，将一个数据自乘许多遍是非常慢的。在本文的实现中，使用了"平方的平方"技巧可以有效减少乘法次数，因此高效得多。其算法如下：

```
while(n!=0){
   if((n&1)==1 ) res = res.mul(base);
   if((n>>=1)!=0 ) base = base.mul(base);
}
```

其中 n 为指数，算法开始时 *res* 和 *base* 分别初始化为1和底数。如果 n 为奇数时，*res* 累乘底数，同时将指数 n 减半，如果 n 不为零，将底数 *base* 平方。不断重复这个过程直到指数降为零为止。考虑计算 a^{15} 的过程，将指数分解成2的幂次之和，我们可以得到 $a^{15}=1*a*a^2*a^4*a^8$，由于我们在计算过程中用 base 保留了2的幂次，因此在计算 a^4 时，可以直接将前面迭代过程中保留的 a^2 直接平方，和自乘4次相比减少了两次乘法。整个过程仅需要做7次乘法，而传统的自乘方法则至少需要14次乘法。对于多精度数据类型来说，在减少乘法次数的同时，也意味着减少了创建对象的次数，从而可以显著提高时空效率。

3.5 除法操作

除法操作是多精度数据算术中最为复杂的操作，它不像乘法那样直接。我们不但不可能一次得到商的所有有效数字，甚至部分商也需要经过若干次调整才能得到，因为除法存在一个猜商的过程。大多数情况下，我们需要猜到这个部分商的一个上界，并且这个上界应该比较接近上确界，否则过多的试商将引入大量的多精度数据乘法、减法和比较操作，最终导致性能的显著降低。另外还需要考虑的一点是除法有两种结果类型，一种是商和余数，另一种是一个"足够精确"的商，这个商的最终长度由系统预定的分辨率或对本次除法所指定的分辨率决定。除法算法的一个大致流程图如下。

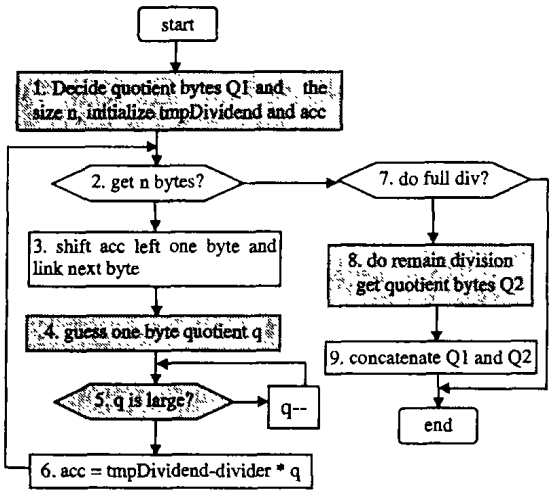

图3 除法算法示意图

表1 除法详细过程

tmpDividend	Quotient	Accumulator
012345	0123/43=02	012345-02 * 4322=003701
370167	3701/43=86, large, 85	370167-86 * 4322 <0 370167-85 * 4322=002797
279789	2797/43=65, large, 64	279789-65 * 4322<0 279789-64 * 4322=003181
318100	3181/43=73	318100-73 * 4322=002594
259400	2594/43=60	259400-60 * 4322=000080
008000	0080/43=01	008000-01 * 4322=003678
367800	3678/43=85	367800-85 * 4322=000430

在进行除法之前，首先需要对齐小数点。这样可以在第一部轻松确定部分商 Q1的存储空间大小为被除数和除数空间大小之差。然后将除数复制到累加器 *acc* 中，从被除数的高字节起复制与除数等长的字节数到部分被除数 *tmpDividend*。为了减少不必要的数组复制，实际的 *acc* 和 *tmpDividend* 的空间大小都比除数多一个字节空间。这样 *q* 可以直接乘到 *acc* 上，而不必额外分配对应的乘积空间。第四步猜商可以采用文[10]中提到的技巧。第五步判断 *q* 是否过大，我们可以直接比较 *acc* 和 *tmpDividend* 的大小，其中 *acc* 已经为除数和商 *q* 的乘积，如果 *acc* 大于 *tmpDividend*，表示猜得的商 *q* 偏大，这时可以直接在 *acc* 的基础上递减除数，直到比 *tmpDividend* 小，这样我们可以避免将除数重新复制到 *acc* 中再与调整后的商 *q* 做乘法。第八步指的是被除数的有效数字已经用尽，我们需要在余数后补零继续作除法。通常需要在系统中预设一个最小的分辨率，比如小数点后18位，或直接在除法参数中指定本次除法所需要达到的分辨率。整个后续除法过程与图3中二至六步相同，只是在第三步中左移一个字节后补零。表1表示了1234567.89和43.22作除法的详细过程，保持到小数点后10位的结果为285.6473600185。

3.6 数据转换

对于字符串表示的小数和十进制编码的数据模型之间的转换仅需要两步，第一步确定数据的符号和存储空间大小，第二步从右至左逐字符地将十进制数字放到对应的四个比特位上即可。相反方向的转换同样只需从左至右以四比特为单位将对应的十进制数字转换为对应的 ASCII 码就可以了。

4 比较试验和性能分析

为了比较各种编码方案的特性，我们使用 Java 语言实现了两种单精度的编码格式，分别是二进制定点格式(BinFix)和二段式编码的确切算术格式(BinExact)，在这两种编码方案中，为了实现的方便性，我们都只实现了加减和乘法运算，同时小数部分都占据16位。在对比试验中，我们计算了5000000个正方形减去其内切圆后的面积，计算公式为：$(r+r)^2-pai*r^2$，其中 pai 取3.1416，r 从0.1开始随循环索引变量增大，同时我们还用二进制浮点格式，即系统直接支持的浮点部件进行了同样的计算。试验结果如表2所示。

表2 单精度编码算术比较

编码方案	BinFix	BinExact	Float(硬件)
运行时间(ms)	219	3860	78
时间比	2.8	49.5	1
结果准确性	No	Yes	No

从上表我们可以看出，由硬件支持的浮点算术仍然是最快的，在二进制定点编码方案中，虽然我们采用补码方式存储整个数据，从而可以用一个整数加法指令完成一次加法操作，但是因为两个32位的操作数作乘法操作会导致一个64位的结果，同时依据二进制定点格式，我们只能取64位结果中的中间32位。在做乘法时，我们不得不将其中的两个操作数都提升为64位的数据类型，否则根据 Java 的类型提升规则只能得到低32位的结果值，这些整数扩展转换和64位数据的乘法操作导致了一些不必要的操作，使得 BinFix 的运行时间达到浮点硬件的2.8倍。而 BinExact 中每一次算术运算都要将整数和小数部分的运算分别进行处理，同时因为其中的算术运算和 BinFix 比起来都相对复杂，因此在实现中，大多用函数调用的方式实现其算术运算功能，这又导致不必要的函数调用开销。

在对多精度数据的实验中，我们实现了十进制编码系数的定点(DecFix)和浮点(DecFlt)格式，在这两种编码格式中我们实现了加减乘除幂五种运算，数据运算的中间结果预先设定为18位整数18位小数。我们选取了一个计算借贷还款的例子，计算公式为：$amount * (m\_rate/(1.0-(1.0/(1.0+m\_rate))^{months}))$，其中 *amount* 为借款量，*m_rate* 为月利率，*months* 为借款月份。对于多精度数据算术，我们做了两组实验，第一组对同一个数据样本我们分别使用 DecFix、DecFlt 和二进制编码系数的浮点(BigDecimal，jdk 库提供)格式重复计算了5000次，第二组实验中，我们使用伪随机数产生了5000个样本，并且采用 DecFlt 和 BigDecimal 格式实现的 Cobol 中的定点数据类型进行了计算，实验结果如表3所示。

表中 hot 操作数据采用 jdk 所提供的 hprof 工具对程序执行采样获得，在第一组实验中 BigDecimal 热函数相对较为分散，因此采用了其中最热的 trace 代替，第二组实验中最热的函数调用已经表现为数据文件读写和字节码文件装入，表中记录数据为与算术运算相关的最热函数。从表中数据我们可以看出，DecFix 虽然采用定点数据格式可以避免加减法操作时的移位操作，但是由于大多数数据并没有占满定点空间，从而导致不必要的与零进行乘法操作，最终使得乘法操作占据了3282ms 中的近1570ms。而同时由于 DecFlt 采用浮点数据格式，也就是按需分配实际数据所占存储空间，使得数组乘法大大减少，仅占984ms 中的250ms 左右。通过进一步分析 BigDecimal 中幂函数的调用轨迹，我们发现在许多以幂函数开始的调用轨迹中都包括一个 digitLength 的函数调用，此函数确定一个二进制编码的系数包括多少位十进制有效数字，系统采用的办法就是使用大整数除法不断和10的幂作除法，由3.5节的除法算法分析我们可以看出，大整数除法也是一个

非常昂贵的过程。也就是说整个过程中，系统大量时间耗费在二进制编码的大整数到十进制的转换上，最终导致 BigDecimal 的运行时间也达到了1953ms。最后我们将 BigDecimal 和 DecFlt 分别应用到实际的 Cobol 事务处理程序中，发现针对5000组不同测试样本 DecFlt 仍然较 BigDecimal 格式快20%左右。

表3 多精度编码算术比较

编码方案	DecFix	BigDecimal	DecFlt
运行时间(ms)	3282	1953	984
Hot 操作	数组乘，47.82%	幂次，32.05%	数组乘，25%
时间比	3.33	1.98	1
运行时间(ms)	n/a	3172	2641
相关 Hot 操作	n/a	除法，2.94%	数组乘，1.41%
时间比	n/a	1.20	1

结论 十进制算术的强烈需求和大多数嵌入式处理器中浮点部件的缺乏使得采用软件方式支持定浮点运算成为必需。本文通过对小数定浮点特性的深入分析，指出了在各种应用要求限制下的选择策略，并且对多精度的定浮点运算给出了以十进制编码系数的数据模型及其详细的算法，实验表明这种数据模型比 Java 中以二进制编码系数的 BigDecimal 数据类型性能提高近一倍。同时我们将此数据模型和算法运用在 Cobol2Java 翻译系统中，将实际的事务处理程序性能提高20%左右。

参 考 文 献

1 Perkins R. EASIAC, A Pseudo-Computer. Communications of the ACM, ACM, 1956, 3(2): 65～72
2 Jones F B, Wymore A W. Floating Point Feature on the IBM Type 1620. IBM Technical Disclosure Bulletin, 05-62, IBM, May, 1962. 43～46
3 Chen T C, Ho I T. Storage-Efficient Representation of Decimal Data. ACM, 1975, 18(2): 49～52
4 Stevenson D, et al. IEEE 754-1985 IEEE Standard for Binary Floating-Point Arithmetic. IEEE, July 1985. 20
5 Cody W J, et al. IEEE 854-1987 IEEE Standard for Radix-Independent Floating-Point Arithmetic. IEEE, March 1987. 14
6 Cowlishaw M F, Schwarz F M, Smith R M, Webb C F. A Decimal Floating-Point Specification. In: Proc. of the 15th IEEE Symposium on Computer Arithmetic, ISBN 0-7695-1150-3, IEEE, June 2001. 147～154
7 Cowlishaw M F. Decimal Floating-Point: Algorism for Computers. In: Proc. of the 16th IEEE Symposium on Computer Arithmetic (ARITH'03), 2003
8 Melton J, et al. ISO/IEC 9075: 1992: Information Technology - Database Language - SQL. ISO, 1992. 626
9 Shibamiya A. Decimal arithmetic in applications and hardware. pers. Comm. June 2000. 2
10 Knuth. The Art of Computer Programming. Addison-Wesley Publishing Company, 1981, 2: 269～271

（上接第141页）

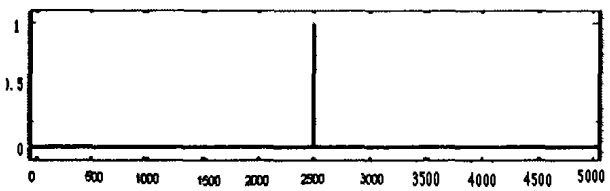

图5 输出序列的自相关函数

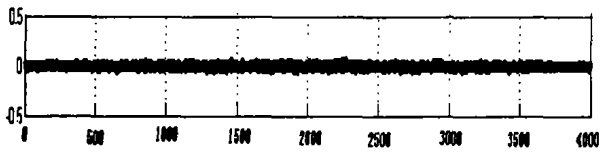

图6 输出序列的互相关函数

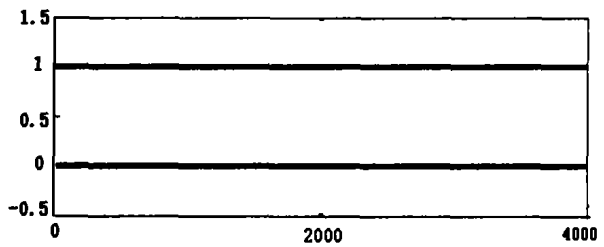

图7 输出序列的0、1分布

结论 本文通过 Logistic 映射和帐篷映射及相关控制技术提出了一种随机性较好的伪随机序列发生器的实现方法。当然只要对该方法稍加修改就可以得到任意位长的位串（密钥）发生器。在使用密码系统通信时，保持加密端和解密端的密钥序列发生器同步是至关重要的。在信息传递过程中，如果一个发生器跳过一个周期或者一个密文位在传输中丢失都可能出现解密错误，因此必须保证在继续进行之前使两个发生器重新同步。关于发生器的同步可参阅文[1,4,7,8]。

参 考 文 献

1 Rueppel R A. Linear complexity and random sequences. In: Advances in Cryptology. EURO CRYPT'85 (LNCS 219), 1986. 167～188
2 Sushchik M, Tsimring L S, Volkovskii A R. Performance analysis of correlation-based communication schemes utilizing chaos. IEEE Trans. on CAS-I FTAA, 2001, 47(12): 1684～1691
3 Gernak J. Digital generators of chaos. Phys. Lett. A, 1996, 214: 151～160
4 Frey D R. Chaotic digital encoding an approach to secure communication. IEEE Trans, On circuits and systems (II), 1993, 40(10): 660～666
5 Habutsu T, Nishio Y, Sasase I. et al. A secret cryptosystem by iterating a chaotic map. In: Advance in cryptology - EUROCRYPT'91, D. W. Davies, ed. LNCS 547, (Springer - Verlag, Berlin), 1991. 127～140
6 Kocarev L. Chaos-based cryptography: A brief overview. IEEE Trans. on CAS-I, 2001, 1(3): 6～21
7 Pecora M, Carroll L. Synchronization in chaotic systems. Phys. Rev. Lett., 1990, 64(8): 821～823
8 Cuomo K M, Oppenhein A V, Strogatz S H. Int. J. Bifurcation and Chaos[M]. Sci. Am., 1993, 269: 62～68
9 Robinson C. Dynamical systems: stability, symbolic dynamics and chaos[M] (2nd Edition). CRC, 1998
10 Jung O, Rul C. Encryption with Statistical Self-Synchronization in Synchronous Broadband Networks. In: Proc. Cryptographic Hardware and Embedded Systems, CHES '99: 1999. 340～352
11 饶妮妮. 一类混合混沌序列及其性能分析. 电子科技大学学报, 2001, 20(2): 115～119
12 周红. 有限精度混沌系统的 m 序列扰动实现. 电子学报, 1997, 25(7): 95～97