Low-level Floating Point Marshalling Between Different Instruction Level Architectures

Iliya Georgiev

Abstract: *The paper suggests a method for marshalling floating point numbers in network computing based on the bit-level converting subroutines. Such subroutines are building blocks of hybrid conversion protocols. The method provides speed that is required by real time and transaction applications. A library of conversion subroutines is implemented that could be linked to different applications on different computers. The library is incorporated in commercial transaction packages.*
Key words: *marshaling, floating point conversion*

INTRODUCTION

Data Marshalling. Marshalling is a transformation (conversion), when the sending program translates the data it wants to transmit from the representation that it uses internally, into a message that can be transmitted over the network. On the receiving side, the application translates this arriving message into a representation that it can then process; that is called unmarshalling. Marshalling transformations are from big endian to little endian byte order, between different integer presentation sizes, ASCII (UNICODE) to EBCDIC symbol tables, serializing of several data objects, and the most difficult *transformation of floating point (FP) numbers*. Some computers represent floating point numbers in IEEE 754 formats (or its dialects), while other machines still use their own non-standard format.

A significant problem rises any time when floating-point data are exchanged [1, 2, 3]. Let us consider some example. Program *A* and program *B* are exchanging floating-point data over the network. Program *A* is working with very small numbers without using normalization, for example - exponent (8 bits): 01_{16} ; fraction (24 bits): $000\ 09a_{16}$. Program *A* is sending this number to the program *B*, which keeps all floating-point data normalized. Trying to normalize the number program *B* causes exponent underflow. The result is *fatal loss of data*. To prevent this, there must be a consistent *floating-point data conversion* from one format to the other.

Motivation and contribution of the paper. Requirements to the floating number network marshalling are very difficult for implementation. Some of them are: performance (for example, in transaction processing no additional overload is appreciated); transparency (no way to issue exceptions in a case of unavoidable data or precision loss); application and data dependence (the conversion method depends on the range and discretization of the presented numbers) and so on.

The paper focuses on transparent low-level floating point conversion that could help the developers of the marshallers to prevent the fuzziness of the floating-point numbers and to create converting hybrid protocols that work with transaction-satisfied speed.

BRIEF REVIEW OF SOME POPULAR FOATING POINT FORMATS

Intel IEEE 754 format *[4].* The formats of the floating-point data match the single and double formats of the IEEE standard 754-1985. The extended format follows the recommendations of the standard but is Intel specific. The short format is 32 bits long and consists of one bit sign, 8 bits exponent (e) and 23bits significant (f, from the other name fraction). The value is represented by the formula: $N = (-1)^s * 2^{e-127} * 1.f.$ The exponent is coded in Excess 127 code. The standard can present normalized and not-normalized numbers. The normalized fraction is greater to 1 and less than 2. The high order integer bit is therefore dropped and assumed to have the value of 1. The long format is 64 bits long and consists of 1 bit sign, 11 bits exponent (Excess 1023) and 52 bits fraction. The value

is: $N = (-1)^s * 2^{e-1023} * 1.f$. The extended Intel format consists of 1 bit sign, 15 bits exponent and 64 bits fraction. The value is: $N = (-1)^s * 2^{e-16,383} * 1.f$

Floating point numbers with an exponent of *0xff* do not represent conventional numbers but are instead reserved to represent quantities that may not be represented as numbers, like √-1 and infinity ∞. Plus ∞ is represented by 0*x3f800000* (single), that is, a biased exponent of 0xff and a zero significant. Minus ∞ is the same as ∞, but the sign bit set, 1xff800000. Infinity will be the result of a floating divide of 1.0 by 0.0. Infinity compared to infinity is considered equal, whereas infinity compared to any other number is considered larger. The presentation of ∞ is useful in such cases as *atan (∞),* which can be programmed to return the correct result of π/2. All others numbers with a biased exponent of *0xff* are considered to represent quantities that can not be represented as numbers. It is customary to initialize all uninitialized single floating variables to integer *-1*, which is, *0xffffffff*, a *NaN* (not a number).

SUN SPARC format [7]. The single and double formats are according IEEE 754 standard, the quad format is SPARC specific. The quad format is 128 bits long and consists of one bit sign, 15 bits exponent and 112 bits fractional part of the significant. The value of the floating-point number is represented by the formula: $N = (-1)^s * 2^{e-16,383} * 1.f$.

IBM XA/370 and SA/390 mainframe format [5]. A floating point number consists of a signed hexadecimal fraction *f* and an unsigned seven-bit binary integer called the characteristic *c*, which represents a signed exponent. The exponent value is considered as *mod 64* value (Excess 64 code). The fraction of a floating-point number is a hexadecimal number because it is considered to be multiplied by a number which is a power of 16. The floating-point numbers have a short (32-bit), a long (64 bit) and a extended (128 bit) format. The value of a floating-point number is the product of its fraction and the number 16 raised to the power of the exponent which is represented by its characteristic: $N = \pm f * 16^{\pm (c-64) mod 64}$. The leftmost bit presents the sign of the digit (0 - positive number, 1 - negative number). The characteristic is the next seven bits (0 to 7) for all formats. The fraction is constituted by the remaining bits - 6 hex digits for short and 14 for the long format. An extended floating-point number has a 28-hexdigit fraction and consists of two 64-bit words which are called the high-order and low-order parts. The high-order part may be any long floating point number and it is considered as the leftmost 14 hex-digits of the 28-digit fraction. The sign and the characteristic of the high-order part are for the all extended number. If the high-order part is normalized, the extended number is considered normalized. The fraction of the low-order 64-bit word should be considered concatenated to the higher part of the fraction.

HP (Tandem) Tandem Non-Stop Series [9]. The single precision floating point number consists of 1 bit sign, 22 bit fraction and 9 bits exponent. The extended precision floating-point number is presented in a 64-bit quadruple word. The sign is one bit, the fraction 54 bits and the exponent 9 bits. The fraction is always normalized, to be greater to 1 and less than 2. The high order integer bit is therefore dropped and assumed to have the value of 1. During all calculations, the sign is temporarily removed and the assumed integer bit reinserted. The integer plus the 22 fraction bits are equivalent to 6.9 decimal digits; the 55 bits of an extended number are equivalent to 16.5 decimal digits. If the value of the number to be represented is zero, the sign is zero, the fraction is 0, and the exponent is zero. The absolute-value range of 32-bit floating-point numbers is $\pm 2^{-256}$ through $\pm(1- 2^{-23})* 2^{256}$. In decimal notation, this is $\pm 8.64 * 10^{-78}$ through $\pm 1.15 * 10^{77}$. For the extended floating-point numbers (64 bits), the range is the same; only the precision is increased: $\pm 2^{-256}$ through $\pm(1- 2^{-55})* 2^{256}$.

METHODS FOR FLOATING POINT CONVERSION

Because any conversion of the floating-point numbers could be data-and-application dependent there is no general-purpose solution. Nevertheless, here we consider three main approaches of floating-point conversion.

Specific Data Conversion Protocol. This approach is extremely data-and-program dependent but is effective and fast. It is convenient for distributed engineering and scientific processing in real-time or non-stop regime. Application programs have their specific very reliable protocol for data conversion. Some of the main principles of such protocol are the following. Every program knows the floating-point formats used by the partner. The data migrating from one computer to another is under control of the both programs, i.e. every program has some supervising function for data marshalling. Every floating-point number or group of numbers is identified during the exchange. Every identifier explicitly defines the format of the floating-point numbers. There is a convention how to make the conversion. Usually this is the receiving program, i.e. the program which will use the floating-point numbers. If the data migrate between more than two programs or computers, all programs, which are going to use these data, are converting the numbers for their own purposes. There is a convention about the alternative: converting to the nearest format with rounding or converting to the higher format without rounding. Usually, the identifier of the floating-point data under transfer explicitly defines which method should be used. Sometimes, when the transferred data are extremely important, the data are kept in buffers, which gives a possibility of backpoints.

Common Data Conversion Protocol. This approach is less data-and-program dependent but is slower than the first one. It is used in the engineering data bases, distributed computer-integrated manufacturing and computer-assisted design systems. The main principle of this approach is that all floating-point data are transferred in a ASCII as a *decimal form scientific notation (*1.192092896e-07). When program A wants to send floating-point numbers to program B, it converts the numbers in a decimal form scientific notation. From the other side program B converts the received data from decimal form to its floating point formats.

Floating Point Specification Table. The following main principles are the basis of this approach. The floating-point data are transferred from the sending program "as are" or in a decimal form scientific notation. If the data are "as are", that means the floating-point number are presented in the formats of the sending program A (no conversion). In this case the integrated speed is higher. When decimal form is used, then the processing speed is like the second approach. Together with the floating-point data the sending program transfers a table, where all floating-point characteristic and limits are described. The table is something like a floating-point specification of the sending architecture. The receiving program converts the floating-point numbers to its own formats. The conversion is based on the parameters, described in the table. Some of the parameters in specification table [7] are: smallest positive number, guard digit, radix of exponent representation, smallest negative number, normalization, number of bits in the significant, presentation of zero, etc.

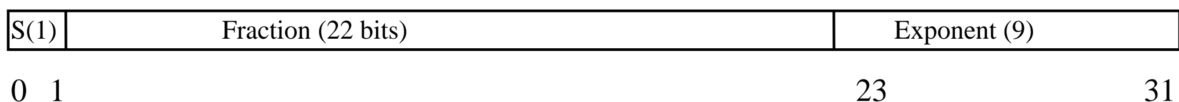
LOW-LEVEL MARSHALLING APROACH

In this paper a low level marshalling principle is suggested that extracts the exponent and significant binary value of each source floating point number and constructs again the binary form of the destination format. The supplied converting functions can be used for the implementation of every of the described above three methods.

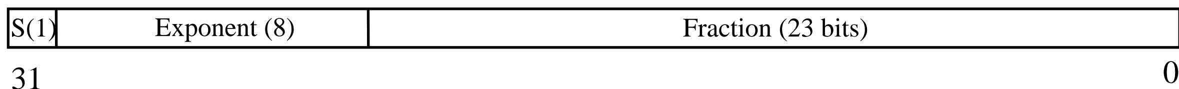
The presented low-level marshalling method is *application program independent*. The conversion is well defined function with precise definition of the restrictions. The application programs can choose functions from the library according to its needs.

The low-level marshalling method is *data dependent*. That means the converting functions are traversing the floating-point numbers "as they are". Rounding and other treating of the result can be fulfilled only by the application program. All functions take a floating-point number in some source format and return a number in the destination format. The functions do not make big-endian to little-endian conversion and vice versa. The functions are implemented as class methods.

Let us explain the low level marshalling principle with a conversion example between the Tandem Non Stop single and IEEE 754 short format. The Tandem single format consists of 1 bit sign, 22 bit fraction and 9 bits exponent:



The short format of IEEE 754 is 32 bits long and consists of one bit sign, 8 bits exponent and 23bits fraction:



In both formats the fraction is normalized, to be greater to 1 and less than 2. The high order integer bit is therefore dropped and assumed to have the value of 1. During all calculations, the sign is temporarily removed and the assumed integer bit reinserted. The conversion method has the following stages: source FP number unpacking, sign processing, exponent processing, fraction processing, and destination number packing. Let us identify the source Tandem number as *TNS(32),* and the destination number as *IEEE(32).* The pseudo code follows. For simplicity and readability some special cases (for zero, not normalized numbers) are not given.

```
// TNS(32) unpacking in S(32) for the sign, E(32) for the exponent and F(32) for
//thefraction // using bitwise AND

  S(32) = TNS(32) and 0x80000000;  //sign is ready for packing
  E(32) = TNS(32) and 0x000001ff;  //E(32) is the unpacked exponent
  F(32) = TNS(32) and 0x7FFFFE00; //F(32) is the unpacked fraction

// Exponent Processing, source exponent is biased by 256 (excess 256 code)
   E(32) = E(32) – 256;  // subtract the bias
   if E(32) > 127          // check if the exponent can fit in the target format
      then E(32) = 255 and error code "Conversion not possible";  // issue error
                                        //code and use saturation encoding
   E(32) = E(32) +127;   //destination exponent is in excess 127 code
   E(32) = E(32) logical left shift 23 bits; //shift the exponent to the destination place

// Fraction Processing, source exponent is biased by 256 (excess 256 code)
  F(32) = F(32) logical right shift 8 bits; //shift the fraction to the destination

// Destination number packing
IEEE (32) = S(32) or E(32) or F(32) //packing using bitwise OR operation
```

Below is the implementation in C/C++ language of the above Tandem to IEEE single floating format conversion.

```
// unions to present the numbers both
//as FP number and char strings (octets)

union ShortFloat {
  float kurz;
  unsigned long kurzint;
  octet kurzString[4];
};
union DoubleFloat {
   double doppelt;
   struct {
     unsigned long firstint;
     unsigned long secondint;
  } twoint;
   octet doppeltString[8];
 };
// ***Tandem single(32 bit)to PENT short (32 bit
// float) ***
// Host:  Pentium;  Sender: Tandem; Receiver:
//Pentium
//The data has been received as a string of char,
//the result of the conversion is a float
//   WARNING: The Tandem dbl exponent is one
//bit more than Pentium short.
// The program checks if the mapping is possible.
//   Returns:   0 - mapping OK;
//              -1 -mapping impossible.

short cvTANDEMdblPENTsh ( octet* FPsource,
float* FPdestination) {
    ShortFloat *psh = new ShortFloat;
 unsigned long aFloat, aSign;
long aExponent;
for (short i=0; i < 4; i++) {
psh->kurzString [i]= FPsource [i];
}
    aFloat = psh->kurzint;
//check for a signed zero
//go out with signed zero
if (0 == (aFloat & 0x7fffffff)) {
*FPdestination=psh->kurz;
delete psh;
 return 0;
}
//check for a signed zero
//go out with signed zero
if (0 == (aFloat & 0x7fffffff)) {
*FPdestination=psh->kurz;
delete psh;
 return 0;
}
// sign processing
  aSign = aFloat & 0x80000000;
// exponent conversion
aExponent = aFloat  & 0x000001ff;
 aExponent = (aExponent- 256) + 127; //exponent biasing
   //check if mapping is possible
    if ((0 >= aExponent)||(aExponent > 255)) {
  aFloat = 0x00800000 | (aFloat & 0x80000000);
   if ((0 == aExponent) && (0 == (psh->kurzint &
0x00ffffff))) {
   goto goout;
   }
   // keep this for emulation call
   //if (0 > aExponent)  goto noMapping;__

else if (aExponent > 255) aFloat =
0x7f7fffff | (aFloat & 0x80000000);

//noMapping:
//printf ("Mapping impossible\n"); // debugging only
psh->kurzint=aFloat;
*FPdestination=psh->kurz;
delete psh;
return -1;
}

aExponent = (aExponent <<23) & 0x7f800000;

// fraction processing
_aFloat = psh->kurzint & 0x7ffffe00;
_aFloat = aFloat >> 8; //fraction forming
_aFloat = aFloat | aExponent | aSign;

// transfer the result to the destination
goout:
psh->kurzint=aFloat;
*FPdestination=psh->kurz;
delete psh;
return 0;
}
```

Hybrid conversion protocol. The suggested low-level conversion approach were used to create the main marshalling functions for hybrid conversion protocols. In this protocol, the applications programs can use the low-level marshalling function for most of the cases. If the low- level conversion function issues some exception that the conversion is not possible, then the floating point number can be presented as very long decimal number and transmitted. The receiving application then converts the decimal into the internal floating point format.

Hybrid conversion protocol is especially efficient to solve special cases [3]: rounding, infinity, etc. On some floating-point architecture every bit pattern represents a valid floating-point number. The IBM XA/370 is an example of this. Other format like IEEE 754 standard has NaNs (*Not a Number*) and infinities. Without any special analysis, there

is no good way to handle exceptional situations like taking the square root of a negative number, other than aborting computation. Under IBM FORTRAN, the default action in response to computing the square root of a negative number like -4 results in the throwing of an error message. Since every bit pattern represents a valid number, the return value of square root must be some floating-point number. In the case of System/370 FORTRAN, $\sqrt{-4} = 2$ is returned. In IEEE 754 arithmetic, a *NaN* is returned in this situation.

IMPLEMENTATION OF A LOW LEVEL FP MARSHALLING LIBRARY

The described method was used to create converting functions to/from practically all floating point formats: IEEE 754 dialects (Intel, Sun SPARC, HP-Perspective architecture, R 6000, JAVA RTE Virtual machine format, MS .NET Common Language Runtime format), IBM XA/370 and SA390 formats, HP (Tandem) Non Stop format.

The library was incorporated in the following commercial applications: TransFuse© (transaction gateway and marshaller) and CodeSync© (language converter) [8].
Information technologies applications obtain access to existing TP Monitor systems and IBM MQ Series programs within a transaction through TransFuse.

TransFuse acts as a bridge between the CORBA and the non-CORBA world. TransFuse binds application components together into a single transaction that can span different TP Monitors or multiple instances of a single TP Monitor--ensuring transaction integrity through two phase commits. In this way, transaction systems can interoperate with each other in a single transaction.

CodeSync enables companies to maximize their investments in COBOL programs while migrating to distributed application systems. CodeSync parses COBOL source code and builds C++ streamable objects, Java, or DDL structures that are used by applications which access enterprise data through TransFuse. This enables the programmer to write new client/server code without rewriting existing COBOL programs.

CONCLUSIONS

In this paper, we share our research and development results applying bit-level programming method to create well-defined functions for floating point marshalling between most popular architectures. Such approach gives conversion speed and flexibility for transaction and real time applications. The implemented functions are used for years in commercial transaction gateway software and could be incorporated as building blocks in different hybrid marshalling packages.

REFERENCES

[1] Allison, C. Where did all my decimals go?', The *Journal of Computing Sciences in Colleges,* vol. 21, No 3, pp. 47-58, 2006.
[2] Cheney, W., Kincaid, D. 'Numerical Mathematics and Computing'*, Thomson/Brooks Cole*, 2004.
[3] Goldberg, D. 'What Every Computer Scientist Should Know About Floating-Point Arithmetic', *ACM Computing Surveys,* vol.23, No1, pp.5-48.
[4] IEEE Library, 'IEEE Standard for Binary Floating-Point Arithmetic,' *ANSI/IEEE Standard No. 754-1985,* The Institute of Electrical and Electronics Engineers, Inc., New York, August 1985.
[5] IBM Corporation, 'Enterprise Systems Architecture/390 Principles of Operation', *IBM library.Order No. SA22-7201-05,* September 1998.
[6] Overton, M. 'Numerical Computing with IEEE Floating Point'*, SIAM*, 2001.
[7] SPARC Architecture & Assembler, *SUN Microsystems*.
[8] VisiBroker, 'TransFuse User's Guide *and the* CodeSync User's Guide - Integrating Host Data with CORBA Applications', http://info.borland.com/techpubs/ *books/its/its10/programmer/itspg13.htm*
[9] HP NonStop Himalaya K10000 Server Description Manual.

ABOUT THE AUTHOR

Iliya Georgiev, Prof. Dr., Department of Mathematical and Computer Sciences, Metro State College of Denver, Campus box 38, P.O. Box 173362, Denver 80217, USA, Phone: +303 673 9403, E-mail: gueorgil@mscd.edu, ilgeorg@gmail.com; URL: http://clem.mscd.edu/~gueorgil/.