REVISITING INTEGER MULTIPLICATION OVERFLOW

Eyas El-Qawasmeh and Ahmed Dalalah
Computer Science Dept.
Jordan University of Science and Technology

ABSTRACT. Online problems arise in various applications ranging from load balancing and scheduling to network and financial problems. However, some of these online applications like financial tools, online calculators and online math programs suffer from the overflow problem caused by the multiply operation of two operands. The overflow occurs whenever the multiplication of any two-integer numbers exceeds the maximum limit available for the result. Many programming languages ignored this problem; therefore, the programmer has to handle it, mostly in "predict and avoid" approach. This paper addresses the detection and control of integer overflow in programming languages. Two examples from C and Java programming languages are considered. The paper suggests "detect and do" algorithms to handle the overflow. In addition, this paper suggests adding a built-in function to test whether an overflow will be generated by multiplication operation in advance.

KEYWORDS. BigInteger, Detection, Multiplication, Overflow

1. INTRODUCTION

Currently, the wide use of the World Wide Web increases the use of online applications. In the online applications, the "whole dataset is not available and the application receives the information piece by piece from the user" [Borodin, et al., 1998]. Online problems arise in a wide variety of applications including scheduling, robot motion planning, load balancing, math and financial problems [Fiat, et al., 1998] [Grove, et al., 1995] [Irani, et al. 1997] [El-Qawasmeh, 2003]. Examples of online mathematical and financial applications include math clubs, statistics tools, calculators, and interactive finance tools.

Some of the online computations must solve the overflow problem in order to get the correct results. For example, the computation of a jackpot prize with one ticket, where the number of combinations of 40 numbers takes 5 at a time is "40 choose 5" which is 40!/(40-5)!5!. The intermediate computation of 40! generates a number that requires more than 50 digits. Another example is the calculation of Fibonacci numbers for 100 and above. In these two examples, the result cannot fit in a typical integer data type and it will produce incorrect results due to the overflow of the multiply operation.

Multiplication is one of the common arithmetic integer operations that a processor performs and it has been an active area in computer science [Gok, 2000] [Schulte, et al. 2000]. When two unsigned n-bit numbers are multiplied together, it is always possible to produce a result with 2n bits; as a result, an overflow occurs. In other words, we are trying to store a number in a memory location that is not large enough to hold it. A simplified example is an 8-bit variable, which can hold a maximum number equal to ($2^7 - 1$) in case of signed integer or ($2^8 - 1$) in case of unsigned integer. However, if we try to store a number equal to or greater than 2^8 into this 8-bit variable, then an overflow will occur since we can't represent that number with 8 bits [Elguibaly, 2000] [Gok, 2000] [Parhamin, 1988].

Many Programming languages, especially the procedural languages, have ignored the overflow problem and shifted the responsibility of solving it toward the programmer. However, programmers are unable to solve it efficiently. They use one of two approaches: "predict and avoid", or "do and detect". The former approach, which predicts the overflow problem and avoids it, works in many cases, but when numbers get very large it does not work since it becomes impossible to avoid the overflow due to the limitations in hardware or in language itself. The later approach, which is a "do and detect", lacks the capability to be implemented in the programming language itself. Therefore, the programmer has to do it using the assembly language. The use of assembly language is very time consuming and tedious process, which might not work on all platforms.

An ideal solution to the overflow problem should "detect and do" the corresponding operation. Either the hardware or the software can solve this problem. A hardware approach might allow a computer word to be of a very big

size. For example, expand the 32-bit computer word to 128-bit. However, this solution is not currently available due to many reasons. An alternative to the hardware approach is the software approach, which can control the overflow by providing a mechanism to detect the overflow and do the arithmetic operation. In this paper, we devoted our attention toward the software to solve the overflow problem.

This paper will review the detection of overflow in integer multiplication. Then, it presents the corresponding algorithms to handle integer overflow in multiplication. The presented algorithms allow us to handle integers of any size (without any extra requirements of hardware) in the procedural languages. This paper assumes that the size of the computer word is 16-bit unless mentioned something else.

The organization of this paper is as follows. Section 2 is an overflow detection process. Section 3 is a C language example. Section 4 is a Java example. Section 5 is the suggested “detect and do” algorithms, and section 6 contains the concluding remarks.

2. OVERFLOW DETECTION

Given unsigned integer A consists of n-bit where A = $a_{n-1}a_{n-2}\ldots\ldots a_1a_0$. To produce an overflow, it is sufficient (in unsigned multiplication) to have a "carry out" of the most significant bit. For example, consider 3-bit representations of 5 and 6. The multiplication of (5*6) = (101 * 110) should yield 11110, but in a 3-bit answer, the most left 2-bits are lost and the result is 110 = 6.

The result of unsigned/signed multiplication will be out-of-range when it generates a carry, which used to flag the error. The output bits remain valid, but only as the n significant bits of 2n-bit result. The full result can be computed using the “carry out” generated from the most significant bits of the result and increasing the number of bits that can hold the number.

A hardware implementation defines its own process that it will execute for some instances of undefined behavior. Undefined behavior occurs when the result is not mathematically defined or when the result is not in the range of representable values for its type. The immediate consequences of a signed integer overflow mostly yield one of the following: 1)- a mathematically incorrect result; 2)- some sort of trap that may appear as an implementation-defined signal; 3)- raising an exception; and 4)- terminating the program.

Integer overflow detection should be implemented before its occurrence, not after its occurrence. The reason for this is that the overflow in some programming languages causes undefined behavior, so on some machines the program will never reach the call to the function that checks whether an overflow occurred or not. As an example, consider the following function skeleton code:

```
// sizeof (int) may be any number
Integer  a, b, c
Boolean Overflow;                          /* Flag */
/* Assume arbitrary values assigned to a and b */
c = a * b;                                 /* Possible overflow */
Overflow = Check_Integer_Overflow ( );     /* True if overflow occurs, or false otherwise */
```

Figure 1: Example of unreachable call to overflow detection after the multiply operation

In a language like C, there is no way to reach the last statement of Figure 1 (provided that there is an overflow in (c = a * b)). Another reason to check for an overflow before its occurrence is the absence of a compiler up to the knowledge of the authors that has a portable way to detect an overflow after its occurrence. Therefore, it is recommended to check for possible overflow before we apply the multiplication on any two numbers that generates overflow.

Multiplication overflow should be detected in advance so that we can take the necessary actions to handle it. There are two types here: unsigned numbers and signed numbers. The unsigned numbers case is a special case of signed numbers. Therefore, we will cover the signed numbers case. If the overflow changes only one sign of the two operands then we can check for overflow by verifying the sign of the answer. For this case, consider the following sign of truth table

Multiplier	Multiplicand	Product
+	+	+
+	-	-
-	+	-
-	-	+

Let us assume that the + sign represent the true value (represented by non-zero integer in C/C++) and – represent the false value (represented by zero value in C/C++). Therefore, we can check for potential overflow from the following equation.

Overflow = ((Multiplier sign XOR Multiplicand sign) !=Product sign)   …(1)

Where overflow is a logical variable that will be evaluated to true or false. If the value of the variable **Overflow** is true then an overflow occurred, otherwise the overflow did not occur probably. The previous test detect for overflow in most cases, but not all of them. The following is a procedure, which detects overflow in all cases

```
Boolean Test_Multiply (int x, int y)
Begin
    int product = x * y;
```

```
    If ( y !=0 AND (product / y) !=x)
        return true;        //  Overflow will occur and normal multiplication should be avoided.
    else
        return false;       //  No Overflow  End If
End
```

Figure 2: C-Like function to detect overflow in the multiplication

The authors recommend using the test in equation (1) to check for overflow. If variable ***Overflow*** in equation (1) is false, then we need to apply procedure Test_Multiply. A question will arise about this procedure when the product generates undefined behavior. The solution to this is to avoid traditional multiplication and use one of the algorithms that described later in this paper for multiplication.

The overflow detection should be followed by a mechanism to do the multiplication although we predict the occurrence of the overflow. The following section discusses how the mechanism of handling it in some programming languages by considering C and Java languages as two examples.

Currently, many programming languages raise an exception, which is the standard solution to this problem. In this case, it will raise an exception without treating the overflow problem. This is a way of detection and reporting the overflow rather than solving it. This treatment does not produce the required answer for the arithmetic operations. The later solution is not standard, but many programmers use it commonly in practice.

The overflow problem can be reduced by using long integer arithmetic. There are three basic strategies for implementing long integer arithmetic. The first strategy, which is called the default strategy, is implemented in the traditional long integer arithmetic package. The second strategy is to use Gnu Multi-Precision Package (GMP) as a supplementary long integer arithmetic package. The third strategy is to use GMP as the primary long integer arithmetic package. The GMP libraries are available at http://www.swox.com/gmp.

3. C LANGAUGE APPROACH

The procedural language, which we will take as an example, is the C language. C does not pay attention to the carry or overflow; it simply leaves the problem to the programmer (it gives no access to the carry or overflow flags, which are needed to verify the occurrence of overflow).

The standard of C language says that overflow is simply undefined behavior. Therefore, you will not be able to detect it after its occurrence because the program will go into the twilight zone. Thus, if you would like to detect overflow, you must explicitly write some code before the multiply operation as we

mentioned previously. Of course, for a given C compiler, overflow of integer arithmetic multiplication may be well defined, but the code will not be portable.

In C, we do not need to generate such an overflow to determine the value of a carry or an overflow flag that would result from multiplication. We can determine whether the multiply operation would overflow before performing it. Take for example a*b where **a** and **b** represent integer values. This operation overflows if the result would be greater than MAX_INT or less than MIN_INT. An expression that determines weather a*b would overflow in C is:

```
Overflow = (((a > 0) & (b > 0)) || ((a < 0) & (b< 0))) ? ((a > MAX_INT / b) || ( b >
        MAX_INT))  :  (( a < MIN_INT / b) || ( b < MAX_INT / a))
```

Where & means logical AND and || means logical OR in C language. Note that C language permits throwing overflow exception on integer overflow, but the implementation is not obligatory. Thus, the programmer must test for overflow by himself. In this context, C does provide any of the basic facilities such as a test for overflow.

4. JAVA APPROACH

Integer overflow in Java specification is not detectable, thus, it is the responsibility of the programmer to verify the values by checking if an overflow occurs or not. In Java, the result of integer overflow operation is specified to be different from the arithmetically correct results by 2^n where n is a 32-bit for type integer.

Java provides a solution to the overflow by using the Big Integer class (*BigInt*) instead of integer. The general mechanism of big integers is as follows: For integers larger than a certain size (2^{32} on most machines, 2^{64} for some others) we use a “large integer” library. All large integer libraries store each integer in multiple machine words. For example, if we have a 128-bit number, and the integer size of our machine is 32 bits, then it will use four machine words to store that number.

The implementation of *BigInt* (Big Integer) class take care of the overflow so that it will not be discarded, rather the results in a carry are handled just like pencil and paper arithmetic. The difference in the implementation is the use of base 2 instead of base 10.

The *BigInt* class in Java allows the programmer to use very large values as long as there is sufficient memory. A big integer is an integer, which does not overflow. Internally, a big Integer (an object of class *BigInt*) is an array of single digit values, and the array can grow as large as we need. Big Integers therefore will not overflow. The *BigInt* class provides many constructors and methods,

among these methods is the multiply method. The multiply method returns the answer (which is an object) through the **this** pointer.

5. DETECT AND DO ALGORITHM

The programming languages that does not support enough cure for the overflow problem should use a different representation. One of these methods is the use of arrays. The other method is to use linked lists. This paper uses three different methods. The first method is the arrays, the second is linked list, and the third method divides the number into two halves. It works for double the size of the word and has limited capabilities.

The arrays method represents a number as a sequence of digits stored in an array of characters. Then, we can write a function to do multiplication on those arrays, and then make them as large as we want. Following is our algorithm for doing the multiply operation based on arrays.

Algorithm Multiply( )

```
// Declare the following
// S1, S2 strings
// Answer is one-dimensional array of type integer where each element contain part of the answer
// LengthMax is an integer temporary variable which contains the length of the largest number
Begin
        Input Number1 and Number2 which you would like to do the multiplication.
        For each number do the following
        Store in strings S1, S2, the contents of Number1 and Number2 respectively (i.e. S1 =
                string (Number1), S2 = string(Number2)). Thus, each string will hold the contents
                of the number but in character form.
        If (strlength (S1) > strlength (S2))
        {
                LengthMax = strlength (S1 )
                LengthMin =  strlength (S2)
        }
         else
        {
                LengthMax = strlength (S2)
                LengthMin = strlength (S1)
        }
        For (I = 0; I < LenghtMin; I ++) do
        {
                count = 0
                For (j=0;j<LengthMax, j++) do
                {
                        Answer[I] = Answer[I] + (integer (S1[J]) * 10^j )* integer (S2[I])
                                                        // casting and multiplication
                        //Increase count by 1
                }
        }
        For (I = 0; I < LenghtMin; I ++) do
                Break Answer[ I ] into its individual bits
        For (I = 0; I < LenghtMin; I ++) do
                Shift Answer[ I ] I bits to the left
```

```
        For (J = 0; J < 2 * LenghtMax; J ++) do
        {
                Total = 0
                Add bit No. J for all answers and print the total
        }
End
```

Figure 3: The multiplication of two numbers

In this algorithm, we convert each number into an array of characters (strings). Then the length of the longest two strings (S_1, S_2) is assigned to a variable called **LengthMax**. Note that we convert each decimal digit into a character form, where each character occupies two bytes (one for the character, and the other unused temporally). Thus, if we have an integer number that consists of two decimal digits, then they will be stored in 4 bytes as character variable. Figure 5 explains how we get the answer.

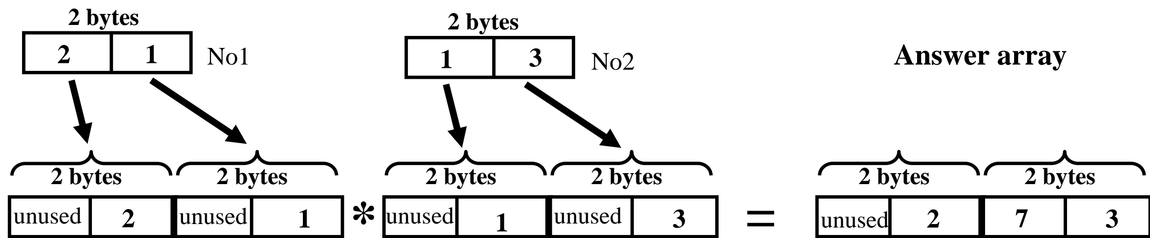

Figure 4: The multiplication of two overflowed numbers

The generated answer will not fit into the added resultant. In order to store the value of the answer in a variable we might do one of the following two approaches: 1)- Frequency Repetition, or 2)-Scaling.

In frequency repetition, we count how many times we have reached the maximum integer available on the machine. Thus, the answer will be the contents of our **answer array** minus the maximum integer; meanwhile we have to create a variable called **Frequency**, which will contain the value 1. If our answer overflows one more time, then the value of the frequency will be incremented whenever saturation occurs.

As an example, consider Figure 4, where the **answer array** contains (2 and 73). To find the result, we need to subtract MAX_INT from **answer array** and increase the frequency by one. This step will be repeated until the content of **answer array** becomes less than MAX_INT. The same thing applies to negative numbers, however, we compare with MIN_INT.

The other method is the scaling method where we divide our number by some factor and store the factor in a variable called **Scaling**. Note that this method might generate rounding in the number.

To correct the output value of our answer, we trace **answer array** element by element to print the required output.

A second algorithm is to use the linked list. Following is the exact algorithm for it
Algorithm Multiply_using_linked_list( )
Begin

```
    Input Number1 and Number2 which you would like to do the multiplication.
    For the first variable, create a linked list and store each digit in a one node
    For the second variable, create a linked list and store each digit in one node.
    Create a linked list of length 2n that will hold  the answer.
    For each node (digit) in the second linked list do
        Multiply the node with the first linked list and store the result in two nodes of the
        answer linked list.
    Print the result using all the nodes of the second linked list
```

End

A third method can be applied for a limited range of numbers. In this method, we split the number into two halves and do normal multiplication. This allows us to achieve the multiplication for 64 bits. However, if we use some special data types such as __int64, which is available in Visual Studio version 6, then we can do the multiplication operation on numbers each one has a length of 128 bits. Following is algorithm *Multiply_using_splitting( )* that divide the number into two halves without using any special data types then it performs the multiplication operation.

Algorithm Multiply_using_splitting (No1, No2)
Begin

```
/* This program computes the entire 64-bits of the product and set
     mask1  =  65535;
    int a_half1  =  No1  &  mask1;      // Get the least 16 bit of a
    int a_half2  =  No1  >> 16;         // shift right to get the most 16 bit of a
    int b_half1  =  No2  &  mask1;   // Get the least 16 bit of b
    int b_half2  =  No2  >> 16;         // Shift right to get the most 16 bit of b
     int r1,  r2,  r3,  r4,  r12,  r34
    r1  =  b_half1  *  a_half1;
    r2  =  b_half1  *  a_half2;
    r3  =  b_half2  *  a_half1;
    r4  =  b_half2  *  a_half2;
    r12  =  r1  +  r2;
    r34  =  r3  +  r4;
    Print the answer which is a concatenation of (r12 ,  r34);     // get 64 bit result
```

End

A fourth multiply algorithm is to use built in data structure called ArrayList which exists in C# (ArrayList handles numbers as arrays but it allows the programmer to use built in methods). The algorithm of it is a kind of similar to algorithm *Multiply()* presented previously. However, it is designed for C#.

The previous paragraphs describes many variations for the Multiply algorithm. Some of these variations are specific to a certain language like C#. Following is a generic “Detect and Do” algorithm that handles overflow in multiplication.

The Detect and Do algorithm will be as follows:

```
Detect_and_Do (variable1, variable2)
// Frequency is an integer variable initialized to zero
Begin
        If there is no overflow then
                Do the normal multiplication
        else
                Apply the multiply algorithm        //any version
                Output the result using the corresponding data structure that holds the answer
        End If
End
```

Figure 5: Detect and Do algorithm

An experiment conducted to see the time required by our algorithm. Table 1 shows the normalized time required by different variations of Multiply algorithm, the time required by regular multiplication, and the use of __int64 in C/C++ language

Approach	Regular	Detect and D0 (Multiply)	Detect and Do (Multply_using _linked List)	Detect and Do (Multiply_usin g_splitting)	__int64
Normalized time	1	3.4	4.4	4.6	1.7

Table 1: The time required by different approaches using C/C++

Note Visual C++ version 6, which contains a C compiler, allows the use of 64-bit integer variables on 32-bit machines using the __int64 data type. In table (1), the “Detect and Do” algorithm requires extra time, but it generates safer arithmetic results. In this table, a value like 3.4 means that the required time for multiplication using “Detect and Do” is about three times if we use regular operations. However, there is a possibility of overflow using regular approach. The same experiment was repeated using Java. The results of Java are listed in table 2.

Approach	Regular	Detect and D0 (Multiply)	Detect and Do (Multply_using _linked List)	Detect and Do (Multiply_usin g_splitting)	Big Integer
Normalized time	1	5.1	3.9	4.2	4.7

Table 2: The time required by different approaches using Java

All the previous programs where run many times and the average run time were taken as a measurement in Table 1 and 2. In these experiments, we tried to freeze all unnecessary operations of CPU while the program is running.

6. CONCLUDING REMARKS

The overflow problem occurs whenever the multiplication of two binary numbers generates a result that does not fit into the same number of bits. The overflow can have a large impact on the execution speed and on the software quality of the final product either directly or indirectly [Burgess, 1995]. Many programming languages do not specify what may happen in the event of overflow. Therefore, the results are not those the programmer is intended to get.

A careful programmer will only rely on a minimum range for every variable, but not on an upper bound. For example, a 32-bit application will handle 16-bit values, but the opposite is not true. However, if the programmer blocked with the maximum size, then he/she can use our suggested approach.

The suggested approach avoids the idea of increasing the capabilities of computer hardware. The compensation for this is an extra cost in the execution time. It is recommended and worth the addition of a built-in function that takes two integer parameters and determines whether an overflow will occur or not. This built-in function should be supported in the programming languages that ignores overflow.

REFEREENCES

1- Borodin, A. El-Yaniv, R. Online Computations and Competitive Analysis. Cambridge University Press, 1998.

2- Burgess, C. J., “Software Quality Issues When Choosing a Programming Language,” Proc. of the Third International Conference on Software Quality Management, Vol. 2 of Software Quality Management III, pp. 25-31, Spain, Apr. 1995.

3- Elguibaly, F., “Overflow Handling in Inner-Product Processors,” IEEE Trans. on Circuits and Systems –II: Analog and Digital Signal Processing, Vol. 47, No. 10, pp. 1086-1090, Oct. 2000.

4- El-Qawasmeh, E., “Handling Overflow in Integer Addition in Inline Computations,” Digital Information Management Journal, India, Vol. 1, No. 3, pp. 129-135, Sept. 2003.

5- Fiat, A. and Woeginger, G. (Eds.), Online Algorithms: The State of the Art. Lecture Notes in Computer Science 1442. Springer, Berlin, 1998.

6- Irani, S. and Karlin, A. Online computation. In: Dorrit Hochbaum (editor), Approximation Algorithms for NP-hard problems, chapter 13, pp. 521-564. PWS Publishing Company, Boston, 1997.

7- Gok, M., “Integer Multiplication with Overflow Detection or Saturation,” Master’s Thesis, Lehigh University, PA, USA, 2000.

8- Grove, E., Kao, M.-Y, Krishnan, P. and Vitter., J., “Online Perfect Matching and Mobile Computing,” Proc. of the Fourth Workshop on Algorithms and Data Structures (WADS '95), Kingston, Ontario, pp. 194-205, Aug. 1995.

9- Lang, T. and Bruguera, J. D., “Multilevel Reverse-Carry Computation for Comparison and for Sign and Overflow Detection in Addition,” Proc. of the International Conference on Computer Design, pp. 73-79, Oct. 1999.

10- Michael J. Schulte, Pablo L. Balzola, Ahmet Aakkas, Robert W. Brocato, "Integer Multiplication With Overflow Detection or Saturation," IEEE Trans. on Computers Vol. 49, No. 7, pp. 681-691, June 2000.

11- Mustafa G. "Integer Multiplication with Overflow Detection or Saturation," Master Thesis, Lehigh, USA, May 2000.

12- Schulte, M., Balzola, P., Akkas, A. and Brocato, R., “Integer Multiplication with Overflow Detection or Saturation,” IEEE on Computers, Vol. 49, No. 7, pp. 681-691, Jul. 2000.

13- Parhamin, B., “Zero, Sign, and Overflow Detection Schemes for Generalized Signed Digit Arithmetic,” Proc. of the 22nd Asilomar Conf. on Signals, Systems, and Computers, Pacific Grove, CA, pp. 636-639, Oct./Nov. 1988.

14- GMP web page. http://www.swox.com/gmp/

Filename: OverflowOct5Conference
Directory: C:\EYAS\PAPERS\Under_Writing\MultiplicationOverflow
Template: C:\Documents and Settings\Eyas\Application Data\Microsoft\Templates\Normal.dot
Title: Revisiting Overflow in Mulitiplication
Subject:
Author: Eyas El-Qawasmeh
Keywords:
Comments:
Creation Date: 10/12/2004 4:17 PM
Change Number: 2
Last Saved On: 10/12/2004 4:17 PM
Last Saved By:
Total Editing Time: 1 Minute
Last Printed On: 10/12/2004 4:17 PM
As of Last Complete Printing
Number of Pages: 12
Number of Words: 3,926 (approx.)
Number of Characters: 22,382 (approx.)

This document was created with Win2PDF available at http://www.daneprairie.com.
The unregistered version of Win2PDF is for evaluation or non-commercial use only.