Performance of an Occam/transputer implementation of interval arithmetic*

Ole Caprani
Computer Science Department, Aarhus University
Kaj Madsen
Institute for Numerical Analysis, Technical University of Denmark

August 1993

Abstract

Rounded interval arithmetic is very easy to implement by means of directed rounding arithmetic operators. Such operators are available in the IEEE floating point arithmetic of the transputer. When a few small pieces of assembly language code is used to access the directed rounding operators, the four basic rounded interval arithmetic operators can easily be expressed in the programming language Occam. The performance of this implementation is assessed and it is shown that the time consuming part of the calculation are not the directed rounding floating point operations as one might have expected. Most of the time is spent with transport of operands to and from the on-chip floating point unit and the procedure call/parameter passing overhead. Based on this experience the implementation is improved. This implementation runs with 0.15 MIOPS (Million Interval Operations Per Second) or 0.30 MFLOPS on an example interval calculation proposed by Moore. Furthermore, it is demonstrated that an advanced interval language compiler may provide a performance of 0.30 MIOPS or 0.59 MFLOPS on this example calculation.

*Presented at the conference on Nomerical Analysis with Automatic Result Verification Lafayette, Louisianna, 1993.

1 Introduction

Rounded interval arithmetic as defined by Moore [1], is easy to define in terms of directed rounding arithmetic operators. E.g. with interval addition defined as:

$$A = [\underline{a}, \overline{a}], B = [\underline{b}, \overline{b}], A + B = [\underline{a} + \underline{b}, \overline{a} + \overline{b}]$$

we obtain the following rounded interval addition:

$$A \oplus B[\underline{a} + < \underline{b}, \overline{a} > + \overline{b}]$$

where $+ <$ is downwardly rounding addition and $> +$ is upwardly rounding addition written with a notation as in PASCAL-XSC, see Wallis [6]. If such directed rounding operators are available in a programming language as in PASCAL-XSC, implementation of rounded interval arithmetic is very easy. E.g. rounded interval addition $\oplus$ can be implemented in PASCAL-XSC as the following procedure, see Wallis:

```
procedure iadd(a, b: interval; var res: interval);
begin
      res.inf := a.inf + < b.inf;
      res.sup := a.sup > + b.sup;
end
```

On processors that conforms to the IEEE Standard 754, [3], implementation of rounded interval arithmetic is straight forward since the IEEE floating point standard includes directed rounding operators. Furthermore, since most processors implement both the normal floating point operations and the directed rounding operations in hardware so the time taken are approximately the same for normal and directed rounding operations it seems that an efficient rounded interval implementation can be obtained on such processors. This is investigated in the following. Efficiency of an IEEE based interval implementation has been assessed for one processor, namely the transputer T800, [4]. This processor has an on-chip floating point unit (FPU) that conforms to the standard.

In the next section it is shown how rounded interval arithmetic can be implemented on the transputer along the lines above in the programming language Occam, [5]. In the third section we assess the performance of this implementation by means of one of the test cases of Moore, [2]. It turns out that the execution time of the Occam/transputer implementation is not dominated by the time it takes to execute the directed rounding operations on the FPU. The dominating factors turn out to be the overhead time of the procedure call/parameter passing, the endpoint inspection in multiplication and division, and the time it takes to transport the operands to and from the on-chip FPU. This is described in the fourth section. In the fifth section some improvements are given and a simple and reasonably efficient implementation of rounded interval arithmetic is obtained. To carry the improvement even further a compiler should be written e.g. for an interval language as PASCAL-XSC to produce efficient code for the interval operations on the transputer. This is demonstrated in the last section. All the experiments have been carried out on a 20 MHz T800 transputer, [4].

2 Simple interval arithmetic on the transputer

Since it is not possible from Occam to access the rounding mode of the floating point unit on the transputer some lines of assembly code is needed to do the job. E.g. the directed rounding operator $>+$ can be written as an Occam function with a body of assembly code:

```
REAL64 FUNCTION AddRoundUp(VAL REAL64 a,b)
 REAL64 result:
 VALOF
  SEQ
   GUY
    LDLP a
    FPLDNLDB     -- FAreg := a
    LDLP b
    FPLDNLD      -- FBreg := a, FAreg := b
    FPURP        -- round to plus infinity
    FPADD        -- FAreg := FAreg >+ FBreg
    LDLP result
```

```
    FPSTNLDB     -- result := a >+ b
  RESULT result
:
```

FAreg and FBreg are the names of the two top elements of the three valued stack of the FPU, [4]. The other rounding operations can be written quite similarly as Occam functions. With these Occam functions the rounded interval arithmetic operations can be programmed as Occam procedures. Floating point intervals are represented as the Occam type [2]REAL64, i.e. an array of two floating point numbers of type REAL64. The array element indexed 0 is the left endpoint (left= 0), array index 1 gives the right endpoint of the interval (right=1). With these definitions the Occam procedure for addition is straight forward:

```
PROC Interval.Add([2]REAL64 c , VAL [2]REAL64 a,b)
 SEQ
  c[left] := AddRoundDown(a[left],b[left])
  c[right]:= AddRoundUp(a[right],b[right])
:
```

Interval subtraction is similar. In the procedure for interval multiplication the signs of the interval operands are analyzed to form the endpoints of the result with as few multiplications of operand endpoints as possible [1]. Division is performed by means of interval multiplication.

3 Performance of simple interval arithmetic

In [2], Moore describes an interval implementation on the PDP 11/40E by means of microprogrammed directed rounding operations. To assess the efficiency of the implementation he calculated the sum:

$$s = \sum_{i=1}^{9000} \left(\frac{(i-1)*(i-2)*(i^2-2)}{(i+1)*(i+2)*(i^2+2)} - 1\right)^2$$

This was carried out by means of floating point arithmetic and rounded interval arithmetic. Moore found that the calculation performed by means of interval arithmetic was 2 times slower than the calculation performed with floating point arithmetic. This is explained as follows: Every interval operation involves approximately 2 floating point operations and the time taken for a floating point operation and a directed rounding operation is approximately the same.

The sum s has also been calculated on the transputer by means of floating point arithmetic (this is denoted realSum) and by means of the interval procedures of the previous section (this is called intervalSum). The programs are included as appendix 1. The results were:

	realSum	intervalSum
time	0.144 sec	1.61 sec
result	8.5780573082	[8.578057308265, 8.578057308282]

The factor of this implementation is thus 11.2. Certainly, this indicates that the floating point operations are not the time consuming part of the interval calculation.

To assess this let us first consider the FPU utilization. The number of floating point operations per second of the two calculations are ($9000 * 16$ floating point operations in realSum; in intervalSum each interval operation is taken as 2 floating point operations):

realSum	intervalSum
1.0 MFLOPS	0.18 MFLOPS

The performance of the realSum calculation is close to other performance results for heavy FPU bound calculations e.g. the Livermore Loops with 1.5 MFLOPS, [4]. With a floating point addition time of 450 nsec, [4], i.e. 2.2 MFLOPS when only addition operations are performed, this seems to indicate that the FPU is busy more than 50 % of the time during the realSum calculation. The utilization of the FPU in the intervalSum calculation suggests that there is room for improvements.

4 Analysis of the performance

Now, the inefficiency of the intervalSum calculation is caused by the procedure call/parameter passing overhead, the investigation of the endpoints to be multiplied in the interval multiplication and division, and the time it takes to transport operands to and from the floating point unit. This can be demonstrated as follows: The intervalSum calculation has been performed with empty bodies of the directed rounding interval functions. This takes 1.22 seconds. This means that 76 % of the execution time of 1.61 seconds is procedure call/parameter passing overhead and investigation of endpoints. To assess the operand transport overhead, realSum has been rewritten so all intermediate results are stored explicitly as in intervalSum. This version of realSum takes 0.216 seconds. This gives 0.66 MFLOPS. The difference of 33 % or 0.072 seconds shows that the utilization of the FPU depends on the use of the operand stack of the unit. In the second version of realSum all floating point operations are preceded by the transport of two operands from memory to the stack and followed by a transport of the intermediate result back to memory. These explicit transports are so time consuming compared to floating point operations that the extra time of the second version is spent with these transports. E.g. the addition time is approximately 450 nsec and the time it takes to transport the two operands and the result is approximately 750 nsec.

Now, if we consider the time for the intervalSum calculation, this can be written as the sum of the three contributions measured above:

$$t_{intervalSum} = t_{call/parameter,endpoint} + 2 * t_{transport} + 2 * t_{realSum}$$

Since 1.61 seconds is approximatelyequal to $1.22+2*0.072+2*0.144$ seconds it seems that we have accounted for most of the contributions to the time for the intervalSum; the major contribution being the procedure call/parameter passing overhead and endpoint investigation.

5 Improved interval arithmetic

The procedure call/parameter passing overhead suggests that an immediate improvement of the simple implementation i8 to program the bodies of the interval procedures in assembly to avoid the overhead of the calls to the directed rounding functions. E.g. interval addition as follows:

```
PROC Interval.Add([2]REAL64 c , VAL [2]REAL64 a,b)
 GUY
    -- c[left] := AddRoundDown(a[left],b[left])
    LDLP a[left]
    FPLDNLB     -- FAreg   := a[left]
    LDLP b[left]
    FPLDNLDB    -- FBreg   := a[left] , FAreg := b[left]
    FPURM       -- round to minus infinity
    FPADD       -- FAreg   FAreg < + FBreg
    LDLP c[left]
    FPSTNLDB    -- c[left] := a[left] <+ b[left]


    -- c[right] := AddRoundUp(a[right],b[right])
    LDLP a[right]
    FPLDNLDB     -- FAreg   := a[right]
    LDLP b[right]
    FPLDNLDB     -- FBreg   := a[right] , FAreg := b[right]
    FPURM        -- round to plus infinity
    FPADD        -- FAreg   FAreg > + FBreg
    LDLP c[right]
    FPSTNLDB     -- c[right]:= a[right] <+ b[right]
:
```

This gives the following execution times:

realSum	intervalSum
0.144 seconds	0.97 seconds
1.0 NFLOPS	0.30 MFLOPS

Now the factor is reduced to 6.7 and the FPU utilization is almost doubled.

6 Optimized interval calculation

To improve the execution time even further the procedure calls of the intervalSum program can be substituted with procedure bodies to obtain in-line code for the interval operations e.g. as produced by a simple compiler. This results in an execution time of 0.75 seconds and a factor of 5.2. Now, the floating point unit utilization has been improved to 0.38 MFLOPS.

This performance can be improved once more if we remove the endpoint investigation in the multiplications of the numerator and denominator in all but the first term of the sum. The reason is that all intervals in the numerator and denominator are non-negative except for the intervals of the first term. Hence, the endpoint investigations can be skipped in the six multiplications of the numerator and denominator. This results in an execution time of 0.58 seconds and a factor of 4.

As a final improvement we can demonstrate how the calculation can be rearranged in order to make use of the stack of the floating point unit. In the realSum the instructions produced by the Occam compiler to calculate the numerator:

$$(i-1)*(i-2)*(i^2-2)$$

make use of the stack so that only 8 operand transfers are made instead of the 18 which would be needed if 3 transfers were involved in every floating point operation.

The stack can be used just as efficiently in the intervalSum calculation if we rearrange the interval calculation of the mlmerator so that we first calculate the left endpoint as:

$$(i- < 1)* < (i- < 2)* < (i* < i- < 2)$$

and then the right endpoint by a similar expression. This is of course only possible for all but the first term of the sum. The execution time is now reduced to 0.49 seconds, the factor is 3.4 and the FPU utilization is 0.59 MFLOPS. This is almost a factor of two faster than the procedure based implementation of the previous section.

There are still room for improvements. The test for division by a zero interval can be removed. An interval power operation can be used instead of the explicit usage of interval multiplication in the outermost second power operation of each term. Etc. However, the improvements reported already suggests that to obtain efficient interval calculation we need compilers for interval languages as PASCAL-XSC that can produce the kind of code that has been produced above by hand for the Moore sum.

The performance of the various versions of the interval calculation presented can be summarized as in the following table:

version	seconds	MIOPS	interval/real
simple	1.61	0.09	11.2
in-line rounding	0.97	0.15	6.7
in-line procedures	0.75	0.19	5.2
reduced endpoint checking	0.58	0.25	4.0
reduced transfers	0.49	0.30	3.4

7 Discussion

Directed rounding in hardware makes it easy to implement the basic interval operations. Such an implementation is efficient in terms of absolute speed (MIOPS) compared to a software implementation. E.g. for a PASCAL-XSC software implementation on IBM-PC, intel 486, we obtain 35 seconds for realSum and 75 seconds for intervalSum (The programs are included in appendix 2). This corresponds to 2000 IOPS, which is rather poor compared to the hardware implementation on the transputer.

It is, however, hard to obtain a good performance of an interval calculation relative to a floating point calculation by means of directed rounding in hardware. A factor of $6-10$ is easily obtained for the implementation on the transputer. But to get close to a factor of 2 requires much effort. The reason is that the overhead of e.g. endpoint inspection has a greater influence on the time for the interval calculation when fast directed rounding operations are used.

8 Conclusion

A reasonably efficient procedure based implementation of the basic interval operations have been described. The performance has been tested on the Moore sum and a performance of 0.15 MIOPS (Million Interval Operations Per Second) is obtained.

It seems that an even more efficient implementation of interval calculations can be obtained e.g. on the transputer if compilers are written for interval languages as PASCAL-XSC to produce code that avoids the call/parameter passing overhead of the procedure based implementation, utilize the stack of the floating point unit and optimize the endpoint investigations of the multiplication and division operations. The efficiency gained by the usage of this kind of optimized code in the Moore sum was a factor 2.

References

[1] [1] Moore,R.E,
Interval Analysis,
Prentice Hall, 1966.

[2] [2] Moore,R.E,
Microprogrammed Interval Arithmetic,
SIGNUM Newsletter, vol. 15, no 2, page 2, 1980.

[3] [3] IEEE,
ANSI/IEEE Standard 754-1985 for Pinary Floating-Point Arithmetic, IEEE Comp. Soc., Los Alamitos, Calif. 1985.

[4] [4] Mark Homewood et. al.,
The IMS T800 Transputer, IEEE Micro, Oct. 1987.

[5] [5] INMOS Limited
Occam2 Reference Manual
Prentice Hall, 1988.

[6] [6] Peter J. L. Wallis (ed.),
Improving Floating-Point Programming,
John Wiley & Sons, 1990.

[7] [7] R. Klatte et al.,
PASCAL-XSC,
Springer-Verlag, 1991.

Appendix 1 realSum and intervalSum in Occam

The real and interval calculation of the Moore sum can be programmed as follows in Occam:

```
PROC RealMoore(REAL64 s)
     VAL REAL64 one  IS 1.0(REAL64):
     VAL REAL64 two  IS 2.0(REAL64):
     VAL REAL64 zero IS 0.0(REAL64):



    INT i:
    REAL64 i.r, sum, term:
    SEQ
       sum:=zero
       SEQ i=1 FOR 9000
           SEQ
               i.r := REAL64 ROUND i
               term:= (i.r-one)*( (i.r-two)*((i.r*i.r)-two) )
               term:= (term/((i.r+one)*((i.r+two)*((i.r*i.r)+two))))-one
               sum := sum + (term*term)
    s:= sum
:


PROC IntervalMoore([2]REAL64 s)
     VAL [2]REAL64 one  IS [1.0(REAL64), 1.0(REAL(64)]:
     VAL [2]REAL64 two  IS [2.0(REAL64), 2.0(REAL(64)]:
     VAL [2]REAL64 zero  IS [0.0(REAL64), 0.0(REAL(64)]:
     INT i:
     [2]REAL64 i.int, sum, t1,t2,t3,t4:
     SEQ
         sum:=zero
         SEQ i=1 FOR 9000
           SEQ
               i.int := [REAL64 ROUND i, REAL64 ROUND i]

               Interval.Sub(t1,i.int,one)
```

```
                Interval.Sub(t2,i.int,two)
                Interval.Mul(t3,t1,t2)
                Interval.Mul(t1,i.int,i.int)
                Interval.Sub(t2,t1,two)
                Interval.Mul(t1,t3,t2)   -- t1 = (i-1)(i-2) (i*i-2)

                Interval.Add(t2,i.int,one)
                Interval.Add(t3,i.int,two)
                Interval.Mul(t4,t2,t3)
                Interval.Mul(t2,i.int,i.int)
                Interval.Add(t3,t2,two)
                Interval.Mul(t2,t4,t3)   -- t2 = (i+1)(i+2)(i*i+2)

                Interval.Div(t3,t1,t2)
                Interval.Sub(t1,t3,one)
                Interval.Mul(t2,t1,t1)   -- t2 = (t1/t2 - 1)**2

                Interval.Add(t1,sum,t2)
                sum := t1
     s:  sum
:
```

Appendix 2 realSum and intervalSum in PASCAL-XSC

The real and interval calculation of the Moore sum can be programmed as follows in PASCAL-XSC:

```
program RealMoore (input,output);
var s, term :  real;
    i       :  integer;
begin
    s := 0.0;
    writeln('start');
    for i:= 1 to 9000 do
    begin
         term:=(i-1.0)*(i-2.0)*(i*i-2.0);
         term:=term /
                     ((i+1.0)*(i+2.0)*(i*i+2.0));
```

```
        term:=term -1.0;
        s := s + term*term;
    end;
    writeln(s);
end.


program IntervalMoore (input,output);
use i_ari;
var s,term,i_r: interval;
    i         : integer;



begin
     s:=intval(0.0);
     writeln('start');
     for i:= 1 to 9000 do
     begin
          i_r:=intval(i);
          term:=(i_r-intval(1.0))*(i_r-intval(2.0))*(i_r*i_r-intval(2.0));
          term:= term /
                 ((i_r+intval(1.0))*(i_r+intval(2.0))*(i_r*i_r+intval(2.0)));
          term:=term-intval(1.0);
          s:=s + term*term;
     end;
     writeln(s);
end.
```