

# INTERVAL ARITHMETIC USING SSE-2

BRANIMIR LAMBOV

**ABSTRACT.** We present an implementation of double precision interval arithmetic using the single-instruction-multiple-data SSE-2 instruction and register set extensions. The implementation is part of a package for exact real arithmetic, which defines the interval arithmetic variation that must be used: incorrect operations such as division by zero cause exceptions and loose evaluation of the operations is in effect. The SSE2 extensions are suitable for the job, because they can be used to operate on a pair of double precision numbers and include separate rounding mode control and detection of the exceptional conditions. The paper describes the ideas we use to fit interval arithmetic to this set of instructions, shows a performance comparison with other freely available interval arithmetic packages, and discusses possible very simple hardware extensions that can significantly increase the performance of interval arithmetic.

## 1. INTRODUCTION

Numerical computations on a computer are plagued by the problem of roundoff error and its accumulation. Very often we trust results obtained using floating point computations, whose truth we are not certain about. Verifying the correctness of such results may be a very difficult task, to solve which various mathematical or programming tools may be used.

In this paper we will describe a package which implements interval arithmetic, one of the methods that can be used as one of the steps towards solving the problem, in a very efficient way. The package itself is part of a bigger library for exact real computation, which can be used to solve the roundoff error problems completely by providing certified accuracy.

Interval arithmetic is a method of finding lower and upper bounds for the possible values of a result by performing a computation in a manner which preserves these bounds (for an introduction, see [4]). The IEEE-754 standard for floating point arithmetic [7] has useful features to aid fast interval arithmetic, namely the directed rounding modes that should be present with every IEEE-754 implementation. Unfortunately, in some processor architectures, notably Intel's x86, it is non-trivial to effectively use them, as switching the rounding mode for an operation requires significantly more time than the operation itself. Even when one takes into account the fact that one of the directed rounding modes can be emulated by operations on negated values rounded in the other direction, an interval arithmetic package has to be aware that users may mix interval with standard floating point arithmetic and would still require repeatedly switching the rounding modes.

Fortunately, the newer generations of the x86 architecture provide an additional set of registers with its own rounding control, the SSE2 double-precision floating point registers [8]. They can coexist with the old x87-style floating point, which is still the register and instruction set used most widely. Thus, to serve all purposes, we can reserve the SSE2

---

*Key words and phrases.* interval arithmetic.

register and operation set for interval arithmetic and leave x87-style floating point for any standard floating point operations that the user may be performing.

The SSE2 instruction set can also work on packed data, as every SSE2 register can contain and operate on a pair of double-precision floating point numbers. Since an interval is in fact a pair of bounds, one SSE2 register can be used to hold an interval, which nullifies the additional register pressure that interval arithmetic would normally exert.

The package described in this paper is part of the *RealLib* library for exact real number computations [5] and makes use of this register and instruction set to implement very fast machine precision interval arithmetic.

Often an interval arithmetic package would be expected to continue computations even if it reaches exceptional conditions, such as division by an interval that contains zero. This does not make sense for interval arithmetic within exact arithmetic, as in the latter one assumes that in any such case the bad interval will improve in subsequent reiterations at higher precision and ultimately one can reach a precision which avoids the exceptional condition. Thus, contrary to the modern trends of interval arithmetics, for us actually throwing an exception in such a case is the desirable action, so that computations can restart at higher precision as soon as possible.

Additionally, the package implements loose interval arithmetic, i.e. it ignores the portions of the argument of an operation that are outside its domain, e.g. the negative parts of the argument in a square root, meaning for example that  $\sqrt{[-1, 4]} = [0, 2]$ . This is the proper mode of operation to ensure that  $\sqrt{0}$  is computable in exact real arithmetic.

## 2. KEY IDEAS

Normally, interval arithmetic based on floating point would use two rounding operations,  $\Delta$  (rounding towards  $+\infty$ ) and  $\nabla$  (rounding towards  $-\infty$ ). By default IEEE-floating point uses rounding to nearest, which is not useful for our purposes.

We already mentioned that switching the rounding mode has a detrimental effect on the performance of floating point operations, thus we would want to avoid all rounding mode switches. We will only do this once, at the beginning of a computation<sup>1</sup>, setting the rounding mode to rounding towards  $-\infty$ . To compute lower bounds of the results, we will directly use the floating point operation. To compute upper bounds, we will make sure that the result of the floating point operation is negated, thus making use of the identity

$$\Delta(x) = -\nabla(-x).$$

Seeing operations in the form above, compilers are usually overzealous<sup>2</sup> to fold the pair of negations and destroy the effect we want to achieve. To avoid this, at the same time keeping down the number of required operations, we make sure that we always keep the high bound of the interval negated, i.e. our representation of the interval  $x = [\underline{x}, \bar{x}]$  is the pair  $\langle \underline{x}, -\bar{x} \rangle$ . (in the rest of this chapter we will assume every interval is represented in this fashion and will simply write  $x$  to mean  $[\underline{x}, \bar{x}]$  and  $\langle \underline{x}, -\bar{x} \rangle$ )

Three observations can be made directly from this:

- in this setting, the sum of  $x$  and  $y$  is evaluated by  $\langle \nabla(\underline{x} + \underline{y}), -\nabla(-\bar{x} - \bar{y}) \rangle$  which is achieved by a single instruction, `_mm_add_pd`.

<sup>1</sup>This is accomplished by the construction of a special object that also takes care of restoring the previous rounding mode after the interval computation has completed.

<sup>2</sup>The two negations have no effect on the rounding-to-nearest mode which is normally in place in C/C++ code, and on which many standard functions rely, thus this optimization is perfectly legal. Only our specific (non-standard) use of floating point makes it unwanted.

- changing the sign of an interval  $x$  is achieved by simply swapping the two bounds, i.e.  $\langle -\bar{x}, \underline{x} \rangle$ , achieved by a single instruction, *\_mm\_shuffle\_pd*,
- joining two intervals (i.e. finding an interval containing all numbers in both, or finding the minimum of the lower bounds and the maximum of the higher bounds) is performed as  $\langle \min(\underline{x}, \underline{y}), -\min((-\bar{x}), (-\bar{y})) \rangle$  in a single instruction, *\_mm\_min\_pd*.

The latter is used extensively in the computation of multiplication, division and other operations.

### 3. OPERATIONS

In this section we will give short remarks on our implementation of the basic operations on intervals. The operations include the arithmetic operators, including the special cases  $-x$ ,  $\frac{1}{x}$ , and  $x^2$ , absolute value and square root.

All the operations give tight bounds (i.e. the best possible enclosures after rounding).

3.1. **Addition.** Definition:

$$x + y = [\underline{x} + \underline{y}, \bar{x} + \bar{y}] \subseteq \langle \nabla(\underline{x} + \underline{y}), -\nabla((-\bar{x}) + (-\bar{y})) \rangle$$

Addition is implemented as a single *\_mm\_add\_pd* instruction. The negated sign of the higher bound ensures the proper direction of the rounding.

3.2. **Sign change.** Definition:

$$-x = [-\bar{x}, -\underline{x}] = \langle -\bar{x}, \underline{x} \rangle$$

This is a single swap of the two values, implemented as a *\_mm\_shuffle\_pd* instruction. No rounding is performed here.

3.3. **Subtraction.** Definition:

$$x - y = [\underline{x} - \bar{y}, \bar{x} - \underline{y}] \subseteq \langle \nabla(\underline{x} + (-\bar{y})), -\nabla((-\bar{x}) + \underline{y}) \rangle$$

Subtraction is implemented as  $x + (-y)$ , which corresponds to two processor instructions. This is the best that can be achieved with packed SSE2 instructions, because the formula requires a combination of the high bound of one of the arguments with the low bound of the other.

3.4. **Multiplication.** Definition:

$$(3.1) \quad xy = [\min(\underline{xy}, \underline{x}\bar{y}, \bar{x}\underline{y}, \bar{xy}), \max(\underline{xy}, \underline{x}\bar{y}, \bar{x}\underline{y}, \bar{xy})]$$

Unfortunately, the rounding steps are inseparable parts of the operations, this the equation above requires 8 multiplications. Using the fact that  $\Delta(\nabla(r) + \epsilon) \geq \Delta(r)$  (for  $\epsilon$  being the smallest representable positive number), one can do with 4 multiplications at the expense of some precision.

In our implementation we chose a different approach where we use four multiplications without sacrificing accuracy, by selecting the multiples based on the signs of  $\underline{x}$  and  $\bar{x}$ . More specifically, we use these observations:

$$(3.2) \quad xy = \begin{cases} [\min(\underline{xy}, \bar{x}\underline{y}), \max(\bar{x}\underline{y}, \underline{xy})], & \text{if } 0 \leq \underline{x} \leq \bar{x} \\ [\min(\underline{x}\bar{y}, \bar{xy}), \max(\bar{xy}, \underline{x}\bar{y})], & \text{if } \underline{x} < 0 \leq \bar{x} \\ [\min(\underline{x}\bar{y}, \bar{xy}), \max(\bar{x}\underline{y}, \underline{xy})], & \text{if } \underline{x} \leq \bar{x} < 0 \end{cases}$$

to conclude that the formula

$$xy \subseteq \langle \min(\nabla(ax), \nabla(b(-\bar{x}))), -\min(\nabla(c(-\bar{x})), \nabla(dx)) \rangle,$$

where

$$\begin{aligned} a &= \begin{cases} \underline{y} & \text{if } 0 \leq \underline{x} \\ -(-\bar{y}) & \text{otherwise} \end{cases} \\ b &= \begin{cases} -\underline{y} & \text{if } (-\bar{x}) \leq 0 \\ (-\bar{y}) & \text{otherwise} \end{cases} \\ c &= \begin{cases} -(-\bar{y}) & \text{if } (-\bar{x}) \leq 0 \\ \underline{y} & \text{otherwise} \end{cases} \\ d &= \begin{cases} (-\bar{y}) & \text{if } 0 \leq \underline{x} \\ -\underline{y} & \text{otherwise} \end{cases} \end{aligned}$$

computes the rounded results of the multiplication formula in (3.1). It uses more instructions than the direct implementation with 8 multiplications, but achieves better performance.

**3.5. Multiplication by a positive number.** When one of the numbers is known to be positive (e.g. a known constant), one can use one of the cases in (3.2) directly:

$$xy \stackrel{x \geq 0}{=} [\min(\underline{xy}, \bar{xy}), \max(\overline{xy}, \underline{xy})]$$

This is significantly faster than the general case multiplication, involving only 5 instructions (4 for constants).

**3.6. Multiplication of two positive numbers.** If both multiples are known to be positive, multiplication can be achieved by simply changing the sign of the higher bound of one of the arguments followed by `_mm_mul_pd`. If one of the numbers is a constant, one can prepare it in a suitable way to avoid the sign change and implement the multiplication as a single instruction.

**3.7. Division.** Definition:

$$\frac{x}{y} = \left[ \min \left( \frac{\underline{x}}{\underline{y}}, \frac{\underline{x}}{\bar{y}}, \frac{\bar{x}}{\underline{y}}, \frac{\bar{x}}{\bar{y}} \right), \max \left( \frac{\underline{x}}{\underline{y}}, \frac{\underline{x}}{\bar{y}}, \frac{\bar{x}}{\underline{y}}, \frac{\bar{x}}{\bar{y}} \right) \right],$$

undefined if  $0 \in y$ .

Again, this computation would require 8 divisions. Unfortunately, division is a rather slow operation, that is why we would prefer to use as few divisions as possible. One way to do this is to use  $\frac{x}{y} = x \frac{1}{y}$ , using the definition below, which uses only two divisions but quite a few other operations.

A more efficient (as well as more precise) approach turns out to be the use of case distinction similar to (3.2). By examining the divisor, we end up with fewer possible cases and easy recognition of the exceptional cases. More specifically, the operation becomes:

$$(3.3) \quad \frac{x}{y} = \begin{cases} \left[ \min \left( \frac{\underline{x}}{\underline{y}}, \frac{\underline{x}}{\bar{y}} \right), \max \left( \frac{\bar{x}}{\underline{y}}, \frac{\bar{x}}{\bar{y}} \right) \right], & \text{if } 0 < \underline{y} \leq \bar{y} \\ \text{exception,} & \text{if } \underline{y} \leq 0 \leq \bar{y} \\ \left[ \min \left( \frac{\bar{x}}{\underline{y}}, \frac{\bar{x}}{\bar{y}} \right), \max \left( \frac{\underline{x}}{\underline{y}}, \frac{\underline{x}}{\bar{y}} \right) \right], & \text{if } \underline{y} \leq \bar{y} < 0 \end{cases}$$

The final formula we use is

$$\frac{x}{y} \subseteq \left\langle \min \left( \nabla \left( \frac{a}{\underline{y}} \right), \nabla \left( \frac{-a}{(-\bar{y})} \right) \right), -\min \left( \nabla \left( \frac{-b}{(-\bar{y})} \right), \nabla \left( \frac{b}{\underline{y}} \right) \right) \right\rangle,$$

where

$$a = \begin{cases} \underline{x} & \text{if } (-\bar{y}) \leq 0 \\ -(-\bar{x}) & \text{otherwise} \end{cases}$$

$$b = \begin{cases} (-\bar{x}) & \text{if } 0 \leq \underline{y} \\ -\underline{x} & \text{otherwise} \end{cases}$$

with an additional check to throw an exception if  $\underline{y} \leq 0 \leq \bar{y}$ .

**3.8. Reciprocal.** Definition:

$$\frac{1}{x} = \left[ \frac{1}{\bar{x}}, \frac{1}{\underline{x}} \right] \subseteq \left\langle \nabla \left( \frac{-1}{(-\bar{x})} \right), \nabla \left( \frac{-1}{\underline{x}} \right) \right\rangle,$$

undefined if  $0 \in x$ .

This is implemented as a check if the argument contains zero, followed by division of  $-1$  by the argument and swapping the two components.

**3.9. Absolute value.** Definition:

$$|x| = [\max(\underline{x}, -\bar{x}, 0), \max(-\underline{x}, \bar{x})] = \langle \max(0, \underline{x}, (-\bar{x})), -\min(\underline{x}, (-\bar{x})) \rangle.$$

**3.10. Square.** Implemented as  $x^2 = |x||x|$ , using multiplication of positive numbers.

**3.11. Square root.** Definition:

$$\sqrt{x} = [\sqrt{\underline{x}}, \sqrt{\bar{x}}]$$

only defined if  $0 \leq x$ .

Since the rounding is an integral part of the square root operation, and in this case we cannot achieve a negated result, we need to use another method to ensure rounding in the correct direction. We use the fact already mentioned in the subsection on multiplication,  $\Delta(r) \leq -\nabla(-\epsilon - \nabla(r))$ .

The formula we use is:

$$\sqrt{x} \subseteq \begin{cases} \left\langle \nabla(\sqrt{\underline{x}}), -\nabla(\sqrt{-(-\bar{x})}) \right\rangle, & \text{if } \nabla(\nabla(\sqrt{-(-\bar{x})}))^2 = -(-\bar{x}) \\ \left\langle \nabla(\sqrt{\underline{x}}), \nabla(\nabla(-\epsilon - \sqrt{-(-\bar{x})})) \right\rangle, & \text{otherwise} \end{cases}$$

(where  $\epsilon$  is the smallest representable positive number).

The condition for making the first choice in this formula is only satisfied if the result of  $\sqrt{-(-\bar{x})}$  is exactly representable, in which case  $\nabla(\sqrt{-(-\bar{x})}) = \Delta(\sqrt{-(-\bar{x})})$ . Otherwise the second choice adjusts the high bound to the next representable number.

Note that if we don't require tight bounds, using only the second choice in the equation above is sufficient to implement interval square root.

If the argument is entirely negative, the implementation will raise an exception. If it contains a negative part, the implementation will crop it to only its non-negative part, to allow that computations such as  $\sqrt{0}$  can be carried out in exact real arithmetic.

#### 4. PERFORMANCE

We compare the performance of this implementation to the performance of two other packages for interval arithmetic freely available on the internet: the interval part of the *Boost* project (version 1.33.0, [6]) and the library *filib++* (version 2.0, [2]). For the latter, we tried the macro version as well as two of the available rounding policies, *multiplicative* and *native\_onesided\_global*, the latter corresponding most closely to our method of rounding.

The results of the benchmark are summarized in the following table, showing the ratio between the performance of the respective library and double precision floating point:

operation	<i>filib++</i> , macro	<i>filib++</i> , onesided	<i>filib++</i> , mult.	<i>Boost</i>	<i>RealLib</i>
+	6.12	2.45	6.22	9.90	1.05
*	6.78	3.57	6.97	124.27	6.35
/	12.33	3.63	9.24	8.62	3.72
$\sqrt{\cdot}$	27.06	62.23	61.79	15.77	1.73
$ \cdot $	30.71	23.76	23.76	2.03	3.16
$\sum_{i=1}^{1000000} \frac{1}{i}$	3.67	1.74	2.33	4.90	1.70
dot product	12.11	6.28	13.19	148.86	3.72

(Pentium-M 1.8GHz, Windows XP + Cygwin, GCC 3.4.4)

*RealLib* is faster almost everywhere, with the notable exception of multiplication in *filib++*'s *native\_onesided\_global* mode. In this case *filib++* uses a case distinction, which in our test only reaches the shortest of the 9 possible paths, giving only the best case performance of *filib++*'s multiplication code. In contrast, our implementation uses only one execution path for all multiplications, thus the ratio given in the table is both best and worst case performance. In the dot product operation the reader can see the effect that varying signs of the argument has on the performance of multiplication. The time spent in evaluating a dot product is dominated by the time spent in multiplications, but since our implementation has constant performance while *filib++*'s efficiency deteriorates, our SSE2 code turns out to be significantly faster.

#### 5. INTEL'S SSE3

The latest multimedia extension set introduced by Intel, the SSE3 [10], aimed at improving complex number computations, does not provide any benefit for interval computations. Intel chose to improve complex multiplications and divisions by introducing the instruction *\_mm\_addsub\_pd*, which combines two packed registers by adding one of the two components and subtracting the other [9]. Unfortunately, the use of this instruction leads to incorrect results if a directed rounding mode is in effect, because the multiplication that precedes the subtraction is rounded in the wrong direction.

A better handling of complex multiplications would have been the introduction of a multiplication instruction "*mulpn*" (for multiply positive negative) that changes the sign of one of the components of one of the arguments. This would require the same effort that the instruction *\_mm\_addsub\_pd* required, but would have the correct behavior in directed rounding modes, i.e. complex multiplication code using *mulpn* would yield upper bounds for the result of the multiplication if rounding towards  $+\infty$  is in effect, and lower bounds in the case of rounding towards  $-\infty$ .

Unlike *\_mm\_addsub\_pd*, a *mulpn* instruction would have been useful and advantageous for interval arithmetic. Multiplication of two positive numbers could be implemented as

a single *mulpn*, which would also speed up the implementations of transcendental interval functions.

## 6. SUGGESTIONS FOR A HARDWARE IMPLEMENTATION

We hope that the presentation until this point has convinced the reader that the use of the storage  $\langle x, -\bar{x} \rangle$  for intervals in SSE2 registers is clearly superior to the traditional method of storing intervals as simply the pair of the two bounds. This mode of storage avoids the need for special rounding modes in a hardware implementation, and even turns some existing instructions into meaningful interval operations.

We propose this storage to be adopted as the preferred storage format for intervals in hardware implementations.

To further speed up computations on intervals, we propose the introduction of a special selection instruction we call *ivchoice* (for interval choice) that can be used to prepare the arguments for multiplication and division. The action of this instruction should correspond to the following function:

```
__m128d ivchoice(__m128d a, __m128d b)
{
    a = _mm_xor_pd(a, _mm_set_pd(0.0, -0.0));
    a = _mm_shuffle_pd(a, a, _mm_movemask_pd(b));
    return a;
}
```

This is pseudocode, because *\_mm\_shuffle\_pd* cannot be performed based on a non-const integer. A software implementation of the above requires a switch statement, which can slow the execution considerably, especially in cases where the signs of the multiples cannot be predicted.

If such an instruction is available, the multiplication algorithm becomes:

```
__m128d IntervalMul(__m128d x, __m128d y)
{
    __m128d a, b;
    a = _mm_shuffle_pd(x, x, 1);
    b = _mm_shuffle_pd(y, y, 1);
    y = ivchoice(y, a);
    b = ivchoice(b, x);
    y = _mm_mul_pd(y, a);
    b = _mm_mul_pd(b, x);
    y = _mm_min_pd(b, y);
    return y;
}
```

If the latency of the proposed instruction can be the same as the latency of *\_mm\_shuffle\_pd*, this sequence of instructions will run about 30% faster than the current implementation.

Moreover, since the multiplications above only use the results of *ivchoice* with the same second argument, it is even possible to fuse *ivchoice* with the multiplication that is applied to the result:

```
__m128d ivmul(__m128d a, __m128d b)
{
    a = _mm_xor_pd(a, _mm_set_pd(0.0, -0.0));
    a = _mm_shuffle_pd(a, a, _mm_movemask_pd(b));
    a = _mm_mul_pd(a, b);
    return a;
}
```

Correspondingly, the *IntervalMul* function will in this case change to:

```
__m128d IntervalMul(__m128d x, __m128d y)
{
    __m128d a, b;
    a = _mm_shuffle_pd(x, x, 1);
    b = _mm_shuffle_pd(y, y, 1);
    y = ivmul(y, a);
    b = ivmul(b, x);
    y = _mm_min_pd(b, y);
    return y;
}
```

The extent to which such fusion can be beneficial depends on the actual hardware implementation. If the latency of *ivchoice* can be folded completely (which seems possible) or partially, interval multiplication using the fused *ivmul* could reach a latency close to the latency of two dependant double precision multiplications.

Apart from an additional test if the divisor contains zero and the use of *\_mm\_div\_pd* instead of *\_mm\_mul\_pd*, the division code is identical to the multiplication one:

```
__m128d IntervalDiv(__m128d y, __m128d x)
{
    __m128d a, b;
    if (_mm_movemask_pd(x) == 3)
        throw exception;
    a = _mm_shuffle_pd(x, x, 1);
    b = _mm_shuffle_pd(y, y, 1);
    y = ivchoice(y, a);
    b = ivchoice(b, x);
    y = _mm_div_pd(y, a);
    b = _mm_div_pd(b, x);
    y = _mm_min_pd(y, b);
    return y;
}
```

Fused *ivchoice* and division (“*ivdiv*”) is also possible, and the changes to the division code are exactly as above.

Of course, one would prefer to have a complete hardware implementation of interval arithmetic that provides instructions for the four basic operations on intervals. In our mode of operation addition already has a hardware implementation as a single instruction. Subtraction would require a fusion of swapping and addition (“*ivsub*”) which should be easy to accomplish in hardware without extra latency compared to addition.

On the other hand, multiplication and division seem too complex to be directly implemented. A pure hardware implementation of multiplication may be able to choose execution paths without the delays associated with incorrect branch predictions, thus probably the preferable hardware design would examine the signs of the four components to choose one of 9 possible combinations and perform a single pair of multiplications in 8 of the possible cases. In the 9<sup>th</sup> case, however, the operation would require the same amount of work as the function *IntervalMul* above.

Since the worst-case latency would be the same as the algorithm above, the latter should not be ignored as a possible basis for a pure hardware implementation of interval multiplication.

To conclude, we suggest that hardware assistance for interval computations should be provided as the adoption of the  $\langle x, -\bar{x} \rangle$  storage format and the introduction of the instructions of one of the following three levels:

basic	<i>mulpn, ivsub, ivchoice</i>
advanced	<i>mulpn, ivsub, ivmul, ivdiv</i>
full	<i>ivsub, IntervalMul, IntervalDiv</i>

The advanced level seems to be the best combination of feasibility and performance.

## 7. RELATED WORK

In [1], von Gudenberg discusses the efficiency of implementations of interval arithmetic using the multimedia extensions Intel's SSE, AMD's 3DNow! and Motorola's AltiVec. The paper concludes that the use of multimedia extensions only leads to a very modest improvement in multiplication with Intel's SSE in comparison to standard floating point, and only due to the fact that four single-precision operations can be executed in parallel.

Unlike SSE, the double precision second version of the extensions, SSE2, is a natural candidate for interval arithmetic because the packed registers hold two double precision values.

Von Gudenberg used a variety of rounding policies, the fastest of which is global onesided rounding, the method we use, but did not store one of the components negated in memory. Consequently, handling the negations required to perform rounding in the proper direction increases the number of instructions needed for every operation. If we were to use SSE2 in a similar mode of operation, the required number of instructions for addition would be four instead of one, for sign change – two instead of one, for subtraction – five instead of two, and for multiplication of positive intervals – three instead of two.

Additionally, instead of 9-case branching on the signs of the 4 components, we prefer to use 4 multiplications with selected arguments (the selection is branch-free), which gives us stable performance that is not affected by branch mispredictions or longer latency execution paths, although with a worse best-case performance.

In [3], Kolla, Vodopivec and von Gudenberg discuss the possibility of hardware extensions supporting interval arithmetic similar to the multimedia extensions 3DNow!, via packed storage of single precision numbers in a double precision register. For addition and subtraction they require special instructions that round each component of the pair in the appropriate direction, and for multiplication they describe a case selection method that can easily be implemented and be very efficient for 8 of the 9 possible cases and requires a sequence of operations and longer latency for the (rare) 9'th case.

We are quite skeptical about the chances of such a complicated multiplication instruction ever being implemented in hardware. Instead, we give a much more modest proposal that can also lead to very good performance at the cost of little extra hardware. It also has the benefit that one of the operations, addition, already has a hardware implementation.

## REFERENCES

- [1] von Gudenberg, J.W., *Interval Arithmetic on Multimedia Architectures*. Reliable Computing Vol. 8 No. 4 (2002).
- [2] Hofschuster, W., Krämer, W., Lerch, M., Tischler G., von Gudenberg, J.W., *The Interval Library fi.lib++ 2.0 Design, Features and Sample Programs*. Preprint 2001/4, Universität Wuppertal (2001). available at [http://www.math.uni-wuppertal.de/wrswt/preprints/prep\\_01\\_4.pdf](http://www.math.uni-wuppertal.de/wrswt/preprints/prep_01_4.pdf)
- [3] Kolla, R., Vodopivec, A., von Gudenberg, J.W., *The IAX Architecture – Interval Arithmetic Extension*. Universität Würzburg, Institut für Informatik, Techn. Report TR225 (1999). available at <http://www2.informatik.uni-wuerzburg.de/mitarbeiter/wvg/Public/iax.ps.gz>

- [4] Kearfott, R. B., *Interval Computations: Introduction, Uses, and Resources*, Euromath Bulletin 2 (1), pp. 95-112 (1996).
- [5] Lambov, B., *RealLib: An Efficient Implementation Exact Real Arithmetic*, to appear in *Mathematical Structures in Computer Science*.  
see also <http://www.brics.dk/~barnie/RealLib/>
- [6] *Boost Interval Arithmetic Library*.  
available at <http://www.boost.org/libs/numeric/interval/doc/interval.htm>
- [7] IEEE Standards Committee 754, *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*. Institute of Electrical and Electronics Engineers, New York (1985). Reprinted in SIGPLAN Notices, 22(2):9–25 (1987).
- [8] Intel Corp., *IA-32 Intel Architecture Software Developer's Manual, Volumes 1-3*.  
available at [http://developer.intel.com/design/pentium4/manuals/index\\_new.htm](http://developer.intel.com/design/pentium4/manuals/index_new.htm)
- [9] Intel Corp., *Using SSE3 Technology in Algorithms with Complex Arithmetic*.  
available at <http://www.intel.com/cd/ids/developer/asm-na/eng/dc/pentium4/optimization/66717.htm>
- [10] Intel Corp., *Next Generation Intel Processor: Software Developers Guide*.  
available at <http://www.intel.com/cd/ids/developer/asm-na/eng/dc/pentium4/optimization/66756.htm>

BRICS, UNIVERSITY OF AARHUS, IT PARKEN, 8200 AARHUS N, DENMARK  
E-mail address: [barnie@brics.dk](mailto:barnie@brics.dk)