Efficient Implementation of Interval Arithmetic Narrowing Using IEEE Arithmetic

Timothy J. Hickey[1] and **Qun Ju**
Computer Science Department
Brandeis University
Waltham, MA, USA, 02254
{tim,qun}@cs.brandeis.edu
http://www.cs.brandeis.edu/~tim

Abstract

We present and analyze several implementations of the interval arithmetic narrowing function for multiplication. Starting from the Cleary algorithm for narrowing multiplication we describe two optimizations which produce code that is 10-15 times faster on the average. Finally, we propose a few simple RISC instructions which would allow efficient execution of interval arithmetic and narrowing routines. These new instructions require a new rounding mode which extends the "round-toward-infinity" mode of the IEEE 754 floating point standard.

1 Introduction

One of the most exciting advances in Constraint Logic Programming over the last few years has been the development of arithmetic constraint solvers based on relational interval arithmetic [1],[10],[3], [6],[11]. These solvers all rely on various types of narrowing procedures for solving the constraints. In this paper we take a close look at the narrowing procedure for the most basic non-trivial constraint, the multiplication relation $x * y = z$, and we describe and analyze several implementations of this procedure both in hardware and software. This work represents the first step of a project to implement highly efficient, provably correct, and widely portable interval arithmetic and narrowing procedures for all of the standard operators and commonly used mathematical functions (e.g., $y = \cos(x)$, $z = x^y$, etc.).

1.1 IEEE 754 arithmetic

Throughout the paper, unless explicitly stated otherwise, we will assume that floating point arithmetic is implemented according to the IEEE 754-1985 Standard on Floating Point Arithmetic [9]. A nice overview of the standard can be found in Goldberg's survey article [5]. The IEEE 754 standard provides a precise definition of how floating point numbers are represented as bit strings and how the arithmetic operations (+,-,*,/,sqrt) are implemented. It also specifies 4 non-standard floating point values: positive and negative infinity, negative zero, and the

[1]This research was partially supported by NSF grant CCR-9403427

not-a-number value `NaN`. The arithmetic operators are defined on all floating point inputs (including the non-standard ones), and they are subject to four different rounding modes which specify what is to be done when the result of an arithmetic operation is not a member of the set of representable floating point numbers. The four rounding modes supported are (1) round-to-nearest, (2) round-toward-positive-infinity, (3) round-toward-negative-infinity, and (4) round toward zero. The arithmetic operators produce `NaN` on some combinations of arguments, for example $0/0$,∞/∞,$-0/0$,$0*(-\infty)$,$-\infty+\infty$, etc. all yield `NaN`, whereas when there is an unambiguous limiting value for the expression, the IEEE standard specifies that value: $\infty*(-\infty)=-\infty$, $-\infty/(-0)=\infty$, $\infty-(-\infty)=\infty$, $0*(-0)=-0$, $1/(-\infty)=-0$, etc. The reader is advised to read the standard for a precise specification of these arithmetic operators. Rather than explicitly setting the rounding modes in our code, we will assume that one has access to the following functions:

```
add_lo(x,y) sub_lo(x,y) mul_lo(x,y) div_lo(x,y), sqrt_lo(x),
add_hi(x,y) sub_hi(x,y) mul_hi(x,y) div_hi(x,y), sqrt_hi(x)
```

where the suffixes `lo` and `hi` specify whether the rounding mode is toward negative infinity or positive infinity respectively.

1.2 The IEEE arithmetic narrowing functions

In the algorithms we describe, a non-empty **IEEE interval** is represented as a pair $[a,b]$ of IEEE double precision numbers satisfying the following properties:

$$-\infty \le a \le b \le \infty \quad \wedge \quad a \ne \infty \quad \wedge \quad -\infty \ne b$$

A non-empty IEEE interval $[a,b]$ is a machine representation of the closed real interval

$$\{x \in \mathbf{R}: \ a \le x \le b\}$$

We also place restrictions on the sign of zeroes in IEEE intervals $[a,b]$. If a is zero then it must be positive zero, whereas if b is zero it must be negative zero. This convention allows us to make use of the fact that $1/(-0)=-\infty$ and $1/0=+\infty$ to more easily compute quotients of IEEE intervals. We represent the empty interval, `EMPTY`, by the pair $[\infty,-\infty]$. The narrowing function N_* for multiplication that we are studying in this paper maps a 3-tuple of IEEE intervals $U=(X,Y,Z)$ to another 3-tuple $U'=(X',Y',Z')$ such that every solution of the constraint $X*Y=Z$ in U is also contained in U', and U' is the smallest 3-tuple of intervals with this property. More formally, we have the following definitions:

Definition 1.1 *Let $\alpha_k(S)$ be the function which maps subsets $S \subset \mathbf{R}^k$ of real numbers to the smallest k-tuple of IEEE intervals which contains S. We write α for α_1. The* **narrowing function for multiplication** *is defined by:*

$$N_*(X,Y,Z) = \alpha_3\left(\{(a,b,c) \in (X,Y,Z): \ a*b=c\}\right)$$

Similarly, the **narrowing function for addition** *is defined by:*

$$N_+(X,Y,Z) = \alpha_3\left(\{(a,b,c) \in (X,Y,Z): \ a+b=c\}\right)$$

For example, $N_*([-1,5],[-1,1],[2,10]) = ([2,5],[0.4,1],[2,5])$.

It is quite easy to evaluate the narrowing function N_+ on IEEE intervals. The following theorem shows that this can be done using the `add_hi/lo` and `sub_hi/lo` functions.

Theorem 1.2 *Let X, Y, and Z be non-empty IEEE intervals, then we can compute $(X', Y', Z') = N_+(X, Y, Z)$ as follows:*

$$X' = X \cap \alpha(Z - Y), \quad Y' = Y \cap \alpha(Z - X), \quad Z' = Z \cap \alpha(X + Y)$$

where $A \circ B = \{a \circ b : \ a \in A, b \in B\}$, for any binary operator $\circ$ and sets A, B of real numbers. Moreover, if $X = [xlo, xhi]$, $Y = [ylo, yhi]$, $Z = [zlo, zhi]$, then

$$\begin{aligned} \alpha(X + Y) = & \ [\texttt{add\_lo}(xlo, ylo), \texttt{add\_hi}(xhi, yhi)] \\ \alpha(Z - Y) = & \ [\texttt{sub\_lo}(zlo, yhi), \texttt{sub\_hi}(zhi, ylo)] \end{aligned}$$

proof: This result is well-known when the endpoints are all finite. Thus, one need only check that the formulas for $\alpha(X+Y)$ and $\alpha(Z-Y)$ also hold when one or more of the four endpoints are infinite. These 15 cases are easily checked by hand. **QED**

Multiplication is more difficult to narrow than addition. Nevertheless, we can express the narrowing $(X', Y', Z') = N_*(X, Y, Z)$ using three separate expressions for X', Y', and Z' as we show in the rest of this section.

Definition 1.3 *For any sets A and B of real numbers, we let $A * B$ and A/B denote the element-wise* **set product** *and* **set quotient** *of the sets A and B. Thus,*

$$A * B = \{a * b : \ a \in A, b \in B\} \quad A/B = \{a/b : \ a \in A, b \in B - \{0\}\}$$

To properly define narrowing of multiplication we need the following modified division operator $A \oslash B$ which can be thought of extending the A/B operator by treating $0/0$ as an undefined quantity that could take any value in the range $[-\infty, \infty]$.

Definition 1.4 *For any sets A,B of real numbers, we define the* **modified quotient** *$A \oslash B$ by*

$$A \oslash B = \begin{cases} A/B & \text{if } 0 \notin A \cap B \\ [-\infty, \infty] & \text{else} \end{cases}$$

These definitions allow us to express the multiplicative narrowing function using three simpler functions, as follows.

Theorem 1.5 *For any IEEE Intervals X, Y, and Z, we have $N_*(X, Y, Z) = (X', Y', Z')$, where*

$$X' = \alpha(X \cap (Z \oslash Y)), \quad Y' = \alpha(Y \cap (Z \oslash X)), \quad Z' = \alpha(Z \cap (X * Y)) = Z \cap \alpha((X * Y))$$

proof: This is a straightforward unwinding of the definitions. There are two points about which one must be careful. First, if both Z and Y contain zero, then $X * 0 = 0$ is a solution to the constraint $x * y = z$ so X will not be narrowed. This justifies the use of $Z \oslash Y$ rather than Z/Y above. The second point is that $X * Y$ is always an interval so we can move the α inward. We can not move α inward in the expressions for X' and Y' because the modified quotient $Z \oslash Y$ of two intervals is not necessarily an interval. **QED**

2 The Partition Algorithm

The simplest algorithm we consider for narrowing the multiplication relation is based on narrowing algorithms (for more general classes of intervals) described by Cleary [2] and Hyvonen [8], and implemented in Scheme by Cleary et. al. as part of a CLP(interval) interpreter [3]. We will call this algorithm the **Partition Algorithm**, because it relies on partitioning the input into subsets which can easily be narrowed and then taking the interval union of the results.

2.1 Narrowing Multiplication of Positive Intervals

The simplest case for multiplicative narrowing is when the arguments X, Y, Z are all non-zero, positive intervals. In this case, Z/X and Z/Y are guaranteed to be IEEE intervals with finite endpoints, and we can compute the narrowing function N_* using Theorem 2.2 which we state and prove in this section.

Definition 2.1 *We say that an IEEE interval* $X = [a, b]$ *is* **semi-definite** *if* $0 \leq a < b < \infty$. *Thus,* X *is non-empty, non-zero, and non-negative. For semi-definite IEEE intervals* $X = [a, b]$, $Y = [c, d]$, $Z = [e, f]$, *we define* `mulppp`, `divppp` *by*

- $\texttt{mulppp}([a, b], [c, d]) = [\texttt{mul\_lo}(a, c), \texttt{mul\_hi}(b, d)]$
- $\texttt{divppp}([a, b], [c, d]) = [\texttt{div\_lo}(a, d), \texttt{div\_hi}(b, c)]$

For all IEEE intervals X *and* Y *we define* $\texttt{inter}(X, Y)$ *by*

- $\texttt{inter}([a, b], [c, d]) = \begin{cases} [max(a, c), min(b, d)] & \textit{if } max(a, c) \leq min(b, d) \\ \texttt{EMPTY} & \textit{otherwise} \end{cases}$
- $\texttt{union}([a, b], [c, d]) = [min(a, c), max(b, d)]$

For any set S *of real numbers, we let* S^p, S^z, *and* S^n *denote the positive, zero, and negative parts of* S, *i.e.,* $S^p = S \cap (0, \infty)$, $S^n = S \cap (-\infty, 0)$, *and* $S^z = S \cap [0, 0]$.

Observe that $\texttt{mulppp}(X, Y) = \alpha(X * Y)$, $\texttt{divppp}(X, Y) = \alpha(X/Y)$, when X, Y are semi-definite and that $\texttt{inter}(X, Y) = \alpha(X \cap Y)$, $\texttt{union}(X, Y) = \alpha(X \cup Y)$ for all IEEE intervals X, Y (even empty ones).

Theorem 2.2 *For IEEE intervals* X, Y, Z, *define* $\texttt{nar\_ppp}(X, Y, Z) = (X', Y', Z')$ *where*

- $X' = \texttt{inter}(X, \texttt{divppp}(Z, Y))$
- $Y' = \texttt{inter}(Y, \texttt{divppp}(Z, X))$
- $Z' = \texttt{inter}(Z, \texttt{mulppp}(X, Y))$

If X, Y, Z *are semi-definite IEEE intervals as in the previous definition, then we have* $\texttt{nar\_ppp}(X, Y, Z) = N_*(X^p, Y^p, Z^p)$ *is the narrowing of all positive solutions to the constraint* $X * Y = Z$.

proof: This result is well-known if none of the endpoints of X, Y, or Z is zero or infinite since in this case one does not need to use the infinite values of IEEE arithmetic. An easy way to check that these formulas for N_* work correctly when the endpoints can be zero or infinity is to observe that by taking logarithms, this problem becomes the addition narrowing problem and the IEEE operations on zero and infinite values are defined so that when a and b are non-negative numbers and at least one of them is zero or infinity then

$$\log(a*b) = \log(a) + \log(b) \quad \log(a/b) = \log(a) - \log(b)$$

where $\log(+0) = -\infty$, $\log(\infty) = \infty$, $\log(NaN) = NaN$. **QED**

2.2 Partitioning the General Case

The general case of multiplicative narrowing is substantially more complicated than the non-zero, positive interval case because of the rules for multiplication and division of signed numbers. To handle these more general cases, the Partition Algorithm relies on the following property of the narrowing operator to reduce to the problem of narrowing intervals whose arguments have constant sign (positive, zero, or negative):

$$N_*(S) = \alpha\Big(\bigcup_{a_1,a_2,a_3 \in \{p,z,n\}} (N_*(S \cap \mathbf{R}^{a_1a_2a_3})\Big)$$

and $\mathbf{R}^{abc} = \mathbf{R}^a \times \mathbf{R}^b \times \mathbf{R}^c$, $\mathbf{R}^p = (0,\infty)$, $\mathbf{R}^z = [0,0]$, $\mathbf{R}^n = (-\infty,-0)$. This allows us to reduce to the problem of narrowing subsets $S^a = S \cap \mathbf{R}^a$ of $\mathbf{R}^a$. It is easy to check that $N_*(S^a)$ is empty unless S^a is one of the following 9 subsets:

$$S^{ppp},\ S^{nnp},\ S^{pnn},\ S^{npn},\ S^{zzz},\ S^{zpz},\ S^{znz},\ S^{pzz},\ S^{nzz},$$

and moreover, $N_*(S^a) = S^a$ for the last five subsets. The first four cases can all be reduced to the first case using the following symmetry functions:

$$I_{pnn}(X,Y,Z) = (X,-Y,-Z),\ \ I_{npn}(X,Y,Z) = (-X,Y,-Z),\ \ I_{nnp}(X,Y,Z) = (-X,-Y,Z)$$

where $-[a,b] = [-b,-a]$. If we the use the fact that for $a \in \{nnp, npn, pnn, ppp\}$ we have

$$N_*(S^a) = I_a(N_{ppp}(I_a(S^a))) \quad I_a(S^a) \subset \mathbf{R}^{ppp}$$

where $N_a(S) = N_*(S^a)$, and hence $N_{ppp}(S) = N_*(S \cap R^{ppp})$, then our algorithm is

$$\begin{aligned} N_*(S) = \quad & \alpha_3(N_{ppp}(S) \cup I_{nnp}(N_{ppp}(I_{nnp}(S^{nnp}))) \\ & \cup I_{npn}(N_{ppp}(I_{npn}(S^{npn}))) \cup I_{pnn}(N_{ppp}(I_{pnn}(S^{pnn}))) \\ & \cup S^{zzz} \cup S^{zpz} \cup S^{znz} \cup S^{pzz} \cup S^{nzz}) \end{aligned}$$

A C procedure `narrow_mul_PA` implementing the Partition Algorithm for narrowing multiplication is shown in Figure 1. (Note that `I3(X,Y,Z)` maps three intervals to the 3-tuple `(X,Y,Z)` if all are non-empty, or to the empty 3-tuple `(E,E,E)` if one or more is empty. Notice that the case which requires the most execution time occurs when all three of the intervals X, Y, Z properly contain zero. But in this case neither X nor Y can be narrowed (as $X * 0 = 0$ and

```
int narrow_mul_PA(INTERVAL *x, INTERVAL *y, INTERVAL *z) {
  INTERVAL xp,xo,xn,yp,yo,yn,zp,zo,zn;   INTERVAL3 R;
  xp = pos(*x); yp = pos(*y); zp= pos(*z);
  xo = zer(*x); yo = zer(*y); zo= zer(*z);
  xn = neg(*x); yn = neg(*y); zn= neg(*z);
  R =             nar_ppp(      I3(xp,yp,zp));
  R = alpha3(I_nnp(nar_ppp(I_nnp(I3(xn,yn,zp)))),R);
  R = alpha3(I_npn(nar_ppp(I_npn(I3(xn,yp,zn)))),R);
  R = alpha3(I_pnn(nar_ppp(I_pnn(I3(xp,yn,zn)))),R);
  R = alpha3(                   I3(xo,yo,zo)   ,R);
  R = alpha3(                   I3(xo,yp,zo)   ,R);
  R = alpha3(                   I3(xo,yn,zo)   ,R);
  R = alpha3(                   I3(xp,yo,zo)   ,R);
  R = alpha3(                   I3(xn,yo,zo)   ,R);
  if (nonempty3(R)) { *x = R.x; *y = R.y; *z=R.z;  return(TRUE); }
  else return(FALSE);
}
```

Figure 1: The Partition Algorithm for Narrowing Multiplication

$0 * Y = 0$ are valid solutions to the constraint). So the most that can be done is to narrow Z, and this can be done by the following macro `nar_sss(X,Y,Z)` which requires only four multiplications and a few comparisons:

```
Z.lo = max(Z.lo,min(mul_lo(X.hi,Y.lo),mul_lo(X.lo,Y.hi)))
Z.hi = min(Z.hi,max(mul_hi(X.lo,Y.lo),mul_hi(X.hi,Y.hi)))
```

The algorithm given in the next section uses the framework of abstract interpretation to make this type of optimization in a principled way.

3 The Abstract Domain Algorithm

The idea behind the **Abstract Domain** algorithm derived in this section is to use abstract interpretation techniques [4] to classify the possible inputs into special cases which can be narrowed using routines optimized for those cases. This method leads to our most efficient narrowing algorithm for multiplication, but other abstract domains may lead to even better algorithms. This is an area which deserves further study.

The six cases we consider are given by the following subsets (whose union contains the set of all intervals):

```
P={[a,b]: 0<a<=b}  G={[0,b]: 0<b}   S={[a,b]: a<0<b}
N={[a,b]: a<=b<0}  L={[a,0]: a<0}   Z={[0,0]}
```

The basic structure of the algorithm to compute $N_*(X, Y, Z)$ is to determine which of these possible cases holds for each of X, Y, and Z, and then to use a switch to jump to a routine optimized for this case. Since there are six cases for each of X, Y, and Z, this requires us to

```
int narrow_mul_AI(Interval *X, Interval *Y, Interval *Y) {
double xlo=X->lo,xhi=X->hi,ylo=Y->lo,yhi=Y->hi,zlo=Z->lo,zhi=Z->hi;
  /* there are 6 cases for each variable: P Z N G S L */
  switch( (CASE(xl,xh)<6)|CASE(yl,yh)<<3)|CASE(zl,zh)) {
    case PPP:                    NAR_ppp         break;
    case PPZ:                         return(0); break;
    case PPN:                         return(0); break;
    case PPG:                    NAR_ppp         break;
    case PPS: FORCE_z_p          NAR_ppp         break;
    case PPL:                         return(0); break;
    ....    ....    ....    ....    ....    ....   ....
    case LLS: FORCE_z_p  SYM_nnp NAR_ggg SYM_nnp break;
    case LLL: FORCE_z_0  break;
    default: return(0); }
  X->lo=xlo;X->hi=xhi; Y->lo=ylo;Y->hi=yhi; Z->lo=zlo;Z->hi=zhi;
  return(1);}
```

Figure 2: Portions of the Abstract Domain Version of the Multiplicative Narrowing Algorithm

write narrowing routines for each of these $6^3 = 216$ cases. We can use abstract interpretation and symmetry to reduce this to 13 cases as follows. Observe that if we let $E = \{\}$ be the empty set and I be the set of all intervals, then the set $\Delta = \{E, Z, P, N, G, L, S, I\}$ forms a lattice (ordered by inclusion) which can be viewed as an abstract domain for the set of intervals. We can lift the narrowing function $N_* : I^3 \rightarrow I^3$ from the domain of intervals to a function $\nu_* : \Delta^3 \rightarrow \Delta^3$ in such a way that $\nu_*(a) = \alpha_I(N_*(\gamma_I(a)))$ where α_I maps a tuple of T intervals to the smallest tuple $D \in \Delta^3$ containing T, and $\gamma_I : \Delta^n \rightarrow I^n$ maps D to the union of all intervals in D. Thus, γ_I is defined on Δ by:

$$\gamma_I(Z) = [0,0] \;\; \gamma_I(P) = (0,\infty) \;\; \gamma_I(N) = [-\infty, 0)$$

$$\gamma_I(G) = [0,\infty) \;\; \gamma_I(L) = [-\infty, 0] \;\; \gamma_I(S) = [-\infty, \infty]$$

The abstract narrowing function has the effect of either returning the empty set (for example, $\nu_*(P,P,N) = \{\}$), in which case abstract narrowing has shown the constraint is unsatisfiable, or it intersects one or more of X, Y, and Z with a smaller abstract element. There are 36 cases (out of 216) in which abstract narrowing returns the empty set. For the remaining 180 cases, the abstract narrowing can be effected using some subset of the following nine operations `FORCE_V_a` with $V \in \{X,Y,Z\}$ and $a \in \{p,z,n\}$. These operations intersect V with $\mathbf{R}^a$ and hence have the effect of setting one or both endpoints to zero, or returning the empty set.

After performing the abstract narrowing, there are 37 cases in which at least one of the arguments is zero. As we have seen with the Partition Algorithm, if any of the arguments is zero, then the result of narrowing is either the empty set or the set itself, so in all of these cases we either FAIL or do nothing.

```
#define NAR_ssp \
 if      ((t1=mul_hi(xhi,yhi))<zlo)           \
 {FORCE_x_n FORCE_y_n SYM_nnp NAR_ppp SYM_nnp} \
 else if ((t2=mul_hi(xlo,ylo))<zlo)           \
 {FORCE_x_p FORCE_y_p         NAR_ppp      }   \
 else if (t1<t2) {if (t2<zhi) zhi=t2;}        \
 else if (t1<zhi) zhi=t1;                      \
```

Figure 3: A Special Case Narrowing Macro

The remaining cases can be further reduced by applying the I_{nnp}, I_{npn}, and I_{pnn} symmetries, which map N to P and L to G. This reduces the number of cases per variable to 3 (P, G, S) and after this reduction there are only 13 cases remaining, for which we have defined special case narrowing procedures:

```
NAR_ppp           NAR_ggg           NAR_ssg
NAR_pgg NAR_gpg   NAR_gsg NAR_sgg   NAR_ssp
NAR_pss NAR_sps   NAR_gss NAR_sgs   NAR_sss
```

Figure 2 shows part of a C program implementing this algorithm. This program was constructed automatically by a simple Prolog program which generated all abstract cases, computed the abstract narrowing, and generated the `FORCE`, `SYM`, and `NAR` macros needed for each case. These cases are all relatively straightforward. We have already seen the code for `NAR_ppp`, and `NAR_sss`. The most complex case is `NAR_ssp` whose code is shown in Figure 3. The narrowing macros were written by hand but they could in principle be generated automatically by partially evaluating the Partition Algorithm.

4 The Interval Arithmetic Algorithm

In this section, we present a narrowing algorithm for multiplication which takes an opposite approach to the case analysis method of the previous two sections. The idea here is to compute the narrowing of (X, Y, Z) as directly as possible using general interval arithmetic operations. We begin with some definitions and a theorem which forms the basis of the algorithm.

Definition 4.1 *Let $X = [a, b]$ be a non-empty IEEE interval, then we say X* **is zero** *if $a = b = 0$, X* **is nonzero** *if $a \neq 0$ or $b \neq 0$, X* **contains zero** *if $a \leq 0 \leq b$, X* **is split** *if $a < 0 < b$. If X and Y are two IEEE intervals, we define the following interval arithmetic functions*

$$\texttt{mul}(X,Y) = \alpha(X * Y) \quad \texttt{odiv}(X,Y) = \alpha(X \oslash Y) \quad \texttt{div}(X,Y) = \alpha(X/Y)$$
$$\texttt{pos}(X) = \alpha(X \cap [0,\infty)) \quad \texttt{neg}(X) = \alpha(X \cap (-\infty, 0])$$

which map IEEE intervals to IEEE intervals.

Theorem 4.2 *Let X, Y, and Z be IEEE intervals, then we can compute $(X', Y', Z') = N_*(X, Y, Z)$ using the following formulas*

$$X' = G(X, Y, Z), \quad Y' = G(Y, X, Z), \quad Z' = \texttt{inter}(Z, \texttt{mul}(X, Y))$$

where

$$G(X, Y, Z) = \texttt{union}(\texttt{inter}(X, (Z \oslash \texttt{neg}(Y))), \texttt{inter}(X, (Z \oslash \texttt{pos}(Y))))$$

If Y is not split $X' = \texttt{inter}(X, \texttt{odiv}(Z, Y))$, and if X' is split, then $Y' = Y$.

proof: This follows from Theorem 1.5 and the fact that if A and B are real intervals then

$$\alpha(A \cap B) = \alpha(\alpha(A) \cap \alpha(B))$$

Thus, if we write $Y = \texttt{pos}(Y) \cup \texttt{neg}(Y)$ we get

$$\begin{aligned} X' = &\ \alpha(X \cap Z \oslash Y) \\ = &\ \alpha((X \cap (Z \oslash \texttt{pos}(Y))) \ \cup \ (X \cap (Z \oslash \texttt{neg}(Y)))) \\ = &\ \texttt{union}((X \cap (Z \oslash \texttt{pos}(Y))), (X \cap (Z \oslash \texttt{neg}(Y)))) \\ = &\ \texttt{union}(\texttt{inter}(X, (Z \oslash \texttt{pos}(Y))), \texttt{inter}(X, (Z \oslash \texttt{neg}(Y)))) \end{aligned}$$

If Y is not split then least one of $\texttt{neg}(Y)$ or $\texttt{pos}(Y)$ is empty or is zero and hence is included in the other. Say that $Y \subset [0, \infty]$ so that $\texttt{neg}(Y) \subset \texttt{pos}(Y) = Y$. So we can simplify the expression for X' to

$$\begin{aligned} X' = &\ \texttt{union}(\texttt{inter}(X, (Z \oslash \texttt{pos}(Y))), \texttt{inter}(X, (Z \oslash \texttt{neg}(Y)))) \\ = &\ \texttt{inter}(X, (Z \oslash \texttt{pos}(Y))) \\ = &\ \texttt{inter}(X, (Z \oslash Y)) \\ = &\ \texttt{inter}(X, \texttt{odiv}(Z, Y)) \end{aligned}$$

If now X' is split, then X must have been split originally and $\texttt{odiv}(Z, Y)$ must be split. Thus, either Z is split or Y is split. If Z contains zero, then $\texttt{odiv}(Z, X) = [-\infty, \infty]$ so $Y' = Y$. If Y is split and Z doesn't contain zero and X' is still split, then there are positive and negative solutions to $x * y = z$ and so $Y' = Y$. **QED**

This theorem allows us to narrow the relation $X * Y = Z$ by evaluating 3 interval arithmetic expressions involving unions, intersections, products and a restricted form of division of intervals. Indeed, observe that the formula in the previous theorem only requires that we compute $A \oslash B$ in the case where the denominator B is not split. Moreover, if X or Y is not split then the corresponding equations can be simplified by removing one of the summands of the union. In the remainder of this section we examine various ways of evaluating `mul`, `odiv`, and $G(X, Y, Z)$. The case when Y is split can be handled relatively easily as the following theorem shows.

Theorem 4.3 *Let $X = [a, b]$, $Y = [c, d]$ and $Z = [e, f]$ be IEEE intervals and assume that Y is split, i.e. $c < 0 < d$. If Z contains 0, then $G(X, Y, Z) = X$. On the other hand, if $e > 0$ then*

$$G(X, Y, Z) = \texttt{union}(([a, b] \cap [-\infty, \texttt{div\_hi}(e, c))), ([a, b] \cap (\texttt{div\_lo}(e, d), \infty]))$$

where the critical point to notice is that $[a,b]$ is being intersected with an open interval in both cases, so it is impossible for either of these intersections to be $[0,0]$. Similarly, if $f<0$ then

$$G(X,Y,Z)=\texttt{union}(([a,b]\cap[-\infty,\texttt{div\_hi}(f,d))),([a,b]\cap(\texttt{div\_lo}(f,c),\infty]))$$

Proof: This follows directly from the definitions. However, the only subtle point is that when one of the endpoints (say the positive endpoint) of Y is infinite, the corresponding interval $Z \oslash \texttt{pos}(Y)$ is an open set $(0,\infty)$ and if $X=[a,0]$, then the intersection will not contain 0, thus we cannot use $\texttt{inter}(X,\texttt{odiv}(Z,Y))$ to compute our result as $\texttt{odiv}(Z,Y)$ will incorrectly return a closed interval. If neither endpoint is infinite, then $Z \oslash Y$ is a closed interval, but we get the same result if we intersect with its interior. **QED**

4.1 Multiplication and Division of Intervals

We next present two methods for computing the product $\texttt{mul}(X,Y)$ and modified quotient $\texttt{odiv}(X,Y)$ of two IEEE intervals, X and Y. If $X=[a,b]$ and $Y=[c,d]$ are real intervals with finite endpoints, then the function $x*y$ takes its maximum and minimum for $x\in X$ and $y\in Y$ when x and y are endpoints of X and Y, so

$$X*Y=[min(ac,ad,bc,bd),max(ac,ad,bc,bd)]$$

There are two basic approaches to computing $\texttt{mul}(X,Y)$ and $\texttt{odiv}(X,Y)$ for IEEE intervals X and Y. The first approach is to handle some exceptional cases and then do a case analysis. The second eliminates the case analysis by using an enhanced version of the IEEE rounding modes.

4.2 The Case Analysis Method

For the product $X*Y$, if either X or Y is zero, then the result is zero. For the modified division $X\oslash Y$, if X and Y both contain zero, then the result is $[-\infty,\infty]$. If $Y=[0,0]$ and X doesn't contain zero, then the result is the `EMPTY` interval. Otherwise, for both multiplication and division there are three cases for X and three for Y as follows: P:$a\geq 0$, N:$b\leq 0$, S:$a<0<b$. In each of these cases $\texttt{mul}(X,Y)$, $\texttt{odiv}(X,Y)$ can be computed using at most two multiplications/divisions, except for the SS case which requires 4 multiplications and no divisions. The case analysis for the non-exceptional cases are as follows;

$case:$	$mul([a,b],[c,d])$	$odiv([a,b],[c,d])$
$PP:$	$[\texttt{mul\_lo}(a,c),\texttt{mul\_hi}(b,d)]$	$[\texttt{div\_lo}(a,d),\texttt{div\_hi}(b,c)]$
$PN:$	$[\texttt{mul\_lo}(b,c),\texttt{mul\_hi}(a,d)]$	$[\texttt{div\_lo}(b,d),\texttt{div\_hi}(a,c)]$
$PS:$	$[\texttt{mul\_lo}(b,c),\texttt{mul\_hi}(b,d)]$	$[-\infty,\infty]$
$NP:$	$[\texttt{mul\_lo}(a,d),\texttt{mul\_hi}(b,c)]$	$[\texttt{div\_lo}(a,c),\texttt{div\_hi}(b,d)]$
$NN:$	$[\texttt{mul\_lo}(b,d),\texttt{mul\_hi}(a,c)]$	$[\texttt{div\_lo}(b,d),\texttt{div\_hi}(a,c)]$
$NS:$	$[\texttt{mul\_lo}(a,d),\texttt{mul\_hi}(a,c)]$	$[-\infty,\infty]$
$SP:$	$[\texttt{mul\_lo}(a,d),\texttt{mul\_hi}(b,d)]$	$[\texttt{div\_lo}(a,c),\texttt{div\_hi}(b,c)]$
$SN:$	$[\texttt{mul\_lo}(b,c),\texttt{mul\_hi}(a,c)]$	$[\texttt{mul\_lo}(b,d),\texttt{mul\_hi}(a,d)]$
$SS:$	$[min(\texttt{mul\_lo}(a,d),\texttt{mul\_lo}(b,c)),$ $max(\texttt{mul\_hi}(a,c),\texttt{mul\_hi}(b,d))]$	$[-\infty,\infty]$

In all of these cases, IEEE rounded arithmetic yields the correct results even for infinite and signed zero values because the case analysis eliminates all NaN cases of $0 * \infty$, $0/0$, etc. Notice that the case in $\mathtt{mul}(X, Y)$ where X or Y is zero must be handled separately, as otherwise $[0, -0] * [0, \infty]$ would result in a NaN value $0 * \infty$, as must the $\mathtt{odiv}(X, [0, -0])$ case.

4.3 The Enhanced Modes Method

There is another way to compute the product which eliminates the case analysis. This method requires an enhancement of the IEEE rounding modes in which `mul_hi`, `mul_lo`, `div_hi`, and `div_lo` are defined (i.e. return non-NaN values) on all finite and infinite arguments.

Definition 4.4 *Let* $\mathtt{mul\_loE}(x, y)$ *(resp.* $\mathtt{mul\_hiE}(x, y)$*) equal* $\mathtt{mul\_lo}(x, y)$ *(resp.* $\mathtt{mul\_hi}(x, y)$*) if the latter is defined, i.e. is a non-NaN value. Otherwise, if either* x *or* y *is NaN, then* $\mathtt{mul\_loE}(x, y)$ *(resp.* $\mathtt{mul\_hiE}(x, y)$*) is NaN. The remaining cases are defined as follows where* $\beta(x, y)$ *denotes the interval* $[\mathtt{mul\_loE}(x, y), \mathtt{mul\_hiE}(x, y)]$*:*

$$\begin{array}{lllll} \beta(0, \infty) & = \beta((-0), -\infty) & = \beta(\infty, 0) & = \beta(-\infty, (-0)) & = [0, \infty] \\ \beta(0, -\infty) & = \beta((-0), \infty) & = \beta(-\infty, 0) & = \beta(\infty, (-0)) & = [-\infty, (-0)] \end{array}$$

We call these the **exceptional** *cases for the enhanced rounding modes for multiplication. Similarly, we define* `div_loE` *and* `div_hiE` *to be extensions of* `div_lo` *and* `div_hi` *where the values on the exceptional cases for division are given by the following, where* $\delta(x, y) = [\mathtt{div\_loE}(x, y), \mathtt{div\_hiE}(x, y)]$*:*

$$\begin{array}{lllll} \delta(0, 0) & = \delta((-0), (-0)) & = \delta(\infty, \infty) & = \delta(-\infty, -\infty) & = [0, \infty] \\ \delta(0, (-0)) & = \delta((-0), 0) & = \delta(\infty, -\infty) & = \delta(-\infty, \infty) & = [-\infty, 0] \end{array}$$

With these enhanced modes we can now compute the product and modified quotient of two IEEE intervals with much less case analysis:

Theorem 4.5 *Let* $X = [a, b]$ *and* $Y = [c, d]$ *be non-empty IEEE intervals. If* X *or* Y *is zero, then* $\mathtt{mul}(X, Y) = [0, 0]$*, otherwise* $\mathtt{mul}(X, Y) = [L, H]$ *where*

$$\begin{aligned} L &= min(\mathtt{mul\_loE}(a, c), \mathtt{mul\_loE}(a, d), \mathtt{mul\_loE}(b, c), \mathtt{mul\_loE}(b, d)) \\ H &= max(\mathtt{mul\_hiE}(a, c), \mathtt{mul\_hiE}(a, d), \mathtt{mul\_hiE}(b, c), \mathtt{mul\_hiE}(b, d)) \end{aligned}$$

proof: The extended multiplication is defined so that if a_i and b_i converge to a and b through finite, non-zero values without changing sign, then

$$\mathtt{mul\_lo}(a, b) \leq a_i * b_i \leq \mathtt{mul\_hi}(a, b)$$

It is easy to check that this definition satisfies the property stated in the theorem since one only needs to check the cases where one or more of the endpoints is zero or infinite. **QED**

Theorem 4.6 *Let* $X = [a, b]$ *and* $Y = [c, d]$ *be non-empty IEEE intervals. If* Y *is zero, then* $\mathtt{odiv}(X, Y)$ *is either* $[-\infty, \infty]$ *if* X *contains zero, or is empty. Otherwise, if both* X *and* Y *contain zero then* $\mathtt{odiv}(X, Y) = [-\infty, \infty]$*. Otherwise,* $\mathtt{odiv}(X, Y) = [L, H]$ *where*

$$\begin{aligned} L &= min(\mathtt{div\_loE}(a, c), \mathtt{div\_loE}(a, d), \mathtt{div\_loE}(b, c), \mathtt{div\_loE}(b, d)) \\ H &= max(\mathtt{div\_hiE}(a, c), \mathtt{div\_hiE}(a, d), \mathtt{div\_hiE}(b, c), \mathtt{div\_hiE}(b, d)) \end{aligned}$$

```
int narrow_mul_IA(Interval *x, Interval *y, Interval *y){
   return( intersect_mulIII(*x,*y,z) &&
   (splitI(*y)?split_divIII(*z,*y,x):intersect_divIII(*z,*y,x)) &&
   (splitI(*x)?    TRUE            :intersect_divIII(*z,*x,y)));}
int intersect_mulIII(Interval x, Interval y, Interval *z) {
    return(nonemptyI(*z=intersectIII(*z,mulIII(x,y))));}
int intersect_divIII(Interval x, Interval y, Interval *z) {
    return(nonemptyI(*z=intersectIII(*z,divIII(x,y))));}
Interval intersectIII(Interval x, Interval y) { Interval z;
    z.hi = min(x.hi,y.hi);  z.lo = max(x.lo,y.lo);
    return( (z.lo>z.hi)?EMPTY:z);}
```

Figure 4: The Interval Arithmetic version of the Multiplication Narrowing Algorithm

proof: The $Y = [0, 0]$ case is clear as is the case where X and Y contain zero. Thus, we may assume that Y is non-zero and either X doesn't contain zero or Y doesn't contain zero. If none of the endpoints of X and Y are zero or infinite, then it is easy to verify the theorem (show the maxima and minima of $x * y$ for fixed x or y occur at the endpoints). To prove the general case we use a limiting argument or check the (many) cases where the endpoints of X and Y are infinite, zero, positive, or negative. **QED**

The `mul_loE`, `mul_hiE`, `div_loE`, `div_hiE` instructions could in theory be implemented in hardware (as the current IEEE rounding modes are) or they can be implemented in software. There are at least two software implementations. The easiest is to use a `switch` or nested `if`s to determine whether one of the exceptional cases holds. For example, the `div_loE` instruction can be implemented as a macro as follows:

```
#define div_loE(X,Y)
  ((isnan(t=div_lo(X,Y)))?((X==0)?NEGINF:(((t=X*Y)<0)?t:0)):t)
```

Another way is to set the processor flags to generate "signalling NaNs" in the exceptional cases and then use interrupt handling to compute the values in these exceptional cases. If the application being executed only rarely generates the exceptional cases then this can be an efficient implementation.

4.4 The IA Multiplicative Narrowing Algorithm

Putting together the results of the last few sections. we see that we can compute N_* using the procedure in Figure 4 where the `mulIII` and `divIII` procedures are defined either using the case analysis method or the enhanced modes method. and `split_div` is defined as in Theorem 4.3.

4.5 Hardware support for Interval Arithmetic

In this section we show that a few simple enhancements to standard RISC chips will enable us to attain very high performance interval arithmetic operations for multiplication and addition,

```
/* X in [R2,R3], Y in [R4,R5], return X*Y in [R6,R7]*/
  mulDDI  R2,R4, R8, R9    /* cycle 1 */
  mulDDI  R2,R5,R10,R11
  mulDDI  R3,R4,R12,R13
  mulDDI  R3,R5 R14,R14
  iszeroI R2,R3,I1
  iszeroI R4,R5,I2
  min     R8,R10,R8        /* cycle 2 */
  max     R9,R11,R9
  min     R12,R14,R12
  max     R13,R15,R13
  min     R8,R12,R6        /* cycle 3 */
  max     R9,R13,R7
  or      I1,I2,I3
  mvifz   I3,R6          /* cycle 4 */
  mvifz   I3,R7
```

Figure 5: 4 cycle interval multiplication on a superscalar processor

which in turn can lead to high performance narrowing routines. It would of course be possible to implement interval arithmetic and narrowing operations directly in hardware, but this is difficult to do in a RISC processor where the goal is to keep the instructions as simple as possible. The new machine instructions which we propose are the following:

```
 mulDDI  R1,R2,R3,R4  <=>  R3 := mul_loE(R1,R2), R4 := mul_hiE(R1,R2)
 divDDI  R1,R2,R3,R4  <=>  R3 := div_loE(R1,R2), R4 := div_hiE(R1,R2)
 min     R1,R2,R3     <=>  R3 := min(R1,R2)
 max     R1,R2,R3     <=>  R3 := max(R1,R2)
 mvifz   R1,R2        <=>  if (R1=0) R2 := 0
 iszeroI R1,R2,I1     <=>  I1 := (R1==0 && R2==0)
```

The instructions `mulDDI` (resp. `divDDI`) compute an IEEE interval $[R3, R4]$ containing $R1 * R2$ (resp. $R1/R2$) with respect to the enhanced rounding modes. Observe that these six instructions do not require branching and hence can optimize the flow in a pipelined architecture. These instructions, together with the usual assembly instructions make it possible to compute an interval multiplication or interval division operation every cycle with low latency on a general purpose pipelined superscalar machine. For example, a pipelined, superscalar machine with at least 4 multiply/divide units and 11 integer units could be used to multiply two intervals every cycle with a latency of 4 cycles using the the assembly code in Figure 5.

	unoptimized code			optimized code		
	min	avg	max	min	avg	max
AD	1.00	4.09	12.00	1.00	2.67	7.00
IA	2.00	7.15	11.00	2.00	6.12	10.00
EM	1.00	38.59	64.00	1.00	14.99	25.00
PA	35.00	47.11	65.00	28.00	37.41	51.00

Figure 6: Benchmarks in microseconds on 150MHz R4400

5 Performance Benchmarks

We have described four narrowing algorithms in the previous sections.

- **PA** The Partition Algorithm in Figure 1 is a straightforward implementation of the the Cleary/Hyvonen algorithm.
- **AD** The Abstract Domain Algorithm in Figure 2 was obtained by using abstract interpretation to optimize PA.
- **IA** The Interval Arithmetic Algorithm defined in Figure 4, where interval multiplication and division are done using case analysis, and
- **EM** The Interval Arithmetic Algorithm but where the extended modes are used to compute products and modified quotients of intervals.

We ran all four algorithms (AD,IE,EM,PA) on 10000 randomly generated narrowing problems and we timed each narrowing with an accuracy of 1 microsecond (by iterating 10000 times using a 0.01 sec timer). The results in Figure 6 were obtained using 1 processor of a 150 MHz SGI Onyx with R4400 processors. The unoptimized code used the `gcc` compiler with no optimization switches, whereas the optimized code used the `-O3` switch. The results show that the AD implementation has about the same range of execution times as IA, but for the "random" distribution we considered AD was twice as fast on the average. The EM and PA implementations are nearly equivalent for unoptimized code. EM is about twice as fast as PA when optimized, but is still 2-3 times slower than the IA. The code used for these experiments can be found in [7].

We have found that many of the standard optimizing C compilers generate code that does not respect the rounding modes, and hence when our narrowing procedures are compiled with optimization switches set, they often produce incorrect results. When the optimization is turned off then, correct code is generated. The problem seems to be that modifying rounding modes is usually done by setting a bit in a control register and the compilers we used would inline the code for changing the mode but then incorrectly move arithmetic operators over the mode-changing instruction apparently without realizing that this would change the semantics of the program.

Ideally one would want the IEEE 754 standard to become a part of the ISO C standard so that all C compilers would correctly handle rounding mode declarations. Until that time one must exercise caution. In particular, critical applications demand that the assembly code be hand checked for correctness, as well as efficiency.

References

[1] Benhamou, F. and Older, W., Applying interval arithmetic to real, integer, and boolean constraints. Journal of Logic Programming.

[2] John C. Cleary, Logical Arithmetic, Future Computing Systems, 2(2), 1987, pp. 125-149.

[3] John C. Cleary, Alan Dewar, Vinit Kaushik and Sue Rempel Scheme Prolog 1.2 `http://www.cs.indiana.edu/scheme-repository/code.lang.html`

[4] Patrick Cousot, and Radhia Cousot, Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixed Points 4th Symposium on Principles of Programming Languages, January 1977, pp. 238-252.

[5] Goldberg, D. What Every Computer Scientist Should Know About Floating-Point Arithmetic ACM Computing Surveys, Vol. 23, No. 1, 1991.

[6] Hickey, T. CLP(F) and Constrained ODEs ECRC Technical report Proceedings of the Workshop and Constraints and Modelling Ithica, NY, 1994

[7] Hickey, T. Smath library. `http://www.cs.brandeis.edu/~tim/smath.html`

[8] Eero Hyvonen. Constraint reasoning based on interval arithmetic: the tolerance propogation approach Artificial Intelligence, 58 (1992), pp. 71-112.

[9] IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985. Institute of Electrical and Electronics Engineers, Inc., 345 East 47th St, NY, NY, 10017, USA.

[10] Older, W. and Vellino, A., Constraint Arithmetic on Real Intervals, in Constraint Logic Programming: Selected Research Colmerauer, A. and Benhamou, F. (eds), MIT Press 1993.

[11] Pascal van Hentenryck, and Laurent Michel. `Newton`: Constraint Programming over Nonlinear Constraints Brown University CS Dept. Tech. report, 1996.