Acceleration of contractor algebra on RISCV in the context of mobile robotic

Filiol Pierre, Jaulin Luc,
Le Lann Jean-Christophe, Bollengier Theotime

June 25, 2023

Table of contents

1. A bit of context
2. The RiscV standard
3. Interval representation
4. Adding support for xinterval
5. Emulation platform
6. Conclusion

Interval arithmetic in robotic

Interval analysis is standardized in the IEEE 1788 standard .

Robotic applications using intervals rely on libraries such as :

1. ibex/gaol
2. libieee1788
3. mpfi

Such libraries have flaws

1. Portability issues.
2. Not designed with robotic in mind
3. Lack of documentation

Interval arithmetic in robotic

Interesting features for a robotic-oriented implementation

Precision is often not the biggest concern in mobile robotic.
However, some other criteria are worth to be optimized :

1. Overall execution speed
2. Portability
3. Guaranteed results (even if pessimistic)
4. Energy efficiency (embedded context).

Hardware implementation can be a solution

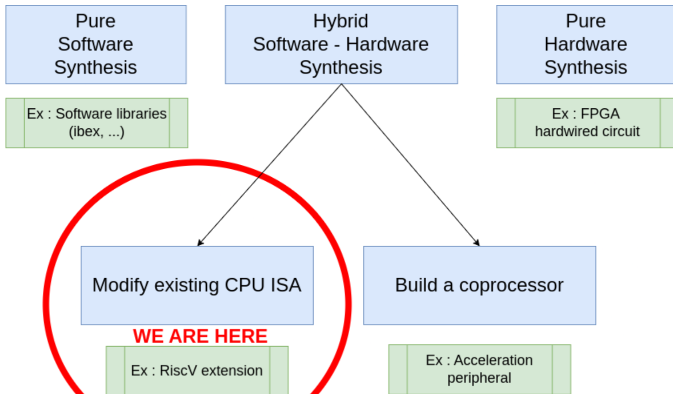
This allows to

1. Reach a more accurate bitwise and timing mastery
2. Portable by essence (you provide the coprocessor)
3. Be power efficient

Interval arithmetic in robotic

Scope of the study

There are several ways to produce hardware accelerators :



The RiscV standard

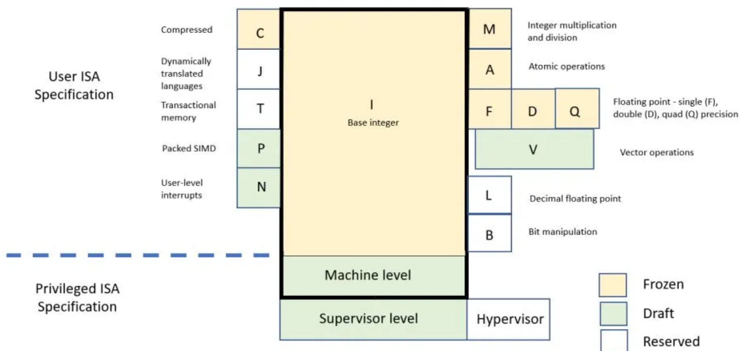
Some key strengths

1. Open specification and standard
2. Modular
3. Extensible
4. Simple (compared to x86/arm/...)
5. Complete software stack (gcc, ...)

Extensibility is very interesting for us

We can insert our interval operations and modify riscv-gcc to use them.

Riscv modular architecture



Extending the RiscV standard

RiscV has several advantages in term of extensibility

1. It offers standard extensions to start building your solution
2. By design, a lot of space is available for additional instructions and extension
3. RiscV gcc compiler can be modified to support new instructions (strong compiler basis).

The goal is to build a RiscV extension called xinterval

1. It is based on standard extensions I,M,A,F,D
2. It provides instructions for computing with intervals
3. The new instructions can be used in C language

Why I,M,A,F,D ?

RV32I brings basic integer instructions + integer registers

Base Integer Instructions: RV32I and RV64I

Category	Name	Fmt	RV32I Base		+RV64I	
Shifts	Shift Left Logical	R	SLL	rd,rs1,rs2	SLLW	rd,rs1,rs2
	Shift Left Log. Imm.	I	SLLI	rd,rs1,shamt	SLLIW	rd,rs1,shamt
	Shift Right Logical	R	SRL	rd,rs1,rs2	SRLW	rd,rs1,rs2
	Shift Right Log. Imm.	I	SRLI	rd,rs1,shamt	SRLIW	rd,rs1,shamt
	Shift Right Arithmetic	R	SRA	rd,rs1,rs2	SRAW	rd,rs1,rs2
	Shift Right Arith. Imm.	I	SRAI	rd,rs1,shamt	SRAIW	rd,rs1,shamt
Arithmetic	ADD	R	ADD	rd,rs1,rs2	ADDW	rd,rs1,rs2
	ADD Immediate	I	ADDI	rd,rs1,imm	ADDIW	rd,rs1,imm
	SUBtract	R	SUB	rd,rs1,rs2	SUBW	rd,rs1,rs2
	Load Upper Imm	U	LUI	rd,imm		
	Add Upper Imm to PC	U	AUIPC	rd,imm		
Logical	XOR	R	XOR	rd,rs1,rs2		
	XOR Immediate	I	XORI	rd,rs1,imm		
	OR	R	OR	rd,rs1,rs2		
	OR Immediate	I	ORI	rd,rs1,imm		
	AND	R	AND	rd,rs1,rs2		
	AND Immediate	I	ANDI	rd,rs1,imm		
Compare	Set <	R	SLT	rd,rs1,rs2		
	Set < Immediate	I	SLTI	rd,rs1,imm		
	Set < Unsigned	R	SLTU	rd,rs1,rs2		
	Set < Imm Unsigned	I	SLTIU	rd,rs1,imm		
Branches	Branch =	B	BEQ	rs1,rs2,imm		
	Branch ≠	B	BNE	rs1,rs2,imm		
	Branch <	B	BLT	rs1,rs2,imm		
	Branch ≥	B	BGE	rs1,rs2,imm		
	Branch < Unsigned	B	BLTU	rs1,rs2,imm		
	Branch ≥ Unsigned	B	BGEU	rs1,rs2,imm		
Jump & Link	J&L	J	JAL	rd,imm		
	Jump & Link Register	I	JALR	rd,rs1,imm		
Synch	Synch thread	I	FENCE			
	Synch Instr & Data	I	FENCE.I			
Environment	CALL	I	ECALL			
	BREAK	I	EBREAK			

Why I,M,A,F,D ?

RV32F and RV32D brings basic floating point instructions

1. arithmetic instructions for float and double
2. special math instructions for float and double
3. conversion instructions
4. memory related instructions

Register configurations

Using I,M,A,F,D provides the following registers :

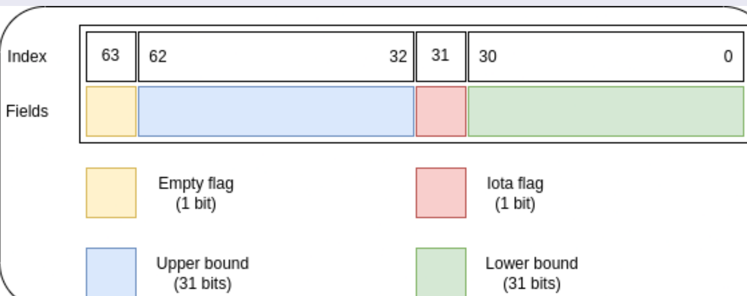
1. 32 registers of size 32 bits for unsigned instructions
2. 32 registers of size 64 bits for floating-point instructions

32 bits RiscV allows to have 64 bits long registers.

How to represent an interval in hardware ?

Intervals are represented using a 64 bits word

They fit inside the 64 bits double registers brought by D extension.



About bounds encoding and flags

Upper and lower bounds are encoded on 31 bits

This differs from the IEEE-754 standard which uses 32 bits.

1. 1 sign bit (same as IEEE-754)
2. 7 bits exponent (against 8 in IEEE-754)
3. 23 bits mantissa (same as IEEE-754)

Overall precision remain the same but the range of number we can represent is smaller. This difference is often pointless in robotic.

This allows to define two flags

1. an empty flag (to represent the empty set)
2. a iota flag

About the iota flag

In one of our previous work, we highlighted a phenomenon which happens in most interval solvers.

Let us consider the following set $\mathbb{X}$

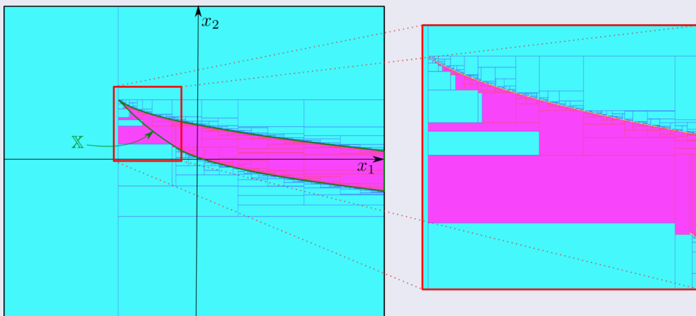
$$\mathbb{X} = \{(x_1, x_2) \mid x_2 + \sqrt{x_1 + x_2} \in [1, 2]\}. \quad (1)$$

Let us try to compute an inner and outer approximations of $\mathbb{X}$ using an existing solver (CODAC) :

```python
from codac import *
from vibes import *
X0=IntervalVector([[-10,10],[-10,10]])
f = Function(x1,x2,x2+sqrt(x1+x2))
S=SepFwdBwd(f,Interval(1,2))
vibes.beginDrawing()
SIVIA(X0,S,0.01)
```

About the iota flag

We obtain the following paving (which is wrong)



About the iota flag

Some box are wrongly classified

This happens because contractor-based methods obtain an inner approximation by considering a contractor for the complentary of $\mathbb{X}$ as

$$\mathbb{X} = \{(x_1, x_2) \mid x_2 + \sqrt{x_1 + x_2} \notin [1, 2]\}. \qquad (2)$$

whereas it should be

$$\overline{\mathbb{X}} = \{(x_1, x_2) \mid x_2 + \sqrt{x_1 + x_2} \notin [1, 2] \text{ or } x_1 + x_2 < 0\}. \qquad (3)$$

This what iota flag is meant for

It is used to mark intervals that are victims of this phenomenon after a chain of forward contractions. Those intervals are then trapped and handled accordingly during backward contraction.

About the iota flag

Some box are wrongly classified

This happens because contractor-based methods obtain an inner approximation by considering a contractor for the complentary of $\mathbb{X}$ as

$$\mathbb{X} = \{(x_1, x_2) \mid x_2 + \sqrt{x_1 + x_2} \notin [1, 2]\}. \tag{4}$$

whereas it should be

$$\overline{\mathbb{X}} = \{(x_1, x_2) \mid x_2 + \sqrt{x_1 + x_2} \notin [1, 2] \text{ or } x_1 + x_2 < 0\}. \tag{5}$$

This what iota flag is meant for

It is used to mark intervals that are victims of this phenomenon after a chain of forward contractions. Those intervals are then trapped and handled accordingly during backward contraction.

The xinterval custom extension

Overview

1. It requires I,M,A,F,D standard extensions.
2. It implements the interval model detailed previously.
3. It consists of a set of assembly instructions which extend RiscV standard.

Main instructions of xinterval

Conversion functions

Name	Type	Effect
itvlcvt.x.inf	D,S	Set the lower bound of the interval stored in rd to be the fp value in rs1.
itvlcvt.x.sup	D,S	Set the upper bound of the interval stored in rd to be the fp value in rs1.
itvlcvt.x.sup	D,S,T	Use the fp values in rs1 and rs2 to build a new interval in rd.

Main instructions of xinterval

Check functions

Name	Type	Effect
itvlisempty	d,S	Tells if the interval stored in rs1 is empty
itvlhasiota	d,S	Tells if the interval stored in rs1 is iota-flagged

Set functions

Name	Type	Effect
itvlinter	D,S,T	Store in rd the intersection of the intervals in rs1 and rs2
itvlunion	D,S,T	Store in rd the union of the intervals in rs1 and rs2

Main instructions of xinterval

Forward and backward contractors

Addition	Subtraction	Multiplication	Division
Square root	Square	Exponential	Logarithm
Cosinus	Sinus		

Using addition forward contractor in assembly

addFwCtc *fs2, fs3, fs4*

Using addition backward contractor in assembly

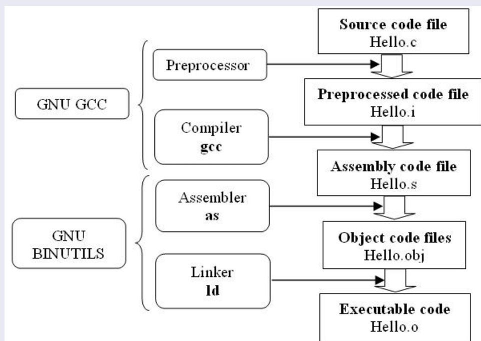
addBwCtc1 *fs3 fs2, fs3, fs4*
addBwCtc2 *fs4 fs2, fs3, fs4*

How to make our new instructions valid in C ?

By default, gcc is totaly ignorant about xinterval

We need to instruct him how to use them. Luckily, this is handled by the riscv-toolchain.

Solution : Modify the assembler and binutils



Modifying binutils

Telling the assembler how to handle forward contractors

```
/* interval arithmetic instructions */
{"itvladd",        0, INSN_CLASS_XINTERVAL, "D,S,T", MATCH_ITVLADD,        MASK_ITVLADD,        match_opcode, 0},
{"itvlsub",        0, INSN_CLASS_XINTERVAL, "D,S,T", MATCH_ITVLSUB,        MASK_ITVLSUB,        match_opcode, 0},
{"itvlmul",        0, INSN_CLASS_XINTERVAL, "D,S,T", MATCH_ITVLMUL,        MASK_ITVLMUL,        match_opcode, 0},
{"itvldiv",        0, INSN_CLASS_XINTERVAL, "D,S,T", MATCH_ITVLDIV,        MASK_ITVLDIV,        match_opcode, 0},
{"itvlinter",      0, INSN_CLASS_XINTERVAL, "D,S,T", MATCH_ITVLINTER,      MASK_ITVLINTER,      match_opcode, 0},
{"itvlunion",      0, INSN_CLASS_XINTERVAL, "D,S,T", MATCH_ITVLUNION,      MASK_ITVLUNION,      match_opcode, 0},
{"itvlsqrt",       0, INSN_CLASS_XINTERVAL, "D,S,T", MATCH_ITVLSQRT,       MASK_ITVLSQRT,       match_opcode, 0},
{"itvlsqr",        0, INSN_CLASS_XINTERVAL, "D,S,T", MATCH_ITVLSQR,        MASK_ITVLSQR,        match_opcode, 0},
{"itvlexp",        0, INSN_CLASS_XINTERVAL, "D,S,T", MATCH_ITVLEXP,        MASK_ITVLEXP,        match_opcode, 0},
{"itvllog",        0, INSN_CLASS_XINTERVAL, "D,S,T", MATCH_ITVLLOG,        MASK_ITVLLOG,        match_opcode, 0},
{"itvlcos",        0, INSN_CLASS_XINTERVAL, "D,S,T", MATCH_ITVLCOS,        MASK_ITVLCOS,        match_opcode, 0},
{"itvlsin",        0, INSN_CLASS_XINTERVAL, "D,S,T", MATCH_ITVLSIN,        MASK_ITVLSIN,        match_opcode, 0},
```

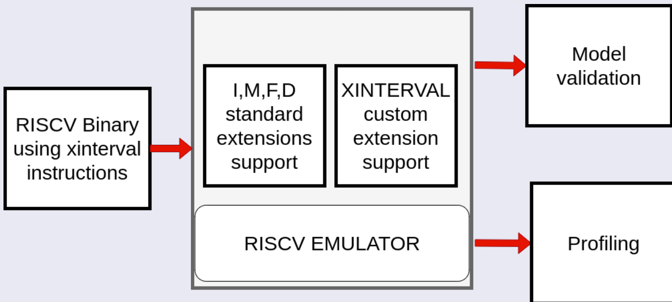
Modifying binutils

Now this code is valid in C

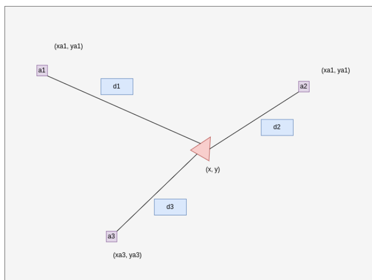
```
1 // inline function to allow the use of xinterval
      instruction addfwctc
2 // This instruction add 2 intervals stored in double
      registers and stores the result in a double
      register
3 inline interval __attribute__((always_inline))
      _addFwCtc(interval itv1, interval itv2) {
4   interval result;
5   asm("addfwctc %0,%1,%2" : "=f"(result) : "f"(itv1),
      "f"(itv2));
6   return result;
7 }
```

Building an emulator to test everything

We have implemented an emulator to test our model



A classic localization problem



Localisation problem using landmarks with known positions

1. We want to estimate the position (x,y) of a robot which navigates around 3 landmarks with known positions.
2. It periodically receives his distance relative to each landmark with a given accuracy ϵ.

A classic localization problem

Localisation problem using landmarks with known positions

1. At each step (x,y) must obey the following constraints :
$(x - a_{1x})^2 + (y - a_{1y})^2 \in [(d_1 - \epsilon)^2, (d_1 + \epsilon)^2]$
$(x - a_{2x})^2 + (y - a_{2y})^2 \in [(d_2 - \epsilon)^2, (d_2 + \epsilon)^2]$
$(x - a_{3x})^2 + (y - a_{3y})^2 \in [(d_3 - \epsilon)^2, (d_3 + \epsilon)^2]$
2. Using properties of forward-backward propagation, we can find a contractor which computes the set of x and y which meet those requirements.

Building the application

The compiled code

The code below corresponds to the contractor to one landmark :

```
box2d CinRing(box2d input, interval cx, interval cy, interval r) {

    interval x = input.x;
    interval y = input.y;

    interval a = _subFwCtc(x, cx);
    interval b = _subFwCtc(y, cy);

    interval a2 = _sqrFwCtc(a);
    interval b2 = _sqrFwCtc(b);
    interval r2 = _sqrFwCtc(r);

    a2 = _addBwCtc1(r2, a2, b2);
    b2 = _addBwCtc2(r2, a2, b2);

    a = _sqrBwCtc(a2, a);
    b = _sqrBwCtc(b2, b);

    x = _subBwCtc1(a, x, cx);
    y = _subBwCtc1(b, y, cy);

    box2d output = {x, y};
    return output;
}
```

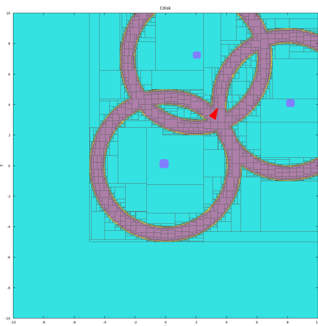
Building the application

Binary dump of the compiled code

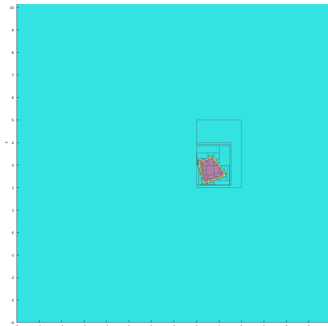
```
80002e24 <CinRing>:
80002e24: e4010113          addi  sp,sp,-448
80002e28: 1a812e23          sw  s0,444(sp)
80002e2c: 1c010413          addi  s0,sp,448
80002e30: 22b587d3          fmv.d fa5,fa1
80002e34: e6c43427          fsd fa2,-408(s0)
80002e38: e6d43027          fsd fa3,-416(s0)
80002e3c: e4e43c27          fsd fa4,-424(s0)
80002e40: e6a43827          fsd fa0,-400(s0)
80002e44: e6f43c27          fsd fa5,-392(s0)
80002e48: e7043787          fld fa5,-400(s0)
80002e4c: fef43427          fsd fa5,-24(s0)
80002e50: e7843787          fld fa5,-392(s0)
80002e54: fef43027          fsd fa5,-32(s0)
80002e58: fe843787          fld fa5,-24(s0)
80002e5c: eaf43c27          fsd fa5,-328(s0)
80002e60: e6843787          fld fa5,-408(s0)
80002e64: eaf43827          fsd fa5,-336(s0)
80002e68: eb843787          fld fa5,-328(s0)
80002e6c: eb043707          fld fa4,-336(s0)
80002e70: 02e7c78b          subfwctc  fa5,fa5,fa4
80002e74: eaf43427          fsd fa5,-344(s0)
80002e78: ea843787          fld fa5,-344(s0)
80002e7c: fcf43c27          fsd fa5,-40(s0)
80002e80: fe043787          fld fa5,-32(s0)
80002e84: ecf43827          fsd fa5,-304(s0)
80002e88: e6043787          fld fa5,-416(s0)
80002e8c: ecf43427          fsd fa5,-312(s0)
80002e90: ed043787          fld fa5,-304(s0)
80002e94: ec843707          fld fa4,-312(s0)
80002e98: 02e7c78b          subfwctc  fa5,fa5,fa4
```

Building the application

The paving obtained from inside the emulator :



(a) Union of contractors



(b) Intersection of contractors

Figure: Using contractors to solve a localisation problem

Building the application

Our emulator allows us to monitor the execution

We can for example :

1. Obtain various metrics (nb of instructions, frequency, ...).
2. Estimate timing and clock cycles.

Conclusion

During this presentation we have shown how to :

1. Build a RiscV extension to compute with intervals.
2. Modify the gcc compiler to use it in C.
3. Build an emulator to test and profile our model.
4. Build an run a small robotic application.