5e Rencontres Arithmétique de l'Informatique Mathématique (RAIM 2012)
Dijon, 20-22 juin 2012

Synthesis of fixed-point programs based on instruction selection

... the case of polynomial evaluation

Amine Najahi

Advisors: Matthieu Martel and Guillaume Revy

Joint work with Christophe Mouilleron

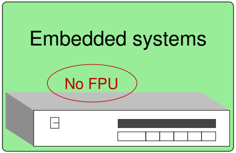
Équipe-projet DALI, Univ. Perpignan Via Domitia
LIRMM, CNRS: UMR 5506 - Univ. Montpellier 2

Motivation

- Embedded systems are ubiquitous
  - microprocessors and/or DSPs dedicated to one or a few specific tasks
  - satisfy constraints: area, energy consumption, conception cost

Motivation

- Embedded systems are ubiquitous
  - microprocessors and/or DSPs dedicated to one or a few specific tasks
  - satisfy constraints: area, energy consumption, conception cost
- Some embedded systems do not have any FPU (floating-point unit)

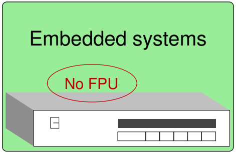


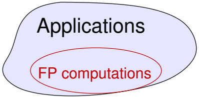
- Highly used in audio and video applications
  - demanding on floating-point computations

Motivation

- Embedded systems are ubiquitous
  - microprocessors and/or DSPs dedicated to one or a few specific tasks
  - satisfy constraints: area, energy consumption, conception cost

- Some embedded systems do not have any FPU (floating-point unit)

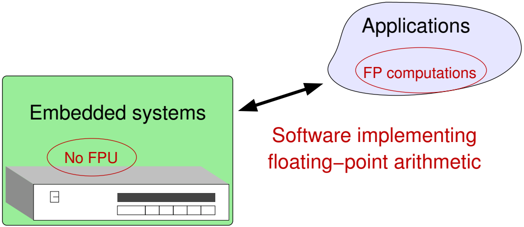




- Highly used in audio and video applications
  - demanding on floating-point computations

Motivation

- Embedded systems are ubiquitous
  - microprocessors and/or DSPs dedicated to one or a few specific tasks
  - satisfy constraints: area, energy consumption, conception cost
- Some embedded systems do not have any FPU (floating-point unit)

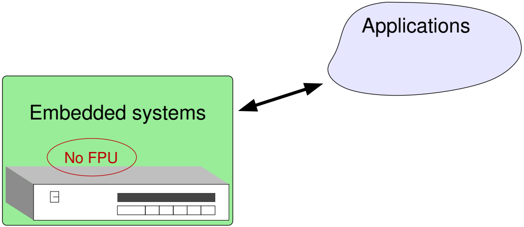


- Highly used in audio and video applications
  - demanding on floating-point computations

Motivation

- Embedded systems are ubiquitous
  - microprocessors and/or DSPs dedicated to one or a few specific tasks
  - satisfy constraints: area, energy consumption, conception cost

- Some embedded systems do not have any FPU (floating-point unit)



- Highly used in audio and video applications
  - demanding on floating-point computations

How to use floating-point programs on embedded systems?

- Two approaches to continue using numerical algorithms on these cores:
  1. convert the entire numerical application from floating to fixed-point arithmetic
  2. write a floating-point emulation library and link the numerical application against it

Fixed-point conversion

- ✓ produces a fast code
- ✓ consumes less energy
- ✗ machine specific: no standard
- ✗ smaller dynamic range than floating-point
- ✗ tedious and time consuming

Floating-point support design

- ✓ tons of code are written using floating-point
- ✓ an algorithm can be synthesized on a PC and then transferred to the device without modifications
- ✗ slower
- ✗ tedious and time consuming

↪ There is a need for the automation of both processes.

Fixed-point conversion *vs.* floating-point emulation design

- Floating to fixed-point conversion tools:
  - addressed by the ANR project DEFIS, with IRISA, LIP6, CEA, THALES, INPIXAL
  - some tools are currently developed: ID.Fix, . . .
  - two main approaches:
    1. statistical methods: perform well, but provide no guarantees and may be slow.
    2. analytical methods: usually quite pessimistic, but they are safer to use.

- Floating-point emulation support:
  - a number of high quality emulation libraries exist: FLIP, SoftFloat,. . .
  - more or less compliant with the IEEE-754 standard
  - FLIP: relies on polynomial evaluation to evaluate division and square root
    - a huge number of schemes for evaluating a given polynomial $\rightsquigarrow$ development of CGPE
    - $\approx$ 50 % of FLIP's code was generated by CGPE.

Outline of the talk

1. The CGPE tool

2. Instruction selection: an extension of CGPE

3. Conclusion and perspectives

Outline of the talk

1. The CGPE tool

2. Instruction selection: an extension of CGPE

3. Conclusion and perspectives

Overview of CGPE

- Goal of CGPE: automate the design of fast and certified C codes for evaluating univariate/bivariate polynomials
  - in fixed-point arithmetic
  - by using the target architecture features (as much as possible)

- Remarks:
  - fast $\rightsquigarrow$ that reduce the evaluation latency on a given target
  - certified $\rightsquigarrow$ for which we can bound the error entailed by the evaluation within the given target's arithmetic

Global architecture of CGPE

- Input of CGPE
  1. polynomial coefficients and variables: value intervals, fixed-point format, ...
  2. set of criteria: maximum error bound and bound on latency (or the lowest)
  3. some architectural constraints: operator cost, parallelism, ...

```
<polynomial>
   <coefficient x="0" y="0" inf="0x00000020" sup="0x00000020" sign="0" integer_part="2" fraction_part="30"/>
   <coefficient x="0" y="1" inf="0x80000000" sup="0x80000000" sign="0" integer_part="1" fraction_part="31"/>
   <coefficient x="1" y="1" inf="0x40000000" sup="0x40000000" sign="0" integer_part="1" fraction_part="31"/>
   <coefficient x="2" y="1" inf="0x10000000" sup="0x10000000" sign="1" integer_part="1" fraction_part="31"/>
   <coefficient x="3" y="1" inf="0x07fe93e4" sup="0x07fe93e4" sign="0" integer_part="1" fraction_part="31"/>
   <coefficient x="4" y="1" inf="0x04eef694" sup="0x04eef694" sign="1" integer_part="1" fraction_part="31"/>
   <coefficient x="5" y="1" inf="0x032d6643" sup="0x032d6643" sign="0" integer_part="1" fraction_part="31"/>
   <coefficient x="6" y="1" inf="0x01c6cebd" sup="0x01c6cebd" sign="1" integer_part="1" fraction_part="31"/>
   <coefficient x="7" y="1" inf="0x00aebe7d" sup="0x00aebe7d" sign="0" integer_part="1" fraction_part="31"/>
   <coefficient x="8" y="1" inf="0x00200000" sup="0x00200000" sign="1" integer_part="1" fraction_part="31"/>
   <variable x="1" y="0" inf="0x00000000" sup="0xfffffe00" sign="0" integer_part="0" fraction_part="32"/>
   <variable x="0" y="1" inf="0x80000000" sup="0xb504f334" sign="0" integer_part="1" fraction_part="31"/>
   <absolute_evalerror value="25081373483158693012463053528118040380976733198921b-191" strict="false"/>
</polynomial>
```

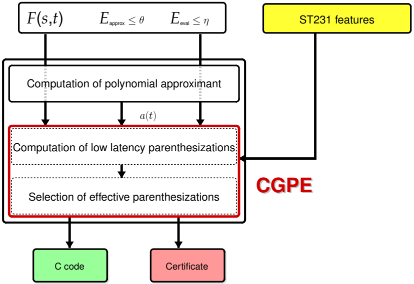
```
cgpe --degree="[8,1]" --xml-input=cgpe-test1.xml --coefs="[100000000111111111]"                    \
     --latency=lowest --gappa-certificate --output                                                 \
     --schedule="[4,2]" --max-kept=5 --operators="[111111111111111111:133333333111333331]" ...
```

Global architecture of CGPE (cont'd)

Internals of CGPE

CGPE proceeds in two steps:

1. Computation step:
   - computes evaluation schemes while reducing their latency on unbounded parallelism
   - considers only two possible arithmetic operations: addition and multiplication
   - produces DAGs that represent the computed efficient schemes



2. Filtering step:
   - prunes the evaluation schemes that do not satisfy different criteria: latency ($\rightsquigarrow$ scheduling filter), accuracy ($\rightsquigarrow$ numerical filter), ...

Global architecture of CGPE (cont'd)

Output of CGPE

```
uint32_t func_d9_0(uint32_t T, uint32_t S)
{
  uint32_t r0  = T >> 2;                    // (+) Q[1.31]
  uint32_t r1  = 0x80000000 + r0;           // (+) Q[1.31]
  uint32_t r2  = mul(S, r1);                // (+) Q[2.30]
  uint32_t r3  = 0x00000020 + r2;           // (+) Q[2.30]
  uint32_t r4  = mul(T, T);                 // (+) Q[0.32]
  uint32_t r5  = mul(S, r4);                // (+) Q[1.31]
  uint32_t r6  = mul(T, 0x07fe93e4);        // (+) Q[1.31]
  uint32_t r7  = 0x10000000 - r6;           // (-) Q[1.31]
  uint32_t r8  = mul(r5, r7);               // (-) Q[2.30]
  uint32_t r9  = r3 - r8;                   // (+) Q[2.30]
  uint32_t r10 = mul(r4, r4);               // (+) Q[0.32]
  uint32_t r11 = mul(S, r10);               // (+) Q[1.31]
  uint32_t r12 = mul(T, 0x032d6643);        // (+) Q[1.31]
  uint32_t r13 = 0x04eef694 - r12;          // (-) Q[1.31]
  uint32_t r14 = mul(T, 0x00aebe7d);        // (+) Q[1.31]
  uint32_t r15 = 0x01c6cebd - r14;          // (-) Q[1.31]
  uint32_t r16 = r4 >> 11;                  // (-) Q[1.31]
  uint32_t r17 = r15 + r16;                 // (-) Q[1.31]
  uint32_t r18 = mul(r4, r17);              // (-) Q[1.31]
  uint32_t r19 = r13 + r18;                 // (-) Q[1.31]
  uint32_t r20 = mul(r11, r19);             // (-) Q[2.30]
  uint32_t r21 = r9 - r20;                  // (+) Q[2.30]
  return r21;
}
```

Listing 1: C code

```
## Coefficients and variables definition
a0 = fixed<-30,dn>(0x00000020p-30);
a1 = fixed<-31,dn>(0x80000000p-31);
a2 = fixed<-31,dn>(0x40000000p-31);
                    ...
a8 = fixed<-31,dn>(0x00aebe7dp-31);
a9 = fixed<-31,dn>(0x00200000p-31);

T = fixed<-32,dn>(fixed<-23,dn>(var0));
S = fixed<-31,dn>(var1);

CertifiedBound =
25081373483158693012463053528118040380976733198921b-191;

## Evaluation scheme
r0 fixed<-31,dn>= T * a2;        Mr0 = T * a2;
r1 fixed<-31,dn>= a1 + r0;       Mr1 = a1 + Mr0;
                    ...
r21 fixed<-30,dn>= r9 - r20;     Mr21 = Mr9 - Mr20;

## Results
{
  (
       var0 in [0x00000000p-32,0xfffffe00p-32]
    /\ var1 in [0x80000000p-31,0xb504f334p-31]
    ->
    /\ r0 in [0,0xffffffffp-31]
    /\ r0 - Mr0 in ?
              ...
    /\ r21 in [0,0xffffffffp-30]
    /\ |r21 - Mr21| - CertifiedBound <= 0
    /\ CertifiedBound in ?
  )
}
```

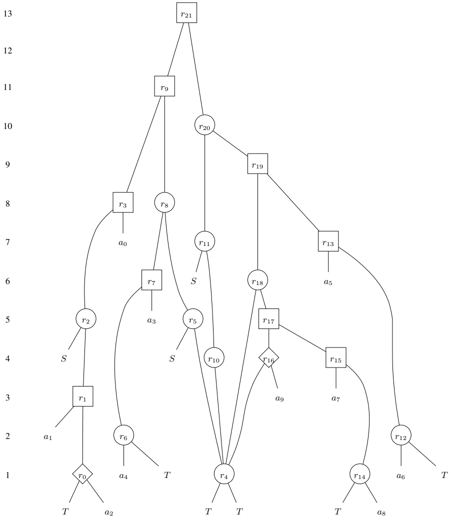
Listing 2: GAPPA certificate

Global architecture of CGPE (cont'd)

Output of CGPE

```
uint32_t func_d9_0(uint32_t T, uint32_t S)
{
  uint32_t r0  = T >> 2;                   // (+) Q[1.31]
  uint32_t r1  = 0x80000000 + r0;          // (+) Q[1.31]
  uint32_t r2  = mul(S, r1);               // (+) Q[2.30]
  uint32_t r3  = 0x00000020 + r2;          // (+) Q[2.30]
  uint32_t r4  = mul(T, T);                // (+) Q[0.32]
  uint32_t r5  = mul(S, r4);               // (+) Q[1.31]
  uint32_t r6  = mul(T, 0x07fe93e4);       // (+) Q[1.31]
  uint32_t r7  = 0x10000000 - r6;          // (-) Q[1.31]
  uint32_t r8  = mul(r5, r7);              // (-) Q[2.30]
  uint32_t r9  = r3 - r8;                  // (+) Q[2.30]
  uint32_t r10 = mul(r4, r4);              // (+) Q[0.32]
  uint32_t r11 = mul(S, r10);              // (+) Q[1.31]
  uint32_t r12 = mul(T, 0x032d6643);       // (+) Q[1.31]
  uint32_t r13 = 0x04eef694 - r12;         // (-) Q[1.31]
  uint32_t r14 = mul(T, 0x00aebe7d);       // (+) Q[1.31]
  uint32_t r15 = 0x01c6cebd - r14;         // (-) Q[1.31]
  uint32_t r16 = r4 >> 11;                 // (-) Q[1.31]
  uint32_t r17 = r15 + r16;                // (-) Q[1.31]
  uint32_t r18 = mul(r4, r17);             // (-) Q[1.31]
  uint32_t r19 = r13 + r18;                // (-) Q[1.31]
  uint32_t r20 = mul(r11, r19);            // (-) Q[2.30]
  uint32_t r21 = r9 - r20;                 // (+) Q[2.30]
  return r21;
}
```

Listing 3: C code



Achievements and lacking features of CGPE

Features achieved by CGPE

- ✓ validated on the ST200 core
- ✓ so far, no ambushes were encountered for $\sqrt{\ }$, $\sqrt[3]{\ }$, $\frac{1}{\sqrt{\ }}$, $\frac{1}{\sqrt[3]{\ }}$ $\cdots$
- ✓ produced optimal schemes for some of the above functions such as $\sqrt{\ }$

Features lacking in CGPE

- ✗ simplistic description of the underlying architecture (*ex.* no handling of advanced operators such as ST200 shift_and_add instruction)
- ✗ the only shifts handled correspond to the multiplication by a power of 2
- ✗ hypotheses are made on the format of the input coefficients

Achievements and lacking features of CGPE

Features achieved by CGPE

- ✓ validated on the ST200 core
- ✓ so far, no ambushes were encountered for $\sqrt{}$, $\sqrt[3]{}$, $\frac{1}{\sqrt{}}$, $\frac{1}{\sqrt[3]{}}$ $\cdots$
- ✓ produced optimal schemes for some of the above functions such as $\sqrt{}$

Features lacking in CGPE

- ✗ simplistic description of the underlying architecture (*ex.* no handling of advanced operators such as ST200 shift_and_add instruction)
- ✗ the only shifts handled correspond to the multiplication by a power of 2
- ✗ hypotheses are made on the format of the input coefficients

Problem: without hypotheses on the formats of the input coefficients, CGPE fails
Solution: add the handling of multiple shifts to CGPE

Shift handling in CGPE

- There are 4 types of shifts to consider:

  1. multiplication by a power of 2 shifts: allows to gain a few cycles
     - shifting is usually less costly than multiplication

Shift handling in CGPE

- There are 4 types of shifts to consider:

  1. multiplication by a power of 2 shifts: allows to gain a few cycles
     - shifting is usually less costly than multiplication
  2. alignment shifts: used to align commas for an arithmetic operation
     - addition of a $Q[1.31]$ and a $Q[2.30]$

Shift handling in CGPE

- There are 4 types of shifts to consider:

  1. multiplication by a power of 2 shifts: allows to gain a few cycles
     - shifting is usually less costly than multiplication
  2. alignment shifts: used to align commas for an arithmetic operation
     - addition of a $Q[1.31]$ and a $Q[2.30]$
  3. leading zeros' elimination shifts: used to gain some bits of precision
     - 0x40000000 in the $Q[2.30]$ format $\rightsquigarrow$ 0x80000000 in the $Q[1.31]$ format

Shift handling in CGPE

- There are 4 types of shifts to consider:

  1. multiplication by a power of 2 shifts: allows to gain a few cycles
     - shifting is usually less costly than multiplication
  2. alignment shifts: used to align commas for an arithmetic operation
     - addition of a $Q[1.31]$ and a $Q[2.30]$
  3. leading zeros' elimination shifts: used to gain some bits of precision
     - 0x40000000 in the $Q[2.30]$ format $\rightsquigarrow$ 0x80000000 in the $Q[1.31]$ format
  4. overflow prevention shifts: used before an arithmetic operation to prevent it from overflowing
     - to prevent the addition of a $Q[1.31]$ and a $Q[1.31]$ from overflowing the $Q[1.31]$ format, both operands are shifted to the $Q[2.30]$ format

- Remark: to detect whether one of these shifts is needed, we rely on:
  - fixed-point arithmetic rules (for case 2)
  - MPFI computations (for cases 1, 3 and 4).

Shift handling in CGPE

- There are 4 types of shifts to consider:

  1. multiplication by a power of 2 shifts: allows to gain a few cycles
     - shifting is usually less costly than multiplication
  2. alignment shifts: used to align commas for an arithmetic operation
     - addition of a $Q[1.31]$ and a $Q[2.30]$
  3. leading zeros' elimination shifts: used to gain some bits of precision
     - 0x40000000 in the $Q[2.30]$ format $\rightsquigarrow$ 0x80000000 in the $Q[1.31]$ format
  4. overflow prevention shifts: used before an arithmetic operation to prevent it from overflowing
     - to prevent the addition of a $Q[1.31]$ and a $Q[1.31]$ from overflowing the $Q[1.31]$ format, both operands are shifted to the $Q[2.30]$ format

Problem: shifts may affect the critical path, potentially increasing the latency of the DAG

Solution: use more advanced instructions to help absorb this increase

- *ex:* shift-and-add instruction available on some fixed-point processors like the ST231

Outline of the talk

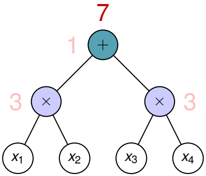
1. The CGPE tool

2. Instruction selection: an extension of CGPE

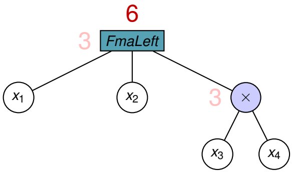
3. Conclusion and perspectives

The problem of instruction selection

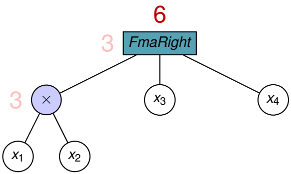
- A well known problem in compilation that was proven to be NP-complete on DAGs.
- Usually solved using a tiling algorithm:
  - input:
    - a DAG representing an arithmetic expression.
    - a set of tiles, with a cost for each.
    - a function that associates a cost to a subtree.
  - output:
    - a set of covering tiles that minimize the cost function.



$$((x_1 \cdot x_2) + (x_3 \cdot x_4))$$



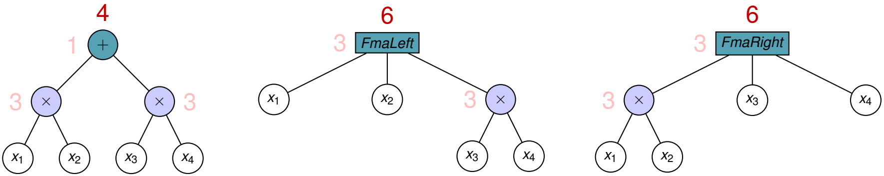
$$FmaLeft(x_1, x_2, (x_3 \cdot x_4))$$



$$FmaRight((x_1 \cdot x_2), x_3, x_4)$$

The problem of instruction selection

- A well known problem in compilation that was proven to be NP-complete on DAGs.
- Usually solved using a tiling algorithm:
  - input:
    - a DAG representing an arithmetic expression.
    - a set of tiles, with a cost for each.
    - a function that associates a cost to a subtree.
  - output:
    - a set of covering tiles that minimize the cost function.



$((x_1 \cdot x_2) + (x_3 \cdot x_4))$ $\quad$ $FmaLeft(x_1, x_2, (x_3 \cdot x_4))$ $\quad$ $FmaRight((x_1 \cdot x_2), x_3, x_4)$

Remark on instruction selection

Some work in the area

Voronenko and Püschel from the Spiral group (2004):

- Automatic Generation of Implementations for DSP Transforms on Fused Multiply-Add Architectures.

✓ They provide a short proof of optimality in the case of trees.

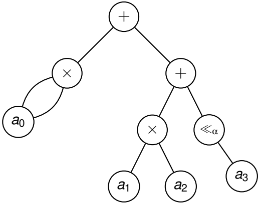
✗ Their method handles FMAs in DAGs but is not generic.

- We wish to integrate numerical verification in the process of instruction selection.

The NOLTIS tiling algorithm

Near-Optimal Instruction Selection algorithm (Koes and Goldstein in CGO-2008)

1: BottomUpDP()
2: TopDownSelect()
3: ImproveCSEDecision()
4: BottomUpDP()
5: TopDownSelect()

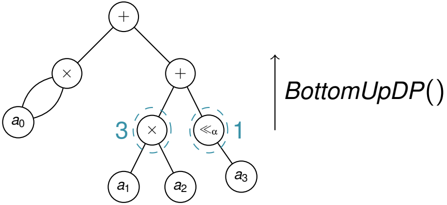


the progress step by step of the tiling algorithm
on the expression $(a_0^2 + ((a_1 \times a_2) + (a_3 \ll \alpha)))$

The NOLTIS tiling algorithm

Near-Optimal Instruction Selection algorithm (Koes and Goldstein in CGO-2008)

1: BottomUpDP()
2: TopDownSelect()
3: ImproveCSEDecision()
4: BottomUpDP()
5: TopDownSelect()

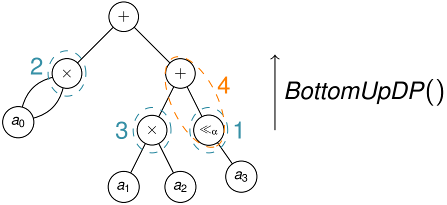


the progress step by step of the tiling algorithm
on the expression $(a_0^2 + ((a_1 \times a_2) + (a_3 \ll \alpha)))$

The NOLTIS tiling algorithm

Near-Optimal Instruction Selection algorithm (Koes and Goldstein in CGO-2008)

```
BottomUpDP()
TopDownSelect()
ImproveCSEDecision()
BottomUpDP()
TopDownSelect()
```

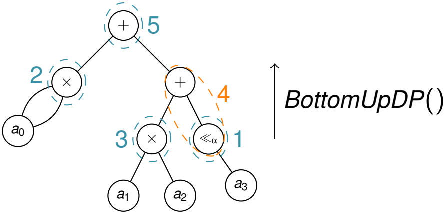


the progress step by step of the tiling algorithm
on the expression $(a_0^2 + ((a_1 \times a_2) + (a_3 \ll \alpha)))$

The NOLTIS tiling algorithm

Near-Optimal Instruction Selection algorithm (Koes and Goldstein in CGO-2008)

1: BottomUpDP()
2: TopDownSelect()
3: ImproveCSEDecision()
4: BottomUpDP()
5: TopDownSelect()

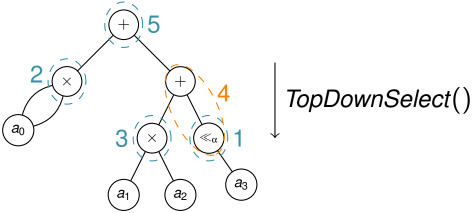


the progress step by step of the tiling algorithm on the expression $(a_0^2 + ((a_1 \times a_2) + (a_3 \ll \alpha)))$

The NOLTIS tiling algorithm

Near-Optimal Instruction Selection algorithm (Koes and Goldstein in CGO-2008)

1: BottomUpDP()
2: TopDownSelect()
3: ImproveCSEDecision()
4: BottomUpDP()
5: TopDownSelect()

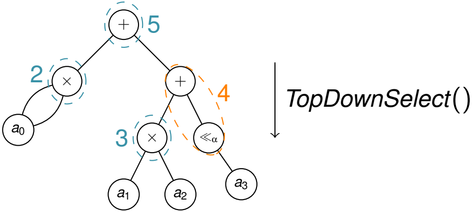


the progress step by step of the tiling algorithm
on the expression $(a_0^2 + ((a_1 \times a_2) + (a_3 \ll \alpha)))$

The NOLTIS tiling algorithm

Near-Optimal Instruction Selection algorithm (Koes and Goldstein in CGO-2008)

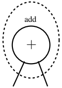
BottomUpDP()
TopDownSelect()
ImproveCSEDecision()
BottomUpDP()
TopDownSelect()

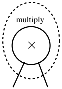


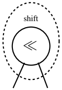
the progress step by step of the tiling algorithm
on the expression $(a_0^2 + ((a_1 \times a_2) + (a_3 \ll \alpha)))$

Instruction tiles considered in CGPE

- Classical tiles
  1. addition tile.
  2. multiplication tile.
  3. shift tile.



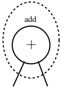


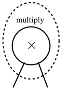


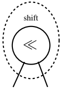
Instruction tiles considered in CGPE

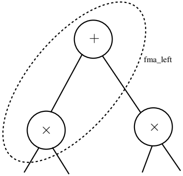
- Classical tiles
  1. addition tile.
  2. multiplication tile.
  3. shift tile.

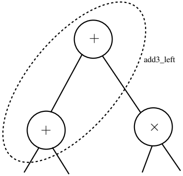
- Advanced tiles
  4. fma tiles (left and right).
  5. add3 tiles (left and right).
  6. shiftAdd tiles (available on the ST200 core).
  7. square tile.

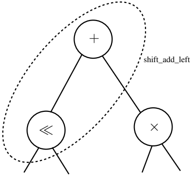


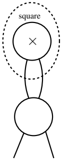












Simple example

- Original code

```
uint32_t func_d9_0(uint32_t T, uint32_t S)
{
  uint32_t r0  = T >> 2;                    // (+) Q[1.31]
  uint32_t r1  = 0x80000000 + r0;           // (+) Q[1.31]
  uint32_t r2  = mul(S, r1);                // (+) Q[2.30]
  uint32_t r3  = 0x00000020 + r2;           // (+) Q[2.30]
  uint32_t r4  = mul(T, T);                 // (+) Q[0.32]
  uint32_t r5  = mul(S, r4);                // (+) Q[1.31]
  uint32_t r6  = mul(T, 0x07fe93e4);        // (+) Q[1.31]
  uint32_t r7  = 0x10000000 - r6;           // (-) Q[1.31]
  uint32_t r8  = mul(r5, r7);               // (-) Q[2.30]
  uint32_t r9  = r3 - r8;                   // (+) Q[2.30]
  uint32_t r10 = mul(r4, r4);               // (+) Q[0.32]
  uint32_t r11 = mul(S, r10);               // (+) Q[1.31]
  uint32_t r12 = mul(T, 0x032d6643);        // (+) Q[1.31]
  uint32_t r13 = 0x04eef694 - r12;          // (-) Q[1.31]
  uint32_t r14 = mul(T, 0x00aebe7d);        // (+) Q[1.31]
  uint32_t r15 = 0x01c6cebd - r14;          // (-) Q[1.31]
  uint32_t r16 = r4 >> 11;                  // (-) Q[1.31]
  uint32_t r17 = r15 + r16;                 // (-) Q[1.31]
  uint32_t r18 = mul(r4, r17);              // (-) Q[1.31]
  uint32_t r19 = r13 + r18;                 // (-) Q[1.31]
  uint32_t r20 = mul(r11, r19);             // (-) Q[2.30]
  uint32_t r21 = r9 - r20;                  // (+) Q[2.30]
  return r21;
}
```

Listing 4: Original C code

- With the fma in 3 cycles and the shift in 1 cycle

```
uint32_t func_tiled(uint32_t T, uint32_t S)
{
  uint32_t r0  = power(T, -2);
  uint32_t r1  = add(0x80000000, r0);
  uint32_t r2  = fma_right(0x00000020, S, r1);
  uint32_t r3  = square(T);
  uint32_t r4  = mul(S, r3);
  uint32_t r5  = mul(T, 0x07fe93e4);
  uint32_t r6  = sub(0x10000000, r5);
  uint32_t r7  = mul(r4, r6);
  uint32_t r8  = sub(r2, r7);
  uint32_t r9  = square(r3);
  uint32_t r10 = mul(S, r9);
  uint32_t r11 = mul(T, 0x032d6643);
  uint32_t r12 = sub(0x04eef694, r11);
  uint32_t r13 = mul(T, 0x00aebe7d);
  uint32_t r14 = sub(0x01c6cebd, r13);
  uint32_t r15 = power(r3, -11);
  uint32_t r16 = add(r14, r15);
  uint32_t r17 = fma_right(r12, r3, r16);
  uint32_t r18 = mul(r10, r17);
  uint32_t r19 = sub(r8, r18);
  return r19;
}
```

Listing 5: Code after tiling

Simple example

Original code

```
uint32_t func_d9_0(uint32_t T, uint32_t S)
{
  uint32_t r0  = T >> 2;                  // (+) Q[1.31]
  uint32_t r1  = 0x80000000 + r0;         // (+) Q[1.31]
  uint32_t r2  = mul(S, r1);              // (+) Q[2.30]
  uint32_t r3  = 0x00000020 + r2;         // (+) Q[2.30]
  uint32_t r4  = mul(T, T);               // (+) Q[0.32]
  uint32_t r5  = mul(S, r4);              // (+) Q[1.31]
  uint32_t r6  = mul(T, 0x07fe93e4);      // (+) Q[1.31]
  uint32_t r7  = 0x10000000 - r6;         // (-) Q[1.31]
  uint32_t r8  = mul(r5, r7);             // (-) Q[2.30]
  uint32_t r9  = r3 - r8;                 // (+) Q[2.30]
  uint32_t r10 = mul(r4, r4);             // (+) Q[0.32]
  uint32_t r11 = mul(S, r10);             // (+) Q[1.31]
  uint32_t r12 = mul(T, 0x032d6643);      // (+) Q[1.31]
  uint32_t r13 = 0x04eef694 - r12;        // (-) Q[1.31]
  uint32_t r14 = mul(T, 0x00aebe7d);      // (+) Q[1.31]
  uint32_t r15 = 0x01c6cebd - r14;        // (-) Q[1.31]
  uint32_t r16 = r4 >> 11;                // (-) Q[1.31]
  uint32_t r17 = r15 + r16;               // (-) Q[1.31]
  uint32_t r18 = mul(r4, r17);            // (-) Q[1.31]
  uint32_t r19 = r13 + r18;               // (-) Q[1.31]
  uint32_t r20 = mul(r11, r19);           // (-) Q[2.30]
  uint32_t r21 = r9 - r20;                // (+) Q[2.30]
  return r21;
}
```

Listing 6: Original C code

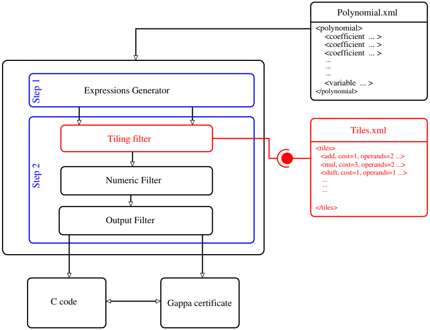
With the fma in 3 cycles and the shift in 3 cycle

```
uint32_t func_tiled(uint32_t T, uint32_t S)
{
  uint32_t r0  = fma_right(0x80000000, T, 0x40000000);
  uint32_t r1  = fma_right(0x00000020, S, r0);
  uint32_t r2  = square(T);
  uint32_t r3  = mul(S, r2);
  uint32_t r4  = mul(T, 0x07fe93e4);
  uint32_t r5  = sub(0x10000000, r4);
  uint32_t r6  = mul(r3, r5);
  uint32_t r7  = sub(r1, r6);
  uint32_t r8  = square(r2);
  uint32_t r9  = mul(S, r8);
  uint32_t r10 = mul(T, 0x032d6643);
  uint32_t r11 = sub(0x04eef694, r10);
  uint32_t r12 = mul(T, 0x00aebe7d);
  uint32_t r13 = sub(0x01c6cebd, r12);
  uint32_t r14 = power(r2, -11);
  uint32_t r15 = add(r13, r14);
  uint32_t r16 = fma_right(r11, r2, r15);
  uint32_t r17 = mul(r9, r16);
  uint32_t r18 = sub(r7, r17);
  return r18;
}
```

Listing 7: Code after tiling

Remarks on instruction selection in CGPE

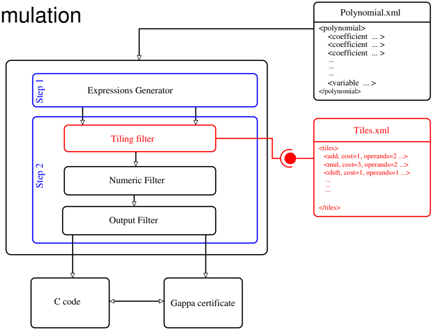
- A separation is achieved between the computation of DAGs (Intermediate Representation) and the code generation process
  - the code can be generated according different criteria ⇝ cost function
  - this general approach allows to tackle other problems (sum, dot-product, ...)



Remarks on instruction selection in CGPE

- A separation is achieved between the computation of DAGs (Intermediate Representation) and the code generation process
  - the code can be generated according different criteria ⇝ cost function
  - this general approach allows to tackle other problems (sum, dot-product, ...)

- We are not bound to use these tiles, we can add many others
  - CGPE can thus serve as a platform of simulation
  - this general approach allows to give some feedback on the eventual need or usefulness of some tiles



Outline of the talk

1. The CGPE tool

2. Instruction selection: an extension of CGPE

3. Conclusion and perspectives

Conclusion

- Code synthesis for fast and certified polynomial evaluation
  - fast and certified C codes, in fixed point arithmetic
  - tool to automate polynomial evaluation implementation, using at best architectural features
  - implemented in the tool CGPE (Code Generation for Polynomial Evaluation)

  `http://cgpe.gforge.inria.fr/`

- Extension of CGPE based on instruction selection:
  - automatic handling of all input formats.
  - better usage of the advanced architectural features (such as fma, add-3, shift-and-add, ...)
  - using a tiling algorithm implies more modularity, as code generation is now an independant process.

Current work and perspectives

Current work

- keep working on instruction selection in CGPE
- make CGPE more general to tackle other problems, like matrix inversion and multiplication, ...

Current work and perspectives

- Current work
  - keep working on instruction selection in CGPE
  - make CGPE more general to tackle other problems, like matrix inversion and multiplication, ...

- Further extensions of CGPE
  - handle other arithmetics like floating-point arithmetic, where the fma tile is more and more ubiquitous
  - target other architectures (like FPGAs)

5e Rencontres Arithmétique de l'Informatique Mathématique (RAIM 2012)
Dijon, 20-22 juin 2012

Synthesis of fixed-point programs based on instruction selection

... the case of polynomial evaluation

Amine Najahi

Advisors: Matthieu Martel and Guillaume Revy

Joint work with Christophe Mouilleron

Équipe-projet DALI, Univ. Perpignan Via Domitia
LIRMM, CNRS: UMR 5506 - Univ. Montpellier 2