DE LA RECHERCHE À L'INDUSTRIE

Self-optimisation using runtime code generation for Wireless Sensor Networks

ComNet-IoT Workshop
ICDCN'16 Singapore

Caroline Quéva Damien Couroussé Henri-Pierre Charles

Univ. Grenoble Alpes, F-38000 Grenoble, France
CEA-LIST, MINATEC Campus, F-38054 Grenoble, France

2016-01-04

Problem statement

- IoT : more and more sensors
  - Low-power sensors
  - Increase lifetime

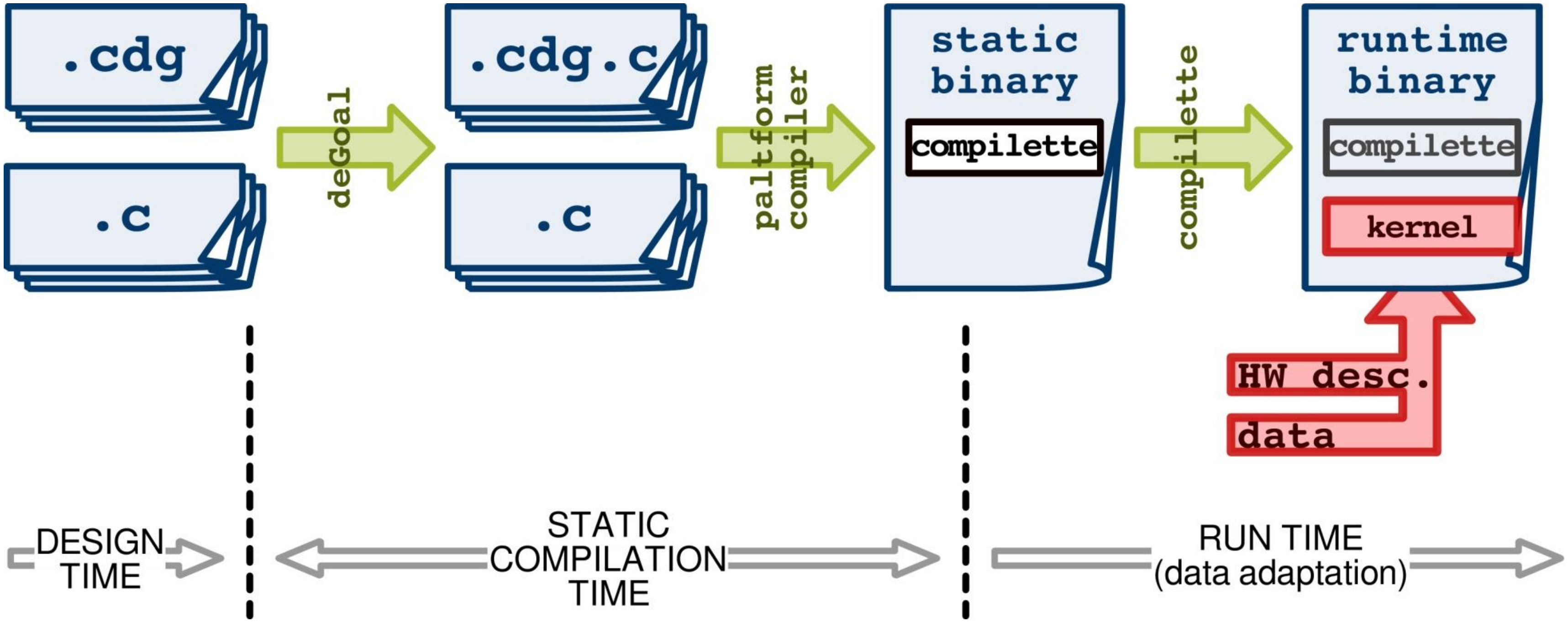
⇒ Self-optimisation

- Some code optimisations are not accessible to static compilers
  - Unknown data or hardware
- Delay code optimisations at runtime
  - Constant propagation, elimination of dead code,
  - Loop unrolling,
  - etc.

Outline

- Code generation with deGoal
- State of the art
- Our approach : Automatisation process
- Results
- Conclusion and future works

Code generation flow



leti & list

deGoal example

Standard code

```
float mul(float a, float b) {
    return a*b;
}



int main()
{
    float result = 0;
    float value = rand();

    for (int i=0; i<5; i++) {
        result += mul(value, (float) i);
    }
}
```

deGoal code

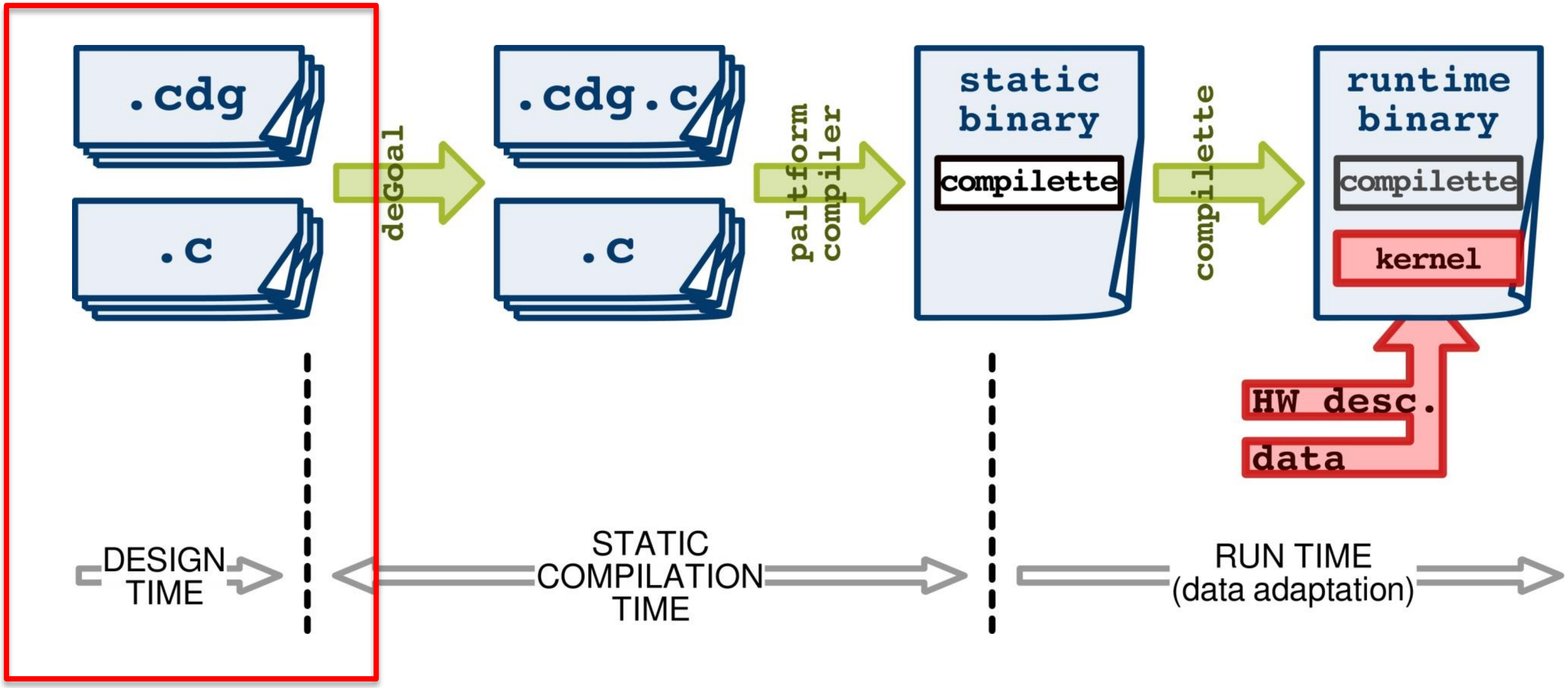
```
void compilette (cdgInsnT *code, float
mulvalue) {
    cdgInsnT *code= CDGALLOC(1024);
    #[
        Begin code Prelude float input
        mul input, input, #(mulvalue)
        rtn
        End
    ]#;
}

int main()
{
    float result = 0;
    float value = rand();
    mulCDG = compilette(value);
    for (int i=0; i<5; i++) {
        result += mulCDG((float) i);
    }
}
```

Features

deGoal

- Reduce execution time
- Runtime portable optimization
- Specialize on runtime data (parameters, hardware)
- Generated code is smaller
- No runtime dependencies with any compiler

deGoal limitation



Written by the developer

JIT

```
Call to f
  If LookupCache(f)
    Execute foptim
  Else
    If ExecCount(f) > Thresh
      foptim <- HotCompile(f_bytecode)
      Execute foptim
    Else
      Interpret f
```

⇒ High memory footprint
⇒ High overhead
⇒ HotCompile is the same for all functions
⇒ No data-dependent optimization

Standard deGoal

```
fspec_val <- Compilette(f, val)
Call to f(val)
  Execute fspec_val
```

⇒ Specialization done by the developer
⇒ Low memory footprint

Self-optimization system

```
Call to f(val)
  If LookupCache(f, val)
    Execute fspec_val
  Else
    If ExecCount(f, val) > Thresh_f
      fspec_val <- Compilette(f, val)
      Execute fspec_val
    Else
      Execute f(val)
```

⇒ Data-dependent self-optimization
⇒ Low memory footprint

Automatisation process

- Library
  - Ready-to-use compilettes (lightweight runtime code generators).
  - No more development cost for the developer

- Code cache
  - Keep several versions of the specialized code
  - Save generation cost
  - Low memory footprint

Use Case : Floating point multiplication

- Floating-point multiplications on MSP430 Wismote platform

- Why ?
  - Standard library function : ~1000 cycles per invocation
  - Micro-controllers lack dedicated HW support for arithmetic computing
  - Linear function often used to convert sensor value to user value
- Specialize on first argument value
- Adjust precision p using mantissa truncation

gcc generic version

```
/* tgcc */
float fmul (float M, float X) {
    return (M*X);
}
```

specialized version

```
/* tgen: code generation */
float (*) (float) fmulM;
fmulM = generate_fmul_code(M, p);

/* tdyn: run the generated routine
float fmul (float X) {
    return fmulM(X);
}
```

Performance metrics

t_{gcc} : execution time of gcc's multiplication routine

t_{gen} : execution time of code generation

t_{dyn} : execution time of the generated function

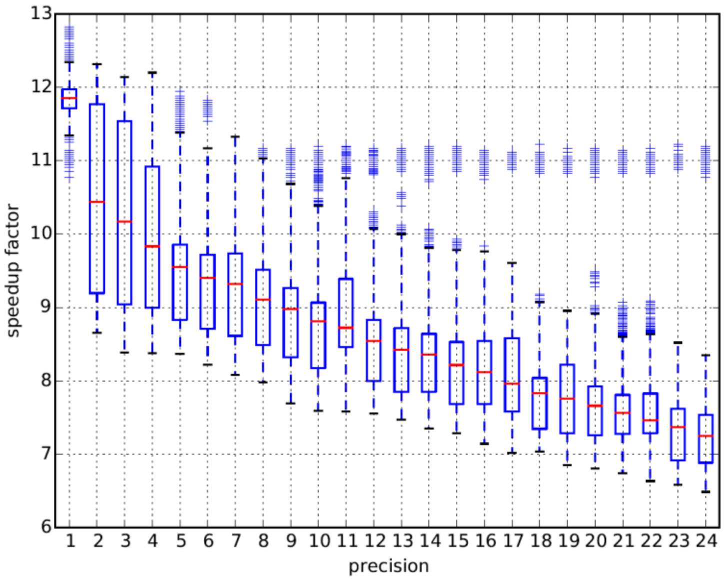
- Speedup :

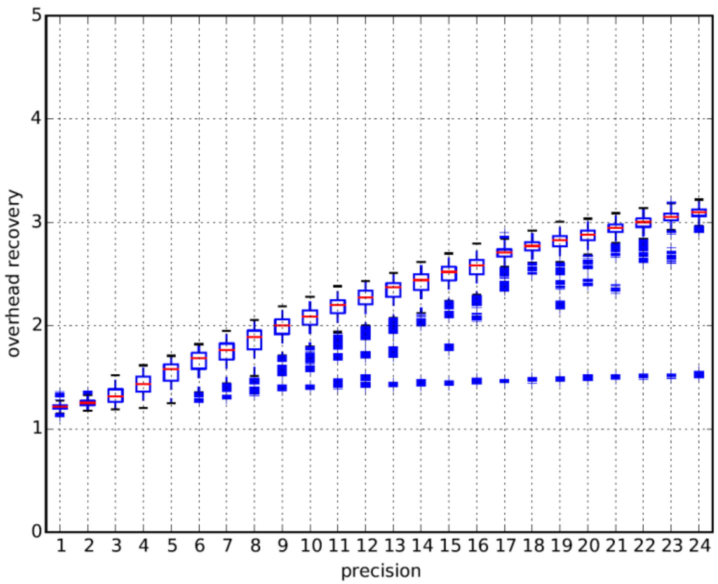
$$s = \frac{t_{dyn}}{t_{gcc}}$$

- Overhead recovery :

$$N = \frac{t_{gen}}{t_{gcc} - t_{dyn}}$$

Results for standard deGoal



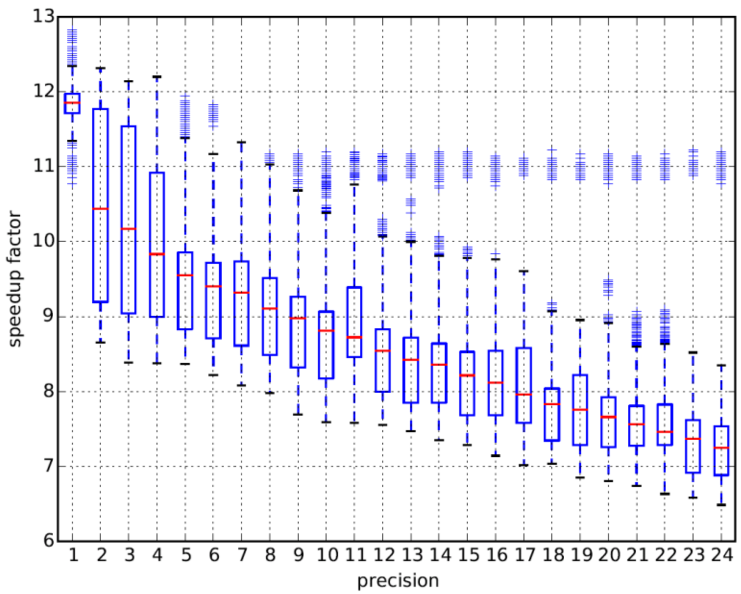


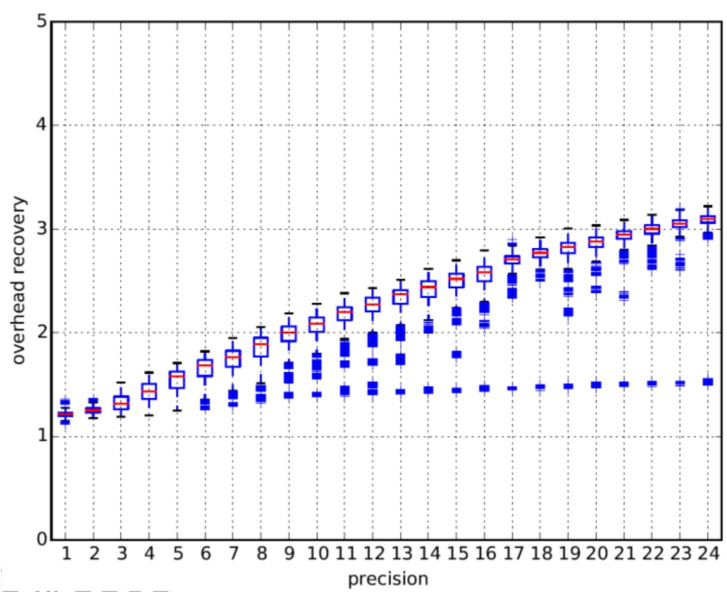
Box plot : Red line is the median, bottom and top of the box are first and third quartiles, individual points are outliers.

- Speedup more than 7
  - and increases if precision is reduced
- Overhead recovery less than 4
  - and decreases if precision is reduced
  - Only need 4 executions of the specialized code to pay off generation time

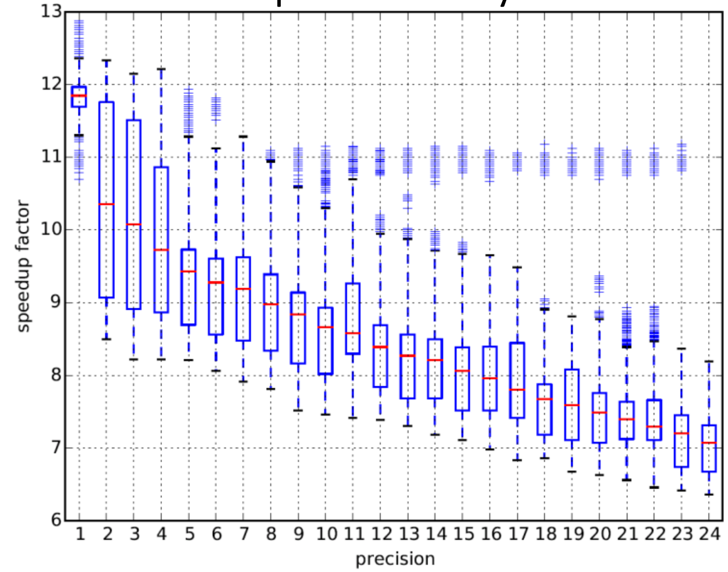
Our results

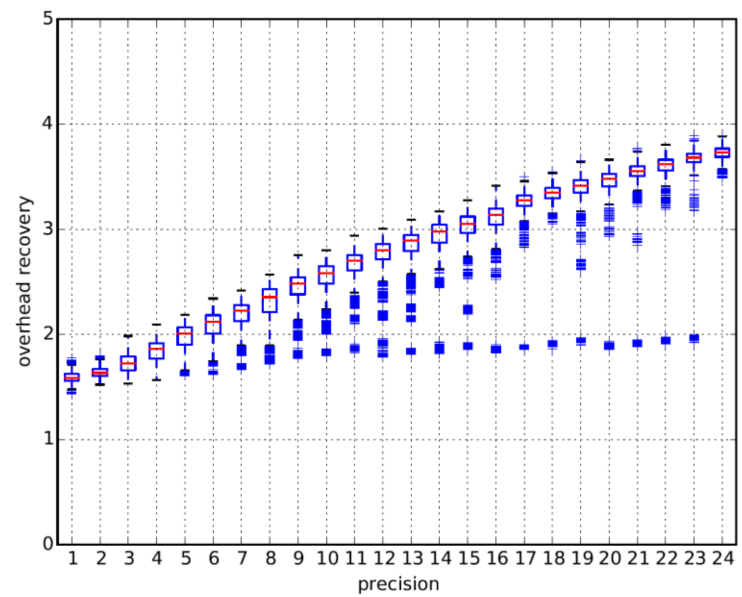
Developer writes the compilette





Self-optimisation system





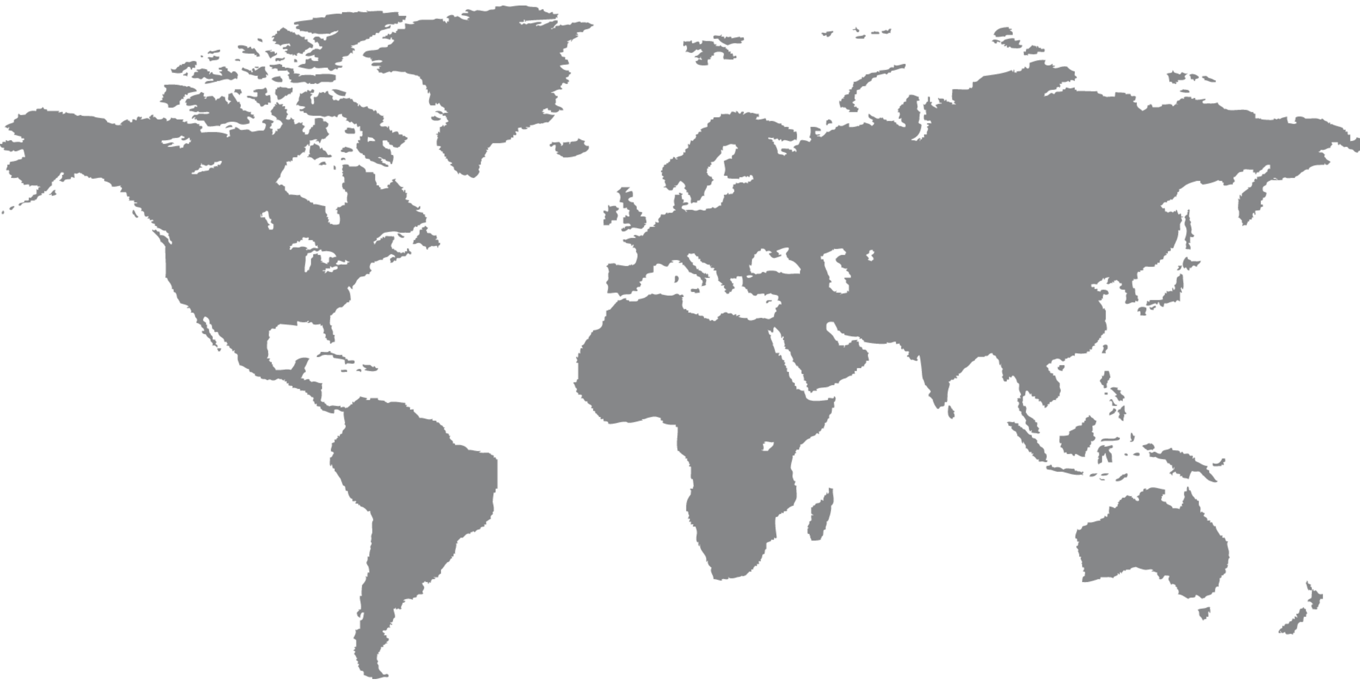
Conclusion

- Data specialization is easy to use by the developer
- Efficiency : around 7 times faster
- Less than 4 calls necessary to pay off code generation
- Extra flexibility on precision

Future works

- Implement an efficient decision algorithm
- Generalize to other operators (e.g. trigonometry)
- Adapt to other platforms

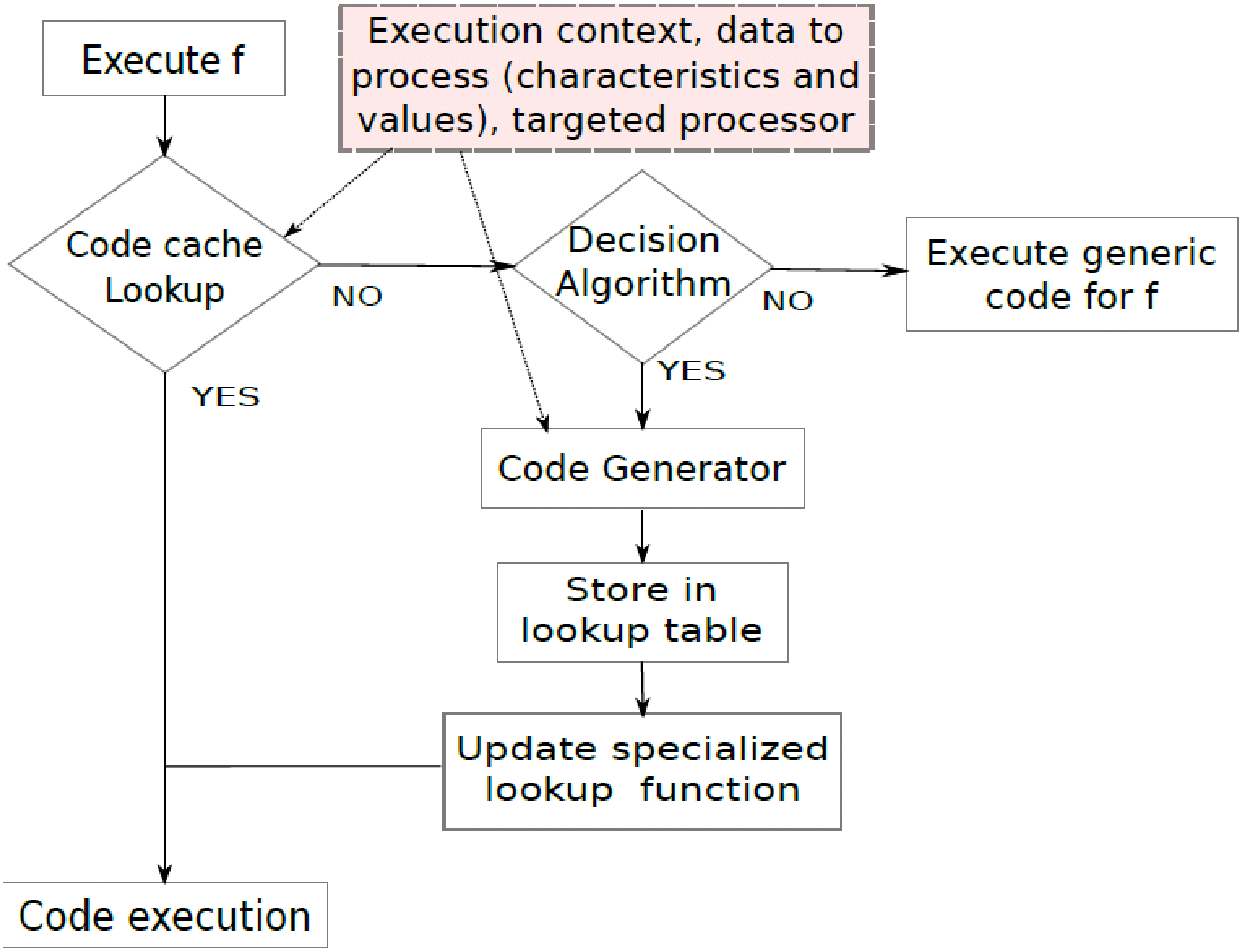
For more questions you can contact :
damien.courousse@cea.fr, henri-pierre.charles@cea.fr



Thank you.

Centre de Grenoble
17 rue des Martyrs
38054 Grenoble Cedex

Centre de Saclay
Nano-Innov PC 172
91191 Gif sur Yvette Cedex


Execute f
Execution context, data to process (characteristics and values), targeted processor
Code cache Lookup
NO
Decision Algorithm
NO
Execute generic code for f
YES
YES
Code Generator
Store in lookup table
Update specialized lookup function
Code execution

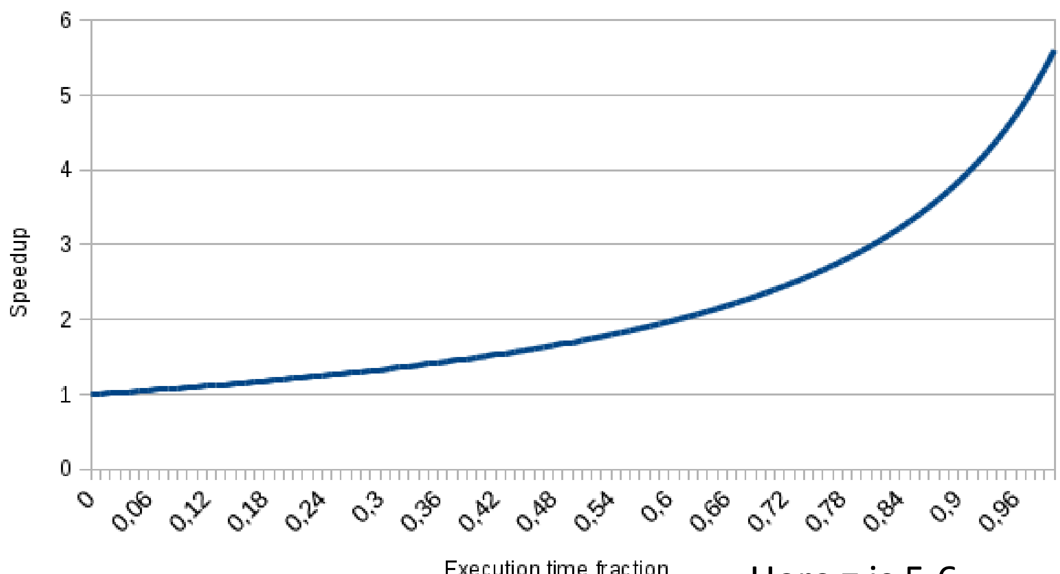
- An application is an overall process

S_{app} : speedup of the overall application

τ : fraction of time initially spent executing the operation to specialize

s : speedup of the specialized function

$$S_{app} = \frac{1}{1 - \tau + \frac{\tau}{s}}$$



Here τ is 5.6

Lookup specialization

Algorithm 1 Generic "lookup" function

```
procedure LOOKUP(id)
    for i = 0; i < NbElem; i++ do
        if id == elem_i then return i
    return −1
procedure MAIN
    ...
    index ← LOOKUP(id)
    if index == −1 then
        f_spec ← GenerateCode(f)
    else
        f_spec ← cache[index]
    res ← f_spec(value)
    ...
```

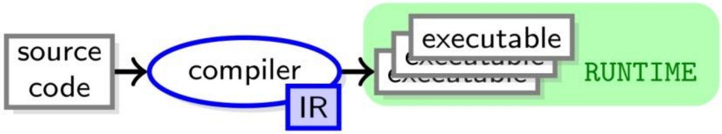
Algorithm 2 Specialised "lookup" function

```
procedure LOOKUP_SPEC(id, value)
    compare(id, elem_0)
    branch @code_spec0
    compare(id, elem_1)
    branch @code_spec1
    ...
    compare(id, elem_X)
    branch @code_specX
    branch @CodeGen
procedure MAIN
    ...
    res ← LOOKUPSPEC(id, value)
    ...
```

- Apply specialization on any runtime data
  - Number of elements in the code cache
  - Loop unrolling
  - Branch directly to the specialized code

Approaches for code specialization

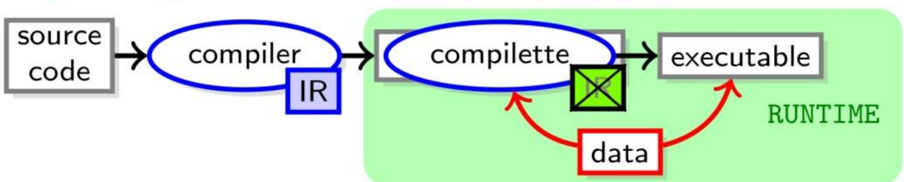
Static code versionning (e.g. C++ Templates)



- static compilation
- runtime: select executable
- memory footprint ++
- limited genericity
- runtime blindness

Runtime code generation, with deGoal

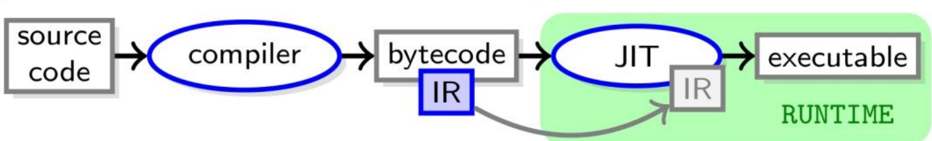
A *compilette* is an ad hoc code generator, targeting one executable



- fast code generation
- memory footprint −−
- **data-driven code generation**

Dynamic compilation

(JITs, e.g. Java Hotspot)



IR Intermediate Representation

- overhead ++
- memory footprint ++
- not designed for data dependant code-optimisations

deGoal supported architectures

ARCHITECTURE	STATUS	FEATURES
ARM32	✓	
ARM Cortex-A, Cortex-M [Thumb-2, VFP, NEON]	✓	SIMD, [IO/OoO]
STxP70 [including FPx] (STHORM / P2012)	✓	SIMD, VLIW (2-way)
K1 (Kalray MPPA)	✓	SIMD, VLIW (5-way)
PTX (Nvidia GPUs)	✓	
MIPS	↺	32-bits
MSP430 (TI microcontroler)	✓	Up to < 1kB RAM
CROSS CODE GENERATION supported (e.g. generate code for STxP70 from an ARM Cortex-A)		

[IO/OoO]: Instruction scheduling for in-order and out-of-order cores

Example of deGoal code

- Simple program example: vector addition

```
void gen_vector_add(void *buffer, int vec_len, int val)
{
#[
    Begin buffer Prelude vec_addr

    Type int_t int 32 #(vec_len)
    Alloc int_t v

    lw v, vec_addr
    add v, v, #(val)
    sw vec_addr, v
]#
}
```

deGoal DSL:
Source to source converted to standard C code

Standard C code

Example of deGoal code

- Simple program example: vector addition

```
void gen_vector_add(void *buffer, int vec_len, int val)
{
#[
    Begin buffer Prelude vec_addr

    Type int_t int 32 #(vec_len)
    Alloc int_t v

    lw v, vec_addr
    add v, v, #(val)
    sw vec_addr, v
]#
}
```

When executed →

Program memory:

```
ldr r1, [r0]
add r1, #1
str r1, [r0]
add r0, #4
ldr r2, [r0]
add r2, #1
str r2, [r0]
add r0, #4
```

Example of deGoal code

- Simple program example: vector addition

```
void gen_vector_add(void *buffer, int vec_len, int val)
{
#[
    Begin buffer Prelude vec_addr

    Type int_t int 32 #(vec_len)
    Alloc int_t v

    lw v, vec_addr
    add v, v, #(val)
    sw vec_addr, v
]#
}
```

Interface: pointer to code buffer and I/O registers (← Begin buffer Prelude vec_addr)

Type definitions and variable allocations (← Type / Alloc)

Instructions (← lw / add / sw)

Example of deGoal code

- Simple program example: vector addition

```
void gen_vector_add(void *buffer, int vec_len, int val)
{
#[
    Begin buffer Prelude vec_addr

    Type int_t int 32 #(vec_len)
    Alloc int_t v

    lw v, vec_addr
    add v, v, #(val)
    sw vec_addr, v
]#
}
```

Determined by the application and fixed in the final machine code

Example of deGoal code

- Simple program example:

```
void gen_vector_add(void *buffer, int vec_len, int val)
{
#[
    Begin buffer Prelude vec_addr

    Type int_t int 32 #(vec_len)
    Alloc int_t v

    lw v, vec_addr
    add v, v, #(val)
    sw vec_addr, v
]#
}
```

Inline run-time constants