Self-optimisation using runtime code generation for Wireless Sensor Networks Internet-of-Things

Caroline Quéva, Damien Couroussé, Henri-Pierre Charles

FROM RESEARCH TO INDUSTRY

cea tech

Self-optimisation using runtime code generation for Wireless Sensor Networks

Internet-of-Things Symposium
ESWeek Amsterdam

Caroline Quéva **Damien Couroussé**
Henri-Pierre Charles

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

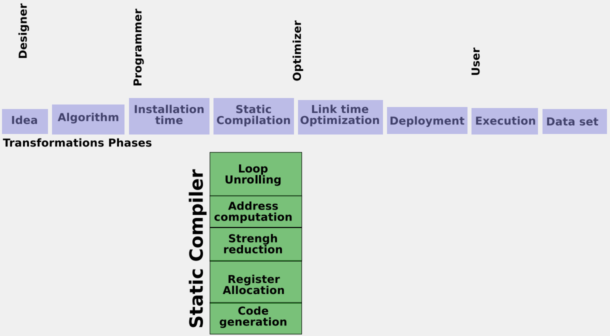
www.cea.fr

leti & list

Classical Compiler architecture : GCC, LLVM, Java JIT

- Driven by performance only
- Mono architecture
- Not energy aware
- Not data dependent

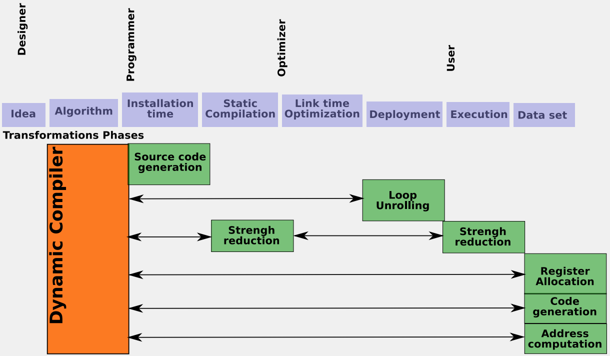
“Compilation time” typology



Future Compilers Architecture

- Multi objective (execution time, power, thermal constraints)
- Multi-target (Heterogeneous multi SoC)
- Data driven (dynamically)

Using "Compilation time"



Definitions

Static compilation "classical" binary code generation (gcc, icc, clang, ...)

Dynamic Compilation binary code generated at run-time (DBT)

JIT run-time dynamic compilation based on complex Intermediate representation (Java, LLVM)

Innovations

Compilette : small binary code generator embedded into application able to optimize code depending on data sets

deGoal : a tool which help to generate *Compilettes*

Supported architectures

Architecture	Stat.	Features
ARM Cortex-A & M, [T2, VFP, NEON]	✓	SIMD, [IO/OoO]
STxP70 (STHORM / P2012)	✓	SIMD, VLIW (2-way)
K1 (Kalray MPPA)	✓	SIMD, VLIW (5-way)
PTX (CUDA Asm language)	✓	
ARM32	✓	
MIPS	↻	32 bits
MPS430 (TI)	✓	Up to < 1kB ram

- Cross code generation supported (generate code for STxP70 from ARM Cortex-A)
- IO / OoO : different insn scheduling for both mode

Writing a compilette with deGoal

- 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
]#
}
```

Source to source converted to standard C code

Standard C code

Writing a compilette with deGoal

- 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
]#
}
```

When executed →

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
```

Writing a compilette with deGoal

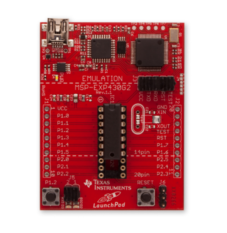
- Simple program example:

```
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

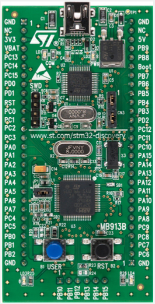
Intro Architectures examples

Example MSP430



- LaunchPad Development Kit
- TI Micro controller
- 16-bit Ultra-Low-Power Microcontroller,
- 1KB Flash
- 128B RAM

Example STM32



- ST Micro microcontroller
- ARM Cortex-M 32 processor

Code specialisation for arithmetic

- Micro-controllers lack dedicated HW support for arithmetic computing
  - No FP unit
  - Sometimes, even no integer multiplier !
- Illustration on floating-point multiplication,
- But applicable to many other operations !

Example : simple float multiplication, compiled for TI's MSP430

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

```
c630: mov r8, r12
c632: mov r9, r13
c634: mov r6, r14
c636: mov r7, r15
c638: call #0xc9d6 ;<__mulsf3>
c63c: mov r14, r12
c63e: mov r15, r13
c640: mov r10, r14
c642: mov r11, r15
```

- `__mulsf3` costs ~**1000 cycles per invocation**

Experiment

Reference

- gcc's generic version of the float multiplication routine
- Precision $p = 24$

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

Our approach

- Runtime specialization of the float multiplication routine
- Precision $p \leq 24$

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

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

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

t_{gen} : execution time of code generation

t_{dyn} : execution time of the generated `fmulM` function

Performance indicators

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

t_{gen} : execution time of code generation

t_{dyn} : execution time of the generated `fmulM` function

Speedup :

$$s = \frac{t_{dyn}}{t_{gcc}} \qquad (1)$$

Overhead recovering :

$$n = \frac{t_{gen}}{t_{gcc} - t_{dyn}} \text{ so that } t_{gen} + n.t_{dyn} \leq n.t_{gcc} \qquad (2)$$

LibMul GeneralAlgo

FP Multiplication algorithm (X, Y)

- Special case handling (denormal, zero, NaN)
- Unpack X, Y
- Mantissa multiplication
- Renormalize
- Repacking
- Rounding

Optimization potential

- Skip unecessary steps
- Value specialization

Optimization principle

`F = mul(M, X)`

ALGORITHM 1: Floating-Point Multiplication with Horner scheme

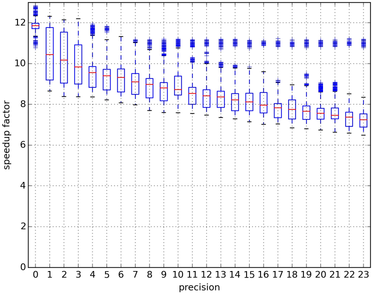
Input: Floating-point operands M and X to be multiplied (M is known, X is unknown).

Output: The result $M \times X$ of the multiplication.

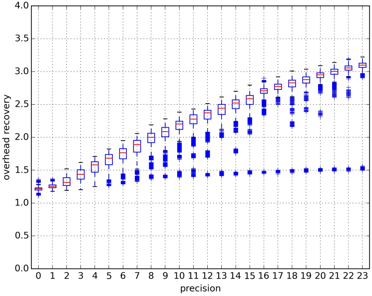
$i \leftarrow 0$;
$detection \leftarrow 1$;
$result \leftarrow X$;
while ***not*** *(mantissa(M) && (1 ≪ i))* **do**
    $i \leftarrow i + 1$;
end
for $i \leftarrow i + 1$ **to** *len(mantissa(M))* **do**
    if *mantissa(M) && (1 ≪ i)* **then**
        $result \leftarrow (result \ll detection) + X$;
        $detection \leftarrow 1$;
    end
    else
        $detection \leftarrow detection + 1$;
    end
end
$result \leftarrow (result \ll detection)$;
return *result*

Results

Speedup



Overhead recovering



Conclusion for multiplication

- Demonstration that runtime code generation is a realistic goal
  - on a 16-bit micro-controller
  - with only 512 bytes of RAM
- Efficiency : 10$\times$ faster floating-point multiplication (when one of the two operands is a runtime constant)
- Greater genericity : extra flexibility on precision
- 50% increase performance on an IIR application (7 coeff.)
- Further work : automatic runtime code generation embedded in `libm`

Future works

In IoT

- Generalize the arithmetic library
  - Other operators (trigonometry, . . .)
  - Network stacks

Other domains

- High Performance Computing
- Power Consumption driven
- Code polymorphism
  - Online optimization
  - Cryptography