



HAL
open science

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

Caroline Quéva, Damien Couroussé, Henri-Pierre Charles

► **To cite this version:**

Caroline Quéva, Damien Couroussé, Henri-Pierre Charles. Self-optimisation using runtime code generation for Wireless Sensor Networks Internet-of-Things. Internet-of-Things Symposium at ESWeek, Marilyn Wolf (Georgia Tech) Jason Xue (City University of Hong Kong), Oct 2015, Amsterdam, Netherlands. cea-01240865

HAL Id: cea-01240865

<https://cea.hal.science/cea-01240865>

Submitted on 9 Dec 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Self-optimisation using runtime code generation for Wireless Sensor Networks

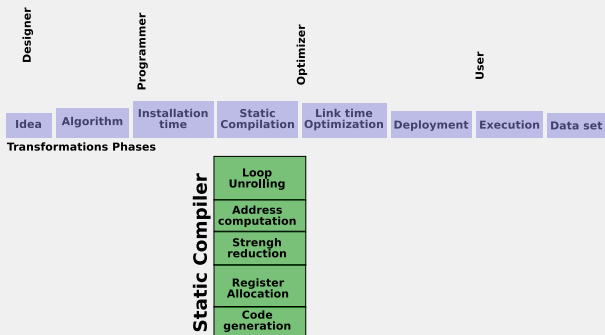
Internet-of-Things Symposium
ESWeek Amsterdam

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

Classical Compiler architecture : GCC, LLVM, Java JIT

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

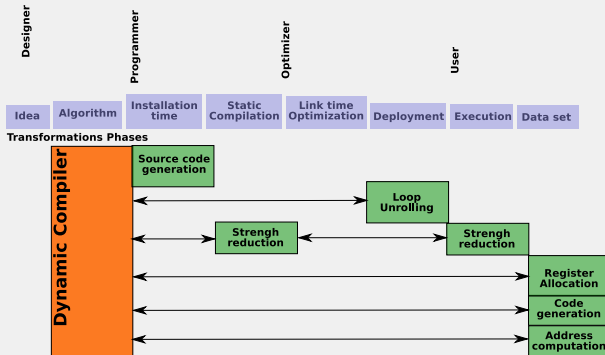
“Compilation time” typology



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

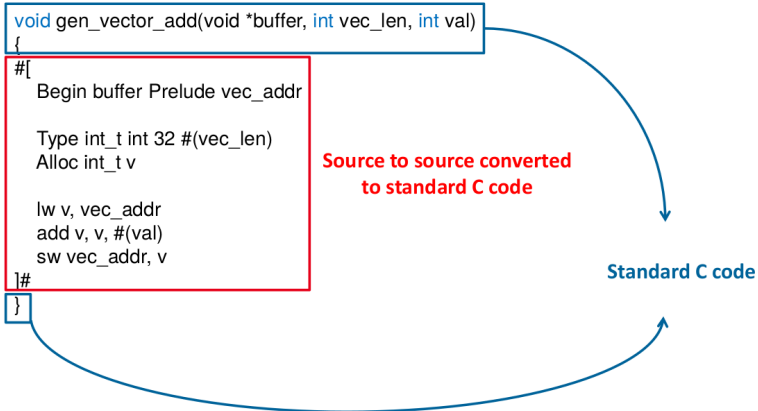
Complette : small binary code generator embedded into application able
to optimize code depending on data sets

deGoal : a tool which help to generate *Complettes*

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

■ Simple program example:



■ 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
```



■ Simple program example:

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

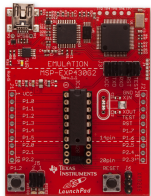
    Type int_t int 32 #(vec_len)
    Alloc int_t v

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

Inline run-time
constants

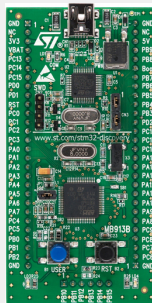


Example MSP430



- 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

- 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

```
float fmul (float a, float b) {
    return (a*b);
}
1  c630: mov r8, r12
2  c632: mov r9, r13
3  c634: mov r6, r14
4  c636: mov r7, r15
5  c638: call #0xc9d6 ;<__mulsf3>
6  c63c: mov r14, r12
7  c63e: mov r15, r13
8  c640: mov r10, r14
9  c642: mov r11, r15
```

- `__mulsf3` costs ~ 1000 cycles per invocation

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 $p \leq 24$

```
1 /* tgen: code generation */  
2 float (*) (float) fmulM;  
3 fmulM = generate_fmuls_code(M, p);  
4  
5 /* tdyn: run the generated routine  
6 float fmul (float X) {  
7     return fmulM(X);  
8 }
```

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

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 :

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

Overhead recovering :

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

FP Multiplication algorithm (X, Y)

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

Optimization potential

- Skip unnecessary steps
- Value specialization

$$F = \text{mul}(M, X)$$

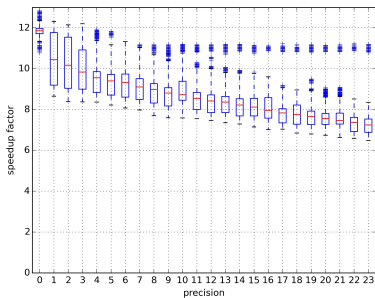
ALGORITHM 1: Floating-Point Multiplication with Horner scheme

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

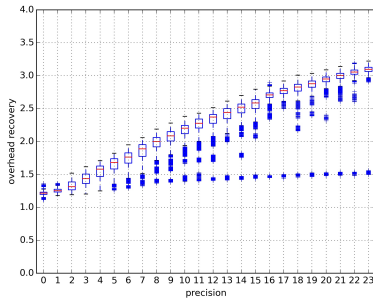
Output: The result $M \times X$ of the multiplication.

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

Speedup



Overhead recovering



- 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`

In IoT

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

Other domains

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