Runtime Code Generation for Performance and Security in Embedded Systems

Damien Couroussé
2015-10-16
Univ. Grenoble Alpes, F-38000 Grenoble, France
CEA-LIST, MINATEC Campus, F-38054 Grenoble, France

Henri-Pierre Charles, Thierno Barry, Fernando Endo, Bruno Robisson, Hassan Noura, Thibault Cattelani, Caroline Queva, Charles Aracil...
COGITO people : Jean-Louis Lanet, Bruno Robisson, Thierno Barry, Philippe Jaillon, Olivier Potin, Hélène Le Bouder...
Runtime Code Generation: Motivation

Pitch: some code optimisations are not accessible to static compilers
- Unknown data
- Sometimes, the hardware is also unknown, at least partially

- Delay code optimisations at runtime
  - Constant propagation, elimination of dead code,
  - Strength reduction,
  - Loop unrolling,
  - Instruction scheduling,
  - etc.

- Drive code performance by runtime-changing constraints
  - Bounds: power / energy / execution time
  - Heterogeneous cores: accelerators, specialised instructions

- Runtime code generation for unusual purposes (e.g. security)?
  (runtime) code specialisation
  (runtime) code polymorphism
Tools for runtime code generation in embedded systems

- ... for performance
- ... for security
Approaches for code specialisation

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

- Static compilation
- Runtime: select executable
- Memory footprint ++
- Limited genericity
- Runtime blindness

**Runtime code generation**, with deGoal
A *compillette* 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)

- Overhead ++
- Memory footprint ++
- Not designed for data dependant code-optimisations
Code generation flow

DESIGN TIME → STATIC COMPILATION TIME → RUN TIME (data adaptation)

- .cdg
- .cdg.c
- static binary
- runtime binary
- .c
- compilette
- HW desc.
- data
- kernel
Simple program example: vector addition

```c
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
Simple program example: vector addition

```c
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

    #}
}
```

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
```
Simple program example: vector addition

```c
void gen_vector_add(void *buffer, int vec_len, int val)
{
    // Interface: pointer to code buffer and I/O registers
    Begin buffer Prelude vec_addr

    // Type definitions and variable allocations
    Type int_t int 32 #(vec_len)
    Alloc int_t v

    // Instructions
    lw v, vec_addr
    add v, v, #(val)
    sw vec_addr, v

}#
}
```
Simple program example: vector addition

```c
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

  ]#
}
```
Simple program example:

```c
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
}
```
Simple program example: vector addition

```c
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

```
add v, v, #(val)
```

```
ADD R0, #4
ADD R0, #90
```

```
ADD_T2 R0, R0 #1024
```
Example of deGoal code

Simple program example: vector addition

```c
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
    ]#

}
```

```
1
Type int_t int 32 #(vec_len)
2
Alloc int_t v
4
 Inline run-time constants
```

```
Run #1
Run #2
Run #N
```

```
ADD R0, #4
Scalar/SISD
ADD R0, #4
ADD R1, #4
```

```
VADD Q0, #4
Vector/SIMD (if enabled)
```
Simple program example: vector addition

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

    Type int_t int 32 1
    Alloc int_t r

    int i;
    for (i = 0; i < vec_len; ++i) {
        #[ lw r, vec_addr
        add r, r, #(val)
        sw vec_addr, r
        add vec_addr, #(sizeof(int))
    }
}
```

Loop unrolling: “Copy-paste” a block of instructions
**Features**

**deGoal**
- Portable DSL
- Complex variables (registers)
  - Typed
  - Vector support, dynamically sized
- Mix runtime data & binary code
- Easily extended with domain-specific or hardware-specific instructions (e.g. multimedia)

**Results**
- Auto-adaptative dynamic libraries
- Runtime portable optimization
- Multiple performance metrics:
  - Faster generated code
  - Smaller generated code
  - Code generation 3 order of magnitude faster than JIT/LLVM
  - Code generators 4 orders of magnitude smaller than JITs/LLVM
## deGoal supported architectures

<table>
<thead>
<tr>
<th>ARCHITECTURE</th>
<th>STATUS</th>
<th>FEATURES</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM32</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>ARM Cortex-A, Cortex-M [Thumb-2, VFP, NEON]</td>
<td>✓</td>
<td>SIMD, [IO/OoO]</td>
</tr>
<tr>
<td>STxP70 [including FPx] (STHORM / P2012)</td>
<td>✓</td>
<td>SIMD, VLIW (2-way)</td>
</tr>
<tr>
<td>K1 (Kalray MPPA)</td>
<td>✓</td>
<td>SIMD, VLIW (5-way)</td>
</tr>
<tr>
<td>PTX (Nvidia GPUs)</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>MIPS</td>
<td>🔄</td>
<td>32-bits</td>
</tr>
<tr>
<td>MSP430 (TI microcontroller)</td>
<td>✓</td>
<td>Up to &lt; 1kB RAM</td>
</tr>
</tbody>
</table>

**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
Tools for runtime code generation in embedded systems

- for performance
  - Application use cases
    - Video compression (Itanium)
    - Linear algebra on GPUs (Nvidia)
    - Arithmetic computing on Micro-controllers for the IoT (ARM Cortex-M, TI MSP430)
    - Auto-tuning for embedded applications (ARM Cortex-M + VFP + NEON)
    - Memory allocation in MPSoCs
    - ...

- for security
Floating-point multiplication

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

**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.

```plaintext
i \leftarrow 0;
detection \leftarrow 1;
result \leftarrow X;
while not (mantissa(M) && (1 \ll i)) do
    i \leftarrow i + 1;
end
for i \leftarrow i + 1 to \text{len}(mantissa(M)) do
    if 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
```
Arithmetic computation on micro-controller

floating-point multiplication

\[ \text{mulM} = \text{gen(M)} \rightarrow F = \text{mulM}(X) \]

**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.

\[ i \leftarrow 0; \]
\[ \text{detection} \leftarrow 1; \]
\[ \text{result} \leftarrow X; \]

while not (mantissa(\( M \)) \&\& (1 \ll i)) do
  \[ i \leftarrow i + 1; \]
end

for \( i \leftarrow i + 1 \) to \( \text{len(mantissa(\( M \)))} \) do
  if mantissa(\( M \)) \&\& (1 \ll i) then
    \[ \text{result} \leftarrow (\text{result} \ll \text{detection}) + X; \]
    \[ \text{detection} \leftarrow 1; \]
  else
    \[ \text{detection} \leftarrow \text{detection} + 1; \]
  end
end

\[ \text{return result} \]
Arithmetic computation on micro-controller

- Target platform: TI MSP430
  → 512 bytes of RAM!! :)
- Specialisation
  - one operand of the multiplication
  - bit precision (width of the mantissa)
- Speedup over 10x
- Overhead recovery < 4
- Genericity: variable precision

Runtime code specialisation of floating-point arithmetics
Tools for runtime code generation in embedded systems

- for performance
- for security

Runtime code generation as a Software Protection for Embedded Systems against Physical Attacks
An attack is usually split between:

1. a first step attack:
   - global inspection of the target
   - identification of the security components involved (HW/SW)
   - identification of weaknesses

2. a second step attack:
   - focused attack
   - on an identified potential weakness
A coarse typology of physical attacks

- **Reverse engineering**
  - HW inspection: decapsulation, abrasion, chemical etching, memory extraction, etc.
  - SW inspection: debug, memory dumps, code analysis, etc.

- **Side channel attacks**: SPA (Simple Power Analysis), DPA (Differential –), CPA (Correlation –).
  - Electromagnetic analysis
  - Power analysis
  - Acoustic analysis
  - Timing attacks

- **Fault injection attacks**
  - under/over voltage drops
  - ion / laser beam, optical illumination
  - glitch attacks
  - ...

> spatial and temporal sensibility
Code polymorphism

Definition

- Regularly changing the behavior of a (secured) component, at runtime, while maintaining unchanged its functional properties, with runtime code generation

What for?

- Protection against reverse engineering of SW
  - the secured code is not available before runtime
  - the secured code regularly changes its form (code generation interval $\omega$)
- Protection against physical attacks
  - polymorphism changes the spatial and temporal properties of the secured code: side channel & fault attacks
  - combine with usual SW protections against focused attacks
- Compatible with State-of-the-Art HW & SW Countermeasures

How?

- deGoal: runtime code generation for embedded systems
  - fast code generation
  - tiny memory footprint: proof of concept on TI's MSP430 (512 B RAM)
Polymorphic code generation: overview

clear text → cipher program → cipher key → ciphered message

alea
Polymorphic code generation: overview

Diagram:
- Clear text
- Alea
- Cipher key
- Cipher program
- Ciphered message
- Polymorphic code generator
- Alea
- Polymorphic cipher instance
- Ciphered message
Opportunities for polymorphic code generation

- Random register allocation
- Random instruction selection
- Instruction shuffling
- Insertion of noise instructions
Application to AES: AddRoundKey()

```c
void addRoundKey_complilette( cdg_insn_t* code
 , uint8_t* key_addr, uint8_t *state_addr)
{
    #[Begin code Prelude

    Type reg32 int 32
    Alloc reg32 state, key, i

    mv i, #(16)
    loop:
        sub i, i, #(1)
        lb state, @(#(state_addr) + i) // state = state_addr[i]
        lb key, @(#(key_addr) + i) // key= key_addr[i]
        xor state, key
        sb @(#(state_addr) + i), state
        bneq loop, i, #(0)
    rtn
    End
    ]#;
}
```
Random register allocation

- Greedy algorithm: each register is allocated among one of the free registers remaining

- Has an impact on:
  - The management of the context (ABI)
  - Instruction selection
**Instruction selection**

- Replace an instruction by a semantically equivalent sequence of one or several instructions
- Select the sequence in a list of equivalences
- Examples:

\[
\begin{align*}
    c & := a \ xor \ b \iff c := ((a \ xor \ r) \ xor \ b) \ xor \ r \\
    c & := a \ xor \ b \iff c := (a \ or \ b) \ xor \ (a \ and \ b) \\
    c & := a - b \iff k := 1 \ ; \ c := (a + k) + (\text{not} \ b) \\
    c & := a - b \iff c := ((a + r) - b) - r
\end{align*}
\]
Instruction shuffling

- Reorder instructions, but do not break the semantics of the code!
  - Defs
  - Uses
  - *Do not* take into account result latency and issue latency
  - Special treatments for... special instructions. E.g. branch instructions
Insertion of noise

- Insertion of fake instructions, that have no effect on the result of the program.
- Controlled by a probability of insertion $p$
- Can insert any instruction:
  - Arithmetic (add, xor...)
  - Memory accesses (lw, lb,...)
  - Power hungry ones (mul, mac...)
  - nop
Illustration on AES

8-bit AES
STM32 (Cortex-M3)
Reference implementation
Experimental validation: CPA on EM traces

Reference implementation

Polymorphic version, code generation interval: 500

Distinguish threshold = 39 traces
Key byte 10

Distinguish threshold = 89 traces
Key byte 02
Experimental validation: CPA on EM traces

Polymorphic version
code generation interval: 20

Polymorphic version,
code generation interval: 500

Distinguish threshold > 10000 traces
Key byte 02

Distinguish threshold = 89 traces
Key byte 02
8 bit AES

Polymorphic code generation:
- AddRoundKey()
- SubBytes()

All overheads included

Execution times (in cycles), over 1000 runs:

<table>
<thead>
<tr>
<th></th>
<th>min</th>
<th>max</th>
<th>average</th>
</tr>
</thead>
<tbody>
<tr>
<td>reference</td>
<td>6385</td>
<td>6385</td>
<td>6385</td>
</tr>
<tr>
<td>code generator</td>
<td>5671</td>
<td>12910</td>
<td>9345</td>
</tr>
<tr>
<td>polymorphic instance</td>
<td>7185</td>
<td>9745</td>
<td>8303</td>
</tr>
</tbody>
</table>

Impact of the code generation interval $\omega$:

<table>
<thead>
<tr>
<th>$\omega$</th>
<th>$k$</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.76</td>
<td>53.0%</td>
</tr>
<tr>
<td>5</td>
<td>1.59</td>
<td>18.4%</td>
</tr>
<tr>
<td>20</td>
<td>1.37</td>
<td>2.1%</td>
</tr>
<tr>
<td>100</td>
<td>1.31</td>
<td>1.1%</td>
</tr>
</tbody>
</table>

$k$: overhead vs. reference implementation
%: percentage contribution of runtime code generation to the performance overhead
Polymorphic code generation library (64 variants)
- Average: 22395.5 bytes
- Min: 19632 bytes
- Max: 25208 bytes

Binary Image: size increase vs the reference implementation
- Reference: 73279 bytes
- Polymorphic version, max size: 76543 bytes. Max difference: 3264 bytes

Size of the polymorphic instance
- $\sim k$ if $\omega \to \infty$
- Roughly proportional to the overhead incurred by the execution of the polymorphic instance
- deGoal: build runtime code generators (a.k.a compilettes)
  - Code specialisation on runtime data values
  - Code specialisation on characteristics of the hardware
  - Applicable to embedded systems constrained wrt computing power and memory resources

- Polymorphic runtime code generation
  - Generic software countermeasure
    - The generated code can exploit hardware characteristics available
    - Compatible with state-of-the-art software countermeasures
  - Variation of the observation in time and space
  - Experimental validation on AES, side channel: the security is increased by a factor of 2000

Thanks!