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

Design Time

STATIC COMPILATION TIME

RUN TIME (data adaptation)

- .cdg
- .cdg.c
- static binary
- runtime binary

- deGoal
- platform compiler
- compilette
- compilette
- HW desc.
data
Standard code

```c
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

```c
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**

Diagram showing the process from design to runtime with the following steps:
- **.cdg**
- **.cdg.c**
- Compiled binary
- Runtime binary

**Design Time**
- **.cdg**
- **.c**

**Static Compilation Time**
- **.cdg.c**
- **.c**
- Compiled binary

**Runtime (Data Adaptation)**
- **Runtime binary**
  - **compilette**
  - **Kernel**
  - **HW desc.**
  - **Data**

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`

**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

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

⇒ High memory footprint
⇒ High overhead
⇒ Low memory footprint
⇒ Specialization done by the developer
⇒ Low memory footprint

⇒ Specialization done by the developer
⇒ 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

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

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

\( t_{\text{gcc}} \) : execution time of gcc’s multiplication routine
\( t_{\text{gen}} \) : execution time of code generation
\( t_{\text{dyn}} \) : execution time of the generated function

- **Speedup**:
  \[ s = \frac{t_{\text{dyn}}}{t_{\text{gcc}}} \]

- **Overhead recovery**:
  \[ N = \frac{t_{\text{gen}}}{t_{\text{gcc}} - t_{\text{dyn}}} \]
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
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.courrousse@cea.fr, henri-pierre.charles@cea.fr
Thank you.
Process

1. **Execute f**
2. **Code cache Lookup**
   - **YES**
     - **Code execution**
   - **NO**
     - **Decision Algorithm**
       - **NO**
         - Execute generic code for f
       - **YES**
         - **Code Generator**
           - **Store in lookup table**
           - **Update specialized lookup function**

*Execution context, data to process (characteristics and values), targeted processor*
Generalisation

An application is an overall process

\[ S_{\text{app}} : \text{speedup of the overall application} \]
\[ \tau : \text{fraction of time initially spent executing the operation to specialize} \]
\[ s : \text{speedup of the specialized function} \]

\[ S_{\text{app}} = \frac{1}{1 - \tau + \frac{\tau}{s}} \]

Here \( \tau \) is 5.6
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)

- **Source code** → **Compiler** → **Executable** → **Runtime**

**Runtime code generation**, with deGoal

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

- **Source code** → **Compiler** → **Compilette** → **Executable** → **Runtime**

**Dynamic compilation**

(JITs, e.g. Java Hotspot)

- **Source code** → **Compiler** → **Bytecode** → **JIT** → **Executable** → **Runtime**

**Intermediate Representation**

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

- **Fast code generation**
- **Memory footprint**
- **Data-driven code generation**

- **Overhead**
- **Memory footprint**
- **Not designed for data dependant code-optimisations**
# 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
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
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
    ]#
}
```

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)
{

#[
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
- **Type definitions and variable allocations**
- **Instructions**
Example of deGoal code

Simple program example: vector addition

```c
void gen_vector_add(void *buffer, int vec_len, int val)
{
    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:

```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
    ]#
}
```