Is dynamic compilation possible for embedded system?

Scopes 2015, St Goar

Victor Lomüller, Henri-Pierre Charles

CEA DACLE / Grenoble

June 2 2015
FAQ

What do you mean by dynamic compilation? : a compilation system where the binary code is generated at run-time.

Like a Java JIT system? Yes exactly! But not only for portability: for performances & data / architecture adaptation.

But Java JIT can not be used on ES! Yes! Because it need a huge memory size, introduce lag, is slow in the interpreted part, takes time to generate binary code, ... because it’s java ;-)

What do you want?

We want to have flexible and fast code generation at run-time. Use it to specialize (simplify) binary code at run-time. There is many application domains!
Classical Compiler architecture: GCC, LLVM, Java JIT

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

"Compilation time" typology

Is dynamic compilation possible for embedded system? | DACLE Division | June 2 2015
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*

**Kahuna**  : an LLVM transformation to implement simple *Compilettes*
deGoal: Data Dependent Code Generation

Compilette
- Code generation at run-time embedded into application
- Data dependant
- Architecture independant (mostly)

Compilation chain

- .cdg
  - C source
  - degoal high-level ASM
- .C
- static binary
  - compilette
- runtime binary
  - compilette
  - kernel

RUN TIME
(data adaptation)

REWRITE TIME
(source to source)

STATIC Compilation TIME

HW desc.
data
### deGoal Features

- Portable “assembly language”
- Source to source compiler
- Registers
  - Typed: `int`, `float`, `complex`, ...
  - Vector support: dynamic size
- Mix runtime data & binary code
- Correct use of any multimedia instruction

### Obtained results

- Auto adaptive dynamic libraries
- Runtime Portable Optimization
- Multiple metrics:
  - Faster generated code
  - Smaller generated code
  - 3 order of magnitude faster than JIT/LLVM
  - 4 order of magnitude smaller than JIT/LLVM

---

**Is dynamic compilation possible for embedded system?**

DACLE Division | June 2 2015 | 7
## deGoal support

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Port status</th>
<th>SIMD support</th>
<th>Instruction bundling</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM Thumb-2 (+NEON/VFP)</td>
<td>✓</td>
<td>✓</td>
<td>OoO/InO</td>
</tr>
<tr>
<td>STxP70 (STHORM / P2012)</td>
<td>✓</td>
<td>N/A</td>
<td>✓</td>
</tr>
<tr>
<td>K1 (Kalray MPPA)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>PTX (GPU NVIDIA)</td>
<td>✓</td>
<td>✓</td>
<td>N/A</td>
</tr>
<tr>
<td>ARM32</td>
<td>✓</td>
<td>X</td>
<td>N/A</td>
</tr>
<tr>
<td>MSP430</td>
<td>✓</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>MIPS</td>
<td>✓</td>
<td>X</td>
<td>N/A</td>
</tr>
<tr>
<td>ARM64</td>
<td>✓</td>
<td>X</td>
<td>N/A</td>
</tr>
</tbody>
</table>
deGoal Example : Simple multiplication

Runtime "constant" multiplication code generator

```c
#include <stdio.h> /* -** c -** */

typedef int (*pifi)(int);

/* Compilette which add a constant value */
pifi multiplyCompile(int multiplyValue)
{
    cdgInsnT *code= CDGALLOC(1024);
    printf("Code\n" generation\n" for\n" multiply\n" value\n" %d\n" code\n" at\n" %p\n", multiplyValue);

    VectorType ScalarInt float 32 1
    RegAlloc ScalarInt in 1

    Begin code Prelude in0

    mul in0, in0, #(multiplyValue)
    rtn
    End
}
return (pifi)code;
}
```

Is dynamic compilation possible for embedded system ? | DACLE Division | June 2 2015 | 9
Kahuna : High Level Idea

Kahuna General idea

- Identify “key” variable (which could be constant during a period of time)
- Generate a template & a specializer
- Specialize on the fly, only the needed instructions
- Based on LLVM

High level idea
**Reuse the same template everytime**

+ : don’t have to generate the whole kernel
+ : no compilette to write
- : need to specialize for every change
- : only instructions “compiler aware”

**Illustration**

- Specializer
- Template
- Data
Reuse multiple template

- : need to generate the whole kernel (copy template + specialize)
+ : no compilette to write
+ : no need to specialize for every change (use cache)
- : only instructions “compiler aware”

Illustration
Idea : Kahuna Recipe

Algorithm
- Start with LLVM IR (language independant)
- Treat annotated variable as “constant”
- Add label for instructions to post-modify
- Generate code for binary specialization
- Compile “as usual”

Implementation
- Based on LLVM 3.2
- Implement annotation handling
- Implement Kahuna process
- Modified backend

Graph:
- Annotated LLVM IR
  - Binding-Time Analysis
    - Dynamic Instructions
      - Instructions with holes
      - Templates
      - Fragments
    - Constant Instructions
    - Code Gen. Fragments
      - Insn Info
      - 11c
    - Code Gen. (LLVA)
      - Code Specializer (LLVA)
      - 11c
      - Binary templates
      - Code Specializer (object)

Is dynamic compilation possible for embedded system?

DACLE Division | June 2 2015 | 13
Results: Benchmark & Experiment

Code using “constants”:

- Pass Band Audio filter (Extracted from SOX)
- Finite impulse response filter
- 2D Convolution

Modified version of LLVM

- Standard static LLVM compilation
- Kahuna
- LLVM + static specialization (inlining)
- deGoal

Architecture

- Speed & memory: 800 MHz Cortex-A8 ARM processor (Beagleboard-xM platform).
  - in-order,
  - dual-issue
  - FPU (theoretical peak at 80 MFlops)
- Energy: modified GEM5 and McPAT ARMv7-A simulation environment

Is dynamic compilation possible for embedded system? | DACLE Division | June 2 2015 | 14
# Results Speed

## Speedup over the static version

<table>
<thead>
<tr>
<th>Application</th>
<th>sox Speedup (%)</th>
<th>FIR Speedup (%)</th>
<th>Convolution Speedup (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLVM</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SpeLLVM</td>
<td>21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>deGoal</td>
<td>27</td>
<td></td>
<td></td>
</tr>
<tr>
<td>kahuna</td>
<td>21</td>
<td>10</td>
<td>48</td>
</tr>
</tbody>
</table>

- SpeLLVM = code with data specialization made “by hand”;
- deGoal = code generator made by hand
- Convolution unrolled for kahuna

## First metric: code speed

- First step: Does it work: yes!
- Code speed as fast as compiler production grade
## Results: Code Generation Speed

### Code generation speed

<table>
<thead>
<tr>
<th>Application</th>
<th>sox Cycles</th>
<th>sox Cycles per Insn.</th>
<th>FIR Cycles</th>
<th>FIR Cycles per Insn.</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLVM</td>
<td>126 M</td>
<td>3 M</td>
<td>223 M</td>
<td>3 M</td>
</tr>
<tr>
<td>SpeLLVM</td>
<td>111 M</td>
<td>3 M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>deGoal</td>
<td>10 753</td>
<td>233</td>
<td></td>
<td></td>
</tr>
<tr>
<td>kahuna</td>
<td>205</td>
<td>20</td>
<td>76</td>
<td>8</td>
</tr>
</tbody>
</table>

(Code generation timing, in cycles took to generate the kernels and cycles took to generate one instruction.)
Results Code Generation Speed

Code Generation result

- LLVM O3: 3M cycles
- LLVM & Spec O3: 3M cycles
- deGoal: 10K cycles
- Kahuna: 233 cycles
- LLTVM O3: 126M cycles
- LLTVM & Spec O3: 111M cycles

Is dynamic compilation possible for embedded system?
Results

- Victor Lomüller PhD thesis and articles
- Tools for new metrics in code generation
- Open the door for new code specialization

Open PhD position

- Java JIT compiler for embedded systems

References

