HybroGen : A compiler for innovative interleaved execution and compilation scenarios

New architectures needs interleaved compilation / execution scenarios

Henri-Pierre CHARLES, Maha KOOLI, Thaddée BRICOUT and Benjamin LACOUR

CEA DSCIN department / Grenoble

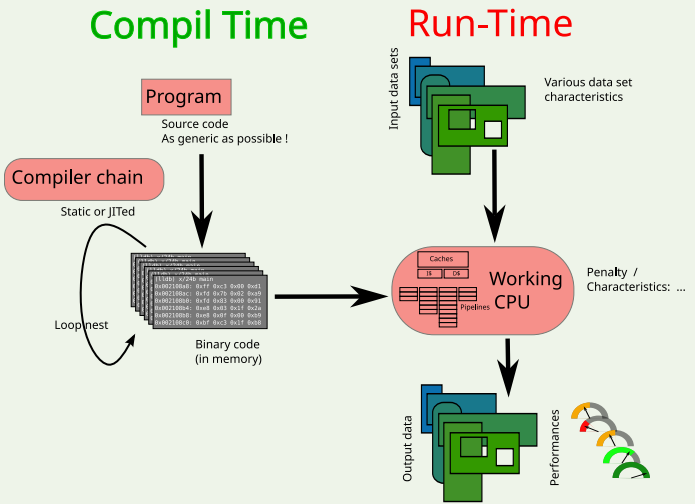
September 2023

Motivation : Static Compiler and notion of "*Compilette*"

Static Compiler

- Run once
- Does not know data set characteristics
- Slow compilation (even with JIT)

Static compilation



Motivation : Static Compiler and notion of "*Compilette*"

Static Compiler

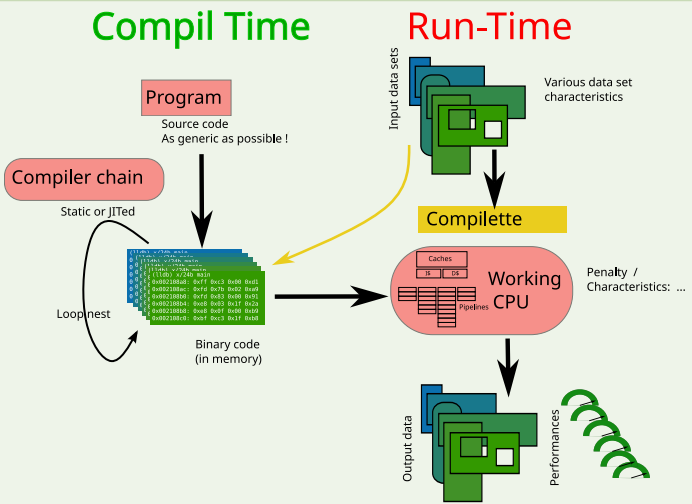
- Run once
- Does not know data set characteristics
- Slow compilation (even with JIT)

Compilette improve performance

- Adapt on the fly
- Knowledge of the architecture
- Knowledge of the application
- Knowledge of the dataset

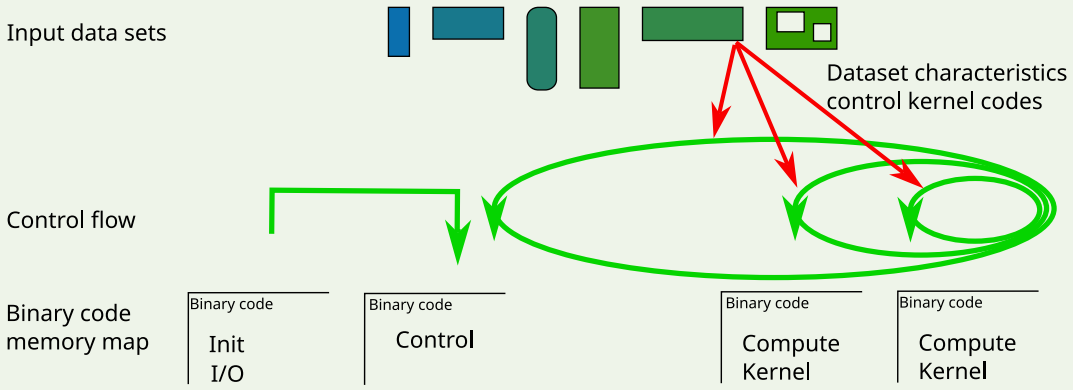
How to implement / program that ?

Static compilation with Dynamic adaptation



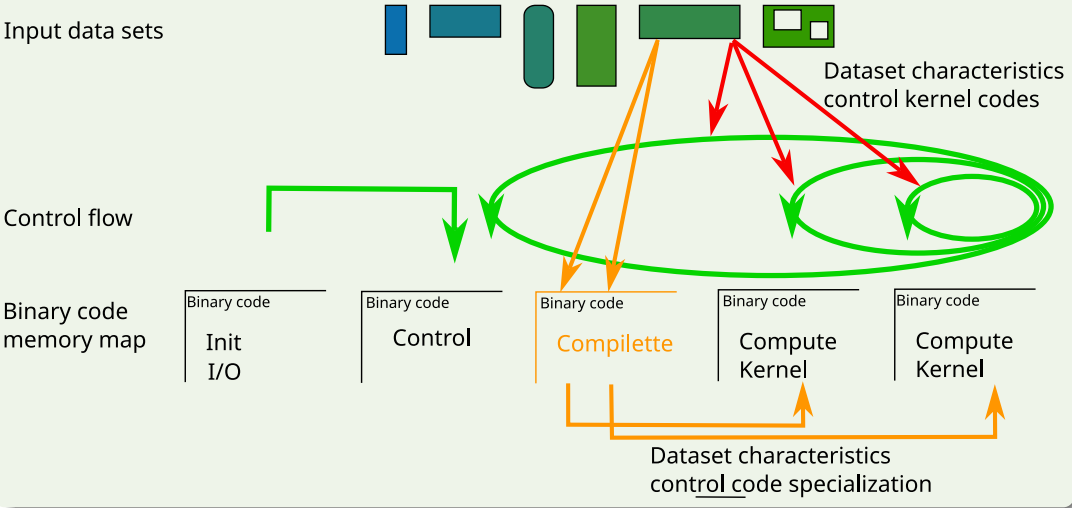
Compilette principle : “Working Example”

Control flow in application



Compilette principle : “Working Example”

Control flow in application with dynamic adaptation



List of Code Generation Scenarios

Compilation scenarios

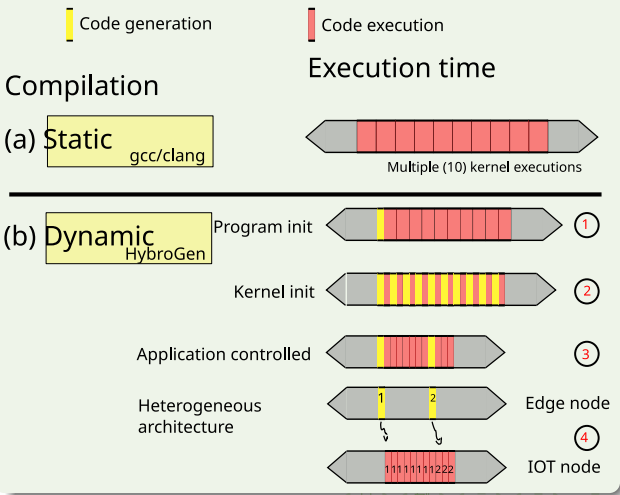
- (a) Static compilation
- (b) Dynamic adaptation
  1. Program initalization
  2. Kernel initialization
  3. Application controlled
  4. Heterogeneous architecture (multi-isa support)

Target list

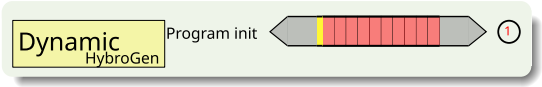
- RISCV (Embedded system)
- CSRAM (Embedded system)
- POWER 8 (HPC computer)
- AARCH64 (both)
- Others (both)

All following scenarios examples works on all platforms

Illustration



Scenario #1 : Application Initialization : Celcius to Farenheit



Generate code at initialization & reuse

- Binary code generated at initialization time
- Demonstrator based on Celcius to Farenheit conversion

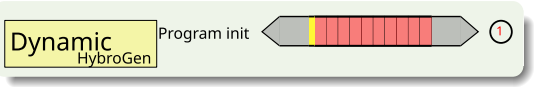
Optimization source

- Optimization based on initialization time knownledge
  - Input parameters
  - Data set characteristics

AARCH64 : gcc -O3 version

```
0000000000402d34 <CelciusToFarenheit>:
 402d34: add w0, w0, w0, lsl #3
 402d38: mov w1, #0x6667    // #26215
 402d3c: movk w1, #0x6666, lsl #16
 402d40: smull x1, w0, w1
 402d44: asr x1, x1, #33
 402d48: sub w0, w1, w0, asr #31
 402d4c: add w0, w0, #0x20
 402d50: ret
```

Scenario #1 : Application Initialization : Celcius to Farenheit



Generate code at initialization & reuse

- Binary code generated at initialization time
- Demonstrator based on Celcius to Farenheit conversion

Optimization source

- Optimization based on initialization time knownledge
  - Input parameters
  - Data set characteristics

AARCH64 : Compilette version

```
(gdb) x/20i fC2F
0x4162a0: mov w10, #0x9
0x4162a4: mul w10, w0, w10
0x4162a8: mov w11, #0x5
0x4162ac: sdiv w11, w10, w11
0x4162b0: add x0, x11, #0x20
0x4162b4: ret
```

Scenario #2 : Kernel Initialization : array scaling

Value injection / dynamic polymorphism

```
#[  int 32 1 mult (int[] 32 1 a,
                   int   32 1 len)  {
    int 32 1 r, i;
    for (i = 0; i < len; i = i + 1) {
        // b value will be included
        //in code generation
        a[i] = a[i] * #(b);
      }
  }  ]#
```

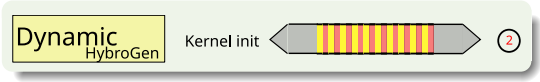
Optimization source

- Optimization based on kernel input parameters

POWER Binary codes specialization on 42

```
(gdb) x/20i fPtr
   0x10021710:  li       r16,10
   0x10021714:  li       r15,0
   0x10021718:  cmpw     r15,r16
   0x1002171c:  bge      0x10021744
   0x10021720:  rotlwi   r17,r15,2
   0x10021724:  add      r17,r3,r17
   0x10021728:  rotlwi   r18,r15,2
   0x1002172c:  add      r18,r3,r18
   0x10021730:  lwz      r18,0(r18)
   0x10021734:  mulli    r18,r18,42
%    Specialized value        ----^
   0x10021738:  stw      r18,0(r17)
   0x1002173c:  addi     r15,r15,1
   0x10021740:  b        0x10021718
   0x10021744:  blr
   0x10021748:  .long 0x0
```

Scenario #2 : Kernel Initialization : array scaling



Value injection / dynamic polymorphism

```
#[  int 32 1 mult (int[] 32 1 a,
                   int   32 1 len)  {
    int 32 1 r, i;
    for (i = 0; i < len; i = i + 1) {
        // b value will be included
        //in code generation
        a[i] = a[i] * #(b);
      }
  }  ]#
```

Optimization source

- Optimization based on kernel input parameters

POWER Binary codes specialization on 0

```
(gdb) x/20i fPtr
   0x10021710:  li       r16,10
   0x10021714:  li       r15,0
   0x10021718:  cmpw     r15,r16
   0x1002171c:  bge      0x10021744
   0x10021720:  rotlwi   r17,r15,2
   0x10021724:  add      r17,r3,r17
   0x10021728:  rotlwi   r18,r15,2
   0x1002172c:  add      r18,r3,r18
   0x10021730:  lwz      r18,0(r18)
   0x10021734:  li       r18,0
% Specialized value 0    ----^
   0x10021738:  stw      r18,0(r17)
   0x1002173c:  addi     r15,r15,1
   0x10021740:  b        0x10021718
```

Scenario #2 : Kernel Initialization : array scaling

Value injection / dynamic polymorphism

```
#[  int 32 1 mult (int[] 32 1 a,
                   int   32 1 len)  {
    int 32 1 r, i;
    for (i = 0; i < len; i = i + 1) {
        // b value will be included
        //in code generation
        a[i] = a[i] * #(b);
      }
  }  ]#
```

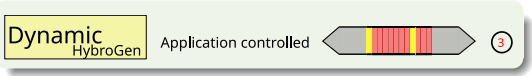
Optimization source

- Optimization based on kernel input parameters

POWER Binary codes specialization on 512

```
(gdb) x/20i fPtr
   0x10021710:  li       r16,10
   0x10021714:  li       r15,0
   0x10021718:  cmpw     r15,r16
   0x1002171c:  bge      0x10021744
   0x10021720:  rotlwi   r17,r15,2
   0x10021724:  add      r17,r3,r17
   0x10021728:  rotlwi   r18,r15,2
   0x1002172c:  add      r18,r3,r18
   0x10021730:  lwz      r18,0(r18)
   0x10021734:  rotlwi   r18,r18,9
% Specialized value 512      ----^
% (shift left instead of mul)
   0x10021738:  stw      r18,0(r17)
   0x1002173c:  addi     r15,r15,1
   0x10021740:  b        0x10021718
   0x10021744:  blr
```

Scenario #3 : Application Controlled : Newton square root



Example Transprecision Newton square root

Declaration **flt #(FloatWidth) 1** allow to specialize variable word width (and related operators) at run time

```
#[
flt #(FloatWidth) 1 iterate(
              flt #(FloatWidth) 1 u,
              flt #(FloatWidth) 1 val,
              flt #(FloatWidth) 1 div )
{
    flt #(FloatWidth) 1 r, tmp1, tmp2;
    tmp1 = val / u;
    tmp2 = u + tmp1;
    return tmp2 / div;
} ]#
```

Optimization source

- Dynamic transprecision
- Dynamic word size adaptation depending on programmer threshold (precision)

Transprecision algorithm

- Generate kernel for 32 bits `float`
- Starts iteration with 32 bits `float`
- Until A threshold
- Regenerate with 64 bits `double`
- Until convergence

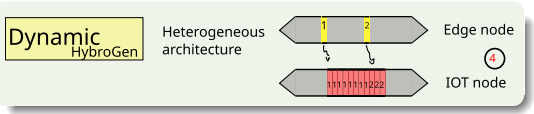
Scenario #4 : Heterogeneous Architectures : RISCV + C-SRAM (computational SRAM): Image Difference

Image Difference

```
int 32 1 subImage(int[] 16 8 a,
  int[] 16 8 b, int[] 16 8 res, int 32 1 l
// a, b, res are array of csram lines
  {
    int 32 1 i; // RISC-V register
    // Control done on RISC-V
    for (i = 0; i < len; i = i + 1)
    { // Workload done on C-SRAM
      res[i] = a[i] - b[i];
    }
  }
  return 0;
```

Optimization source

- Correct usage of specialized accelerator
- Code interleaving (RISCV, CSRAM)



Computational SRAM

- RISCV for the control part
- C-SRAM for the workload

Code generation principle

- Specialized CSRAM compiler plugin to
  - Recognize CSRAM instruction
  - Generate RISCV instructions

Other demonstrations

- TensorFlowLite convolution adaptation for CSRAM

HybroGen Internals

HybroGen description

- DB for instructions characteristics PostgreSQL
- Lexer / Parser ANTLR
- Intermediate representation (in house with C code injection capabilities)
- Classic optimization (reg allocaltion, constant expression)
- Plugins for target specific platforms
- **Generator Code Generator**

Running platforms

- Self hosted platform + OS
- QEMU : used for debug and with performance model
- Bare metal platform

Compilation Steps

- HybroGen
  - Parse & build IR
  - Transform pass
    - Arch specific plugins
    - Constant optimizer
    - Window optimizer
    - Register allocation
  - C code generator generation
- C cross compiler (gcc)
- Execution with run time code specialization

Resulting C code

- Compilette: parametric run-time binary code generator
- No run-time complex IR to manage
- No run-time tools (no assembly, no VM)

Scientific Methodology

Rationale

- To have a binary code genarator "simple" but powerfull enough to make experiments on compilation scenarios
- Code generation toolbox for
  - innovative compilation scenarios (4 preceding examples)
  - Code generation for "in house" CEA processor accelerator
  - Improve programmability of silicon based accelerators

Metrics per platforms

all	Generated Code efficiency : execution time and energy : real or based on model
all	Speed of code generation time : few cycle per generated instruction
embed. systems	Code generator size ($< Kb$)
embed. systems	Specific heterogeneous systems (C-SRAM)

Conclusion / Future Works

Main message :

- Static compiler are
  - complex to evolve
  - tied with input programming language & classic computer architecture
- Dynamic binary code adaptation is the way to go
- Let's prepare new tools & new scenarios

HybroGen future releases

- Two release per year (when ready)
- V4.0 done or soon
- Improve public demonstrators
- Open source licence (CECILL-B i.e. French BSD like)
- Stay as small as possible ($\simeq$11KLOC of python)

Future works

- Working on other programmable accelerators through HybroGen plugin :
  - Support for CEA VRP (variable precision accelerator) : Generalize the transprecision support
  - Support code generation for future CSRAM instruction set
- New code specialization applications : trigonometric functions, BLAS, MPFR library ...
- Application level optimizations

Access / Contact

- HybroGen on GitHub
- Mail me : HP Charles