Low level execution model
Tools to adapt code
Whath is an execution model and how to use it ?

Henri-Pierre CHARLES

CEA DSCIN department / Grenoble

30 nov 2022

Introduction : CPU generic algorithm

CPU Algorithm

- Forever do :
  1. Fetch instruction (IF)
  2. Increment PC
  3. Decode instruction (DE)
  4. Execute (EXE)
  5. Store result (WB)

Comments

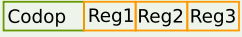
1. Algorithm duration = sum of each instructions
2. Irregular duration : memory access, complex
3. Each step can be divided in sub steps

CPU : Instruction Encoding

Link between architecture and instruction encoding

Memory

Instruction encoding



Register Bank

Instruction examples

```
add r1, r2, r9
ld r3, [r3, r5]
```

Functionnal units

Latency
Delay

SOA ARCH : ARM big.LITTLE

big

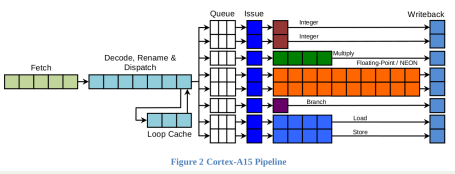


Figure 2 Cortex-A15 Pipeline

LITTLE

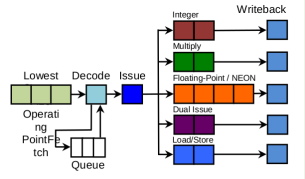


Figure 1 Cortex-A7 Pipeline

How many per day ?

ARM : “more than 30 billion processors sold with more than 16M sold every day ARM” (Nov 2013)
`http://www.arm.com/products/processors/index.php`

- 4 big processors + 4 little
- Same ISA, ...
- (even for vector operation)
- Low latencie switch

big.LITTLE notion

HWParallelismLevel uArch

Argumentation

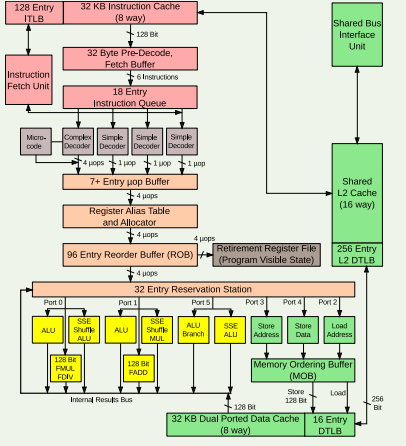
- Hidden micro architecture
-

Intel Example

- 700 high level instructions
- RISC internal uInstructions

Intel Micro architecture

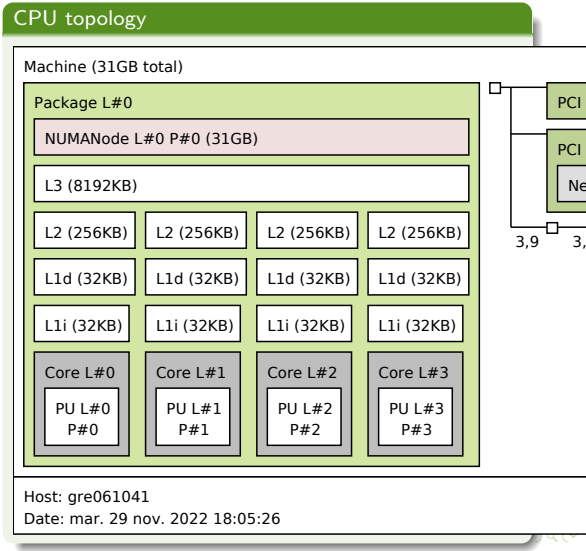
Intel Core2 micro arch



Intel Core 2 Architecture

CPU : Caches Hierarchy

- Linux command : `lstop`
- Huge impact on programs



Models : C Language for Architecture

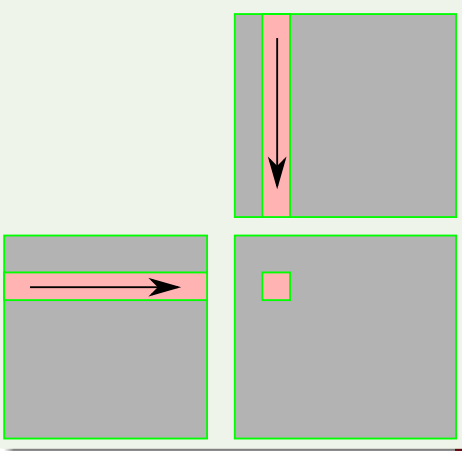
Matrix multiply (sketch)

```
1  for (int l = 0; l < SIZE; l++)
2    for (int c = 0; c < SIZE; c++)
3      for (int k = 0; k < SIZE; k++)
4        R[l][c] += A[l][k] * B[k][c];
```

"Real world"

```
1  for (c= 0; c<NCOL; c+=cacheLineSize)
2   for (l= 0; l<NLINE; l+=halfCacheLine)
3    for (c2= 0; c2<NCOL; c2+=halfCacheLine)
4      for (lk= 0; lk<halfCacheLine; lk++)
5       for (c2k= 0; c2k<halfCacheLine; c2k++)
6        for (ck= 0; ck<cacheLineSize; ck++)
7          res[l+lk][c2+c2k]+=      a[l+lk] [c+ck]* b[c2+c2k][c+ck];
```

Learn to program = learn to serialize / schedule on defined hardware !



Other "interesting examples"

SOA C compil chain

Description

- Language support
  - C language (K&R), C90 up to C11
  - Normalized
  - gcc, clang, icc, tcc, ...
  - A lot of LEGACY
- Open source is mandatory : Apple, Intel
- Economic step !

Compilation chain

- Static compilation
  - Preprocessing (# stuff)
  - Compilation
    - Lexical / Syntaxic
    - IR form
    - Phases / passes
    - Instructions selection
  - Assembly
  - Load
- Dynamic part
  - OS : `ld`
  - Addresses resolution
  - Cache interaction ?

`https://en.wikipedia.org/wiki/C_(programming_language)`

Rappels Static Compilation chain

Static compilation (on C language):

1. Preprocessor (all # stuff : rewriting) `cc -E`
2. Compilation (from C to textual assembly) `cc -S`
3. Assembly (from textual asm to binary asm) `cc -c`
4. Executable (binary + dynamic library)

Optional

- Profiling: Compile (use `-pg`) produce `File`; Run `File`; Use `gprof`
- `cc -da` dump all intermediate representation

(Use gcc -v to see all the steps)
Don't stop at static time (Operating system + processor): Load in memory, dynamic linking; Branch resolution; Cache warmup

SW-SOA GCC

Description

- https://gcc.gnu.org/
- Many platforms supported
- Since 1987
- C, C++, Fortran, Objectice-C, Ada
- OpenMP, OPenACC
- GPL Licence (base compiler for Linux distro)
- Written in C, rewritten in C++ since some years

Compilation chain

- Front-end (L to syntax tree)
- GENERIC and GIMPLE (SSA) intermediate representation
- RTL intermediate representation
- Back-end

Features

- LTO
- Plugins
- Transactional memory support

GCC : GCC-Research

Code Legacy

- Invent a new architecture for new application
- Port to a new architecture
- Write C application
- Run it ... on which platform ?
- What is your metric ?

Structure optimisation

- Invent a new optimization
  - Based on code structure
  - Based on application structure
- Use accelerator
- Performance portability

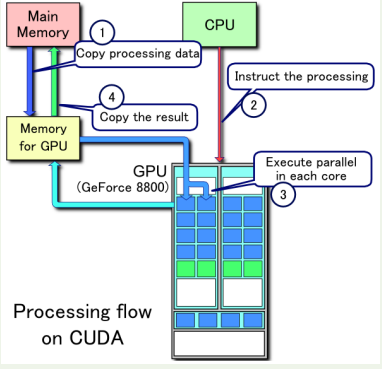
SWParallelismLevel CUDA

- Programming language for data parallelism
-

Illustration

- Normalized: No
- Portable : No (NVIDIA only)
- Scalable : No, why ?

Example of CUDA processing flow



SW-SOA CUDA-Example

Header and CUDA code

```
1  import pycuda.compiler as comp
2  import pycuda.driver as drv
3  import numpy
4  import pycuda.autoinit
5
6  mod = comp.SourceModule("""
7  __global__ void multiply_them(float *dest, float
8  {
9    const int i = threadIdx.x;
10   dest[i] = a[i] * b[i];
11 }
12 """)
```

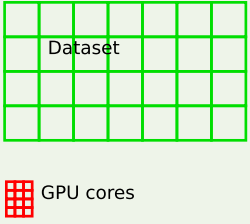
Python code

```
1
2  multiply_them = mod.get_function("multiply_them")
3
4  a = numpy.random.randn(400).astype(numpy.float32)
5  b = numpy.random.randn(400).astype(numpy.float32)
6
7  dest = numpy.zeros_like(a)
8  multiply_them(
9          drv.Out(dest), drv.In(a), drv.In(b),
10         block=(400,1,1))
11
12 print dest-a*b
```

GPU : ThreadModel

CPU side

- Tile dataset (in cuda code)
- Map data on compute grid
- Iterate on memory move / compute



GPU : CompilationChain

CUDA side

- nvcc : From CUDA program to PTX (GPU assembly)
- gcc : C to binary for the host

GPU Driver side

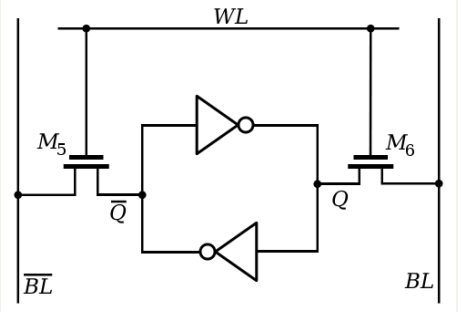
- CPU : Launch host process on CPU
- GPU driver : Compile PTX code to binary
- CPU : copy data from host mem to GPU mem
- CPU : map dataset and launch GPU execution
- GPU : run GPU code
- CPU : copy data from GPU mem to CPU mem

Introduction : Remember Memory Cell 101

SRAM memory cell depicting Inverter Loop as gates

- 6T memory cell
- Only 1 stable mode
- Read : “open” `WL`, read value
- Write : “open” `WL`, write value

Illustration



Memory_cell_(computing)

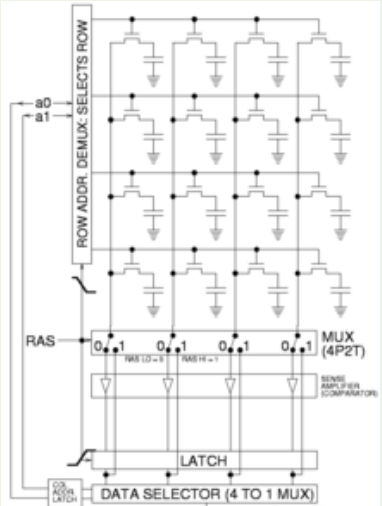
Introduction : Remember Memory 102

Functions

- Select line
- Read or write
- Potentially select word in a line
- Low voltage used; “Sense amp” to normalize

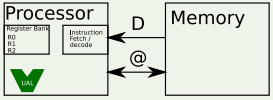
Sense_amplifier
What every programmer should know about memory

Memory array

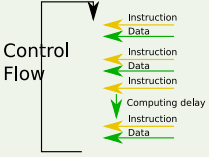


Inverted Von Neumann Programming Model

Choosen Programming model



Bus transactions Code

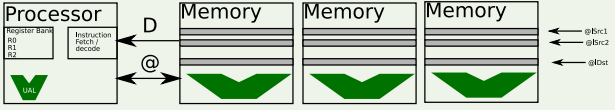


ld r0 = @A

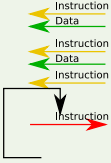
ld r1 = @B

add r2 = r0, r1

st r2 = @C



Bus transactions Code



ld r0 = @lSrc1

ld r1 = @lSrc2

ld r2 = @lDst

add lDst = lSrc1, lSrc2

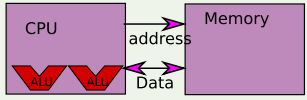
increment line adresses

Why ?

- Allows scalability :
  - Any vector size
  - Any tile number
  - Any system configuration : near or far IMC
- Works with any processor

Ideas : Inverted Von Neumann

Inverted model



CPU	Memory
control flow, address compute application workload,Mem I/O	answer CPU
control flow, address compute	answer CPU (less), application workload

Von neuman

- Stored instructions
- Bottleneck = limited bandwidth (data, insn, L1, L2, L3)
- Programm with data choregraphy

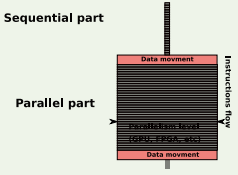
Non von Neumann

- Break model and bottleneck
- Problems
  - Send insns
  - Synchronize
  - Data layout

Von Neuman broked, what about Amhdal Law's ?

Ahmdal law's: "Speedup is limited by the sequential part"

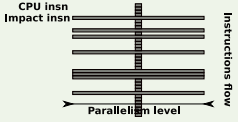
Classical approach



Programmer approach

- Has to maximize parallel part
- Deal with data "choregraphy" between CPU and GPU.

CSRAM approach



Programmer approach

- Ease to interlace scalar instruction and IMPACT instructions
- Do not move data

HybroGen : Initial Objectives : New Code Generation Paradigm

Objectives: Application with Binary Automodification

- Without external libraries
- As fast as possible : Code generation speed $\tilde{1}0$ clock cycles per instruction)
- As small as possible : Code generators $\tilde{1}$ KB
- Portable accross architecture : RISCV, IBM Power 8 / 9, Kalray, CxRAM

Benefits

- Data set code generation dependant : values, size, strides
- User programmable code generation
- Heterogeneous ISA code generation

Domain Specific Language : HybroLang

By The Way :

Do we need a new programming language for a new progamming model ?
YES !
[Hennessy-Patterson "A New Golden Age for Computer Architecture"]

Why a new programming language ?

- Want to make data dependent code generation, on the fly.
- Do not want to find vectorization / parallelization : use it !
- Want to **implement** "Inverted Von Neumann Model" (slides on IMC)
- Want to use specific arithmetic : integer, saturated, arithmetic, ... IP@, stochastic, geometric shapes, ...

HybroLang features

- Similar to C syntax
- Variables = Hardware elements (register, memory line)
- Arithmetic Operators = existing processor UAL
- Run-time delayed code generation

Motivation : Static Compiler Versus Compilette

Static Compiler

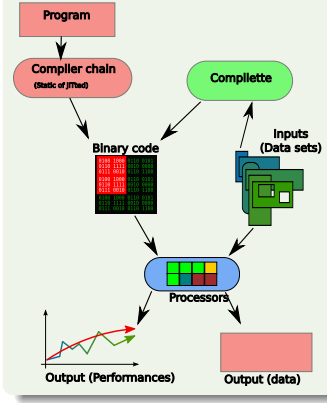
- Run once
- Does not know data set characteristics
- Slow compilation (even with JIT)

Compilette

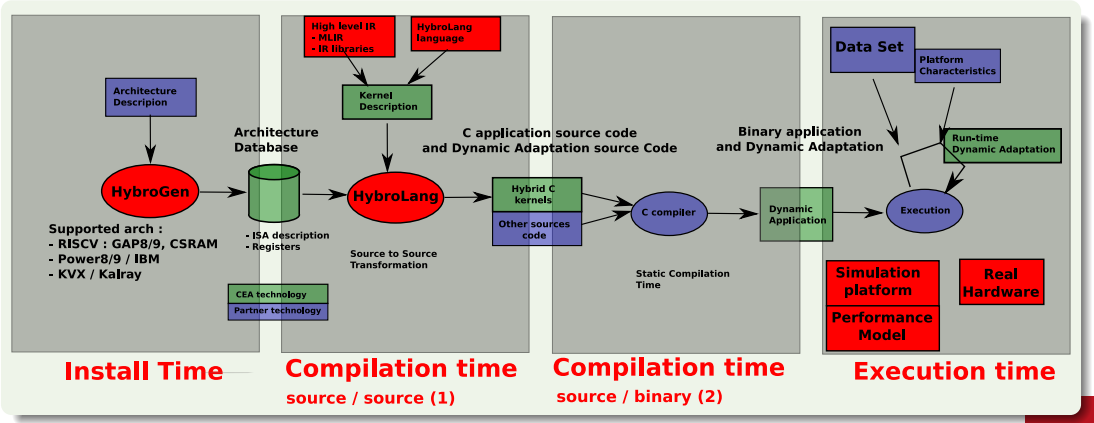
- Adapt on the fly
- Knowledge of the architecture
- Knowledge of the application

Need tools !

Static compilation versus dynamic adaptation



HybroGen: General View



HybroGen : H2 Language Specific

Language Level

- Link to hardware arithmetics datatype :int, float
- To be done : complex, ip, pixel, char
- Adapt & Link to vector size
- C like syntax
- Leaf function level
- Loop, Arithmetic expression

Supported architecture

- RISCV : ETHZ riscy, GreenWave / GAP8/9
- CxRAM : CEA / RISCV + CXRAM
- POWER : IBM / Power8
- Kalray : Kalray / KVX

Run time specific

- Implement innovative code generation scenarios
- Implement innovative code optimization scenarios
- Make dataset responsive applications

HybroGen : Objectives

Application domains

- Stochastic number support
- Packet filtering : Datatype ipv4, ipv6 addresses
- Transprecision algorithms : usage on mathematical iterative methods
- Stencil processing

Already done

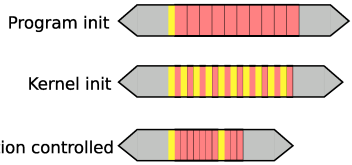
- On the fly code generation for heterogeneous architectures
- Transprecision support : on the fly code generation for precision adaptation. Working demonstration on Newton algorithm : Power / RISCV / Kalray
- Support for In Memory Computing (next slides)
- Target processor modeling (QEMU plugin)

Compilation

Execution time

(a) Static

(b)Dynamic



HybroGen : Optimization Phases

Compilation strategy

- Read code
- Gather needed instructions
- Transform in IR
- Optimize
- Generate C code generators
  - Macro instruction level
  - Instruction generators using macro instruction
  - Compilette rewriting using instruction generators

Optimization levels

- Static :
  - Register allocation
  - Classic code optimization
- Dynamic ($\tilde{40}$ machine cycle per instructions)
  - Instruction generation
  - Instruciton selection
  - Data interleaving

HybroGen : Optimization-Targe-Specific-Phases

RISC architecture

POWER Specific optimization on compare & branch optimization

Kalray Specific optimization on arithmetic operators

CxRAM

- Implement a code generator, generator generator
  - Detect CxRAM instructions
  - Generate RISCV instruciton to generate the CxRAM instruction
  - Generate the RISCV generators

HybroGen : Programmation

Find a strategy

- Is your program compute bound / memory bound ?
- What parameters could be specialized ?
- What high level optimization is usefull ?
- Do you need to use a special instruction
- Regeneration frequency
  - At each function call ?
  - At each data set change ?
  - At a specific frequency ?
  - At a specific event ?

Use Hybrogen

- Use the legacy `HybroGen`
- Implement a specific backend optimization
- Add a new architecture

HybroGen : Simple Add Principle

Not so simple example

- Code specialization / Parameter reduction
- `add(int a, int b)` by `add(int a)` for a known `b` value
- Simple, but much more efficient because at run-time

Benefits

- Reduce memory pressure
- Reduce aritmetic (could optimize on `b` value)
- “Small” tutorial example
- Example on RISCV (works on all architectures)

HybroGen : Simple-Add-Source

Simple Addition with specialization

```
1
2  typedef  int (*pifi)(int);
3
4  h2_insn_t * genAdd(h2_insn_t * ptr, int  b)
5  {
6    #[
7    int 32 1 add (int 32 1 a)
8    {
9      int 32 1 r;
10     r = #(b) + a; // b values will be included in code generation
11     return r;
12   }
13   ]#
14   return (h2_insn_t *) ptr;
15 }
```

HybroGen : Simple how to Build

How to run example

```
1  gre061041:CodeExamples/>./RunDemo.py -a riscv -i Add-With-Specialization
2  Namespace(arch=['riscv'], clean=False, debug=False, inputfile=['Add-With-Specialization'], verbose=False)
3  -->rm -f Add-With-Specialization Add-With-Specialization.c
4  -->which riscv32-unknown-elf-gcc
5  -->../HybroLang.py --toC --arch riscv --inputfile Add-With-Specialization.hl
6  -->riscv32-unknown-elf-gcc -Wall -o Add-With-Specialization Add-With-Specialization.c
7  ('3', '25')
8  -->qemu-riscv32 Add-With-Specialization 3 25
9  gre061041:CodeExamples/>qemu-riscv32 Add-With-Specialization 3 25
10 // Compilette for simple addition between 1 variable with
11 // code specialization on value = 3
12 3 + 25 = 28
```

HybroGen : Simple-Add : Generated Source

Code Macro instructions

```
1  #define riscv_G32(INSN){ *(h2_asm_pc++) = (INSN);}
2
3  #define RV32I_RET__I_32_1() /* RET */ \
4  do { \
5          riscv_G32(((0x8067 >> 0) & 0xffffffff)); \
6  } while(0)
7
8  #define RV32I_ADDI_RRI_I_32_1(r1,r0,i0) /* ADD */ \
9  do { \
10         riscv_G32(((i0 & 0xfff) << 20)|((r0 & 0x1f) << 15)|((0x0 & 0x7) << 12)|((r1 & 0x1f) << 7)|((0x13 & 0x7f) >> 0)); \
11 } while(0)
```

HybroGen : Simple-Add Generated Code

Compilette

```
1         h2_asm_pc = (h2_insn_t *) ptr;
2         h2_codeGenerationOK = true;
3         riscv_genMV_2(h2_00000003, (h2_sValue_t) {VALUE, 'i', 1, 32, 0, (b)});
4         riscv_genADD_3(r, h2_00000003, a);
5         riscv_genMV_2(h2_outputVarName, r);
6         riscv_genRET_0();
7         /* Call back code for loops */
8         h2_save_asm_pc = h2_asm_pc;
9         h2_asm_pc = h2_save_asm_pc;
10        h2_iflush(ptr, h2_asm_pc);
```

HybroGen : Simple-Add-Generated

Instruction Selector

```
1  void riscv_genADD_3(h2_sValue_t P0, h2_sValue_t P1, h2_sValue_t P2)
2  {
3
4      if ((P0.arith == 'i') && (P0.wLen <= 32) && (P0.vLen == 1) && P0.ValOrReg == REGISTER && P1.ValOrReg == REGISTER && P2.ValOrReg ==
5      {
6          RV32I_ADDI_RRI_I_32_1(P0.regNro, P1.regNro, P2.valueImm);
7      }
8
9      else if ((P0.arith == 'i') && (P0.wLen <= 32) && (P0.vLen == 1) && P0.ValOrReg == REGISTER && P1.ValOrReg == REGISTER && P2.ValOrR
10     {
11         RV32I_ADD_RRR_I_32_1(P0.regNro, P1.regNro, P2.regNro);
12     }
```

HybroGen : Simple-Add Generated code

Main code generator

```
1  h2_insn_t * genAdd(h2_insn_t * ptr, int  b)
2  {
3  /* Code Generation of 4 instructions */
4  /* Symbol table :*/
5          /*VarName = { ValOrLen, arith, vectorLen, wordLen, regNo, Value} */
6          h2_sValue_t a = {REGISTER, 'i', 1, 32, 10, 0};
7          h2_sValue_t h2_outputVarName = {REGISTER, 'i', 1, 32, 10, 0};
8          h2_sValue_t r = {REGISTER, 'i', 1, 32, 5, 0};
9          h2_sValue_t h2_00000003 = {REGISTER, 'i', 1, 32, 6, 0};
10
11
12 /* Label  table :*/
13 #define riscv_genLABEL(LABEL_ID) labelAddresses[LABEL_ID] = h2_asm_pc;
14 h2_insn_t   * labelAddresses []={
15         };
16
17
18         h2_asm_pc = (h2_insn_t *) ptr;
```

HybroGen : Simple-Add-Exec

Simple run

```
1 gre061041:CodeExamples/>qemu-riscv32 Add-With-Specialization 3 40
2 // Compilette for simple addition between 1 variable with
3 // code specialization on value = 3
4 3 + 40 = 43
```

Run with debug

```
1 qemu-riscv32 Add-With-Specialization 3 25
2 // Compilette for simple addition between 1 variable with
3 // code specialization on value = 3
4 0x19008 : RV32I_MV_RI_I_32_1
5 0x1900c : RV32I_ADD_RRR_I_32_1
6 0x19010 : RV32I_MV_RR_I_32_1
7 0x19014 : RV32I_RET__I_32_1
8 3 + 25 = 28
```

HybroGen : Simple-Add-Source

Simple Addition with specialization

```
1  int main(int argc, char * argv[])
2  {
3    h2_insn_t * ptr;
4    int in0, in1, res;
5    pifi fPtr;
6
7    if (argc < 3)
8      {
9        printf("Give␣2␣values\n");
10       exit(-1);
11     }
12   in0  = atoi (argv[1]);   // Get the users values in1 & in2
13   in1  = atoi (argv[2]);
14   ptr  = h2_malloc (1024);  // Allocate memory for 1024 instructions
15   printf("//␣Compilette␣for␣simple␣addition␣between␣1␣variable␣with\n");
16   printf("//␣code␣specialization␣on␣value␣=␣%d\n", in0);
17   fPtr = (pifi) genAdd (ptr, in0); // Generate instructions
18   res  = fPtr(in1);  // Call generated code
19   printf("%d␣+␣%d␣=␣%d\n", in0, in1, res);
```

HybroGen : Simple-Add Debug

1 shell to Run / interact

```
gre061041:CodeExamples/>qemu-riscv32 -g \
    7777 Add-With-Specialization 3 25
// Compilette for simple addition
// between 1 variable with
// code specialization on value = 3
0x19008 : RV32I_MV_RI_I_32_1
0x1900c : RV32I_ADD_RRR_I_32_1
0x19010 : RV32I_MV_RR_I_32_1
0x19014 : RV32I_RET__I_32_1
```

1 shell to Debug / observe

```
riscv32-unknown-elf-gdb Add-With-Specialization
GNU gdb (GDB) 9.2
../..
(gdb) target remote :7777
(gdb) break main
Breakpoint 1 at 0x107c6: file Add-With-Specialization.c, line 201.
(gdb) c
Continuing.
Breakpoint 1, main (argc=3, argv=0x40800374) at Add-With-Specializat
201         if (argc < 3)
(gdb) n
206         in0  = atoi (argv[1]);
// Get the users values in1 & in2
211         fPtr = (pifi) genAdd (ptr, in0); // Generate instructions
(gdb)
212         res  = fPtr(in1);  // Call generated code
(gdb) x/4i fPtr
   0x19008:     ori     t1,zero,3
   0x1900c:     add     t0,t1,a0
   0x19010:     mv      a0,t0
   0x19014:     ret
(gdb)
```

HybroGen : Transprecision Source H2

Transprecision square root source code

```
1  /* Newton square root demonstration with variable precision */
2  h2_insn_t * genIterate(h2_insn_t * ptr, int FloatWidth)
3  {
4    #[
5      flt #(FloatWidth) 1 iterate(flt #(FloatWidth) 1 u, flt #(FloatWidth) 1 val, flt #(FloatWidth) 1 div )
6      {
7          flt #(FloatWidth) 1 r, tmp1, tmp2;
8          tmp1 = val / u;
9          tmp2 = u + tmp1;
10         return tmp2 / div;
11     }
12   ]#       /* r = (u + (#(value) / u))  / 2.0*/
13
14   return (h2_insn_t *) ptr;
15 }
```

HybroGen : Transprecision Generated Macros

Macro instruction generation

```
1  /* Begin Header autogenerated part */
2  #include "h2-power-power.h"
3  #define power_G32(INSN){ *(h2_asm_pc++) = (INSN);}
4
5  void P1_BLR__I_32(void){ /* ret */
6  #ifdef H2_DEBUG
7      printf("%p : P1_BLR__I_32\n", h2_asm_pc);
8  #endif
9          power_G32(((0x4e800020 >> 0) & 0xffffffff)); \
10 }
11
12 void PPC_FADDS_RRR_F_32(int r0, int r1, int r2){ /* add */
13 #ifdef H2_DEBUG
14     printf("%p : PPC_FADDS_RRR_F_32\n", h2_asm_pc);
15 #endif
16         power_G32(((0x3b & 0x3f) << 26)|((r0 & 0x1f) << 21)|((r1 & 0x1f) << 16)|((r2 & 0x1f) << 11)|((0x2a & 0x7ff) >> 0)); \
17 }
```

HybroGen : Transprecision-Generated-Compilette

Compilette Generation

```
1  /* Newton square root demonstration with variable precision */
2  h2_insn_t * genIterate(h2_insn_t * ptr, int FloatWidth)
3  {
4  /* Code Generation of 4 instructions */
5  /* Symbol table :*/
6          /*VarName = { ValOrLen, arith, vectorLen, wordLen, regNo, Value} */
7          h2_sValue_t u = {REGISTER, 'f', 1, (FloatWidth), 1, 0};
8          h2_sValue_t val = {REGISTER, 'f', 1, (FloatWidth), 2, 0};
9          h2_sValue_t div = {REGISTER, 'f', 1, (FloatWidth), 3, 0};
10         h2_sValue_t h2_outputVarName = {REGISTER, 'f', 1, (FloatWidth), 1, 0};
11         h2_sValue_t r = {REGISTER, 'f', 1, (FloatWidth), 14, 0};
12         h2_sValue_t tmp1 = {REGISTER, 'f', 1, (FloatWidth), 15, 0};
13         h2_sValue_t tmp2 = {REGISTER, 'f', 1, (FloatWidth), 16, 0};
14         h2_sValue_t h2_00000000 = {REGISTER, 'f', 1, (FloatWidth), 17, 0};
```

HybroGen : Transprecision-Generated-Compilette

Compilette Generation Code generator

```
1         h2_asm_pc = (h2_insn_t *) ptr;
2         h2_codeGenerationOK = 1;
3         power_genDIV_3(h2_00000000, val, u);
4         power_genMV_2(tmp1, h2_00000000);
5         power_genADD_3(h2_00000000, u, tmp1);
6         power_genMV_2(tmp2, h2_00000000);
7         power_genDIV_3(h2_00000000, tmp2, div);
8         power_genMV_2(h2_outputVarName, h2_00000000);
9         power_genRET_0();
10        /* Call back code for loops */
11        h2_save_asm_pc = h2_asm_pc;
12        h2_asm_pc = h2_save_asm_pc;
13        iflush(ptr, h2_asm_pc);
14          /* r = (u + (#(value) / u))  / 2.0*/
```

HybroGen : Transprecision-Generated-InstructionSelector

Macro instruction generation

```
1  void power_genADD_3(h2_sValue_t P0, h2_sValue_t P1, h2_sValue_t P2)
2  {
3      if ((P0.arith == 'f') && (P0.wLen <= 32) && (P0.vLen == 1) && P0.ValOrReg == REGISTER && P1.ValOrReg == REGISTER && P2.ValOrReg ==
4      {
5          PPC_FADDS_RRR_F_32(P0.regNro, P1.regNro, P2.regNro);
6      }
7      else if ((P0.arith == 'f') && (P0.wLen <= 32) && (P0.vLen == 1) && P0.ValOrReg == REGISTER && P1.ValOrReg == REGISTER && P2.ValOrR
8      {
9          PPC_FADDS__RRR_F_32(P0.regNro, P1.regNro, P2.regNro);
10     }
11     else if ((P0.arith == 'f') && (P0.wLen <= 32) && (P0.vLen == 4) && P0.ValOrReg == REGISTER && P1.ValOrReg == REGISTER && P2.ValOrR
12     {
13         V2_03_VADDFP_RRR_F_32(P0.regNro, P1.regNro, P2.regNro);
14     }
15     else if ((P0.arith == 'f') && (P0.wLen <= 64) && (P0.vLen == 1) && P0.ValOrReg == REGISTER && P1.ValOrReg == REGISTER && P2.ValOrR
16     {
17         P1_FADD_RRR_F_64(P0.regNro, P1.regNro, P2.regNro);
18     }
19     else if ((P0.arith == 'f') && (P0.wLen <= 64) && (P0.vLen == 1) && P0.ValOrReg == REGISTER && P1.ValOrReg == REGISTER && P2.ValOrR
20     {
21         P1_FADD__RRR_F_64(P0.regNro, P1.regNro, P2.regNro);
22     }
23
24     else
```

HybroGen : Transprecision-Exec

Macro instruction generation

```
qemu-ppc64le Newton.power 65536 1e-13
Compute square root of 65536.000000
With precision of 1.000000e+01 (float)
With precision of 1.000000e-13 (double)
0x100212a0 : PPC_FDIVS_RRR_F_32
0x100212a4 : P1_FMR_RR_F_32
0x100212a8 : PPC_FADDS_RRR_F_32
0x100212ac : P1_FMR_RR_F_32
0x100212b0 : PPC_FDIVS_RRR_F_32
0x100212b4 : P1_FMR_RR_F_32
0x100212b8 : P1_BLR__I_32
  0 float  : 32768.50000000000000000000, 1.000000e+01
  1 float  : 16385.25000000000000000000, 1.000000e+01
  2 float  : 8194.62500000000000000000, 1.000000e+01
  3 float  : 4101.31103515625000000000, 1.000000e+01
  4 float  : 2058.64526367187500000000, 1.000000e+01
  5 float  : 1045.23986816406250000000, 1.000000e+01
  6 float  : 553.96966552734375000000, 1.000000e+01
```

HybroGen : Transprecision Exec 2nd part

Macro instruction generation

```
1     7 float  : 336.13607788085937500000, 1.000000e+01
2     8 float  : 265.55236816406250000000, 1.000000e+01
3     9 float  : 256.17181396484375000000, 1.000000e+01
4   0x100212a0 : P1_FDIV_RRR_F_64
5   0x100212a4 : P1_FMR_RR_F_64
6   0x100212a8 : P1_FADD_RRR_F_64
7   0x100212ac : P1_FMR_RR_F_64
8   0x100212b0 : P1_FDIV_RRR_F_64
9   0x100212b4 : P1_FMR_RR_F_64
10  0x100212b8 : P1_BLR__I_32
11   10 double : 256.00005761765521583584, 1.000000e-13
12   11 double : 256.00000000000648014975, 1.000000e-13
13   12 double : 256.00000000000000000000, 1.000000e-13
14   13 double : 256.00000000000000000000, 1.000000e-13
```

HybroGen : Transprecision Source Main

Transprecision square root source code (main control)

```
1   fPtr1 = (piff) genIterate (ptr, FLOAT);
2   do
3   {
4     if ((diff < precf) && isFloat)
5       {              /* Code re-generation with double for better precision */
6         fPtr2 = (pidd) genIterate (ptr, DOUBLE);
7         isFloat = False;
8       }
9     value = next;
10    next = (isFloat)?fPtr1(value, af, 2.0):fPtr2(value, af, 2.0);
11    diff = ABS(next - value);
```

Conclusion :

Have fun with binary code generation

- Many scenario to be invented

Intersting period

- End of the Moore's Law
- Amdhal law still valid
- Energy challenge = Specialized architecture
- Specialized architecture = Fun with compilers