Dynamic Compilation for Transprecision Applications on Heterogeneous Platform

Julie Dumas, Henri-Pierre Charles, Kévin Mambu, Maha Kooli

MDPI

Article

Dynamic Compilation for Transprecision Applications on Heterogeneous Platform

Julie Dumas [1], Henri-Pierre Charles [1], Kévin Mambu[1] and Maha Kooli [1]

Univ Grenoble Alpes, CEA, LIST, F-38000 Grenoble, France
[1] Cea-List; firstname.name@cea.fr
* Correspondence: Julie.Dumas@cea.fr

Abstract: This article describes a software environment called *HybroGen* , which helps to experiment binary code generation at run-time. As computing architectures are getting more complex, the application performances become data-dependent. The proposed experimental platform is helpful in programming applications that can be reconfigured at run-time in order adapted for a new data environment. *HybroGen* platform is adapted to heterogeneous architectures and can generate instructions for different target. This platform allows to go farther than classical JIT compilation in many directions: the code generator is smaller by three orders of magnitude, faster by three orders of magnitude compared to JIT (Just-In-Time) platforms and allows making code transformation that is impossible in traditional compilation scheme like code generation for non Von Neumann accelerators or dynamic code transformations for transprecision. The latter will be illustrated in a code example: the square root with Newton's algorithm. We also illustre the proposed *HybroGen* platform with two others examples: a multiplication with a specialization on a value determine at run-time and a convertion of degree Celcius to degree Fahrenheit. This article presents a proof of concept of the proposed *HybroGen* platform in terms of its functionalities, and demonstrate the working status.

Keywords: transprecision; dynamic compilation; heterogeneous; just in time; code specialization

Citation: Dumas, J.; Charles, H.; Mambu K.; Kooli M. Dynamic Compilation for Transprecision Applications on Heterogeneous Platform. *J. Low Power Electron. Appl.* **2021**, *1*, 0. https://doi.org/

Received: 2021-04-30
Accepted:
Published:

1. Introduction

Compilation and code generation[1] are 50 years old research domains, parallel to the computer architecture research domain [1]. Compilers have the difficult task to transform a source code application into a running binary code.

Due to the constant evolution of both application domains and computing systems, this task becomes more and more complex. The difficulty comes from the fact that those evolution goes in the opposite direction.

From **the classical application development** point of view, the priority is to make programmers efficient by providing richer programming environments. As an illustration the Java SDK environment contained around 100 classes in the 1.0 release (1995) and 13,367 for the 1.5 (J2SE 5.0 2004) release (by counting the .class objects). These two orders of magnitude in complexity gives a very rich programming environment which makes programmers very efficient because of the "job oriented API" (JDBC for database applications, Graphic for gaming, etc).

This organization improves programmer efficiency but augments the "distance" between the computing architecture and the problem to solve: the programmer generally focuses on problem-solving by using complex APIs based on high level layers, which augments differences between the data to compute on, and the hardware capabilities.

A first solution to this problem was to delay the code generation at run-time by using JIT (Just-In-Time) compilation in Java. The hotspot compiler [2] is efficient, compiles

[1] This work was supported by theEuropean H2020 FET project OPRECOMP under Grant 732631

on demand but is based on method count, not on hardware counter, nor on data set characteristics. JIT compilation needs a huge memory footprint and needs a long latency to react to new application behavior.

In another domain, **scientific computation applications**, the programmer is aware of the underlying hardware, takes care of the performance by using efficient compilers and uses algorithms where the data accuracy is computed at the application level. To illustrate this, many examples are available in the classical book "Numerical Recipes" [3]. A typical example is the *conjugate gradient algorithm* where the iteration number relies on a *residue* value. This value decreases during the computation and controls the end of the program.

In this domain it could be interesting to do the computation with a reduced precision, which allows computing faster because by reducing the memory bandwidth pressure, and switching to an improved precision at the end. This simple scheme is very complex to setup practically.

In this article we present a new compilation infrastructure proof of concept that allows solving the two identified difficulties that classical compilation chain does not solve:

- Make applications aware of the data-set characteristics and allow to take advantage of this knowledge to optimize code.
- Render possible dynamic transprecision, *i.e.* allow transforming the binary code at run time to change the data types during the application run.

We presents our compilation flow and the different steps. We also show three demonstrations examples to demonstrate the capabilities of our tool: a conversion of degree Celcius to degree Fahrenheit, a multiplication with a specialization on a data fixed at run-time and finally the computation of the square root with Newton's algorithms.

The article is composed of a section 2 where we introduce some other compilation approaches. The, section 3 presents the compilation objectives ans explains as well as the targeted compilation scenarios. Section 4 illustrates how the compilation chain works on small tutorial examples. Section 5 discusses the future evolution of this compilation infrastructure. Finally, section 6 concludes this paper.

2. Related Works

Many compiler works can be cited about compilation, but there are not so many works related to delayed code generation or at least a compilation scenario which allows taking optimization decision at a different time than the static compilation.

2.1. Code Specialization

All standard C compilers are able to do partial evaluation and, for example, able to replace expression containing constants by a resulting value.

The initial idea to do run-time code specialization (i.e. partial evaluation) for the C language came from C. Consel in the 90' [4]. But at that time the underlying hardware was simpler in terms of memory hierarchy and ALU capability.

2.2. Install Time

Many works have tried to do detect possible optimization during the program install on a new machine.

ATLAS in 2001 [5] is a BLAS implementation with semi-automatic optimization detection. Other works including source code generation push the limit farther for other mathematical kernels: FFTW in 2005 [6] for FFT implementations and SPIRAL for linear algebra kernels [7].

Interestingly, FFTW has a dynamic scheduler which chooses the best implementation at run-time, depending on the FFT signal size.

2.3. High Level Intermediate Representation

Leaving the C language offer opportunity to rely on high-level intermediate format. Java hostspot compiler [2] has an interesting strategy using different compiler strategy. It starts the execution by interpreting the code, then depending on the number of function calls, it applies different aggressiveness levels of compilation. But the strategy is only based on function call statistics and execution timing. There is no direct relation between the dataset and the compilation strategy and no vectorization strategy.

JavaScript also uses just-in-time compilation strategy but both Java and Javascript have a very costly compilation phase as described in [8].

A similar approach is described in this Vapor SIMD article [9] but no practical implementation is proposed.

2.4. deGoal

Another attempt was made with the deGoal tool [10]. This tool allows implementing similar compilation scenarios. The programming language was portable across similar SIMD architectures but was at assembly level which makes complex applications difficult do implement.

3. Compiler Level Support for Transprecision

This section presents challenges in terms of compilation for applications using transprecision.

3.1. Transprecision and Challenges for Compilers

Transprecision computing [11] is explored by the H2020 European project OPRECOMP. The idea is to reduce energy consumption by using approximate computing. For example, the precision can be decreased using small float, *i.e.* 8-bit or 16-bit floating point numbers. The precision is adapted during the computation with criteria to use more precision at the end of the computation. This is particularly convenient for iterative mathematical applications where the iteration number is controlled by a diminishing value.

One of the characteristics of transprecision is the fact that is adaptated at run-time and controlled by the application of the data size which is not known at compilation time. Compiler optimization in particular loop statements cannot be used in this case because the compiler does know when applications move to more precision.

3.2. HybroLang : a New Language

In this paper we propose *HybroLang* , a new language, developed within the proposed *HybroGen* compilation platform. It permits to declare more complex data types, like vector, data with varing length and multiple arithmetics: integer, float, complex numbvers, pixels, IP addresses, etc.

3.2.1. Compiler: When Code Generation Should Arise ?

There are different *code generation times* to generate a code for an application. In a standard development flow, the code is generated at static compilation time. Data values are resolved at run-time and with only one step of compilation before the execution, some optimization cannot be applied.

Usually programmers want to write a small code with good performance on all computers. This is not the case in the real life, programmers need to specialize code for a specific architecture. Moreover, programmers develop different version of the code to adapt programs to data types like float, integer and different word size.

With *HybroGen* we propose to generate instructions during the execution to take advantage of data values resolved at run-time. Our platform allows experimentations on multiple scenarios of *code generation time*.

3.2.2. Run-time Code Generation Scenarios

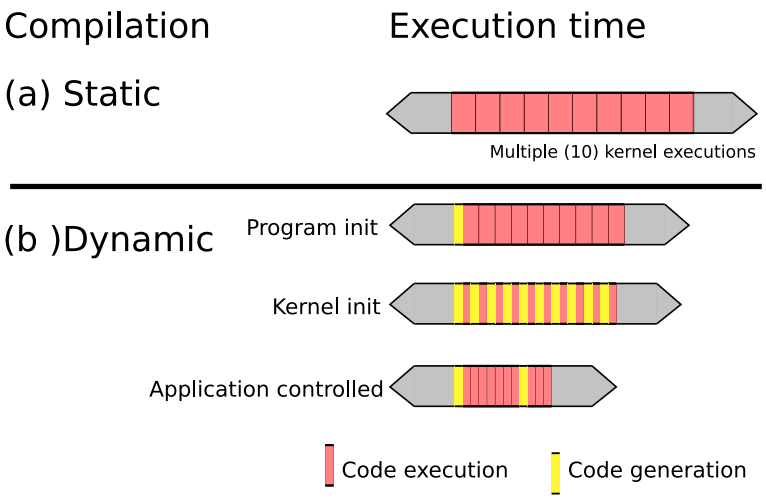

Figure 1. Chronograms of different compilation scenarios. On the top the static compiler (a), without any code specialization at run time. On the bottom our proposition for dynamic compilation (b) with 3 scenarios: binary code specialization at program init, kernel init or application controlled

Figure 1 explains our goal in terms of code generation. On the top of the figure, we illustrate the timing between static compilation (a) and execution time. We illustrate a classical use case where the execution is composed of a prelude, multiple kernel executions and a postlude.

When the data set is not known at compile time, whatever the static compilation time devoted to the kernel compilation, the compiler should be conservative and could not take into account the data characteristics which could be used for optimization (loop bound, data values, needed precision, *etc.*).

In our case we want to generate the binary code at run-time and use dynamic compilation (b). We list the following code generation scenarios:

Program init: the code generation takes place at the beginning of the application and a minimal knowledge make small optimization possible. The binary code is generated once, and the binary code is called many times.

Kernel init: in the second scenario, code generation is done at each kernel invocation. The data set information are so rich that the generated code is very efficient, thanks to the optimizer contained in the code generator. The code generation time could be amortized at each kernel call.

Application driven: in some situation, the application has a knowledge of the context and the programmer want to have the control of the code generation. For example many mathematical applications have loops controlled by a "residue" value. This value can be used to decide when the code generation should be called to improve the precision.

Those scenarios will be illustrated on some tutorial examples in section 4.

3.2.3. Language

In this paper, we propose *HybroLang* is a new language with syntax close to C programming language. We develop *HybroLang* to add support for dynamic compilation of applications with different targeted architectures. This language uses specific data types which are defined with a triplet type, vector size and word size. This language is used only to describe the part of kernel that we want to optimize, we named this part a *compilette*. The other part of the program is written with the language targeted, in this

paper we have chosen C language, but we can imagine other languages like JavaScript or python.

3.2.4. Data and Code Generation Interleaving

The main characteristics of our *HybroGen* environment are:

- The possibility to delay the code generation and have versatile code generation scheme that will be demonstrated later in this article,
- Variables are hardware registers,
- There is no parenthesis expression to avoid local register allocation,
- Special constructions `#(expression)` allows plugging expression results into the binary code. This point is very important; it allows to:
  - Insert values into binary code; thus avoid a memory access,
  - Change vector length at run-time,
  - Change the data type length at run-time.

Those characteristics will be demonstrated in the later examples.

3.3. HybroGen Platform

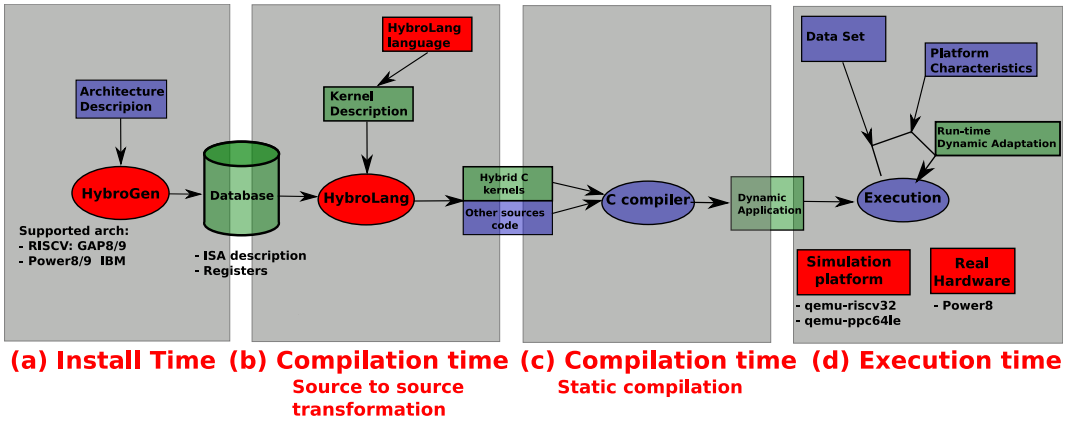

Figure 2. Overview of *HybroGen* platform with 4 steps: (a) install, (b) compilation source to source, (c) compilation source to binary and (d) execution

HybroGen is composed of 4 steps, shown in figure 2 which correspond to different times: install time, source to source compilation time, source to binary compilation time and execution time. At **install time** (a), the description of the instruction set architecture (ISA) is stored in a database. The **compilation time** (b) on figure 2 maps to the compilation source to source and is specific to the proposed *HybroGen* compilation flow. The input is a kernel described with *HybroLang* language described previously. Our compiler *HybroGen* implements different passes of compilation, like a classic compiler but at the output it produces a code in an existing programming language, for this paper it produces C code. *HybroLang* requests the database to construct a code generator which writes the correct encoding at run time. After using *HybroLang* , we use a compiler for the second **compilation time** (c) like `gcc` or `clang`, in this paper we have used `gcc`. Finally, depending on the scenario choice, at **execution time** (d), the code of the compilette is executed which generates the instructions that are executed at the backend.

4. Demonstration of *HybroGen* for Transprecision Applications

In this section we present demonstrations and results of using *HybroGen* for transprecision applications on different platforms like power or RISC-V.

4.1. Experimental Platform

To run demonstrations a system-level simulator and real platform are used. Qemu [12] is used to simulate POWER8 and RISC-V architectures. This simulator exists for different

architectures like x86, MIPS or ARM and support different variants. For example, there is a version for RISC-V 32 bits and another for the 64 bits architecture. In this paper we use qemu-riscv32 version 5.0.0 and qemu-ppc64le version 5.0.0. We also verify our results on a physical POWER8. For the static compilation, *e.g.*, the source to binary compilation, we use riscv32-elf-gcc-9.3.0 and powerpc64le-linux-gnu-gcc-8.

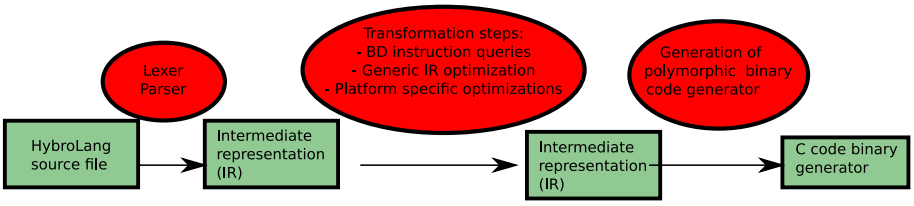

Figure 3. HybroLang compilation chain. Containing classical compilation steps: lexer and parser, generic and specific IR optimizations and also specific ones: using a SQL data base to store instructions specifications and C code generation, which will act as polymorphic binary code generator at run-time

The figure 3 shows the different steps of our compilation chain from the figure 2 section (b) which rewrites the *HybroLang* section of the application to a C version which will be able to generate multiple binary version. This capability to generate multiple binary version is very important on muliple contexts: adapt to hardware characteristics, dataset parameter and, as we focus on this article, on dataset precision.

4.2. Application Scenarios of HybroGen

Dynamic compilation allows code generation at different times during the execution of a program. Figure 4 presents three moments to generate instructions corresponding to the kernel similar to Figure 1 but in an algorithmic form. In the first case, the code is generated only once and at program initialization. This case illustrates a situation where the generated code is execute more than once, N times in the figure, to amortize the cost of dynamic compilation. In the figure 4 we can see that parameter of the execution, *i.e.* `i`, is not used for the code generation but it can use in parameter of the code generated call. We also see the specialization parameter `s` which is a parameter of the generated function `genAdd` in this figure. In the second case figure 4 part (2), the code generation takes place just before the execution of the kernel, this maps to the kernel initialization. In this case, we want to generate the most optimized code which is specific to one execution with constant injection. The cost of the code generation can be amortized because this compilette uses less instructions than a classical compiler. In the last case figure 4 part (3), code generation can take place several times during the execution and it sets off by a condition on data value. The code generation is driven by the application and especially the execution and results values. In transprecision applications, this ability is very useful to adapt precision according to data value.

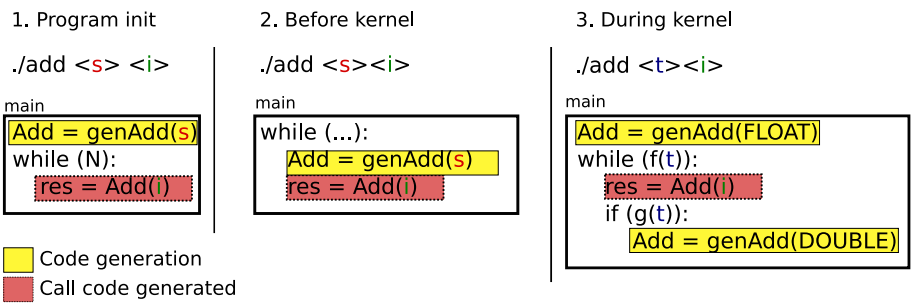

Figure 4. Code generation time where i is the compilette parameters, s corresponds to the parameters to specialize to compilette and t is the threshold which is a condition to re-generate code

4.3. Demonstration Example

To illustrate the three cases described in figure 4, we have chosen three examples: conversion Celcius to Fahrenheit, multiplication with a specialization on a constant value and square root with Newton's algorithm. The latter example demonstrates the advantage of using *HybroGen* for transprecision applications.

4.3.1. Celcius to Fahrenheit

The first demonstration is the conversion of degree Celcius to degree Fahrenheit. This example illustrates the case of an expression of mutliple arithmetics operations. We have chosen this code example for its simplicity. The code of the compilette is described with *HybroLang* language as following:

Listing 1: Celcius to Fahenreit compilette

```
h2_insn_t * genC2F(h2_insn_t * ptr)
{
  #[
  int 32 1 C2F (int 32 1 a)
  {
    int 32 1 r;
    r = a * 9 / 5 + 32;
    return r;
  }
  ]#
  return (h2_insn_t *) ptr;
}
```

In all code examples, the compilette begins with the two symbols #[and finish with]#. Only the compilette is rewrite in C language by *HybroLang* . Other lines, corresponds to the prototype of the C function and the last line is the return of the function.

This example only uses arithmetic operations and we can see that *HybroLang* allows affectation with more than one operator. The function named `genC2F` contains the description of the compilette and returns a pointer to the beginning of the code generated. The compilette `C2F` is called in the main program with different parameters which corresponds to temperature values that we want to convert. This is a typical example of one code generation for several calls to generated code. A more sophisticated version can have data types (int in this example) in parameters and the same compilette can generate code for int, float 16 bits, float 32 bits or whatever. The data value description is very flexible with *HybroLang* .

4.3.2. Multiplication with Specialization

In this part we focus on a small example of using specialization with *HybroLang* , which illustrates a functionality specific to *HybroGen* : injected value at run-time. This code is as small as possible to focus on specialization on data at run-time. We have chosen a compilette which computes a multiplication of a value by a constant. This constant is not known at static compilation time, but only at execution time. *HybroGen* can inject the value, `b` in the following code identified by #(b) in the compilette , during the execution. The generator named `genMult` written using C language and the compilette `mult` described with *HybroLang* are given below:

Listing 2: Multiplication with specialization compilette

```
h2_insn_t * genMult(h2_insn_t * ptr, int  b)
{
  #[
  int 32 1 mult (int 32 1 a)
  {
```

```
    int 32 1 r;
    r = #(b) * a;
    return r;
  }
  ]#
  return (h2_insn_t *) ptr;
}
```

The sentence, which contains `#(b)` is an example of data injection which implements code specialization at run time. The result of this *compilette* is a function which multiplies by the specific constant `b`.

4.3.3. Square Root with Newton's Algorithm

This application is a perfect example of using transprecision for computation. The computation of the square root with Newton's algorithm uses a function for one step of the iteration. At each step, an approximate value of the square root of the value `u` is computed with the formula: $(u + (val/u))/2$ where val is the precision. This function written with *HybroLang* is:

Listing 3: Square root with Newton's algorithm compilette

```
 h2_insn_t * genIterate(h2_insn_t * ptr, int FloatWidth)
{
  #[
    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;
    }
  ]#

  return (h2_insn_t *) ptr;
}
```

At the beginning of the application, float precision is sufficient to compute an approximate result but during the execution if there is no difference between the current and the previous result, then the application generates a new code with better precision. The program stops when a step of the iteration has achieved a result with the required precision. The main program is described below:

Listing 4: Square root with Newton's algorithm main

```
int main(int argc, char **argv)
{
    ...
    fPtr1 = (piff) genIterate (ptr, FLOAT);
    do
    {
        if ((diff < precf) && isFloat)
        { /* Code generation with double for better precision */
            fPtr2 = (pidd) genIterate (ptr, DOUBLE);
            isFloat = False;
        }
        value = next;
        next = (isFloat)?fPtr1(value, af, 2.0):fPtr2(value, af, 2.0);
```

```
        diff = ABS(next - value);
    } while ( isFloat || (!isFloat && (diff > precd)));
}
```

In this code, we can see two calls to genIterate, which is the function responsible for code generation, the first with float precision and the second with double precision. The computation of one step of iterations corresponds to the call of fPtr1 or fPtr2 where `value` maps to the previous result, `af` is the precision that we want to obtain. At the end of the loop, we compute the difference between the current and the previous result to decide if the precision has been reached to change to double precision or to stop the program.

4.4. An example of HybroGen Compilation for Multiplication with Specialization

To detail the different step of *HybroGen* flow, we provide an example of the application `Multiplication with specialization` described previously.

4.4.1. Static Compilation with *HybroLang*

HybroLang compiler transforms the compilette in C Code which is composed of call to generate function. Each instruction of the compilette corresponds to one or more call to generate functions which are in charge to select instructions based on data types and type of each operand. In `Multiplication with specialization`, the main operation is the multiplication, *HybroLang* converts that in a call to power_genMUl_3 or riscv_genMUl_3 respectively for POWER and RISC-V architecture, the number 3 refers to the number of operands because C language does not allow overloaded function. In this example, the first and the second parameters are the same and maps to an integer register with a word size fixed to 32 bits and initialized with value of b to specialize in this code on b. Finally this register contains the result of the multiplication. The third parameter maps to the first register in input which is represented by the variable `a` in the initial code. To summaize, the generation of the code for the multiplication of `a` by `b` to transform on two functions of generation as following:

```
h2_sValue_t a = {REGISTER, 'i', 1, 32, 10, 0};
h2_sValue_t h2_0 = {REGISTER, 'i', 1, 32, 6, 0};
riscv_genMV_2(h2_0, (h2_sValue_t) {VALUE, 'i', 1, 32, 0, (b)});
riscv_genMUL_3(h2_0, h2_0, a);
```

4.4.2. Back-end Code Generation Using Database Request

Generation functions are composed of a conditional structure to select the best instruction. For example, `riscv_genMV_2` is used to select instruction for `move` operation with 2 operands. This function is generated by *HybroLang* as following:

```
  void riscv_genMV_2(h2_sValue_t P0, h2_sValue_t P1){
    if ((P0.arith == 'i') && (P0.wLen <= 32) && (P0.vLen == 1)
    && isRegister(P0) && isRegister(P1)) {
        RV32I_MV_RR_I_32(P0.regNro, P1.regNro);
    }
    else if ((P0.arith == 'i') && (P0.wLen <= 32) && (P0.vLen == 1)
    &&  isRegister(P0) && isValue(P1)) {
        RV32I_MV_RI_I_32(P0.regNro, P1.valueImm);
    }
    else {
                h2_codeGenerationOK = 0;
    }
  }
```

In this function, the first case maps to move operation from register P1 to register P0, the second case corresponds to move operation of integer P1 to register P0. We also generated error messages if there is no operation for this operand. For example, move a float into a register is not possible with this selector function. Functions like `RV32I_MV_RI_I_32` called by selector function write instruction encodage. This macro is generated with SQL request to a database which contains instructions encodage and format for different architectures and variants. The Application Binary Interface (ABI) is also described in the database and requested by *HybroLang* to build c code.

4.4.3. Binary Code Generation at Execution

Finally, the execution of this compilette on RISC-V architecture produces these instructions:

```
0x19008:    ori     t1,zero,3
0x1900c:    mul     t1,t1,a0
0x19010:    mv      t0,t1
0x19014:    mv      a0,t0
0x19018:    ret
```

The same program executed on POWER gives this result:

```
0x4000021260:    li      r15,3
0x4000021264:    mullw   r15,r15,r3
0x4000021268:    addi    r14,r15,0
0x400002126c:    addi    r3,r14,0
0x4000021270:    blr
```

Register `t1` and `r15` respectively for RISC-V and POWER, contain the specialize value: 3 in this execution. This value is multiplying with instructions `mull` and `mullw` by the value in input register `a0` or `r15`. To improve the performances of *HybroGen* , some passes of optimization are needed to reduce to number of move. For example, the result of the multiplication can be stored in t1 or r3, the ouput register for respectively RISC-V and POWER.

The example below shows instructions generated with a specialize value fixed to -5:

```
0x4000021260:    li      r15,-5
0x4000021264:    mullw   r15,r15,r3
0x4000021268:    addi    r14,r15,0
0x400002126c:    addi    r3,r14,0
0x4000021270:    blr
```

A careful reader has noticed that those tutorial codes are not optimal. We know that there is specialized instructions which use constant values, these examples are only to explain the workflow and shows that *HybroGen* could generate multiple binary code from the same compilette.

4.5. Metrics and Evaluation of HybroGen Flow

To evaluate *HybroGen* compilation flow, the number of Lines of Code (LoC) is a good indicator to evaluate the extra cost to port C code to hybrid *HybroLang* and C code. Table 1 presents the number of LoC for three applications and for the different parts of the code. Compilette code is written with *HybroLang* and we can see that it is very small 12 and 14 lines depending on the application. This code is compiled with *HybroLang* which generates C code specific for an architecture. For all the applications and the two targeted architecture, the number of lines of C code generated is between 96 and 284. This difference can be explained by the number of instructions in the database for each architecture and for the semantic instructions and arithmetic use in the application. The latter column corresponds to C code use for management like call to the generators, call to generated code and parameters for management. This code is

the same for all architectures and depends on applications. For these applications the number of lines of code is between 28 and 55.

Applications	**Compilette code (*HybroLang*)**	**Generated code (C)**		**Main without compilette (C)**
		RISC-V	POWER	
Celcius to Fahrenheit	12	179	284	36
Multiplication with specialization	12	96	102	28
Square root with Newton algorithm	14	159	188	55

Table 1: Lines of Code (LoC) of C and *HybroLang* for demonstration applications

5. Discussion and working direction

This article has presented a new compilation infrastructure called *HybroGen* . We have shown that our tool is already working on small examples, which was a challenge in term of compilation chain complexity.

Our technical targeted metrics are (1) code generation speed and (2) code generation size. As we use the same code generation scheme than deGoal [10] we already know that those two metrics are similar and faster and smaller than any JIT compiler.

As scientific targets we want to follow two main objectives which are:

Scientific support for transprecision: we target to support applications containing run-time transprecision and support scientific transprecision applications. This objective is very useful on hardware platforms which contains many floating point representations. For example, RISC-V platform from GreenWaves, the GAP9, has support for floating point variants of 8, 16 and 32 bits. RISC-V standard platform has support for 32 and 64 bits while IBM Power8 platform has support for 32, 64 bits. Those platforms are good candidates.

Compilation support for non Von Neumann architecture: we also support code generation for "in memory computing" devices[13]. On those devices the difficulty comes from the fact that there is two synchronized instruction flows to generate.

This platform is not in the scope of this article.

Metrics to fight with : this article has showed a proof of concept an initial results. We'll continue to improve our *HybroGen* tool and in the future experiments we will focus on other metrics which are :

Speedup for scientific applications : thanks to our run-time optimization we will have speedups that will help scientific applications which need run-time transprecision support. Mainly those whose rely on a residue value that decrease.

Code generation speed : as we can regenerate the binary code very often, it's very important to generate it as fast as possible. *HybroGen* is designed to generate binary code generator which are very fast because our compiler is able to restrict the code generation to the only instructions that are needed by the application.

Code generator size thanks to the previous point, our final code generator are very small (KB order or magnitude), does not rely on external library and can be suited for embedded systems.

The table 2 summarized the current supported platforms.

Our *HybroGen* infrastructure will be open source but is not yet ready for a public release. Nevertheless, it is possible to the *HybroLang* input sources, the output C and

ISA	Instruction set emulator	Hardware
RISCV	qemu-riscv32	GreenWave / Gap9 platform
CSRAM	qemu-riscv32 + In Memory Processing plugin	CEA / RiscV + In MemoryComputing
Power	qemu-ppc64le	IBM / Power8 systems
Kalray	kvx-mppa	Kalray / Coolidge

Table 2: Supported hardware platforms, working both in simulation mode and on hardware platforms

a `Makefile` containing the commands to run the application. The public repository is https://github.com/oprecomp/HybroLang and contains a README which explains how to reproduce the experimentation and run the applications.

6. Conclusion

In this article we have demonstrated the opportunity to break classical compilation static strategies and open the door to make applications auto-adaptive to the context.

We have demonstrated three new code generation scenarios which have binary code generation at run-time in common. The first one shows only binary code generation, the second code specialization at run time and the third shows a code specialization based on transprecision.

Our *HybroGen* infrastructure proof of concept give to the programmer the possibility to control his application and link data parameter to the architecture.

We have shown in this article that those capabilities are useful, does not rely on complex and big JIT infrastructures and the binary code is small and fast.

We continue to extend our *HybroGen* infrastructure and develop demonstrations of its capabilities in two directions: (1) on scientific demonstrators of the transprecision capabilities because it is a challenge for IA applications and (2) on heterogeneity, i.e. the capability to generate binary code at run-time for multiple processors from high performance Power8 up to small RISC-V compute nodes.

Our *HybroGen* infrastructure will be open-source but has not reached a release quality. Nevertheless, we share the code examples version (*HybroLang* , generated C code) in the following repository https://github.com/oprecomp/HybroLang. Running those code allows reproducing the code generation scenarios described in this article.

References

1. Patterson, D. 50 Years of computer architecture: From the mainframe CPU to the domain-specific tpu and the open RISC-V instruction set. 2018 IEEE International Solid - State Circuits Conference - (ISSCC), 2018, pp. 27–31. doi:10.1109/ISSCC.2018.8310168.
2. Paleczny, M.; Vick, C.; Click, C. The java hotspot TM server compiler. Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium-Volume 1. USENIX Association, 2001, pp. 1–1.
3. Press, W.H., Ed. *Numerical recipes: the art of scientific computing*, 3rd ed ed.; Cambridge University Press: Cambridge, UK ; New York, 2007. OCLC: ocn123285342.
4. Consel, C.; Noël, F. A general approach for run-time specialization and its application to C. Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '96; ACM Press: St. Petersburg Beach, Florida, United States, 1996; pp. 145–156. doi:10.1145/237721.237767.
5. Whaley, R.C.; Petitet, A.; Dongarra, J.J. Automated empirical optimizations of software and the ATLAS project q. *Parallel Computing* **2001**, p. 33.
6. Frigo, M.; Johnson, S.G. The design and implementation of FFTW3. *Proceedings of the IEEE* **2005**, *93*, 216–231.
7. Puschel, M.; Moura, J.M.; Johnson, J.R.; Padua, D.; Veloso, M.M.; Singer, B.W.; Xiong, J.; Franchetti, F.; Gacic, A.; Voronenko, Y.; others. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE* **2005**, *93*, 232–275.
8. Park, H.; Kim, S.; Park, J.G.; Moon, S.M. Reusing the Optimized Code for JavaScript Ahead-of-Time Compilation. *ACM Transactions on Architecture and Code Optimization* **2018**, *15*, 1–20. doi:10.1145/3291056.
9. Nuzman, D.; Dyshel, S.; Rohou, E.; Rosen, I.; Williams, K.; Yuste, D.; Cohen, A.; Zaks, A. Vapor SIMD: Auto-vectorize once, run everywhere. Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization. IEEE Computer Society, 2011, pp. 151–160.

10. Charles, H.P.; Couroussé, D.; Lomüller, V.; Endo, F.A.; Gauguey, R. deGoal a tool to embed dynamic code generators into applications. International Conference on Compiler Construction. Springer, 2014, pp. 107–112.
11. Malossi, A.C.I.; Schaffner, M.; Molnos, A.; Gammaitoni, L.; Tagliavini, G.; Emerson, A.; Tomás, A.; Nikolopoulos, D.S.; Flamand, E.; Wehn, N. The transprecision computing paradigm: Concept, design, and applications. 2018 Design, Automation Test in Europe Conference Exhibition (DATE), 2018, pp. 1105–1110. doi:10.23919/DATE.2018.8342176.
12. Bellard, F. QEMU, a fast and portable dynamic translator. USENIX annual technical conference, FREENIX Track. Califor-nia, USA, 2005, Vol. 41, p. 46.
13. Noel, J.P.; Pezzin, M.; Gauchi, R.; Christmann, J.F.; Kooli, M.; Charles, H.P.; Ciampolini, L.; Diallo, M.; Lepin, F.; Blampey, B.; Vivet, P.; Mitra, S.; Giraud, B. A 35.6 TOPS/W/mm^2 3-Stage Pipelined Computational SRAM With Adjustable Form Factor for Highly Data-Centric Applications. *IEEE Solid-State Circuits Letters* **2020**, *3*, 286–289. doi:10.1109/LSSC.2020.3010377.