Journal of
Low Power Electronics and Applications

Article

Dynamic Compilation for Transprecision Applications on Heterogeneous Platform

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

Univ. Grenoble Alpes, CEA, LIST, F-38000 Grenoble, France; henri-pierre.charles@cea.fr (H.-P.C.); Kevin.Mambu@cea.fr (K.M.); Maha.Kooli@cea.fr (M.K.)
* 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 performance is becoming data-dependent. The proposed experimental platform is helpful in programming applications that can be reconfigured at run time in order to be adapted for a new data environment. The *HybroGen* platform is adapted to heterogeneous architectures and can generate instructions for different targets. This platform allows to go farther than classical JIT compilation in many directions: the code generator is smaller by three orders of magnitude and faster by three orders of magnitude, compared to JIT (Just-In-Time) platforms, and allows making code transformation that is impossible in traditional compilation schemes, such as code generation for non von Neumann accelerators or dynamic code transformations for transprecision. The latter is illustrated in a code example: the square root with Newton's algorithm. We also illustrate the proposed *HybroGen* platform with two other examples: a multiplication with a specialization on a value determined at run time, and a conversion of degrees Celsius to degrees Fahrenheit. This article presents a proof of concept of the proposed *HybroGen* platform in terms of its functionalities, and demonstrates the working status.

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

Citation: Dumas, J.; Charles, H.-P.; Mambu, K.; Kooli, M. Dynamic Compilation for Transprecision Applications on Heterogeneous Platform. *J. Low Power Electron. Appl.* **2021**, *11*, 28. https://doi.org/10.3390/jlpea11030028

Academic Editors: Francesco Barchi and Andrea Acquaviva

Received: 5 May 2021
Accepted: 25 June 2021
Published: 29 June 2021

1. Introduction

Compilation and code generation are 50- year-old research domains, parallel to the computer architecture research domain [1]. Compilers have the difficult task of transforming 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 evolutions go in opposite directions.

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 create a very rich programming environment that 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 the differences between the data to compute on and the hardware capabilities.

A first solution to deal with this problem is to delay the code generation at run time by using JIT (Just-In-Time) compilation in Java. The hotspot compiler [2] is efficient and compiles on demand but is based on method count, not on a 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. They take care of the performance by using efficient compilers and use 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 is interesting to do the computation with reduced precision, which allows computing faster, thanks to the reduction in the memory bandwidth pressure, and switching to an improved precision at the end. This simple scheme is very complex to set up practically.

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

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

We present *HybroGen* , a new software environment that allows to experiment binary code generation at run time. We detail the compilation flow and the different code generation steps. We also show three demonstrations examples to experiment and prove the capabilities of our tool: a conversion of degrees Celsius to degrees Fahrenheit, a multiplication with specialization on data fixed at run time, and finally, the computation of the square root with Newton's algorithms.

The article is composed of he following. In Section 2, we introduce some other compilation approaches. Then, Section 3 presents the compilation objectives and also explains the targeted compilation scenarios. Section 4 illustrates how the proposed 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 existing compilers 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 coming from C. Consel in the 1990s [4]. However, at that time, the underlying hardware was simpler in terms of memory hierarchy and ALU capability.

The library approach in [5] proposes a software emulation which cannot use real hardware.

The AIR approach [6] uses a similar approach based on software functions which do not generate code at run time.

Difference from *HybroGen* : real implementation, generates binary code at run time to use hardware accelerators.

2.2. Install Time

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

ATLAS in 2001 [7] 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 [8] for FFT implementations and SPIRAL for linear algebra kernels [9].

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

Difference from *HybroGen*: we generate binary code at runtime, not only choosing the most efficient version, but a gain in code size.

2.3. High Level Intermediate Representation

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

JavaScript also uses a JIT compilation strategy, but both Java and Javascript have a very costly compilation phase as described in [10].

A similar approach is described in this Vapor SIMD article [11] but no practical implementation is proposed.

Difference from *HybroGen*: real implementation and code generation directly based on data values, not on statistical method count and gain in code transformation speed.

2.4. deGoal

Another attempt was made with the deGoal tool [12]. This tool allows implementing similar compilation scenarios. The programming language is portable across similar SIMD architectures but is at assembly level, which makes complex applications difficult to implement.

Difference from *HybroGen*: implementation based on a programming language, not assembly level: gain in expressivity and code generation quality.

Those main approaches are summarized in Table 1 where the tool activity is shown depending on the "compilation time".

Table 1. State of the art comparison of compilation scenarios, and comments on the different transformation steps. The green X shows where the tool has an activity.

Tool Name	**Compilation Time**		
	Install	**Static**	**Execution Time**
Atlas [7]	X: Archi. testing		
FFTW [8]	X: Code eval.:		X: Code selectors
Consel [4]		X: No actual implem.	
VaporSIMD [11]		X: agnostic vectors	X: No actual implem.
Java [2]		X: agnostic bytecode	X: Method count
deGoal [12]	X: ISA description	X: Assembly level	
HybroGen	X: ISA description	X: Generate generators	X: Data dep. Code gen.

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 [13] is explored by the H2020 European project OPRECOMP [14].

The idea is to reduce energy consumption by using approximate computing. For example, the precision can be decreased by using small float, i.e., 8-bit or 16-bit, floating point numbers. The precision is adapted during the computation with the 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 reduced value.

Several articles present work around transprecision computing, such as FlexFloat [5], AIR [6] or the work in [15]. For FlexFloat, Grosser et al. describe the C/C++ library for transprecision computing. AIR presents an algorithmic approach to use transprecision.

These proposals address the challenge of transprecision computing during the application development. One of the characteristics of transprecision is the fact that it is adapted 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 knows when applications move toward 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, such as vectors, data with varying length and multiple arithmetics, integers, floats, complex numbers, pixels, IP addresses, etc.

3.2.1. Compiler: When Should Code Generation 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 times. Data values are resolved at run time and with only one step of compilation before the execution, some optimizations cannot be applied.

Usually, programmers want to write simple code with good performance on all computers. This is not the case in real life. In fact, programmers need to specialize code for a specific architecture. Moreover, programmers develop different versions of the code to adapt programs to data types, such as 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 experimentation on multiple scenarios of *code generation time*.

3.2.2. Run-Time Code Generation Scenarios

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

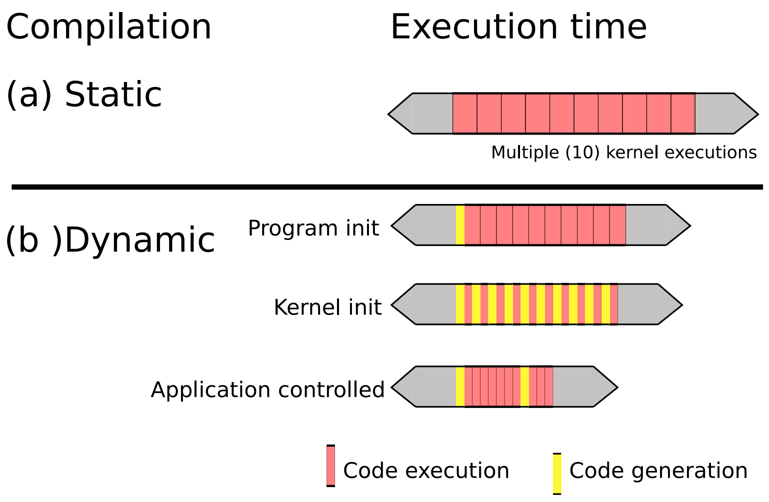

Figure 1. Chronograms of different compilation scenarios. On the top, the static compiler (**a**), without any code specialization at run time. On the bottom, our proposition for dynamic compilation (**b**) with three scenarios: binary code specialization at program init, kernel init or application controlled.

When the data set is not known at the compilation time, whatever the static compilation time devoted to the kernel compilation, the compiler should be conservative and cannot 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 the dynamic compilation in Figure 1b. We list the following code generation scenarios:

- **Program init:** The code generation takes place at the beginning of the application and minimal knowledge makes 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 is so rich that the generated code is very efficient, thanks to the optimizer contained in the code generator. The code generation time can be amortized at each kernel call.
- **Application driven:** In some situations, the application has a knowledge of the context, and the programmer wants to have 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 are illustrated on some tutorial examples in Section 4.

3.2.3. Language Features

In this paper, we propose *HybroLang* as a new language with syntax close to the 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 in triplicate: type, vector size and word size. This language is used only to describe the part of kernel that we want to optimize; we name this part a *compilette*. The other part of the program is written with the language targeted. In this paper, we chose C language, but we can imagine other languages such as JavaScript or Python.

3.2.4. Data and Code Generation Interleaving

The main characteristics of our *HybroGen* environment are the following:

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

Those characteristics are demonstrated in the later examples.

3.3. HybroGen *Platform*

HybroGen is composed of four 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** (Figure 2a), the description of the instruction set architecture (ISA) is stored, 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 as described previously. Our compiler *HybroGen* implements different passes of compilation, such as 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** (Figure 2c) such as `gcc` or `clang`. In this paper, we used `gcc`. Finally, depending on the scenario choice, at **execution time** (Figure 2d), the code of the compilette is executed, which generates the instructions that are executed at the backend.

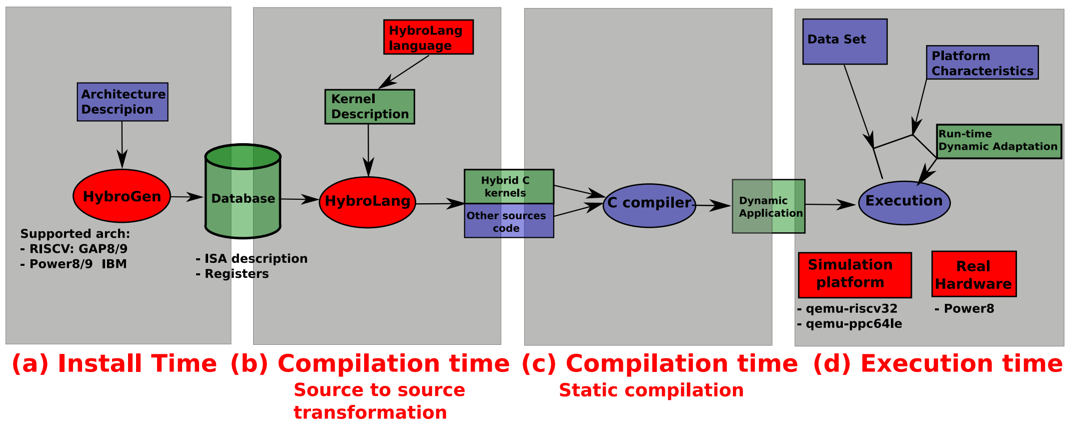

Figure 2. Overview of *HybroGen* platform with four steps: (**a**) install time, (**b**) compilation source-to-source transformation, (**c**) compilation source-to-binary transformation and (**d**) execution time

4. Demonstration of *HybroGen* for Transprecision Applications

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

4.1. Experimental Platform

To run the demonstrations, a system-level simulator and real platform are used. Qemu [16] is used to simulate POWER8 and RISC-V architectures. This simulator exists for different architectures, such as x86, MIPS or ARM, and supports different variants. For example, there is a version for RISC-V 32 bits and another for the 64-bit 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 shows the different steps of our compilation chain from Figure 2b, which rewrites the *HybroLang* section of the application to a C version that is able to generate the multiple binary version. This capability to generate the multiple binary version is very important in multiple contexts: adapt to hardware characteristics, data set parameter and, as we focus on this article, data set precision.

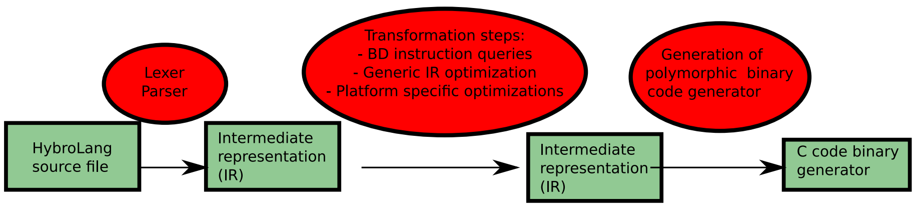
Lexer
Parser
Transformation steps:
- BD instruction queries
- Generic IR optimization
- Platform specific optimizations
Generation of polymorphic binary code generator
HybroLang source file
Intermediate representation (IR)
Intermediate representation (IR)
C code binary generator

Figure 3. HybroLang compilation chain. Contains classical compilation steps—lexer and parser, generic and specific IR optimizations— and also specific ones, using an SQL database to store instruction specifications and C code generation, which act as the polymorphic binary code generator at run time.

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 executed more than once, *N* times in the figure, to amortize the cost of dynamic compilation. In Figure 4, we can see that parameter of the execution, i.e., `i`, is not used for the code generation, but it can be used in the 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, and 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 fewer instructions than a classical compiler. In the last case, Figure 4 part (3), code generation can take place several times during the execution; it sets off by a condition on the data value. The code generation is driven by the application and especially by the execution and results values. In transprecision applications, this ability is very useful to adapt precision according to the data value.

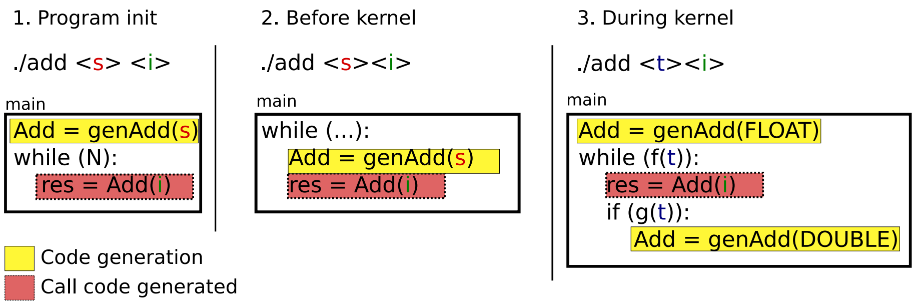

Figure 4. Code generation time where *i* is the compilette parameters, *s* corresponds to the parameters to specialize to the compilette and *t* is the threshold which is a condition to re-generate code.

4.3. Demonstration Example

To illustrate the three cases described in Figure 4, we chose three examples: conversion of Celsius 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. Celsius to Fahrenheit

The first demonstration is the conversion of degrees Celsius to degrees Fahrenheit. This example illustrates the case of an expression of multiple arithmetic operations. We chose this code example for its simplicity. The code of the compilette is described with *HybroLang* language in Listing 1.

Listing 1. Celsius 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 finishes with]#. Only the compilette is rewritten in C language by *HybroLang* . Other lines correspond to the prototype of the C function, and the last line is the return of the function.

This example only uses arithmetic operations; 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 correspond 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, and so on. 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* . It illustrates a specific functionality to *HybroGen* : injected value at run time. This code is as small as possible to focus on specialization of data at run time. We select a compilette which computes a multiplication of a value by a constant. This constant is not known at the static compilation time, only at the 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 in Listing 2.

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 provided in Listing 3.

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, the 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 achieves a result with the required precision. The main program is described in Listing 4.

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, and `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 was reached to either 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*

The *HybroLang* compiler transforms the compilette in C code, which is composed of a call to generate function. Each instruction of the compilette corresponds to one or more call to generate functions, which are in charge of selecting instructions based on data types and the type of each operand. In `multiplication with specialization`, the main operation is the multiplication; *HybroLang* converts that into 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 an overloaded function. In this example, the first and the second parameters are the same, and map to an integer register with a word size fixed to 32 bits and initialized with a 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 the input, which is represented by the variable `a` in the initial code. To summarize, the generation of the code for the multiplication of `a` by `b` to transform on two functions of generation is as follows:

```
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 the `move` operation with 2 operands. This function is generated by *HybroLang* as follows:

```
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 the move operation of integer P1 to register P0. We also generate error messages if there is no operation for this operand. For example, moving a float into a register is not possible with this selector function. Functions such as `RV32I_MV_RI_I_32` are called by the selector function to write instruction encoding. This macro is generated with a SQL request to a database which contains instructions for encoding and the 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 the following 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
```

Registers `t1` and `r15`, respectively for RISC-V and POWER, contain the specialized value: 3, in this execution. This value is multiplied with instructions `mull` and `mullw` by the value in the input register `a0` or `r15`. To improve the performance of *HybroGen*, some passes of optimization are needed to reduce the number of move instructions. For example, the results of the multiplication are stored in t1 or r3, the output registers for, respectively, RISC-V and POWER.

The example below shows instructions generated with a specialized 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 are specialized instructions that use constant values; these examples are only to explain the workflow and show that the proposed *HybroGen* compiler environment is able to generate multiple binary codes from the same compilette.

4.5. Metrics and Evaluation of *HybroGen* Flow

To evaluate the *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 2 presents the number of LoC for the three experimented applications and for the different parts of the code. The compilette code is written with *HybroLang* and we can see that it is very small: 12 to 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 in particular the two targeted architectures, 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: the semantic instructions and arithmetic use in the application. The latter column corresponds to C code used for the management, such as 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 LoC is between 28 and 55.

Table 2. Lines of Code (LoC) of C and *HybroLang* for demonstration applications.

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

5. Discussion and Working Direction

This article presents a new compilation infrastructure called *HybroGen* . We have shown that our tool is already working on small examples, which are a challenge in terms 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 as deGoal [12], we already know that those two metrics are similar to, faster than and smaller than any JIT compiler.

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

- **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 contain many floating point representations. For example, the RISC-V platform from GreenWaves, GAP9, has support for floating point variants of 8, 16 and 32 bits. The RISC-V standard platform has support for 32 and 64 bits, while the IBM Power8 platform has support for 32 and 64 bits. Those platforms are good candidates.
- **Compilation support for non von Neumann architecture:** We also support code generation for "in memory computing" devices [17]. On those devices, the difficulty comes from the fact that there are two synchronized instruction flows to generate. This platform is not in the scope of this article.

This article showed a proof of concept of the initial results. We will continue to improve our *HybroGen* tool, and in future experiments, we will focus on other metrics which are as follows:

- **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 who rely on a residue value that decreases.
- **Code generation speed:** As we can regenerate the binary code very often, it is very important to generate it as fast as possible. *HybroGen* is designed to generate the binary code generator which is 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 is very small (KB order of magnitude), does not rely on an external library and can be suited for embedded systems.

Table 3 summarizes the current supported platforms.

Table 3. Supported hardware platforms, working both in simulation mode and on hardware platforms.

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

Our *HybroGen* infrastructure will be open source, but it is not yet ready for public release. Nevertheless, it is possible to take the *HybroLang* input sources, the output C and a `Makefile` containing the commands to run the application. The public repository is https://github.com/oprecomp/HybroLang (accessed on 28 June 2021) and contains a README which explains how to reproduce the experimentation and run the applications.

6. Conclusions

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

We 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 allows code specialization at run time and the third shows a code specialization based on transprecision.

Our *HybroGen* infrastructure proof of concept gives to the programmer the possibility to control their application, and links the data parameter to the architecture.

We showed in this article that those capabilities are useful, do 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 release quality. Nevertheless, we share the code example version (*HybroLang* , generated C code) in the following repository: https://github.com/oprecomp/HybroLang (accessed on 28 June 2021). this code allows reproducing the code generation scenarios described in this article.

Author Contributions: Conceptualization, J.D., H.-P.C., K.M. and M.K.; methodology, all authors; software, all authors; validation, all authors; formal analysis, all authors; investigation, all authots; resources, all authors; data curation, all authots; writing—original J.D., H.-P.C.; writing—review and editing all authors.; visualization, all authors; supervision, all authors; project administration, H.-P.C.; funding acquisition, H.-P.C. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the EU H2020 project 732631 "OPRECOMP". Open Transprecision Computing.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Input source codes and generated files can be found on https://github.com/oprecomp/HybroLang (accessed on 28 June 2021).

Conflicts of Interest: The authors declare no conflict of interest.

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. In Proceedings of the 2018 IEEE International Solid—State Circuits Conference— (ISSCC), San Francisco, CA, USA, 11–15 February 2018; pp. 27–31. doi:10.1109/ISSCC.2018.8310168.. [CrossRef]
2. Paleczny, M.; Vick, C.; Click, C. The java hotspot TM server compiler. In Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium, Monterey, CA, USA, 23–24 April 2001; Volume 1.
3. Press, W.H. (Ed.) *Numerical Recipes: The Art of Scientific Computing*, 3rd ed.; Cambridge University Press: Cambridge, UK; New York, NY, USA, 2007; ISBN 978-0-521-88068-8.
4. Consel, C.; Noël, F. A general approach for run-time specialization and its application to C. In Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages—POPL '96, St. Petersburg Beach, FL, USA, 21 January 1996; ACM Press: New York, NY, USA, 1996; pp. 145–156. doi:10.1145/237721.237767. [CrossRef]
5. Tagliavini, G.; Marongiu, A.; Benini, L. Flexfloat: A software library for transprecision computing. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2018**, *39*, 145–156. [CrossRef]
6. Lee, J.; Peterson, G.D.; Nikolopoulos, D.S.; Vandierendonck, H. AIR: Iterative refinement acceleration using arbitrary dynamic precision. *Parallel Comput.* **2020**, *97*, 102663. [CrossRef]
7. Whaley, R.C.; Petitet, A.; Dongarra, J.J. Automated empirical optimizations of software and the ATLAS project q. *Parallel Comput.* **2001**, doi:10.1016/S0167-8191(00)00087-9 [CrossRef]
8. Frigo, M.; Johnson, S.G. The design and implementation of FFTW3. *Proc. IEEE* **2005**, *93*, 216–231. [CrossRef]
9. Puschel, M.; Moura, J.M.; Johnson, J.R.; Padua, D.; Veloso, M.M.; Singer, B.W.; Xiong, J.; Franchetti, F.; Gacic, A.; Voronenko, Y.; et al. SPIRAL: Code generation for DSP transforms. *Proc. IEEE* **2005**, *93*, 232–275. [CrossRef]
10. Park, H.; Kim, S.; Park, J.G.; Moon, S.M. Reusing the Optimized Code for JavaScript Ahead-of-Time Compilation. *ACM Trans. Archit. Code Optim.* **2018**, *15*, 1–20. doi:10.1145/3291056. [CrossRef]
11. Nuzman, D.; Dyshel, S.; Rohou, E.; Rosen, I.; Williams, K.; Yuste, D.; Cohen, A.; Zaks, A. Vapor SIMD: Auto-vectorize once, run everywhere. In Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization, Chamonix, France, 2–6 April 2011; IEEE Computer Society: Washington, DC, USA, 2011; pp. 151–160.
12. Charles, H.P.; Couroussé, D.; Lomüller, V.; Endo, F.A.; Gauguey, R. deGoal a tool to embed dynamic code generators into applications. In Proceedings of the International Conference on Compiler Construction, London, UK, 11–18 April 2014; Springer: Berlin/Heidelberg, Germany, 2014, pp. 107–112.
13. 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. In Proceedings of the 2018 Design, Automation Test in Europe Conference Exhibition (DATE), Dresden, Germany, 19–23 March 2018; pp. 1105–1110. doi:10.23919/DATE.2018.8342176. [CrossRef]
14. OPRECOMP. 2021. Available online: http://oprecomp.eu (accessed on 28 June 2021).
15. Grosser, T.; Theodoridis, T.; Falkenstein, M.; Pitchanathan, A.; Kruse, M.; Rigger, M.; Su, Z.; Hoefler, T. Fast linear programming through transprecision computing on small and sparse data. *Proc. ACM Program. Lang.* **2020**, *4*, 1–28. [CrossRef]
16. Bellard, F. QEMU, a fast and portable dynamic translator. In Proceedings of the USENIX Annual Technical Conference, FREENIX Track, Anaheim, CA, USA, 10–15 April 2005; Volume 41, p. 46.
17. Noel, J.P.; Pezzin, M.; Gauchi, R.; Christmann, J.F.; Kooli, M.; Charles, H.P.; Ciampolini, L.; Diallo, M.; Lepin, F.; Blampey, B.; et al. 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 Lett.* **2020**, *3*, 286–289. doi:10.1109/LSSC.2020.3010377. [CrossRef]