



HAL
open science

Dynamic Compilation for Transprecision Applications on Heterogeneous Platform

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

► **To cite this version:**

Julie Dumas, Henri-Pierre Charles, Kévin Mambu, Maha Kooli. Dynamic Compilation for Transprecision Applications on Heterogeneous Platform. *Journal of Low Power Electronics and Applications*, 2021, 11 (3), <https://doi.org/10.3390/jlpea11030028>. 10.3390/jlpea11030028 . cea-03313560

HAL Id: cea-03313560

<https://cea.hal.science/cea-03313560>

Submitted on 4 Aug 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution 4.0 International License

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 illustrate the proposed *HybroGen* platform with two others examples: a multiplication with a specialization on a value determine at run-time and a conversion 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:

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2021 by the authors. Submitted to *J. Low Power Electron. Appl.* for possible open access publication under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Compilation and code generation¹ 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 the European H2020 FET project OPRECOMP under Grant 732631

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

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

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

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

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

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

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

67 2. Related Works

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

71 2.1. Code Specialization

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

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

77 2.2. Install Time

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

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

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

86 2.3. High Level Intermediate Representation

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

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

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

97 2.4. deGoal

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

102 3. Compiler Level Support for Transprecision

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

105 3.1. Transprecision and Challenges for Compilers

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

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

117 3.2. HybroLang: a New Language

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

122 3.2.1. Compiler: When Code Generation Should Arise ?

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

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

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

134 3.2.2. Run-time Code Generation Scenarios

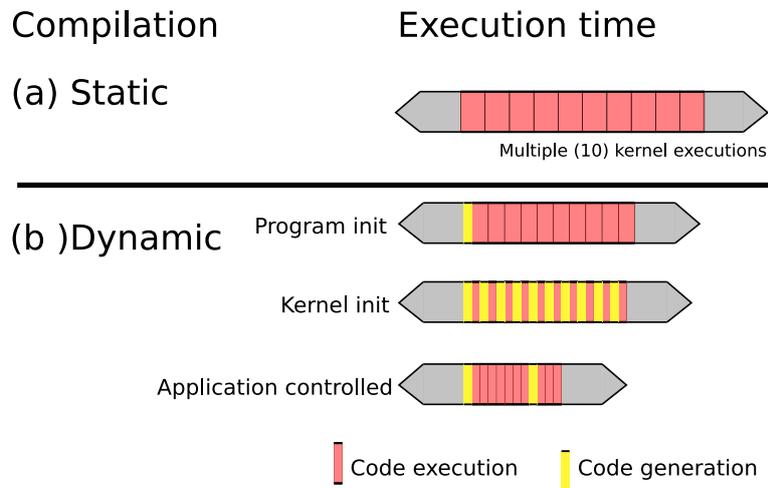


Figure 1. Chronograms of different compilation scenarios. On the top the static compiler (a), without any code specialization at run time. On the bottom our proposition for dynamic compilation (b) with 3 scenarios: binary code specialization at program init, kernel init or application controlled

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

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

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

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

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

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

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

158 3.2.3. Language

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

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

167 3.2.4. Data and Code Generation Interleaving

168 The main characteristics of our *HybroGen* environment are:

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

178 Those characteristics will be demonstrated in the later examples.

179 3.3. *HybroGen* Platform

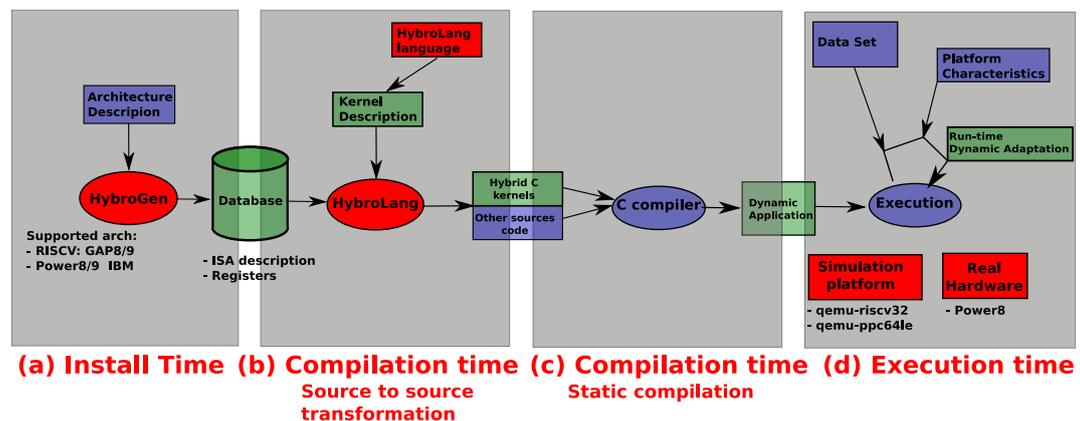


Figure 2. Overview of *HybroGen* platform with 4 steps: (a) install, (b) compilation source to source, (c) compilation source to binary and (d) execution

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

193 4. Demonstration of *HybroGen* for Transprecision Applications

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

196 4.1. Experimental Platform

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

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

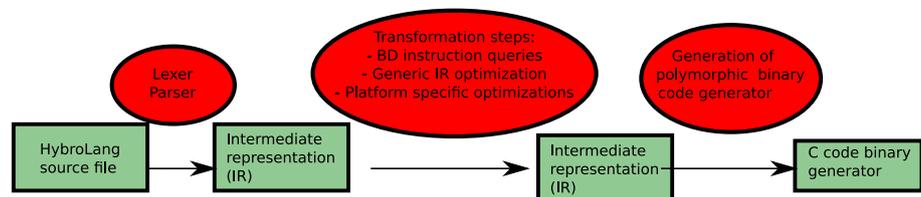


Figure 3. HybroLang compilation chain. Containing classical compilation steps: lexer and parser, generic and specific IR optimizations and also specific ones: using a SQL data base to store instructions specifications and C code generation, which will act as polymorphic binary code generator at run-time

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

209 4.2. Application Scenarios of HybroGen

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

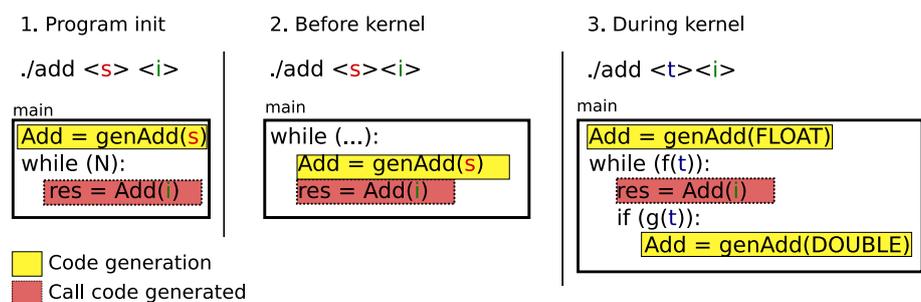


Figure 4. Code generation time where i is the compilette parameters, s corresponds to the parameters to specialize to compilette and t is the threshold which is a condition to re-generate code

227 4.3. Demonstration Example

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

232 4.3.1. Celcius to Fahrenheit

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

Listing 1: Celcius to Fahrenheit compilette

```

237 h2_insn_t * genC2F(h2_insn_t * ptr)
238 {
239     #[
240     int 32 1 C2F (int 32 1 a)
241     {
242         int 32 1 r;
243         r = a * 9 / 5 + 32;
244         return r;
245     }
246     ]#
247     return (h2_insn_t *) ptr;
248 }

```

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

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

261 4.3.2. Multiplication with Specialization

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

Listing 2: Multiplication with specialization compilette

```

270 h2_insn_t * genMult(h2_insn_t * ptr, int b)
271 {
272     #[
273     int 32 1 mult (int 32 1 a)
274     {

```

```

275     int 32 1 r;
276     r = #(b) * a;
277     return r;
278 }
279 ]#
280 return (h2_insn_t *) ptr;
281 }

```

282 The sentence, which contains #(b) is an example of data injection which imple-
 283 ments code specialization at run time. The result of this *complete* is a function which
 284 multiplies by the specific constant b.

285 4.3.3. Square Root with Newton's Algorithm

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

Listing 3: Square root with Newton's algorithm complete

```

291 h2_insn_t * genIterate(h2_insn_t * ptr, int FloatWidth)
292 {
293     #[
294     flt #(FloatWidth) 1 iterate(flt #(FloatWidth) 1 u,
295     flt #(FloatWidth) 1 val, flt #(FloatWidth) 1 div )
296     {
297         flt #(FloatWidth) 1 r, tmp1, tmp2;
298         tmp1 = val / u;
299         tmp2 = u + tmp1;
300         return tmp2 / div;
301     }
302     ]#
303
304     return (h2_insn_t *) ptr;
305 }

```

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

Listing 4: Square root with Newton's algorithm main

```

311 int main(int argc, char **argv)
312 {
313     ...
314     fPtr1 = (piff) genIterate (ptr, FLOAT);
315     do
316     {
317         if ((diff < precf) && isFloat)
318         { /* Code generation with double for better precision */
319             fPtr2 = (pidd) genIterate (ptr, DOUBLE);
320             isFloat = False;
321         }
322         value = next;
323         next = (isFloat)? fPtr1(value, af, 2.0): fPtr2(value, af, 2.0);

```

```

324         diff = ABS(next - value);
325     } while ( isFloat || (!isFloat && (diff > precd)));
326 }

```

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

334 4.4. An example of *HybroGen* Compilation for Multiplication with Specialization

335 To detail the different step of *HybroGen* flow, we provide an example of the appli-
336 cation `Multiplication with specialization` described previously.

337 4.4.1. Static Compilation with *HybroLang*

338 *HybroLang* compiler transforms the compilette in C Code which is composed of
339 call to generate function. Each instruction of the compilette corresponds to one or
340 more call to generate functions which are in charge to select instructions based on data
341 types and type of each operand. In `Multiplication with specialization`, the main
342 operation is the multiplication, *HybroLang* converts that in a call to `power_genMUI_3` or
343 `riscv_genMUI_3` respectively for POWER and RISC-V architecture, the number 3 refers
344 to the number of operands because C language does not allow overloaded function. In
345 this example, the first and the second parameters are the same and maps to an integer
346 register with a word size fixed to 32 bits and initialized with value of `b` to specialize in
347 this code on `b`. Finally this register contains the result of the multiplication. The third
348 parameter maps to the first register in input which is represented by the variable `a` in the
349 initial code. To summarize, the generation of the code for the multiplication of `a` by `b` to
350 transform on two functions of generation as following:

```

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

```

355 4.4.2. Back-end Code Generation Using Database Request

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

```

359 void riscv_genMV_2(h2_sValue_t P0, h2_sValue_t P1){
360     if ((P0.arith == 'i') && (P0.wLen <= 32) && (P0.vLen == 1)
361     && isRegister(P0) && isRegister(P1)) {
362         RV32I_MV_RR_I_32(P0.regNro, P1.regNro);
363     }
364     else if ((P0.arith == 'i') && (P0.wLen <= 32) && (P0.vLen == 1)
365     && isRegister(P0) && isValue(P1)) {
366         RV32I_MV_RI_I_32(P0.regNro, P1.valueImm);
367     }
368     else {
369         h2_codeGenerationOK = 0;
370     }
371 }

```

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

380 4.4.3. Binary Code Generation at Execution

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

```
383 0x19008 :    ori    t1 , zero , 3
384 0x1900c :    mul    t1 , t1 , a0
385 0x19010 :    mv     t0 , t1
386 0x19014 :    mv     a0 , t0
387 0x19018 :    ret
```

388 The same program executed on POWER gives this result:

```
389 0x4000021260 :    li     r15 , 3
390 0x4000021264 :    mullw r15 , r15 , r3
391 0x4000021268 :    addi  r14 , r15 , 0
392 0x400002126c :    addi  r3 , r14 , 0
393 0x4000021270 :    blr
```

394 Register `t1` and `r15` respectively for RISC-V and POWER, contain the specialize value:
 395 3 in this execution. This value is multiplying with instructions `mull` and `mullw` by the
 396 value in input register `a0` or `r15`. To improve the performances of *HybroGen*, some
 397 passes of optimization are needed to reduce to number of move. For example, the result
 398 of the multiplication can be stored in `t1` or `r3`, the output register for respectively RISC-V
 399 and POWER.

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

```
401 0x4000021260 :    li     r15 , -5
402 0x4000021264 :    mullw r15 , r15 , r3
403 0x4000021268 :    addi  r14 , r15 , 0
404 0x400002126c :    addi  r3 , r14 , 0
405 0x4000021270 :    blr
```

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

410 4.5. Metrics and Evaluation of *HybroGen* Flow

411 To evaluate *HybroGen* compilation flow, the number of Lines of Code (LoC) is a
 412 good indicator to evaluate the extra cost to port C code to hybrid *HybroLang* and C
 413 code. Table 1 presents the number of LoC for three applications and for the different
 414 parts of the code. Compilette code is written with *HybroLang* and we can see that it is
 415 very small 12 and 14 lines depending on the application. This code is compiled with
 416 *HybroLang* which generates C code specific for an architecture. For all the applications
 417 and the two targeted architecture, the number of lines of C code generated is between
 418 96 and 284. This difference can be explained by the number of instructions in the
 419 database for each architecture and for the semantic instructions and arithmetic use in
 420 the application. The latter column corresponds to C code use for management like call
 421 to the generators, call to generated code and parameters for management. This code is

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

Applications	Compiletime code (<i>HybroLang</i>)	Generated code (C)		Main without compiletime (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

423

424 5. Discussion and working direction

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

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

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

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

439 **Compilation support for non Von Neumann architecture:** we also support code gen-
440 eration for “in memory computing” devices[13]. On those devices the difficulty
441 comes from the fact that there is two synchronized instruction flows to generate.

442 This platform is not in the scope of this article.

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

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

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

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

458 The table 2 summarized the current supported platforms.

459 Our *HybroGen* infrastructure will be open source but is not yet ready for a public
460 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 Memory Computing
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

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

464 6. Conclusion

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

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

471 Our *HybroGen* infrastructure proof of concept give to the programmer the possibil-
 472 ity to control his application and link data parameter to the architecture.

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

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

480 Our *HybroGen* infrastructure will be open-source but has not reached a release
 481 quality. Nevertheless, we share the code examples version (*HybroLang*, generated C
 482 code) in the following repository <https://github.com/oprecomp/HybroLang>. Running
 483 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*. California, 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/mm² 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.