Software Acceleration of Floating-point Multiplication using Runtime Code Generation

– Student Paper –

Charles Aracil and Damien Couroussé
CEA, LIST, Laboratoire Infrastructures Atelier Logiciel pour Puce (LIALP)
Email: firstname.surname@cea.fr

***Abstract*—Floating-point units are seldom in highly constrained systems, due to silicon and energy footprint, but emulated instead in algorithms based on integer arithmetic. In this paper, we use runtime code generation to generate outperforming flexible and optimized floating-point routines. On a Texas Instrument MSP430 fitted with only 512 bytes of RAM, we achieved mean speedups of 1032 % and 52 %, with tuning features enabling peaks up to 2012 % and 64 %, respectively for floating-point multiplication and an applicative case. At the best of our knowledge, runtime code generation was never achieved with such few computing and memory resources.**

I. Introduction

Embedded systems typically exhibit features far from those of general purpose computing systems. The instruction set is reduced to the basis and memories (either flash and RAM) can be several orders of magnitude under those of general purpose computing systems. In order to satisfy cost, silicon surface and energy requirements, they use simpler hardware architecture, involving more stress on software algorithms. Because of their silicon and energy footprint, floating-point units (FPU) are either seldom or reluctantly included in specific branches of embedded systems wherein energy efficiency prevails on computation speed. Sensor Networks are a prime example of such branch, because they are composed of minimalist nodes disconnected from any power outlet, dedicated to specific and relatively simple tasks that cannot afford heavy architectures, and because they prevail cost and energy autonomy on raw computation power.

There are however cases where one cannot afford a dedicated hardware, and where the acceleration of floating-point processing is still desirable. This is the main motivation for the work presented in this paper.

If the target processor lacks a FPU, the static compiler selects software emulation for floating-point processing. These emulation routines have a strong impact on performance, because the processing of mantissa and exponent is performed with integer arithmetic, and rounding operation is then necessary to comply with the IEEE-754 floating-point representation. CPU architectures less than 32 bits, which still compose the major part of sensor network nodes, turn these routines heavier. They cannot handle easily the 32-bits word length of single-precision floating-point format, further increasing register and memory pressure.

Static compilers are blind concerning the values to be computed, preventing any optimization of runtime values even when they are set just once along program execution. This is the main weakness we tackle in this paper by using runtime code generation. Runtime optimization will exploit the knowledge of the values of ”configuration” variables. Such variables are likely to change along application lifetime and cannot be the target of constant propagation done by a static compilation. However, they are constant over the time of a processing step, which makes them a perfect match for optimization with dynamic code generation.

Given the a priori knowledge, at runtime, of a part of the floating-point data involved in computation, we aim at generating *ad hoc* floating-point routines optimized for the data already known at the time of runtime code generation. Our approach is based on deGoal, a tool for building fast and portable runtime code generators designed at CEA-LIST. At the best of our knowledge, it is the first time that runtime code generation is achieved on small processors below 32-bit with very limited memory resources: our target platform is the MSP430 microcontroller from Texas Instruments, which offers 16-bit integer arithmetic and 512B of RAM. On this platform, we are thus able to achieve speedups above $10\times$ compared with the standard emulation code produced by gcc. The memory footprint of the runtime-generated emulation routine is configurable, depending on the required precision. We focus on floating-point multiplication, but our results are easy to extend as division by a number is a multiplication by its reciprocal.

The paper is organised as follows: Section II introduces related works, Section III describes our runtime code generation method, Section IV presents the experimental setup and the results obtained on floating-point multiplication. Section V illustrates deGoal ability to fairly improve runtime performance in a real case application.

II. Related Works

Dynamic compilation and runtime code generation have been a long-standing topic in computer science papers. Famous examples of dynamic compilation are Just-In-Time compilation (JIT) used in Java virtual machines [1] and in the LLVM framework [2]. Such approaches intend to be fully automatic without requiring extra effort from application developers. The drawback of this genericity lies in a heavyweight memory footprint and the need for important computation resources, which is not compatible with the resources available on constrained embedded systems such as nodes in sensor networks.

Other approaches try to alleviate runtime code generation by exploiting the knowledge of the set of applicative cases to handle. Static compiler produces a specialized runtime code generator with a limited scope, but able to produce highly optimised machine code with a low memory footprint overhead. Consel at al. have applied this approach to specialize interpreters [3]. Our approach is similar to deferred compilation as practiced by Leone et al. [4]: deGoal embeds runtime code generators in an application. Each code generator is statically compiled and highly specialized for a dedicated processing algorithm and a target computing architecture. This way, we are able to achieve runtime code generation on targets that are out of reach of traditional runtime code generation techniques.

There are numerous papers about optimization of floating-point processing, either at algorithmic level [5] or implementation level [6]. A lot of papers cover the implementation of floating-point processing in dedicated accelerators [7] or in FPGAs [8]. At the best of our knowledge, it is the first time runtime code generation was used to accelerate floating-point computation.

III. Experimental Framework

A. Code Generation Framework

We used the runtime code generation tool deGoal [9] to build binary code generators. Within the deGoal environment, we use a specific vocabulary:

a) kernel: A kernel is a small portion of code part of a larger application which is most of the time under strong performance constraints. In our case it is the floating-point multiplication function.

b) compilette: The compilette is the runtime code generator that generates the binary code of the kernel.

c) seed: The seed is the runtime-known data required by the compilette to generate the optimized function. Actually, the seed can consist in one or several input values for the compilette.

The development and execution process of applications using deGoal is dispatched between static and run time. At static time, the main application is developed as usual while the compilette is developed using deGoal specific tools. Then the entire application, including both *main code* and *compilette*, is linked using the platform toolchain and loaded into the platform. At runtime, the program starts and the seed becomes known. The compilette is called and creates an optimized binary code based on the knowing of this seed. Finally, this binary kernel can be invocated by the application like any other function.

In the context of constrained embedded systems, the code generated could be stored either in RAM, which is more accessible, or in flash in order to alleviate RAM pressure, especially with the gap in size between both memories. However, the write endurance is limited in flash technologies to tens of thousands, preventing developers from using it for test purpose. Yet it could be used by a fully-fledged application re-generating code at "reasonable" frequency in order to free RAM space, keep runtime code safe from crash or any feature

```
typedef kernel (float (f*)(float);
kernel mul_M(float);

float fmul(float M, float X);
  /* M is known; we generate the binary
  code for the multiplication kernel */
  mul_M = compilette(M);

  /* X is known,
  result = mul_M * X */
  return(mul_M(X));
}
```

Fig. 1. A sketch of the runtime generation of a kernel for floating-point multiplication, in pseudo C code

developers could think of. In the case of our experimental setup, we have chosen to store all generated code in RAM. This was a particularly challenging constraint because our target platform offered only 512 bytes of RAM.

Embedded systems powered by a fluctuating source of energy such as solar panels are also a perfect match for runtime code generation as it enables temporal relocation of CPU charge. More explicitly, deGoal can generate energy-saving code when the source of energy is functional and use this efficient code when it is not, in order to lengthen the on-battery system lifetime.

B. Runtime Code Generation in Floating-Point Multiplication

The compilette generates a specialised kernel for floating-point multiplication tuned for the first operand of the multiplication. We generate a libgcc-like function but optimizing each step of floating-point multiplication (unpacking, mantissa multiplication, exponent addition, renormalization, rounding and repacking) for the runtime-known operand `M`.

In particular, mantissa multiplication undergoes a complete overhaul based on a polynomial root approximation method known as Horner scheme [10]. We won't explain the mathematical background of Horner scheme, just how it can be applied to our problem [11]. Let be $A = (a_i)_{i\in[0..22]}$ the mantissa of the seed and X the mantissa of the unknown operand. In order to process $A \times X$, we construct the sequence $(c_i)_{i\in[0..N-1]}$, N being the number of set bits in A, containing the distances from right to left between a set bit and its closest next one. Then we compute the terms of the sequence $(X_i)_{i\in[0..N]}$ described in Equation 1, so that X_N will be the result of the multiplication.

$$(X_i)_{i\in[0...N]} \left\{ \begin{array}{lcll} X_0 & = & X & \\ X_i & = & X_{i-1} >> c_{i-1} + X & \forall i \in [1 \ldots N-1] \\ X_N & = & X_{N-1} >> c_{N-1} & \end{array} \right. \quad (1)$$

Actually, other algorithms could have been run to further improve optimization, such as repeating-pattern recognition and Common Subexpression Elimination (CSE). However, they turned out to be too greedy compared with either overhead recovering or memory footprint. The implementation of the Horner scheme using deGoal is described in Algorithm 1.

ALGORITHM 1: Floating-Point Multiplication with Horner scheme

```
Input: Floating-point operands $M$ and $X$ to be multiplied ($M$ is
       known, $X$ is unknown).
Output: The result $M \times X$ of the multiplication.
$i \leftarrow 0$;
$detection \leftarrow 1$;
$result \leftarrow X$;
while not (mantissa(M) && $(1 \ll i)$) do
    $i \leftarrow i + 1$;
end
for $i \leftarrow i + 1$ to len(mantissa(M)) do
    if mantissa(M) && $(1 \ll i)$ then
        $result \leftarrow (result \ll detection) + X$;
        $detection \leftarrow 1$;
    end
    else
        $detection \leftarrow detection + 1$;
    end
end
$result \leftarrow (result \ll detection)$;
return result
```

C. Adaptation of kernel performance vs. precision

In many applications, computation speed and energy efficiency prevail on precision. Short computation chains with limited propagation of errors, critical energy-saving, or computation based on inaccurate sensors, are prime examples of such case. Using mantissa truncation, our algorithm enables to adjust precision according to the desired performance: the less mantissa is treated, the less accurate but the fastest the function will be. In this article, a X-bits truncation denotes a discard of $23-X$ bits, namely only X bits of the mantissa are processed. In order to measure the error introduced by truncation, we use percent error [12]. Percent error is defined by Equations 2, $\hat{x}$ is the approximate value resulting from the processing of our kernel, and x the theoretical floating point value, namely the value computed by the standard algorithm.

$$p_{err} = \frac{|\hat{x} - x|}{|x|} \times 100 \qquad (2)$$

IV. Runtime Performance

A. Target Architecture

We used the microcontroller MSP430-G2553 from Texas Instruments, an ultra-low power 16-bit RISC microprocessor with 27 core instructions and 7 addressing modes. The development platform is fitted with 16 kB of flash and only 512 bytes of RAM. We chose such as minimal platform to illustrate how fair results can be obtained on minimalist architectures with limited memory resources, because they are often used in specific technologies prevailing autonomy and number of units over raw computation power, such as sensor networks. A full cross-compilation toolchain is provided for compiling, debugging and flashing the program into flash memory. We use is the `gcc` toolchain `msp430-gcc` version 4.6.3. All our examples have been compiled using `-Os -fdata-sections -ffunction-sections --gc-sections` options. Option `-O3` has been tested against `-Os` to ensure that `-Os` gives equal or better results than `-O3`.

B. Setup for the Measurement of Execution Times

We describe now the configuration of the MSP430 in order to perform measurements of execution times. We set up clock frequency to 1 Mhz and turned off the watchdogs. A timer is launched at the same frequency than the CPU so that it can provide a cycle-accurate value of time. This accuracy has been checked by setting events on GPIO and monitoring their activity with an oscilloscope. A UART driver performs communication with a computer.

For our experimental setup to be valid, we needed to ensure there is no difference in execution times between code residing in RAM and code residing in flash, because our optimized code is generated into RAM, while the reference code from gcc we bench against is executed from flash. Indeed, programs can be executed in-place from flash memory on our target platform [13]. The clock frequency of the microcontroller is set to 1 MHz so that memory fetches from flash are hidden in one processor cycle.

By recoding manually exact copies of gcc routines such as integer multiplication and division, we confirmed that execution times from RAM and from flash were the same.

We also measured that kernel execution times never varies with same data entries. Indeed, our platform has no OS, virtualization support, branch predictors, cache memories and so on. Interrupts are disabled during kernel executions, so that interruption routines used in the platform, such as UART handlers, would not interfere with our measures of the execution time.

C. Performance Metrics

We base our results on three metrics.

number of generated instructions: The number of generated instructions is monitored to ensure that our algorithm isn't too greedy and to ensure lower memory footprint (than standard library algorithm) and no memory overflow.

speedup: Speedup is the ratio between execution times of standard library algorithm and ours. An effective optimization shall reveal a speedup greater than 1.

overhead recovering: Runtime code generation incurs an execution time overhead, so it makes sense only when saved time overcomes lost time. The overhead recovering is the number of optimized code executions required to actually overcome the time spent in code generation.

To formalize these notions (Equations 3, 4), t_{gen}, t_{lib} and t_{dyn} denote the execution time of respectively code generation, standard algorithm and optimized algorithm. N denotes overhead recovering. We assume here $t_{lib} > t_{dyn}$ as there would be no point in generating a function having same or lower performance than gcc implementation.

$$speedup = \frac{t_{lib}}{t_{dyn}} \quad (3) \qquad N = \frac{t_{gen}}{t_{lib} - t_{dyn}} \quad (4)$$

Actually, the number of generated instructions is a tricky metric, because it depends on precise application constraints.

	min	max	mean	SD
nb of instructions	38	132	106.4	9.38
ratio (%)	56.33	73.58	68.88	51.13
speedup	9.29	20.12	10.32	1.09
overhead recovering	3.28	3.58	3.46	0.04

Fig. 2. Floating-Point Multiplication Results

deGoal memory footprint is the sum of dynamically generated code (the multiplication routine) *and* statically loaded code (the compilette). Runtime code generation has a limited application spectrum, as overhead recovering requires seeds to be unchanged for some executions. As a result, compilette might not substitute standard library algorithms but act as a sidekick triggered when input data can afford optimization. Thus, a relevant metric is closely tied to the application requirements. If generated instructions are to be stored in RAM, and flash is big enough not to worry about, then the number of generated instructions is a fair metric and we target a memory footprint shrinkage. On the contrary, if the overall memory footprint including both flash and RAM is under heavy constraints then a reasonable memory footprint overhead is at stake. Thenceforward, we will focus both on the raw number of generated instructions and the memory footprint overhead. Considering this overhead as a shrinkage (for standard library substitution) or an increase (for standard library support) is up to the developer. This memory footprint overhead is clarified in Equation 5, where $nbinsn$ denotes the number of generated instructions, $scompilette$ the compilette size, $slibgcc$ the code size of standard library floating-point multiplication and $ratio$ the percentage of our code compared to standard library.

$$ratio = \frac{scompilette + nbinsn}{slibgcc} \times 100 \quad (5)$$

D. Runtime efficiency

To measure the performance of our algorithm we processed four hundreds multiplications of two randomly-picked operands, one of them being the seed of the partial application. We processed three special seeds corresponding to the worst, average and best case scenarios, respectively being full-set, a half-set and an empty mantissa. Thus, best and worst case will provide extreme measures, namely minimum and maximum values. For each multiplication, we recorded the operand, the number of instructions generated, the time of generation and static/dynamic execution (see Section IV-C for definition) so that we could deduce speedup and overhead recovering. Table 2 describes the results obtained for the experimentation. For each metric, we focused on two sets of indicators.

minimum and maximum: Because of small memory sizes, too heavy code could result in stack overflows, since our kernels are currently generated in RAM. Extreme values are monitored to prevent such issues.

mean and standard deviation (SD): They are standards in statistical analysis, providing a fair first idea of the result distribution.

E. Performance of the generated kernels

We measure the variation of error with truncation from 0 to 23 bits on 4000 samples per truncation level (Figure 3(a)). The mean of speedup and kernel size are measured too (Figure 3(b)).

Besides, we extract from bytecode static sizes needed to evaluate memory footprint: the compilette size is 284 bytes long while the code size of standard library floating-point multiplication is 544 bytes long, leaving 260 bytes free for runtime-generated code before deGoal memory footprint exceeds standard library floating-point multiplication memory footprint.

Our dynamic algorithm accelerates computation up to 2012 % without any loss of accuracy relative to standard library results, and produces a binary code size ($compilette + multiplication$) averaging 69 % the standard algorithm size.

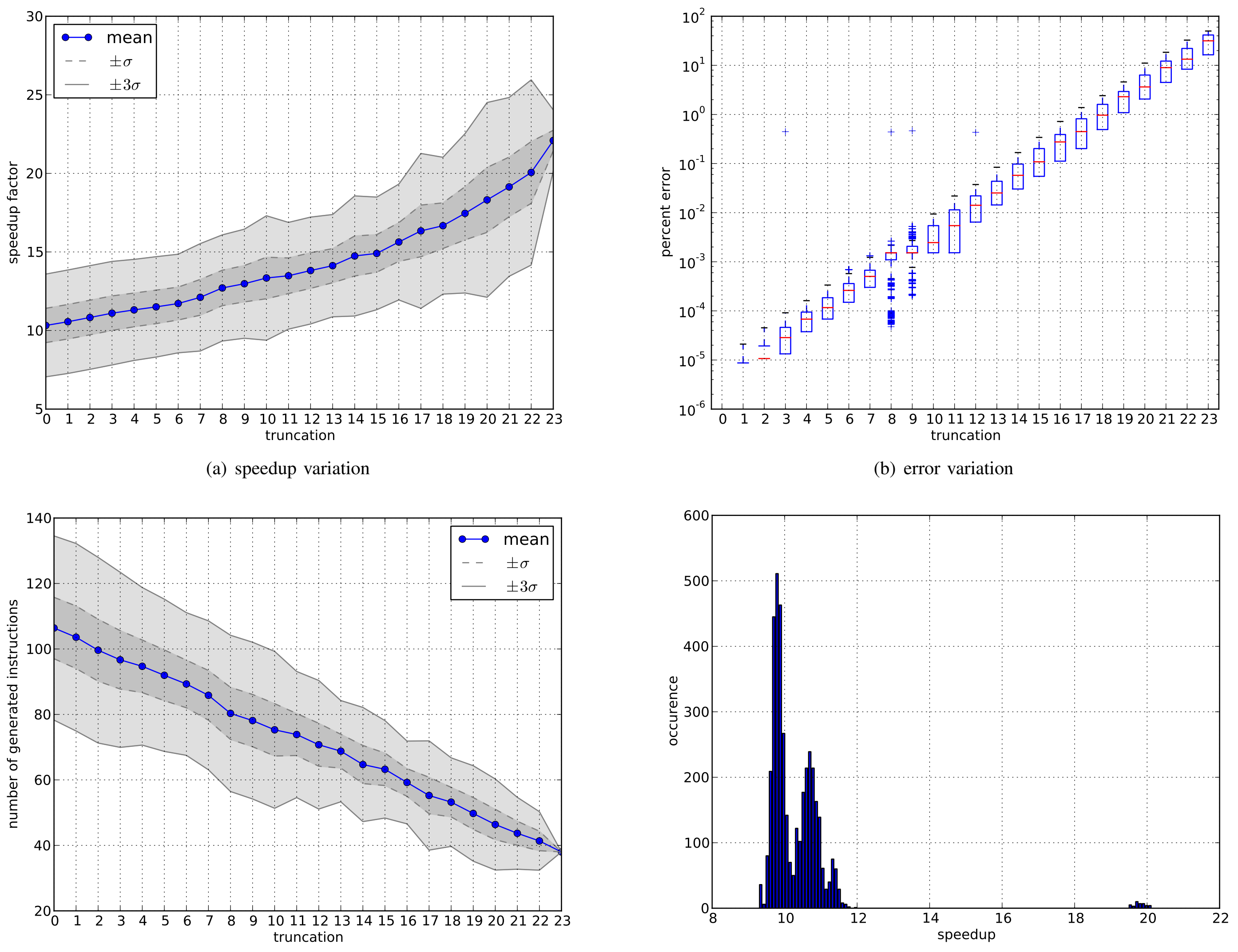
Considering $error \leq 2^{-t}$ where t is the position of the most significant truncated bit, and $log_b(x) = \frac{log_a(x)}{log_a(b)}$, the error quite follows a linear tendency in $(x, log_{10}(y))$ space (see Figure3(b)) [12].

We can see that the mantissa can be truncated up to 7 bits without much negative impact on accuracy (less than 1 % of error), the upper quartile being at 0.2 %). Meanwhile, this truncation more than halves the number of generated instructions and reduces the mean size ratio by more than 10 %. As for speedup, 4-bit truncation enhanced its mean value up to 1500 %.

F. Discussion

Our algorithm clearly gives better results in all aspects of performance: speedup, energy efficiency and memory footprint (in the understanding discussed in Section IV-C). The low overhead recovering enables to use it easily every time an operand remains constant for more than three executions. Moreover, performance is never going down standard implementation's one because *a*) standard deviation assures major part of values are close to the mean; *b*) even extreme values (minimum and maximum) give fair performance features; *c*) energy efficiency is assured: optimization saves battery life by reducing the number of instructions the CPU executes. Indeed, memory and stack accesses are costly in energy and cycles, but associated instructions are rarer in our generated code than in standard library, and our code is lighter, so it necessarily heads to better energy efficiency.

Mean and standard deviation provide a fair idea of reachable performance, but it is entirely relevant only for normal law distribution. We observe a normal distribution for the number of generated instructions (not detailed in this paper for the sake of brevity). Overhead recovering is in the interval $[3.28, 3.58]$: the extra computation cost required by the runtime code generation is always reached after only 4 executions of the generated kernel. As for speedup, we can see in Figure 3(d) that the distribution is not centered on the mean, as it reveals local optima around 9.76, 10.66, 11.30 and 19.6. In the worst case scenario, speedups around 9.76 are the likeliest. Performance tuning further improves reachable speedup and memory footprint cuts providing low percent error results so



(a) speedup variation

(b) error variation

(c) size variation

(d) speedup distribution

Fig. 3. Performance metrics for the kernel tuning of floating-point multiplication

that the design can perfectly match with variable performance requirements.

V. Real case : FIR optimization

A. Motivation

Previous sections highlight achievable improvement on floating-point multiplication. However, an application seldom includes a unique operation. This section introduces a real case of application, focusing on a low-pass Infinite Impulse Response filter implementation (IIR filter).

Wireless sensor networks are primitive systems on both software and hardware matters. They are mainly used for retrieving and sending raw data from sensors to a base station. As they increasingly become widespread, developers now want them to perform a few processing in order to reduce the cost of over-the-air communication [14]. Data filtering, erroneous measure handling and data encryption are prime examples of such new needs.

An inherent problem of physical measurement is error injection. Sensor inaccuracy, algorithmic errors such as collisions on shared resources, biased scheduling, overflows, and spatial/temporal site perturbation (fluctuating energy supply, intense electromagnetic field, vibrations and extreme conditions) are possible origins of such errors. As a result, these data must be filtered to provide clean and treatable signals (data cleaning). For our purpose, we choose a low-pass IIR filter, which interest shall appears on noised signal filtering.

B. Target platform

We ran our experiment on a Zolertia Z1, a platform heavily used in sensor networks and fitted with a MSP430-F2617,

precision	min	mean	max	SD
maximal	7.4	50.5	57.7	9.3
average	12.0	52.3	59.7	9.3
minimal	34.7	56.3	64.0	5.3

Fig. 4. Speedup of optimized IIR filter

a second generation model of MSP430 processors. Memory resources are far more comfortable than those of its predecessors, as they include 8 kB of RAM and no less than 92 kB of FLASH. Combining both advantages of low-power consumption and fair hardware architecture (including for instance a 16×16 integer hardware multiplier), the MSP430-F2617 enables the design of smart applications for the Internet of Things (IoT).

An operating system is often embedded into the platform. ContikiOS is an open source operating system designed for tiny, low-cost, battery-operated and low-power systems. It provides memory allocation, full IP networking, 6LowPan, hardware platform handling, protothreading and a bunch of useful tools for developers such as UART communication drivers and real-time timers for performance measuring.

C. Low-pass filter

The general pattern of a IIR filters is described in Equation 6.

$$H(z) = \frac{\sum_{i=0}^{N} a_i z^{-i}}{\sum_{i=0}^{M} b_i z^{-i}} \tag{6}$$

Consequently, next sample processing mainly consists in addition/subtraction, multiplication and memory accesses. Multiplication code will be optimized for coefficients $(a_i)_{i \in [0..N]}$ and $(b_i)_{i \in [0..M]}$, providing a flexible filter whose coefficients can be set directly at runtime to adapt various sets of data. Such a runtime-generated filter actually becomes a multi-purpose filter enable to respond several requirements without any need of dedicated hardware or hard software reconfiguration.

D. Results

For our experiment, we chose the filter described in Equation 7. It features a bandwidth up to 500 Hz with a notch over 1 kHz, a 2dB mitigation in bandwidth and 20dB mitigation in notch band. We gave this filter a signal composed of two sinusoids in addition, a 250 Hz one with an amplitude of 1, and a 1.5 kHz with an amplitude of 0.5. The sample frequency is 4 kHz. The resulting filter is explained in Equation 7, and underwent the optimization process for all possible accuracy levels.

$$H(z) = \frac{0.0408z + 0.1224z^{-1} + 0.1224z^{-2} + 0.0408z^{-3}}{-1.2978z + 0.7875z^{-1} - 0.1632z^{-2}} \tag{7}$$

The speedup experimental results (Figure 4) focus on minimal, average and maximal accuracy to highlight speedup boundaries. Optimization is always successful and fits with our previous discussion (see Section IV-F). Focusing on mean values, our optimized application reaches a 50 % speedup. Because of the significant time spent out of the kernel, the precision variation doesn't trigger a dramatic improvement in speedup values. However, such slight extra savings in time and energy can benefit developers considering targeted platforms are long-term autonomous systems, meaning any saving could have a substantial impact on the overall life expectancy.

As highlighted in Section IV-E, the number of generated instructions is drastically decreased by mantissa truncation, more than halving dynamic code size. With no truncation, generated code is 973 bytes long. With truncation, it decreases to 357 bytes.

Thus, our application example illustrates how our tool can adapt both application/platform memory and energy specifications.

VI. Summary

This article presents a new algorithm for floating-point arithmetic leveraging runtime code generation. We use our tool called deGoal to build at runtime light and fast routines for software FPU emulation. We are able to improve performance up to 2012 % producing a code half the size of the FPU emulation algorithm of gcc. On a real application, our tool demonstrates speedup up to 50 % and introduces features enabling to tune performance according to the specific requirements of the application.

References

[1] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox, "Design of the java hotspot client compiler for java 6," *ACM TACO*, no. 1, May 2008.

[2] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO'04*, 2004.

[3] S. Thibault, C. Consel, J. Lawall, R. Marlet, and G. Muller, "Static and dynamic program compilation by interpreter specialization," *Higher-Order and Symbolic Computation*, vol. 13, no. 3, pp. 161–178, 2000.

[4] M. Leone and P. Lee, "Dynamic specialization in the fabius system," *ACM Comput. Surv.*, vol. 30, 1998.

[5] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres, *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.

[6] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," 1991.

[7] A. W. Brown, P. H. J. Kelly, and W. Luk, "Profile-directed speculative optimization of reconfigurable floating point data paths."

[8] C. H. Ho, C. W. Yu, P. Leong, W. Luk, and S. J. E. Wilton, "Floating-point fpga: Architecture and modeling," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 17, no. 12, pp. 1709–1718, 2009.

[9] D. Couroussé and H.-P. Charles, "Dynamic code generation: An experiment on matrix multiplication," in *Proceedings of the Work-in-Progress Session, LCTES*, 2012.

[10] K. Venkat, "Efficient multiplication and division using msp430," TexasInstrument, Application Report SLAA329, 2006.

[11] D. E.Knuth, *Seminumerical Algorithms Vol.2, The Art of Computer Programming*. Addison Wesley, 2000.

[12] M. Gilli, "Methode numerique," Department of Econometrie, University of Geneve, Application Report, 2006.

[13] T. Instrument, "Msp430x2xx family user's guide," Tech. Rep. Slau144i, January 2012.

[14] P. Gupta and P. R. Kumar, "Critical power for asymptotic connectivity in wireless networks," 1998, pp. 547–566.