Approximate Computing with Runtime Code Generation on Resource-Constrained Embedded Devices

Damien Couroussé, Caroline Quéva, Henri-Pierre Charles
Univ. Grenoble Alpes, F-38000 Grenoble, France
CEA, LIST, MINATEC Campus
F-38054 Grenoble, France
Email: firstname.lastname@cea.fr

***Abstract*—Approximate computing systems aim at slightly reducing the output quality of service, or precision, of a program in order to save computing operations, reduce the execution time and the energy consumption of the system. However, to the best of our knowledge, in all the approximate computing systems presented in the research literature, the implementation of the components that support the approximation is left to the developer.**

In this paper, we describe the implementation of a precision-aware computing library that saves the developer from the implementation of approximated functions. Efficient implementations of the approximated functions are achieved with runtime code generation. Our implementation of runtime code generation is fast and memory-lightweight, and its overhead can is amortised in a few executions of the generated code. We illustrate the performance and the lightness of our implementation on the WisMote, a MSP430-based platform with only 16 kB of RAM and 256 kB of flash memory. When the generated code is specialised on one of the input arguments of the approximated function, we achieve a speedup above $7\times$.

***Index Terms*—approximate; runtime code generation; compilation; code specialization; precision; floating-point; Wireless Sensor Nework**

I. Introduction

Approximate Computing is a powerful emerging concept, currently bringing a lot of interest in research works. As far as we understand it, it covers two issues that share some ideas [1]: (1) how a system can be resilient in the presence of (hardware) errors but still provide correct results, for example when the hardware runs below its lowest supported operating voltage; (2) how to balance the output precision or Quality of Service of a program in acceptable terms in order to improve its energy and performance efficiency.

In this paper, we address this second issue: we strive for ways to adapt the output precision of a program in acceptable limits in order to improve program performance or energy consumption. The research literature presents a fair amount of papers on this topic. Some works propose tools to analyse the quality output of a program under approximation. ASAC [2] performs a sensitivity analysis of a program in order to identify the parts of the program that are the most sensitive to approximation. Green [3] is a framework that computes statistical QoS (Quality of Service) guarantees of a program; the program is annotated with approximation annotations by the developer. Chan et al. [4] perform statistical analysis to determine how the errors propagate in a circuit composed of accurate and approximato modules.

Another approach is to provide to the developer programming paradigms that support approximate computing. Vassiliadis et al. [5] propose a task-based programming model where the developer describes the significance of program parts and their contribution to the global quality result. The annotations are exploited by the runtime, mainly in loop perforation, to increase program performance w.r.t. result approximation. EnerJ [6] provides specific type annotations to the developer, in order to distinguish between approximate and precise computations. The developer can provide approximate implementations of a program component, but the system also supports approximate-aware hardware.

We hence distinguish between the works than aim to determine the behaviour of a program w.r.t approximation, and the works that provide software developers with tools operating at various levels (compilation, runtime, programming languages) to implement approximate-aware systems. There exists an overlap between these two approximate (pun intended) categories of works, and some works can be classified in these two research domains. However, we observe that in all of these works, the implementation of the component in charge of the approximation is always left to the developer, or to the availability of approximate-aware hardware. In our point of view, this remains a lacking corner-stone in order to build approximate systems.

In this paper, we describe a precision-aware computing library for approximate systems. Our library is lightweight enough to be executed on resource constrained embedded systems. Furthermore, precision awareness is achieved by using runtime code generation in order to reduce the runtime overhead to an acceptable bound. To illustrate our words, we provide performance figures of our precision-aware library on an small computing node typical of the platforms found in Wireless Sensor Networks, fitted with only 16 kB of RAM.

II. Runtime Code Specialisation for Approximate Computing

We present in this section a general description of our approach. Our aim is to provide a precision-aware implementation of the function f, an n-ary function that takes as inputs the n arguments $x_1, \ldots, x_n$, and computes the output y (Equation 1).

$$y = f(x_1, \ldots, x_n) \tag{1}$$

The precision-aware implementation of f is described in Equation 2, where p denotes the precision criterion of the function (e.g. number of bits considered in the mantissa of floating-point numbers). In this equation, the notation underlines the fact that the system generates a new implementation of f (f_p) each time a new value of p is set. We could provide an implementation of f that is generic over all the acceptable values of p, but this solution would come at the price of a performance overhead. Instead, our system uses runtime code generation to *specialise* the implementation of f: the machine code of f_p is generated on the fly, at runtime, according to the value of p. The relation with the code generator is described in equation 3, where gen_f is a runtime code generator of f.

$$y = f_p(x_1, \ldots, x_n) \tag{2}$$

$$f_p = \text{gen}_f(p) \tag{3}$$

We can go one step further by exploiting runtime code generation to specialise the implementation of f on part or all of its input arguments. It indeed happens often that some program variables (for example configuration values) keep the same constant value during an important part of the execution time. If the function f is specialised on its $n-1$ input parameters, the code generation is expressed as in equation 4, and the new output value y is computed as in equation 5. We emphasise on the fact that, because of the specialisation, f has a reduced number of arguments (in this case only 1). One intuitively understands that this specialised version is more efficient in terms of execution time and/or energy consumption.

$$f_{p,x_1,\ldots,x_{n-1}} = \text{gen}_f(p, x_1, \ldots, x_{n-1}) \tag{4}$$

$$y = f_{p,x_1,\ldots,x_{n-1}}(x_n) \tag{5}$$

III. An Approximate Library for Arithmetic Computing with Runtime Code Generation

A. Overview

We describe in this section the working principles of our precision-aware library. The library performs the code specialisation of several target functions thanks to runtime code generation, in order to adapt their implementation (in machine code) according to a precision setting, and to the known values of one or several of the input arguments of the functions. The library supports the specialisation of as many approximated functions as required, but we describe the library for the approximated implementation of a function f.

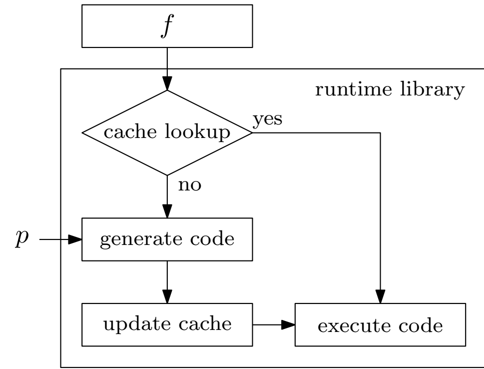


Fig. 1. Overview of the code generation system embedded in the runtime library

We consider the case where the functions supporting approximation are identified a priori, either by the developer or by one of the tools described in the introduction of this paper. The function code generators are added to the library at design time. The code generation system is described below, in section III-B.

Figure 1 illustrates the working principle of our library. Our main concern is to reduce the runtime overhead incurred by code generation: to do so, the specialised functions are stored in a software-managed code cache in order to pay off the cost of runtime code generation. If the specialised code for function f is found in the code cache, it is executed immediately, avoiding the cost of a new runtime code generation. If the specialised code for function f is not found in the code cache, a new code is generated, stored in the code cache, and then executed.

B. deGoal: runtime code generation for embedded devices

Runtime code generation is achieved with deGoal, a tool for embedding runtime code generators, called *compilettes*, in applications [7]. Compilettes can be understood as *ad hoc* runtime code generators specialised to generate the binary (machine) code of a software component. The implementation of the target component is known before runtime, and as a consequence, runtime code generation is fast and code generators present a limited memory footprint. At runtime, when target data for code specialisation are known, the compilette is executed and creates a specialised binary code using the knowledge of the data. The produced binary code is then used the same way as any other function.

IV. Performance evaluation

A. Experimental setup

We use the WisMote platform from Arago Systems [8], which is representative of a low power system with limited memory resources and limited power computation, often used as a node in a Wireless Sensor Network. The WisMote uses the

16-bit MSP430F5437 micro-controller from Texas Instrument, fitted with 256 kB of flash and 16 kB of RAM. We use the open source operating system Contiki version 2.7, and the code is compiled with the gcc toolchain in version 4.7 provided for TinyOS [9]. The clock frequency of the CPU is set to 2.45 MHz, and to measure execution time, we configure an internal timer of the CPU at its maximum frequency (2.45 MHz$/4$). Hence our execution time measurements have a precision of 4 CPU cycles.

All the performance evaluations consider the floating-point multiplication routine. Our reference is the implementation provided in the `libfp` library that comes with the platform gcc toolchain. The implementation of the specialised version generated in our library is optimised using a polynomial root approximation method known as Horner scheme [10]. This method allows for a variable number of precision bits used in the mantissa of the floating-point numbers, and for the specialisation of one of the multiplication operands.

B. Performance metrics

We use two performance metrics representative of the execution cost of our implementation: the speedup, and the overhead recovery. These metrics are based on three performance measurements: t_{ref}, t_{gen}, t_{spec} that respectively denote the execution times of the reference implementation, of the runtime code generator, and of the specialised function.

The *speedup* represents the acceleration factor of our implementation as compared to the reference implementation. It is the ratio between execution cost of the generic application and the specialised application (Equation 6):

$$\text{speedup} = \frac{t_{\text{ref}}}{t_{\text{spec}}} \tag{6}$$

The *overhead recovery*, denoted N, represents the number of executions of the specialised code that are necessary to amortise the cost of code generation (Equation 7).

$$N = \frac{t_{\text{gen}}}{t_{\text{ref}} - t_{\text{spec}}} \tag{7}$$

C. Performance figures

The multiplication routine is generated with a variable precision p that denotes the number of bits of the mantissa of the floating-point numbers taken into account. For 32-bit floating-point values, the mantissa is represented on 24 bits (with one implicit bit). Hence, we vary the precision p in the range $[1; 24]$.

The evaluation uses, for each value of p, of 4000 floating-point multiplications of two sets of randomly-picked operands. Three special values are always included in our sets of input values: 2.0, which has an empty mantissa; 3.3333333, which has an half-set floating-point mantissa; and 3.9999998, which has a full-set mantissa. In the case where we perform code specialisation on the value of the first multiplication operand, these values will respectively correspond to the shortest, average and longest code generation time because the code generation time is proportional to the number of bits set to 1 in the mantissa.

1) Variable precision, no value specialisation: We first describe the performance figures obtained where the multiplication routine is specialised over the precision p only (Figure 2). The performance improvement is modest as compared to the reference implementation. With an equivalent precision ($p = 24$), the speedup is close to 1. This explains why the corresponding values of the overhead recovery are high: considering the minor performance improvements, it requires a high number of executions of the specialised code to amortise the cost of code generation. However, with a lower precision, our specialised version executes faster, up to more than $2\times$ when $p < 6$; the overhead recovery is also acceptable, smaller than 10 for $p < 13$.

2) Variable precision, value specialisation on one of the operands: In this case the runtime library also specialises the multiplication function over one of the operands. Hence, the multiplication function $y = f(a, b)$ is specialised as follows. The code generator specialises the function f over the value of a: $f_a = gen(a)$. The result of the multiplication is then obtained from the execution of $y = f_a(b)$. The performance achieved is illustrated in Figure 3.

As compared to the previous case, the value specialisation on one of the operands brings a considerable performance improvement. With the precision as our reference implementation in the libfp ($p = 24$), our implementation is $7\times$ faster in average, up to $11\times$ for particular specialisation values. Even in the worst case, the speedup is still above $6\times$ faster than the reference implementation. When the precision is lowered, the speedup is even better. The median speedup is around $8\times$ for $p < 17$.

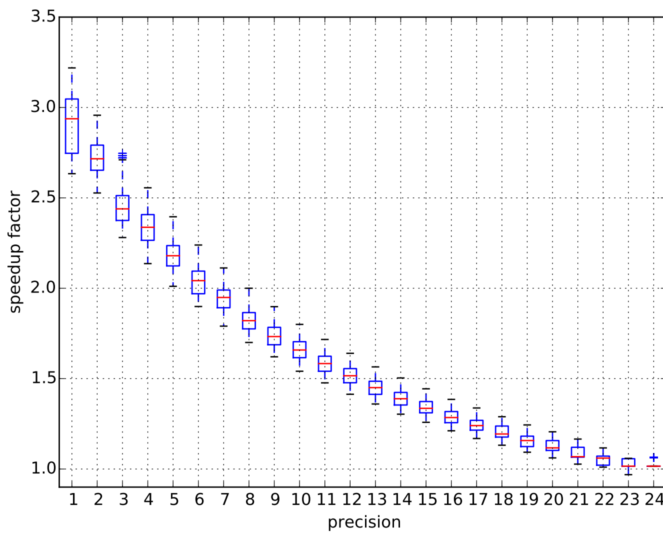
The overhead recovery is variable, depending on the precision setting. This is explained by the fact that the multiplication of the mantissa is unrolled according to the contents of the mantissa of the specialisation value. For $p = 24$, the overhead recovery is always below 5, which means that the cost of runtime code generation is paid-off as soon as the specialised code is executed 5 times. With a lower precision setting, the overhead recovery is even better: the code specialisation is amortised in 3 executions only when $p < 10$.

V. Conclusion

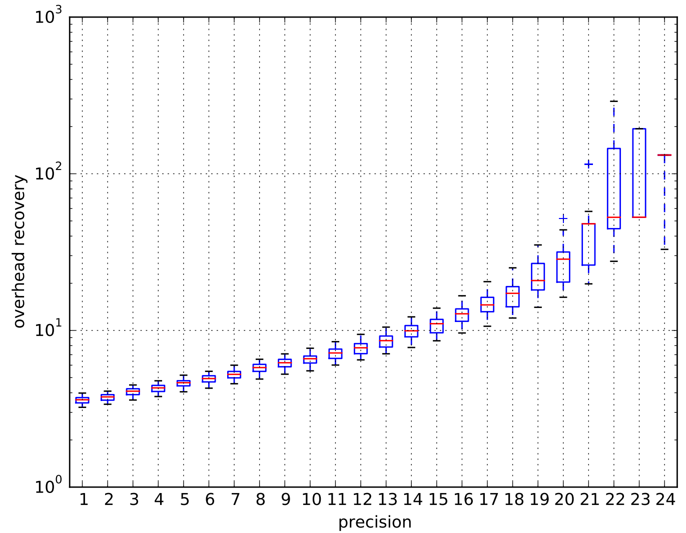
We have presented a precision-aware library for arithmetic computing, suitable even for constrained embedded devices with low computing power and memory resources. The performance figures presented in this paper illustrate that our implementation presents a low runtime overhead, and that it can bring interesting performance improvements as soon as some data are known to keep the same values for a few iterations of the computation. Future works will establish a bridge with higher-level tools that tackle approximation at the semantic level of a program.

Acknowledgments

This work has been partly funded by the Artemis ARROWHEAD project under grant agreement number 332987

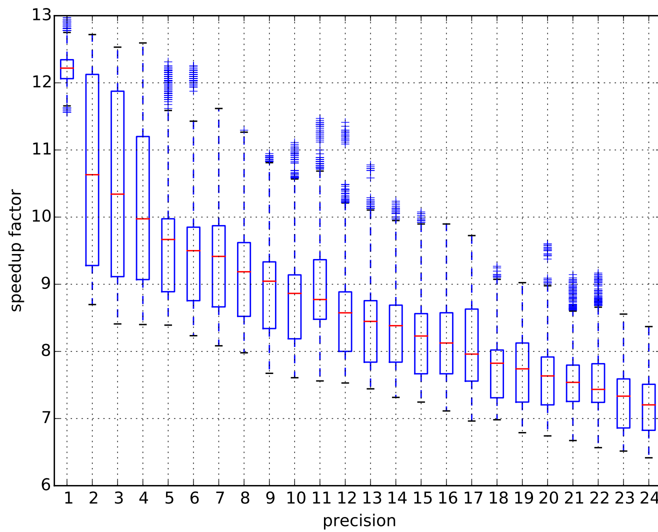


(a) Speedup

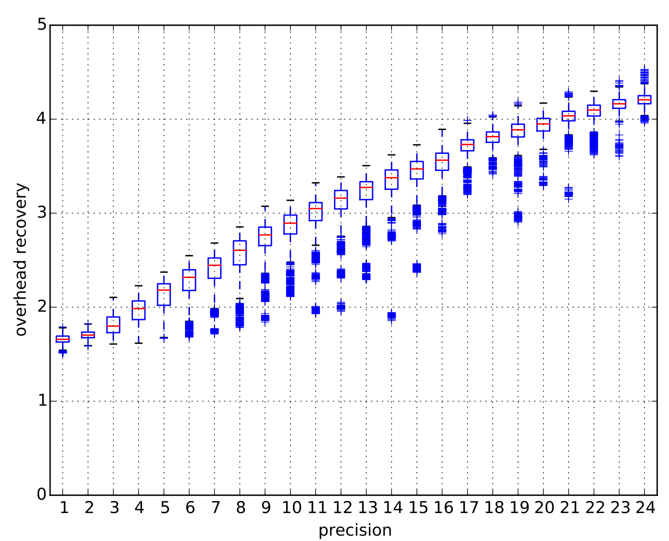


(b) Overhead recovery

Fig. 2. Performance results of our implementation of the floating-point multiplication, specialised over the values of the bit precision only.



(a) Speedup



(b) Overhead recovery

Fig. 3. Performance results of our implementation of the floating-point multiplication, using value specialisation on the first multiplication operand.

(ARTEMIS/ECSEL Joint Undertaking, supported by the European Commission and French Public Authorities).

REFERENCES

[1] J. Han and M. Orshansky, “Approximate computing: An emerging paradigm for energy-efficient design,” in *ETS*, 2013, pp. 1–6.

[2] P. Roy, R. Ray, C. Wang, and W. F. Wong, “Asac: Automatic sensitivity analysis for approximate computing,” *SIGPLAN Not.*, vol. 49, no. 5, pp. 95–104, Jun. 2014.

[3] W. Baek and T. M. Chilimbi, “Green: A framework for supporting energy-conscious programming using controlled approximation,” in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2010, pp. 198–209.

[4] W.-T. Chan, A. Kahng, S. Kang, R. Kumar, and J. Sartori, “Statistical analysis and modeling for error composition in approximate computation circuits,” in *ICCD*, 2013, pp. 47–53.

[5] V. Vassiliadis, K. Parasyris, C. Chalios, C. D. Antonopoulos, S. Lalis, N. Bellas, H. Vandierendonck, and D. S. Nikolopoulos, “A programming model and runtime system for significance-aware energy-efficient computing,” *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2015.

[6] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, “Enerj: Approximate data types for safe and general low-power computation,” *SIGPLAN Not.*, vol. 46, no. 6, pp. 164–174, 2011.

[7] H.-P. Charles, D. Couroussé, V. Lomüller, F. Endo, and R. Gauguey, “deGoal a Tool to Embed Dynamic Code Generators into Applications,” in *Compiler Construction*. Springer, 2014, vol. 8409, pp. 107–112.

[8] Arago-Systems, “Wismote platform,” last visited 2015-11-20, http://www.aragosystems.com/en/wisnet-item/wisnet-wismote-item.html.

[9] “Tinyos (tinyprod) Debian Development Repository,” last visited 2015-11-20, http://tinyprod.net/repos/debian.

[10] C. Aracil and D. Couroussé, “Software acceleration of floating-point multiplication using runtime code generation,” *ICEAC*, pp. 18–23, 2013.