Self-optimisation using runtime code generation for wireless sensor networks

Caroline Quéva, Damien Couroussé, Henri-Pierre Charles

Self-optimisation using runtime code generation for Wireless Sensor Networks

Caroline Quéva, Damien Couroussé, Henri-Pierre Charles
Univ. Grenoble Alpes; F-38000 Grenoble, France
CEA, LIST, Design Architectures and Embedded Software Division, F-38054 Grenoble, France
firstname.lastname@cea.fr

ABSTRACT

This paper addresses the use of runtime code specialisation in resource-constrained embedded systems such as nodes of a Wireless Sensor Network (WSN), in order to improve software efficiency, hence the lifetime of WSN nodes. In our approach, runtime code specialisation is achieved with in-place runtime code generation.

We present a self-optimising system using runtime code generation. Our system is able to automatically make the decision to generate specialised code and use it each time an improvement is observed in application performance. In the Internet of Things (IoT), devices usually have limited precision; our system adapts to theses devices decreasing precision in order to increase performance. We evaluate our system on floating point multiplication using the WisMote platform, where the specialised code executes more than 7 times faster than generic code, all overheads included. To the best of our knowledge, it is the first time that a runtime code generation system is used to automatically optimise code in such constrained devices as WSN nodes.

CCS Concepts

•Software and its engineering → Dynamic compilers; Runtime environments; •General and reference → *Performance;* **•Computer systems organization** → *Embedded software;*

Keywords

Runtime Code Generation, Runtime Code Specialisation, Embedded Software, Self-Optimisation, Compilation, IoT

1. INTRODUCTION

The Internet of Things (IoT) is currently developing, and wireless sensors are more and more encountered in every day life. These systems need to process a growing amount of data and to embed more intelligence despite the fact that their computation capabilities and memory resources are strongly limited. Moreover, the most important characteristic of these low-power sensors is lifetime.

Wireless Sensor Network (WSN) nodes run throughout their lifetime only a few number of applications or services, which configuration settings rarely change during the node lifetime. In this paper, we develop the idea that performing code specialisation of runtime variables (e.g. configuration settings) is a potential source of performance improvement, especially because the overhead of code specialisation is likely to be amortised during the long runtime of a WSN node.

Code specialisation allows to produce more efficient implementations of an application at the expense of loosing generality of use of the specialised code instance. It can be achieved with static techniques (e.g. C++ templates), but in order to specialise code over runtime values, one needs the ability to generate native code *at runtime*. Just-In-Time compilers (JIT) are well-known examples of runtime code generation systems. They however incur a large memory footprint that make them inapplicable to WSN nodes. Furthermore, to the best of our knowledge only a few works have applied code specialisation on runtime data in JITs, for example [6]. Language extensions have been specifically developed for the purpose of code specialisation on runtime data values, for example for C or ML [5, 11, 12] but these works have never been applied to such memory-constrained devices as WSN nodes.

In this paper, we present a lightweight approach that leverages runtime code generation to increase the performance of applications in WSN nodes, hence their lifetime. We extend the idea introduced in [2, 4] showing that runtime code generation is applicable even to constrained embedded devices with low memory resources such as WSN nodes. The limitation of these previous approaches is that a part of the machinery of runtime code generation was (purposefully) exposed to the programmer. As a consequence, the application developer needs a human expertise in runtime code generation. The motivation of the approach proposed in this paper is to hide the management of runtime code specialisation to the application developer. In order to achieve this, we present a generic approach that allows to embed runtime code specialisation in any domain-specific library to improve its performance as compared to a functionally equivalent reference.

The rest of this paper is organised as follows. Section 2 presents the self-optimisation system, Section 3 describes the experimental evaluation, Section 4 describes related works and Section 5 concludes.

ComNet-IoT January 4–7, 2016, Singapore

2. SELF OPTIMISATION SYSTEM

2.1 System Overview

The code specialisation system consists of several elements, each of them dedicated to a specific task, as illustrated in Fig. 1. The system stores the generated code in a software-managed code cache in order to pay off the high cost of runtime code generation by executing the same specialised code several times.

Functions that may be specialised are identified off-line during static compilation step. At runtime when the identified functions are called, the code specialisation systems runs. The first step is to verify in the code cache if the function has already been specialised using a procedure called "lookup function" (*Code cache lookup*, Fig. 1). If the specialised code associated to the function is in the code cache, then it can be used immediately, avoiding the cost of code generation.

If the function has not yet been specialised, an algorithm is used to determine the potential cost saving provided by specialisation (*Decision algorithm*, Fig. 1). If the overall cost of the application is estimated to be increased by code specialisation, then the generic code is executed. However, if the specialisation is estimated to be beneficial regarding the overall cost of the application, then the code generator is used. The specialised code is stored in the code cache to be directly executed if the function is called again. The "lookup function" is then specialised on the new lookup table.

2.2 Code generation tool

Our work is based on deGoal, a tool for building runtime code generators [2]. Using deGoal, the main application is developed at static time as usual. The runtime code generator, called *compilette*, is also developed at static time using deGoal specific tools. Then, at runtime, when data are known, the compilette is called and creates a specialised binary code using the knowledge of the data. This binary code is then used the same way as any other function.

For each application using deGoal, compilettes are written in a dedicated language and, up to now, it was up to the software developer to decide when and how the code generation is triggered. Our system hides deGoal tools and compilettes in libraries and automatically triggers code generation. Software developers have no other task than developing their application as usual.

2.3 Code cache organisation

In our system, several versions of the specialised code can be associated to one original code. Indeed, the code generator uses specific parameters to generate specialised code. These parameters are values unknown during static compilation but known at runtime. For each parameter value encountered during the life of the application, the generated code is different. It's necessary to adapt the code cache in order to deal with those different versions of the specialised code.

To tackle this problem, we use the associativity of the software code cache. For each part of code we want to store in the code cache, all the arguments necessary to the code generator are stored on the same cache line. As shown in Table 1, when a function is specialised, the name of the function, the generated code address and the value of the parameters used for specialisation are stored in the cache line. As the number of parameters used by the code generator is not constant, the number of elements to read is specified at the beginning of the cache line (*Line size*).

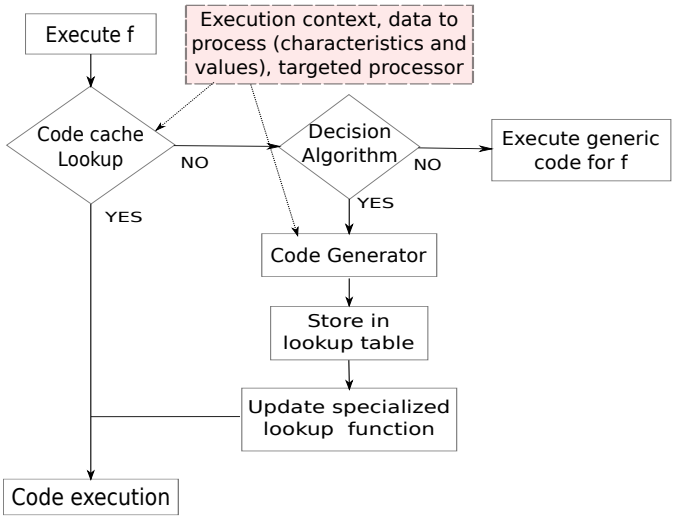


Figure 1: Runtime code generation system

Table 1: Function f is specialised on the first two parameters with values val0, val1 for the first cache line and with values val0,val2 the second time. Function g is specialised on the first parameter with value int0.

Name	Line size	Specialized code	Arguments				
f	3	$@f\_spec_{val0,val1}$	val0	val1			
	3	$@f\_spec_{val0,val2}$	val0	val2			
g	2	$@g\_spec_{int0}$	int0				

When a function a priori identified as an interesting function is called, the code cache is inspected to verify if the specialised code has already been generated. First the name of the function is used to find the set of cache lines corresponding to the function we are looking for. Then the number of element in the cache line is read and each element is compared to the function's context. If all the arguments in the cache line matches with the context, then the specialised code is retrieved from the cache line and executed, without the need for code generation.

As memory size is limited in such systems, our implementation requires only a few hundreds of bytes.

2.4 Decision algorithm

Code generation has a cost that needs to be amortised. Unlike JIT systems in which a function is compiled only if its execution frequency exceeds a constant threshold estimated off-line, we propose an algorithm that decides if a specific function should be specialised or not based on a function-dependant criteria. The algorithm determines if the overall cost of the application will be decreased by using specialisation. The cost metric can be related to execution time of the application, energy consumption, power consumption or a combination of several physical quantities such as the Energy-Delay product.

The decision algorithm has to determine if the code gener-

ation cost would be amortised over the life time of the application. Several solutions can be implemented for this. One solution is to respect the following rule; in which K_{gen}, K_{lib} and K_{spe} denote the execution cost of respectively code generation, standard algorithm and specialised algorithm and N denotes the number of executions of the function to specialise: if $N*(K_{lib}-K_{spe}) > K_{gen}$, then the function should be specialised as soon as it is executed at least N times.

In the rest of the paper, the decision algorithm is not implemented and we use a simpler solution which consists in systematically generating and storing specialised code.

2.5 Specialisation of code cache lookup

Specialisation can be applied to improve performance of each part of an application using any parameter that remains constant for several executions. In the system described in this paper, the lookup function verifies in the code cache to find if a part of code has already been specialised or not. Our idea is to use specialisation to improve performance of this function.

The code cache is modified each time a new specialised code is stored. The first time the lookup function is called, the code cache is empty. Later on, the cache size will increase. Specialisation can be used to optimise the lookup function using the number of elements currently stored in the code cache.

With this method, a new specialised code for the lookup function needs to be generated each time the number of elements in the code cache increases. Algorithm 1 describes a generic function to check if a code can be found in the code cache. Algorithm 2 illustrates a specialised function, knowing that the code cache contains $X+1$ elements. The specialised lookup function finds the code in the code cache and branches directly to the code, avoiding coming back to the main function beforehand.

Algorithm 1 Generic "lookup" function

```
1: procedure LOOKUP(id)
2:     for i = 0; i < NbElem; i++ do
3:         if id == elem_i then return i
       return −1
4: procedure MAIN
5:     ...
6:     index ← LOOKUP(id)
7:     if index == −1 then
8:         f_spec ← GenerateCode(f)
9:     else
10:        f_spec ← cache[index]
11:    res ← f_spec(value)
12:    ...
```

3. PERFORMANCE EVALUATION

3.1 Experimental setup

The implementation proposed in this paper consists of overwriting common functions generally implemented in standard libraries, as for example the GNU's mathematical library libm. The runtime code generator is wrapped in a function to check the code stored in the code cache and, according to the content of the code cache, decide if the specialised function needs to be generated or not (see Section 2.1 for details). For the evaluation of the code cache use, specialised code is generated at the first execution of the function regardless the cost of code generation.

Algorithm 2 Specialised "lookup" function

```
1: procedure LOOKUP_SPEC(id, value)
2:     compare(id, elem_0)
3:     branch @code_spec_0
4:     compare(id, elem_1)
5:     branch @code_spec_1
6:     ...
7:     compare(id, elem_X)
8:     branch @code_spec_X
9:     branch @CodeGen
10: procedure MAIN
11:    ...
12:    res ← LOOKUP_SPEC(id, value)
13:    ...
```

Experimentation has been done on the WisMote platform [3] from Arago Systems. We have chosen this platform because it has the characteristics of a low power system with limited memory resources and limited power computation. The platform uses the 16-bit MSP430F5437 micro-controller from Texas Instrument, fitted with 256 kB of flash and 16 kB of RAM, the CPU clock frequency is set to 2.45 MHz. To measure time, we set up the timer frequency at 2.45 MHz divided by 4. So, one timer tick equals to 4 CPU cycles. The accuracy of the timer has been checked using a GPIO and monitoring activity with an oscilloscope.

WisMote platform embeds Contiki, a small footprint and open source operating system. To perform measurements of execution times, we use Contiki's Energest tool.

3.2 Performance metrics

To evaluate the performance of our solution, we use two metrics based on execution cost. The first metric is the speedup, which is the ratio between execution cost of the generic application and the specialised application. If the specialisation reduces execution cost, then the speedup is greater than 1. The speedup is defined in (1), where K_{lib} is the cost of the standard algorithm and K_{spe} is the cost of the specialised algorithm.

$$speedup = \frac{K_{lib}}{K_{spe}} \tag{1}$$

The second metric is the overhead recovery [2], defined as the number of executions of the specialised code that is required to overcome the cost of code generation. To formalise this notion, K_{gen} denotes the execution cost of code generation and N denotes overhead recovery. We assume here $K_{lib} > K_{spe}$ as there would be no point in generating a function having same or lower performance than the non optimised implementation.

$$N = \frac{K_{gen}}{K_{lib} - K_{spe}} \tag{2}$$

3.3 Application to floating point multiplication

3.3.1 Use case description

We evaluate the performance of our system using floating point multiplication. Floating-point arithmetic is used in several applications in WSN. For example, sensors, as hu-

midity sensors, temperature sensors or force sensors, generally output a digital value which needs to be converted using a formula given by the sensor manual [1]. This formula is often a linear function as described in (3), where *output* is the digital value from the sensor, *value* is the value the user is interested in (temperature, humidity...) and k_1 and k_2 are constant values.

$$value = k_1 * output + k_2 \quad (3)$$

As k_1 is a constant value, using a specialized code can significantly decrease application cost.

Floating-point multiplication can be optimised using a polynomial root approximation method known as Horner scheme. This method is not detailed in this paper for the sake of brevity.

As Aracil et al. explained [2], energy consumption can prevail on precision in specific applications. Moreover, in the IoT, many sensors don't have an accuracy higher than 12 bits. The algorithm used to optimise floating-point multiplication enables to adjust precision using mantissa truncation. The less bits of mantissa are used, the shorter the execution time is.

The evaluation consists of four hundreds floating-point multiplications of two randomly-picked operands, one of them being the parameter used for code specialisation. Multiplication is executed with a number of mantissa bits used from 1 to 24 bits.

For each multiplication, time for code generation, standard library execution and specialised code execution are recorded. Speedup and overhead recovery are deduced from these measures.

3.3.2 Performance results

Fig. 2 details the results of our algorithm and provide a comparison with the results previously obtained by Aracil et al. [2] in which specialisation was managed by the developer. In Fig. 2(a) and 2(c), we observe that for accurate precision the overhead recovery is close to 3.1 and the speedup is around 7.3. Overhead recovery decreases and speedup increases as precision decreases.

The automatic management of the code specialisation and of the code cache takes some time to manage, which increases code generation overhead in our automatic specialisation (Fig. 2(d)) as compared to the case where the specialisation is managed by the developer (Fig. 2(c)). For the same reason, speedup is slightly decreased by code cache management.

The overhead due to code generation is also increased by generation of the specialised lookup function. Each time an element is added to the code cache, the lookup function is generated. This increases generation time overhead and consequently increases overhead recovery value. Then, for a long-time application on a platform with low memory resources, the specialised lookup function decreases application's execution time, as code generation will be payed off after 4 executions.

Automatic triggering of code specialisation with a code cache gives results as good as code specialisation managed by the software developer. Only 4 executions of specialised code stored in the code cache are needed to amortise code specialisation. Moreover, our method is easier to use and doesn't require any effort from the developer.

3.3.3 Specialisation on hardware characteristics

Code specialisation is used to improve application's performance knowing runtime data. Platform specificities is another type of runtime data that can be used to generate efficient code. Moreover, in the IoT, applications are usually deployed on many different platforms, one can think that optimisation can be done statically to exploit platform characteristics, but we observed that the libm floating point multiplication doesn't exploit the integer hardware multiplier to improve performance.

In this section we take advantage of the integer hardware multiplier available on the WisMote platform to multiply mantissa. Instead of implementing Horner scheme (see Section 3.3.1) to multiply mantissa, we load mantissa values in registers and call the hardware multiplier.

For the sake of brevity, no figure is included to illustrate the results presented in this section. Using the integer hardware multiplier, we get a speedup factor around 6.5, which is less than in section 3.3.2. But generation time is divided by 2 as code to use the hardware multiplier is shorter than Horner scheme algorithm: specialised code generation takes around 4800 cycles for Horner scheme, whereas it takes around 2300 cycles when using hardware multiplier. Hence the overhead recovery value for the compilette using the hardware multiplier is less than 2, whereas the value was close to 4 for an accurate multiplication.

3.3.4 Discussion

Our work illustrates that data specialisation can be used easily by software developer to improve application performance. With an overhead recovery less than 4, code specialisation for floating-point multiplication is efficient every time an operand remains constant for more than three executions. These results can be improved using mantissa truncation to get an overhead recovery less than two.

Moreover, the algorithm using the hardware multiplier gives better results. If a hardware multiplier is available on the platform, code specialisation with accurate precision becomes efficient every time an operand remains constant for at least two executions.

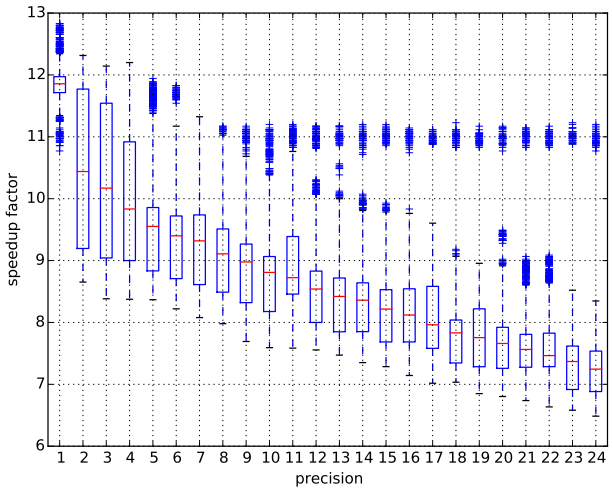
3.4 Global application generalisation

3.4.1 Speedup of a complete application

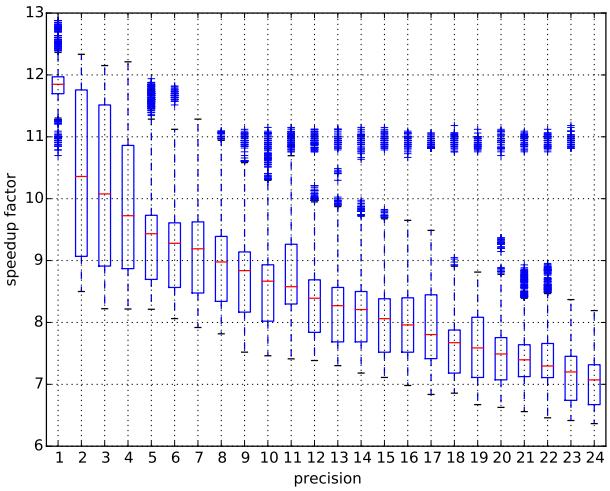
Previous sections highlight a speedup between 6 and 7 on floating-point multiplication. However, an application is an overall process and doesn't include a unique operation. In this section, we introduce a model to calculate the speedup of a complete application based on the speedup obtained by specialising a specific operation. The speedup of an overall application S_{app} is formalised in Equation 4 [8], where s denotes the speedup for the specialised operation (i.e. around 7 for the multiplication) and τ denotes the fraction of execution time spent executing the operation to specialize.

$$S_{app} = \frac{1}{1 - \tau + \frac{\tau}{s}} \quad (4)$$

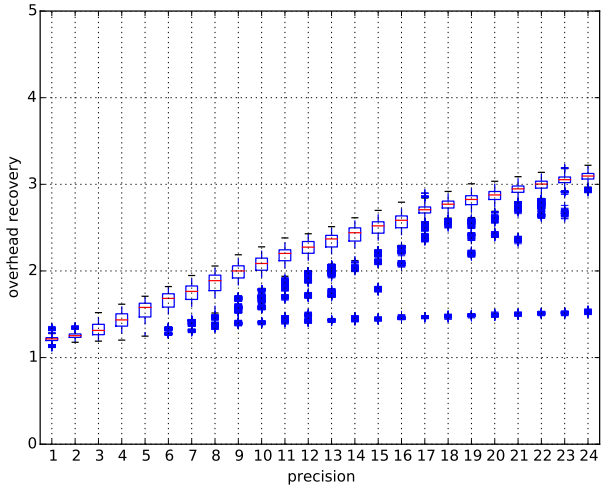
In Equation 4, we observe that speedup of a complete application is correlated with the fraction of time spent in the specialisation target. Fig. 3 illustrates the evolution of the speedup of a complete application depending on the fraction of time τ spent in the floating point multiplication. If τ is



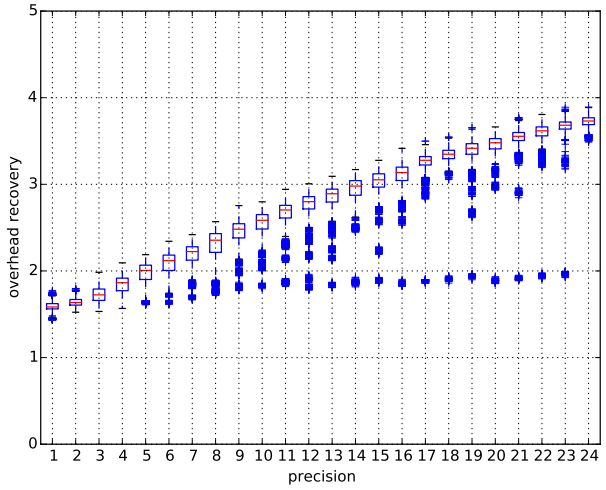
(a) Speedup, manual specialisation scheme [2].



(b) Speedup for our self-optimising system.



(c) Overhead recovery, manual specialisation scheme [2].



(d) Overhead recovery for our self-optimising system.

Figure 2: Performance results of automatic specialisation against specialisation managed by the developer

close to 0, the speedup of the application is close to 1, which means the application's execution time is not changed. If $\tau \to 1$, the speedup of the application is close to the speedup obtained for floating point multiplication s, here 5.6

$$S_{app} = \frac{1}{1 - \sum_{i=1}^{n} \tau_i + \sum_{i=1}^{n} \frac{\tau_i}{s_i}} \tag{5}$$

Equation 5 is a generalisation of Equation 4, modelling the case where specialisation is applied to several functions. n denotes the number of specialised functions, s_i the speedup for the specialised function i and τ_i the fraction of execution time spent executing the function i.

3.4.2 *Illustration*

To illustrate performance improvements on a complete application, the MSP430 Competitive benchmarking described by Texas Instrument [7] is used. The Floating Point math example is compound of a floating point addition, multipli-

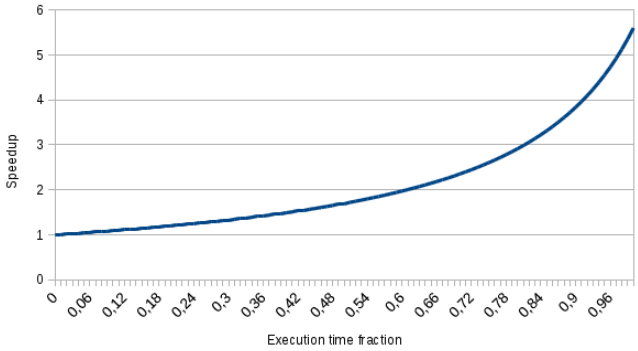


Figure 3: Speedup of the whole application as a function of the fraction of execution time τ spent in the specialised function.

cation and division. We tested it on the WisMote platform with the experimental setup described in section 3.1.

We measure that $\tau = 49\%$ of the execution time is spent in the multiplication routine. The multiplication routine takes 381 CPU cycles without specialisation and 68 cycle counts using our self-optimising system. This leads to a speedup factor s equal to 5.6.

The whole floating point math example takes 465 CPU cycles using our self-optimising system and 778 CPU cycles without specialisation, the speedup factor of the complete application S_{app} is 1.67. These results confirm Equation 4.

4. RELATED WORKS

Many research works have been presented about data code specialisation, for example [5, 11]. A JIT for JavaScript able to perform specialisation on data values has been proposed by Costa et al [6]. The system is based on the hypothesis that many functions are called with the same parameter values. Hence specialisation is done for the first execution of the function and a code cache is used to store the specialised code and use it several times. For each new function call using the same data, the optimised code is used. Khan [10] has also described a system that specialises a binary template and stores different versions in a code cache. However, our system does not limit code specialisation to the first execution of the function and several specialised versions of a generic function can be generated depending on the data, our code cache is then adapted to store more than one optimised code for each original code.

Herring and Wrighton [9] describe a system that generates code at runtime and stores this code to use it several times. They associate an identifier to the function and its arguments values in order to find the specialised code in the code cache. Using an identifier, generally calculated with a hash function, leads to add guards at the beginning of the specialised code. Indeed, when using hash functions to encode values, several initial values can have the same hash code. Guard instructions at the beginning of the specialised code are necessary to verify the exactness of the encoded values. Our system is adapted to processors with limited memory resources and doesn't use any identifier in order to avoid adding guards. Herring and Wrighton specialise functions on arguments and we propose to specialise functions on parameters that can also be platform specificities for example. Moreover, our lookup function is specialised in order to improve its efficiency.

5. CONCLUSION

This paper presents a self-optimising system using runtime code generation, in order to increase the lifetime of WSN nodes. Our approach hides the management of runtime code specialisation to the application developer, embedding code specialisation in a library. Our performance evaluation for floating point multiplication illustrates how the system improves the execution time, with a specialised code between 6 or 7 faster than the reference implementation. Our system enables to decrease precision and improve performance results in order to adapt to IoT devices. The application of code specialisation on other runtime characteristics, such as hardware specificities (e.g. use of a hardware multiplier), reduces even more the execution time; code specialisation is efficient as soon as an operand remains constant for at least two executions. Discussion on the benefit for a complete application leads to the conclusion that the speedup obtained is correlated to the time initially spent in the specialisation targets.

Future work will integrate an efficient decision algorithm in the system to automatically trigger code generation or not depending on the application and to implement other specialised functions in order to create a complete library for arithmetic. The decision algorithm will be able to specialise code if it will improve overall application performance or use generic code otherwise.

6. ACKNOWLEDGMENTS

This work has been partly funded by the Artemis ARROWHEAD project under grant agreement number 332987 (ARTEMIS/ECSEL Joint Undertaking, supported by the European Commission and French Public Authorities).

7. REFERENCES

[1] Phidgets 1125: Humidity/temperature sensor, last visited 2015-11-20. http://www.electronicaestudio.com/docs/ph1125.pdf.

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

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

[4] H.-P. Charles and V. Lomüller. Is dynamic compilation possible for embedded systems? SCOPES, pages 80–83, 2015.

[5] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *POPL*, pages 145–156, 1996.

[6] I. R. de Assis Costa, H. N. Santos, P. R. Alves, and F. M. Q. Pereira. Just-in-time value specialization. *Computer Languages, Systems and Structures*, 40(2):37 – 52, 2014.

[7] W. Goh and K. Venkat. Msp430 competitive benchmarking. application note SLAA205c. Technical report.

[8] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., 2011.

[9] N. Herring and D. Wrighton. Caching runtime generated code. (US2010/0095284 A1), 04 2010.

[10] M. A. Khan. *Techniques de spécialisation de code pour des architectures à hautes performances.* PhD thesis, 2008. 2008VERS0032.

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

[12] M. Poletto, W. C. Hsieh, D. R. Engler, and M. F. Kaashoek. 'C and tcc: A language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems*, 21(2), 1999.