Embedded Operating System Optimization through Floating to Fixed Point Compiler Transformation

Daniele Cattaneo, Antonio Di Bello, Stefano Cherubin, Federico Terraneo and Giovanni Agosta
Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano
Milano, Italy
Email: daniele3.cattaneo@mail.polimi.it antonio.dibello@mail.polimi.it
stefano.cherubin@polimi.it federico.terraneo@polimi.it agosta@acm.org

***Abstract*—Architectures targeted at embedded systems often have limited floating point computation capabilities, and in many cases do not provide any hardware support. In this work, we propose a self-contained compiler transformation pass implemented within LLVM to perform floating point to fixed point conversion. This pass is used to optimize the scheduler of the MIOSIX[1] embedded real-time operating system. We compare the proposed approach with the original floating point implementation, a hand-tuned fixed point one, and a solution based on a C++ library for fixed-point arithmetic. Our solution achieves speedups with respect to original floating point implementation up to 3.1 $\times$.**

***Index Terms*—Compilers, Fixed Point, Task Scheduler**

I. Introduction

Due to area and power constraints, most embedded architectures either do not provide hardware support for floating point computation or provide limited support. The solution is to convert floating point code to its fixed point equivalent, which is however a time-consuming and error-prone task. Several methodologies and tools to automate this conversion have been proposed [1]–[4], but such tools usually carry the burden of complex frameworks that are designed to support specific architectures or programming languages – usually ANSI C.

However, the embedded system environment is evolving to support a wider range of programming paradigms, driven by changes including the Internet of Things and Industry 4.0. Support for more modern native programming languages such as C++ is already being provided by mainstream platforms such as Arduino and MBED. As a result, an increasing number of embedded applications make use of features and paradigms not supported by ANSI C.

To overcome this limitation, we tackle the problem from a language-independent point of view. Our solution is based on the LLVM compiler framework [5], and performs the floating to fixed point conversion at the LLVM-IR level. The compiler frontend and backend are not changed, thus achieving language and architecture independence.

As a case study for the proposed technique, we employ the real time operating system MIOSIX. This OS is written almost entirely in C++. It targets embedded systems, thus it is a representative of the class of emerging embedded C++ codebases. Moreover, MIOSIX uses floating point code in the scheduler. We can thus demonstrate the ability of our approach to support the case of converted code spanning across interrupt service routines, a critical case for embedded systems.

This work provides the following main contributions:

1) An LLVM pass providing source language-independent, target-independent floating- to fixed-point conversion;
2) An optimised version of the MIOSIX task scheduler, and its assessment on three ARM-based embedded development platforms.

II. Analysis of the Case Study

MIOSIX is a real time OS targeting embedded system. The kernel is used as a platform for academic research, such as to design clock synchronization solutions [6] and schedulers.

Focusing on the scheduler, the kernel supports different scheduling algorithms, one of which is based on control theory [7] and uses floating point operations. This scheduler is the subject of our optimization. The goal of the control-based scheduler is to guarantee on average to each task the CPU share the programmer assigned to it. Scheduling is performed in rounds, at the end of which a control-theoretical regulator is run to compute the next execution time for each task, using the time previously used by tasks to provide feedback.

For each task, a floating point variable `alfa` represents its CPU share set point. Two different operations involve computations on `alfa`. The `IRQrecalculateAlfa()` function is called whenever the CPU distribution requirements change. It partitions the scheduling round among threads. It gives a higher share to those with higher priority whilst it keeps the invariant $\sum_{t \in T} \mathtt{alfa}_t = 1$ where T is the set of tasks to be scheduled. The function `IRQrunRegulator()` is instead called by the context switch interrupt service routine, and implements the control theoretical regulator.

To be able to guarantee that no overflows will occur in the fixed point computation, we perform a range analysis of the algorithm. Figure 1 shows the analysis result, restricted to the relevant part of the MIOSIX scheduler. Certain variables such as the measured CPU time burst of each tasks have well defined ranges, while for the maximum number of tasks and priorities we set a limit to 64 for the fixed point conversion.

Based on these assumptions it is possible to compute the initial range of values for each of the involved variables.

[1]https://miosix.org

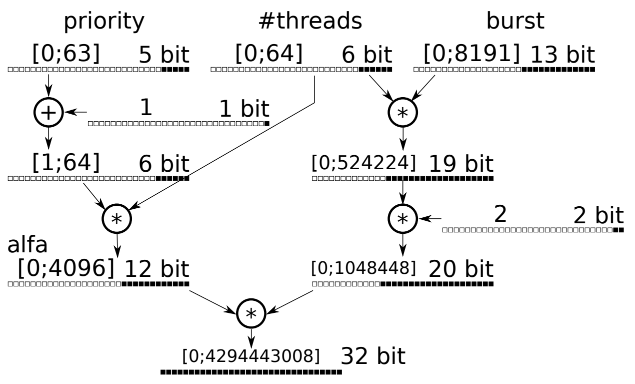


Fig. 1: Data flow analysis of the floating point values used in a fragment of the MIOSIX scheduler. For each node we report the range and the minimum data width required.

III. PROPOSED SOLUTION

We propose a solution to transform a given portion of floating point code into semantically equivalent code that exploits fixed point computation. In this work we specifically target the scheduler component of the MIOSIX operating system.

We rely on the LLVM compiler framework (version 4.0.0) and on its front-end CLANG (same version). We aim at applying the floating point to fixed point transformation without the need to customize the compiler. Our proposed solution exploits only on the compiler middle-end APIs which can be invoked through pluggable external modules called passes.

In this scenario, we ask the programmer to selectively annotate the source code via custom `annotate` attributes on floating point variables. This allows the programmer to focus only performance-critical code, without affecting the correctness of the rest of the program. Those attributes are properly parsed by the vanilla compiler front-end. Our compiler pass collects those annotations in the middle-end and properly propagates them to intermediate values. Then, it creates instructions based on fixed point arithmetic which are semantically equivalent to the original floating point code. After the conversion, we compile the converted code for the target architecture and we integrate the object code into the MIOSIX build system.

A. Annotations of the Source Code

Annotations specify which floating point variables should be converted to fixed point type, and provide hints to the later stages about requirements on desired precision or data width. These informations should be attached to a variable declaration via the `annotate` attribute.

Listing 1 shows an example of annotation, indicating that variables `a` and `b` should be represented with fixed point numbers. Every instruction that uses them is transformed to an equivalent sequence of instructions that uses fixed point numbers: namely, the assignment to `a` on line 5, the assignment to `b` on line 6 and the addition on line 7. Moreover, the annotation on `b` variable should be propagated also to all the floating point uses in its data flow. As a result, the multiplication on line 6 is now transformed to a fixed point operation, whilst it was left unchanged in the previous case on line 5.

```
1  float a __attribute((annotate("no_float")));
2  float b __attribute(
3        (annotate("force_no_float␣24␣8␣unsigned")));
4  int c = 98;
5  a = c * 2.0;
6  b = c * 2.0;
7  a += 10.0;
```

Listing 1: Example of annotated C code where the programmer is asking to transform the variables `a` and `b` to a fixed point representation using our solution. The `c` variable is unaffected in the first multiplication, whilst in the second one is transformed to perform a fixed point operation.

As shown in the example, the programmer may specify additional parameters to the conversion pass. The `"force_no_float"` keyword, used instead of `"no_float"`, specifies that computations that are not part of the data flow of the variable but whose result is used by the annotated variable should be annotated as well. The number of integer bits and the number of fractional bits to be used in the fixed point representation can also be explicitly included in the annotation. Finally, it is possible to use an unsigned fixed point representation instead of the default signed representation.

In the annotation of the MIOSIX kernel control scheduler, we exploit the propagation of annotations to limit the number of manual annotations to six variables and three constants.

B. LLVM Conversion Pass

Our solution transforms the LLVM-IR as if a type change to fixed point was performed in the original source code, and it preserves the original semantic meaning of the code as much as possible. First, we collect the set of instructions which need to be transformed by the algorithm. Then, we perform a data flow analysis to compute the minimum data width required by each value. Finally, the transformation is performed by generating fixed point code equivalent to the original floating point one.

1) Annotation Propagation and Data Flow Analysis: In this first step, we collect all the variables annotated by the programmer. Then, we enqueue for transformation the LLVM instructions that are part of the tree rooted in the instructions which define the annotated variables. When it comes to *forced* variables we also consider for conversion all the floating point values whose floating point descendants are used by the instructions in the aforementioned tree. Given the initial annotations, we can derive through a data flow analysis the data width required by each intermediate value.

It is worth noticing that an additional constraint in the selection of the data width comes from dynamic structures. Dynamically allocated structures are handled by choosing fixed point representations having the same size of the corresponding floating point values. This allows us to preserve the structure size. Then, we change the type of the pointers to the dynamically allocated structure or arrays.

Figure 1 shows an example of the result of the data flow analysis applied to the source code of MIOSIX, with the limit of 32 bit data width for the representation of dynamically allocated values.

2) Transformation: Once the queue has been created, every instruction in it is converted to one or more new instructions to use fixed point arguments and arithmetic.

When the pass meets an instruction with no known conversion, to guarantee the semantic equivalence, it restores the original data type and leaves the instruction unchanged. In the scheduler code there are no unsupported instructions.

For the case study, there is no need to consider floating point values which do not have a fixed point representation – such as NaN and $\pm inf$ – as the value range analysis ensures they are not eligible values for any of the variables.

IV. EXPERIMENTAL EVALUATION

We compare our proposed solution against three other alternatives: a fixed point C++ header library, a manual porting of the algorithms from floating point to integer arithmetic, and the original floating point implementation. In the rest of this section we refer to our solution as the LLVM *pass* version.

We call *reference* version the implementation that exploits floating point variables and arithmetic. Note that for architectures without hardware floating point support, MIOSIX relies on the default GCC software floating point emulation support.

We also generate a manually ported and optimized version of the original algorithm, using only integer arithmetic. We refer to this optimized version as the *manual* version.

Finally, we also convert the floating point code into fixed point equivalent code by exploiting a template-based C++ library that provides an implementation of both signed and unsigned fixed point data types. This is an open-source library [2] which is designed for approximate computing purposes in HPC [8] as part of the ANTAREX project [9], [10]. We refer to this version as the *C++ lib* version.

All the aforementioned conversion approaches require the software developer some effort to be applied. We quantify such effort in terms of newly inserted or modified lines of code (LOCs). The *manual* version requires a complete rework of the scheduler component. It represents the most costly solution in terms of LOCs to be modified. The *C++ lib* solution requires the insertion of 10 LOCs, and the modification of 6 LOCs.The LLVM *pass* solution requires the insertion of attributes near the floating point variables to characterize their initial value range. This procedure costs 9 LOCs and represents the lowest-effort solution.

A. Hardware Setup

We selected two representative off-the-shelf development boards among those supported by MIOSIX, one without floating point hardware support, and one with single precision floating point hardware support:

[2]source code available at https://github.com/skeru/fixedpoint

f207 An STM3220G-EVAL board featuring a 120MHz ARM Cortex M3 microcontroller without hardware floating point support. This board has 1 MByte of on-chip flash memory from which code is executed, and 2 MByte of off-chip SRAM used for the kernel and application data.

f469 An STM32F469I-DISCO board featuring a 168MHz ARM Cortex M4 microcontroller which has hardware support for single precision floating point. This board has 2 MByte of on-chip flash memory from which code is executed, and 16 MByte of off-chip SDRAM used for the kernel and application data.

lpc2138 A development board featuring a 59MHz ARM7TDMI microcontroller without hardware floating point support, using the ARM 32 bit instruction set rather than the mixed 16/32 bit instruction encoding Thumb2 instruction set of the other two boards. The board has 512 KB of on-chip flash memory from which code is executed, and 32 KB of on-chip SRAM.

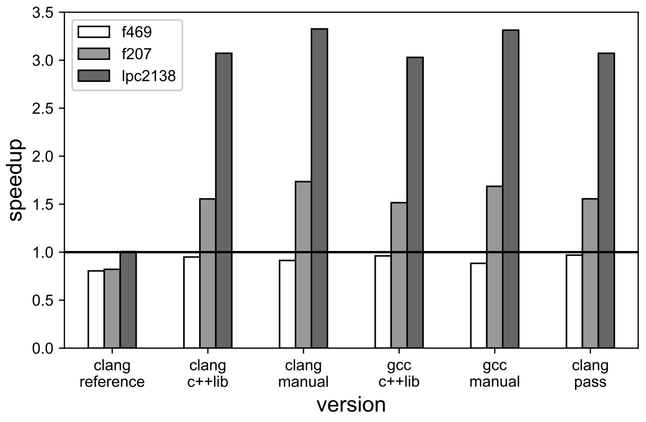
B. Software Setup

For each board we run two series of experiments. We run all of the above mentioned versions of the scheduler with the Hartstone uniprocessor benchmark suite [11] and with benchmarks from the MiBench suite [12]. We rely on the official compiler of the MIOSIX toolchain, which is GCC 4.7.3 with some minor patches to the standard library, and on its default compiler optimization set enabled by the `-O3` optimization level. Our solution compiles only the scheduler component via CLANG 4.0.0 with the same optimization level, and it integrates the compiled scheduler within the original MIOSIX compiler toolchain. To measure the time spent in the scheduling functions that have been affected by the transformation, we added a device driver that interfaces with a hardware timer/counter. The counter was clocked at the maximum frequency possible, corresponding to a timestamping resolution of two CPU clock cycles. To measure the execution time of the whole benchmark, we instead relied on standard C++ library APIs.

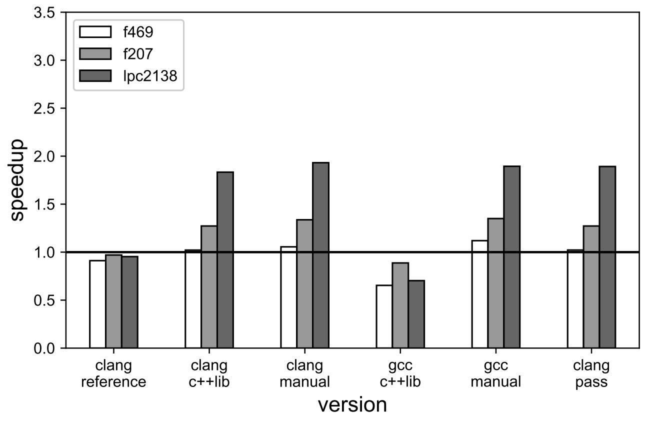
C. Result Analysis

In Section III we described how the LLVM *pass* solution relies on the LLVM compiler framework and it uses CLANG as compiler frontend for the C++ language. However, MIOSIX is ordinarily compiled using a slightly customized GCC toolkit. Thus, in Figure 2 we use the *reference* version compiled with GCC as baseline to measure the speedup of the other versions. On the other hand, as GCC and CLANG are different compilers as for design and implementation, we use both of them to compile the other versions to evaluate the performance differences due solely to the compiler.

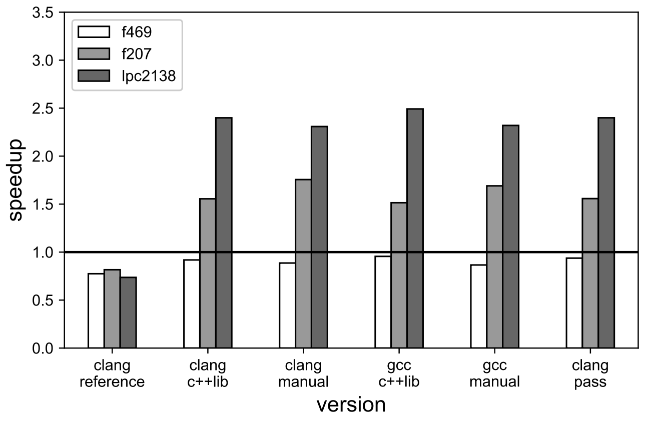
We find that both CLANG and GCC produce code with negligible performance differences, except in two cases. First, the *reference* version is generally slower when compiled with CLANG than with GCC due to the differences between the architectural model used by LLVM and GCC. Second, the `IRQrecalculateAlfa` function in the *C++ lib* version is



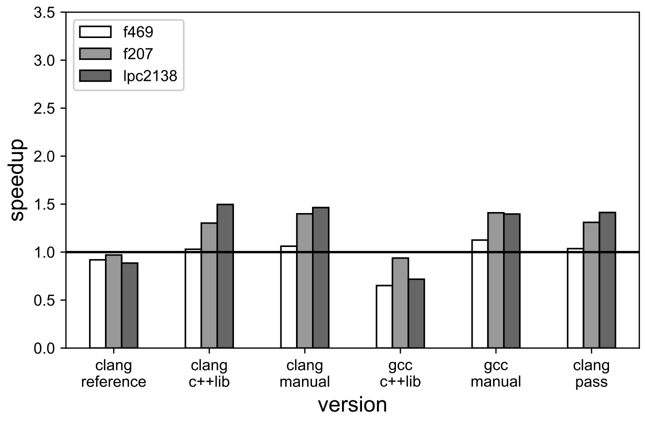
(a) Speedup of `IRQrunRegulator` during the MiBench benchmark.



(b) Speedup of `IRQrecalculateAlfa` during the MiBench benchmark.



(c) Speedup of `IRQrunRegulator` during the Hartstone benchmark.



(d) Speedup of `IRQrecalculateAlfa` during the Hartstone benchmark.

Fig. 2: Average speedup of the fixed point versions of the `IRQrunRegulator` and `IRQrecalculateAlfa` methods compared to their *reference* version compiled with GCC, measured during the execution of the MiBench benchmark and the execution of the Hartstone benchmark. The speedup has been computed as the average execution time of one call to each non-reference version, divided by the average execution time of one call to the *reference* version.

slower when compiled with GCC with respect to the same code when it is compiled using CLANG, because we observe different machine-independent optimizations. In particular, the *C++ lib* version creates a multiplication with two 64 bit integers operands. Those operands are 32 bit values which have been sign-extended to avoid precision loss. CLANG optimizes this pattern of multiplication to a 32 bit by 32 bit operation with a 64 bit result, whilst GCC does not. Indeed, GCC produces a 64 bit by 32 bit multiplication with a 64 bit result. Thus, the use of CLANG over GCC gives a slight advantage in very specific conditions while it becomes a disadvantage in other conditions. For this reason we show data both for CLANG and GCC.

When it comes to using the execution time as a quality indicator, the most important method to consider is `IRQrunRegulator`, as it runs every scheduling round.The `IRQrecalculateAlfa` method is run only once every workload change. Thus, it has a minor impact on real-world applications.

In Figure 2a and in Figure 2c we show the speedups measured on `IRQrunRegulator`. Values refer to the average time spent while executing `IRQrunRegulator`. We run each version 10 times and we report the speedup measured on the median value. We observe the speedup achieved by the fixed point representation is consistent across the two sets of benchmarks. We get a slowdown on the `f469` board as that board has support for hardware floating point, and thus we are effectively measuring the overhead intrinsic to fixed point computations. On the `f207` board, we measure a consistent speedup of roughly 1.5 to 1.8 times for all fixed point versions. Among the fixed point implementations, the *C++ lib* version and the *pass* version are equivalent, and they place only sightly below the *manual* version. Finally, the `lpc2138` board is the one which benefits of the highest speedup, roughly up to 3.3 times. This is due to the ARM7TDMI CPU architecture, which features a more limited pipelining with respect to the Cortex architecture.

In Figure 2b and in Figure 2d we show the speedups measured on `IRQrecalculateAlfa`. Similarly to the previous case, the results are consistent across the two set of benchmarks. On the `f207` and `lpc2138` boards the fixed point versions achieve speedups of roughly 1.2 times (for the `f207` board) and roughly 1.7 times (for the `lpc2138` board). We also achieve small speedups for the `f469` board, up to 1.1 times. The *C++ lib* version compiled with GCC is an outlier because of a missed optimization, as previously discussed. Overall, the *manual* version always sightly exceeds both the *C++ lib* version and the *pass* version, due to inter-procedural optimization performed by hand.

The assembly code generated by the *C++ lib* version compiled with CLANG and the code generated by the LLVM *pass* version are identical for all the boards, except for very small optimizations. Performance-wise, the difference between these two versions is always less than the timer resolution.

Finally, we evaluate the fixed point based solutions over the functional properties of the scheduler being discussed, which must not be invalidated. To this end we compare the quality metrics reported by the Hartstone benchmark, which represent the number of iterations before at least one thread misses a deadline. These indicators are consistent with the *reference* version for every fixed point version.

V. Related Works

The conversion of code designed for embedded system from floating point to a fixed point equivalent version is a long-established problem. A method to convert floating point ANSI C code into equivalent fixed point code are described in [2] and [4], but neither approach supports C++ code.

The *ID.Fix* plugin [13] of the *GeCoS* framework [1] has also been employed to provide a proof of concept for the case of co-optimization of fixed point data type width and SIMD operation when converting floating to fixed point arithmetic [14]. *GeCoS*, however, is primarily an interactive design environment targeting custom hardware, rather than a full compiler. As such, it is not easy to use in the context of embedded software development.

Compared to the tools reported above, ours relies on a state-of-the-art compiler framework – LLVM– which makes it more suitable for adoption in real-world applications.

VI. Conclusions

In this work, we present a source language-independent and target-independent method for performing floating to fixed point conversion through a compiler pass, and exert the resulting toolchain to optimise the performance of the task scheduler of the Miosix real time operating system.

As a result, we achieve a reduction of the overhead imposed by the operating system for task scheduling operations by up to 3.1 times when compared with the original floating point implementation, and an overhead of less than 9% when compared to a manually optimized fixed point implementation of the same algorithm. Our solution reaches the same performance of the language-dependent library version – which represents the most common approach adopted in the state of the art – with less effort from the programmer and better precision over constant propagation.

Future directions include the design of a more comprehensive dataflow analysis allowing the estimation of error, and an early estimation of performance improvements.

Acknowledgements

This work is partially supported by the European Union's Horizon 2020 research and innovation program under Grant agreements No.671623 FET-HPC ANTAREX and No 671668 FET-HPC MANGO. Authors would like to thank STMicroelectronics which provided the boards *f207* and *f469*.

References

[1] A. Floc'h *et al.*, "GeCoS: A framework for prototyping custom hardware design flows," in *2013 IEEE 13th Int'l Working Conf on Source Code Analysis and Manipulation (SCAM)*, Sept 2013, pp. 100–105.

[2] M. Willems, V. Bürsgens, H. Keding, T. Grötker, and H. Meyr, "System level fixed-point design based on an interpolative approach," in *Proceedings of the 34th Annual Design Automation Conference*, ser. DAC '97. New York, NY, USA: ACM, 1997, pp. 293–298.

[3] R. Nobre, L. Reis, J. a. Bispo, T. Carvalho, J. a. M. P. Cardoso, S. Cherubin, and G. Agosta, "Aspect-driven mixed-precision tuning targeting gpus," in *PARMA-DITAM 2018*, Jan 2018.

[4] K.-I. Kum, J. Kang, and W. Sung, "Autoscaler for c: an optimizing floating-point to integer c program converter for fixed-point digital signal processors," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 47, no. 9, pp. 840–848, Sep 2000.

[5] C. Lattner and V. S. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *CGO*, 2004, pp. 75–88.

[6] F. Terraneo, A. Leva, S. Seva, M. Maggio, and A. V. Papadopoulos, "Reverse flooding: Exploiting radio interference for efficient propagation delay compensation in WSN clock synchronization," in *2015 IEEE Real-Time Systems Symposium, RTSS 2015, San Antonio, Texas, USA, December 1-4, 2015*, 2015, pp. 175–184.

[7] M. Maggio, F. Terraneo, and A. Leva, "Task scheduling: a control-theoretical viewpoint for a general and flexible solution," *Transactions on Embedded Computing Systems*, vol. 13, no. 4, pp. 1–22, 2014.

[8] S. Cherubin, G. Agosta, I. Lasri, E. Rohou, and O. Sentieys, "Implications of Reduced-Precision Computations in HPC: Performance, Energy and Error," in *Int'l Conf on Parallel Computing (ParCo)*, Sep 2017.

[9] C. Silvano, G. Agosta, S. Cherubin *et al.*, "The antarex approach to autotuning and adaptivity for energy efficient hpc systems," in *Proceedings of the ACM International Conference on Computing Frontiers*, ser. CF '16. New York, NY, USA: ACM, 2016, pp. 288–293.

[10] C. Silvano, A. Bartolini, A. Beccari, C. Manelfi, C. Cavazzoni, D. Gadioli, E. Rohou, G. Palermo, G. Agosta, J. Martinovič, J. Bispo, J. M. P. Cardoso, J. Barbosa, K. Slaninová, L. Benini, M. Palkovič, N. Sanna, P. Pinto, R. Cmar, R. Nobre, and S. Cherubin, "The ANTAREX Tool Flow for Monitoring and Autotuning Energy Efficient HPC Systems," in *SAMOS 2017 - Int'l Conf on Embedded Computer Systems: Architecture, Modeling and Simulation*, Pythagorion, Greece, Jul. 2017.

[11] N. H. Weiderman and N. I. Kamenoff, "Hartstone uniprocessor benchmark: Definitions and experiments for real-time systems," *Real-Time Systems*, vol. 4, no. 4, pp. 353–382, Dec 1992.

[12] M. R. Guthaus *et al.*, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, Dec 2001, pp. 3–14.

[13] N. Simon, D. Menard, and O. Sentieys, "ID.Fix-infrastructure for the design of fixed-point systems," in *University Booth of the Conference on Design, Automation and Test in Europe (DATE)*, vol. 38, 2011.

[14] A. H. El Moussawi and S. Derrien, "Demo: SLP-aware word length optimization," in *2016 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, Oct 2016, pp. 233–234.