Inter-Procedural Strength Reduction for Embedded Systems

Giovanni Agosta
Politecnico di Milano, Italy
Consorzio Interuniversitario Nazionale per l'Informatica, Roma, Italy

Abstract

Embedded systems often rely on code generated from model-based design tools, which can result in inefficient implementations due to the loss of high-level semantic information during code generation. This paper explores an inter-procedural extension of the strength reduction transformation, traditionally applied within loops, to optimize repeated computations across function calls. The proposed technique identifies parameters acting as counters and replaces costly operations – such as exponentiation or multiplication – with incremental updates based on recurrence relations, using static variables to preserve state between calls. We formalize the transformation, discuss its applicability conditions, and analyse tradeoffs between computation and memory access costs. Experimental evaluation on ARMv8, AVR microcontrollers, and x86_64 platforms demonstrates significant speedups for power operations (up to 9 ×), while highlighting limitations for simpler operations due to memory overhead.

2012 ACM Subject Classification Computer systems organization → Embedded software; Software and its engineering → Compilers

Keywords and phrases Compiler Optimization, Strength Reduction

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2026.4

1 Introduction

The design of embedded system is a complex task [4], that requires specialized knowledge both on the programming of resource constrained platforms, often not supporting the more widely known APIs used in general purpose programming (e.g., POSIX), and of a specific application domain, usually dealing with sensing or actuation within a particular environment or device – e.g., appliances, white goods, or vehicles. As a result, the development process requires a small team of experts with in depth knowledge and wide expertise to achieve good performance and/or energy consumption [3, 11]. This approach however does not scale to large corporate environments, where development is distributed across large teams where each member has a focused competence and limited insight on the overall design process [3, 4, 11]. To support this latter scenario, model-based design tools, visual programming and code generation environments are employed, such as Simulink or LabVIEW [16]. These tools generate code (typically in the C language) requiring significant post-generation optimization. This is often hampered by the loss of information originally expressed in the high level model but not preserved in the generated C code. While usually these tools are closed source, it is possible to observe such effects on similar open source tools and language, such as Modelica, for which it is known that compiling through C language leads to a loss of information that is damaging for performance [1]. At the embedded software level, another example is that of physical meaning of data types, which can provide useful information for value range analysis [12].

A solution to these issues would be to perform code optimization on the high-level representation of the model. This is indeed a solution that would be supported by the rise of MLIR [15], which enables domain-specific dialects to represent specific semantics of the high-level model. While the idea has been proven at least at an exploratory level [6], the most

17th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 15th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2026).
Editors: Davide Baroffio, Paola Busia, Lev Denisov, and Nitin Shukla; Article No. 4; pp. 4:1–4:10

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

commonly employed tools in the industry are proprietary, and do not have MLIR back-ends. Thus, at the present time, alternative options should be investigated, working at the level of the C compiler.

Often, embedded systems perform repetitive tasks [13, 18] in the form of a "main loop", within which a series of activities, represented by subprograms, are carried out. The "function call within a loop" pattern makes code less convenient to optimize, as compilers rarely apply inter-procedural optimization, yet many loop-based optimizations are available, e.g, loop-based strength reduction [5]. When *function inlining* is convenient, it is usually the easiest way to enable inter-procedural optimization [19]. However, several factors, particularly code size expansion, can hamper the widespread use of inlining in embedded software compilation.

In this paper, we explore the tradeoffs in applying inter-procedural loop-based strength reduction, showing how, under certain conditions, it can actually provide significant speedups, but also highlighting the need to perform careful profiling or static analysis to avoid de-optimizing effects due to trading off computation for memory access costs.

The rest of this paper is organized as follows. In Section 2 we introduce the background notions of strength reduction and its variants. In Section 3 we discuss the proposed inter-procedural strength reduction transformation, while in Section 4 we provide experimental data to support the proposed transformation. Finally, in Section 5 we draw some conclusions and highlight future research directions.

2 Background

Strength reduction is a well-known compiler transformation that consists in replacing an operation with a semantically equivalent, but less costly, one [5]. In its simplest form, a direct replacement of a costly operation with a less costly one can be directly performed, as shown in the following examples:

$$\begin{aligned} x \leftarrow y^2 &\Rightarrow x \leftarrow y \times y \\ x \leftarrow y/2^n &\Rightarrow x \leftarrow y >> n \quad (n > 0) \end{aligned}$$

The above examples show how the transformation is clearly optimizing, but also highlight some limitations – in both case, the transformation can be performed only for particular values – small constant exponents in the first case, and powers of 2 in the second.

A more complex, but also more widely applicable, extension is *loop-based strength reduction* [5], which is applied to arithmetic computations dependent on an induction variable i:

$$\begin{aligned} \forall i \in [i_0, i_n] : x \leftarrow y^i &\Rightarrow \forall i \in [i_0, i_n] : x \leftarrow x \times y \\ \forall i \in [i_0, i_n] : x \leftarrow y \times i &\Rightarrow \forall i \in [i_0, i_n] : x \leftarrow x + y \end{aligned}$$

Where an appropriate initial value of x, x_0, must be provided before the loop start. It is worth noting that, in the loop-based version, there is a tradeoff between computation and memory: the value of x at iteration j is now used to compute the value of x at iteration $j+1$, thus introducing a loop-carried dependency which was not present in the original code. The practical impact is, usually, an increased register pressure, since one register is employed to store the value of x and avoid memory accesses, which would lower the performance gain.

3 Inter-procedural Strength Reduction

The proposed inter-procedural version of the strength transformation is applied to a subprogram $f(p_0, \ldots, p_n, i)$ that is called repeatedly, with parameter i being computed in a way such that for successive calls f_n and f_{n+1}, $i_{f_{n+1}} = i_{f_n} + \delta$, where δ is a constant stride. In practice, it is usually sufficient to support the transformation for the unitary stride $\delta = 1$, but in principle is can be stated in a more general way. Also, while the simplest case is that of f being called within a loop, this is not necessary – the key point is that i behaves as a counter. In the case of a loop, this can be directly detected by a scalar evolution analysis (e.g., in LLVM [14]), or from inspection of the loop structure (e.g., in a higher-level compiler framework such as MLIR [15]). Otherwise, it may be inferred from an annotation coming from the programmer, or from a code-generation tool, allowing a wider applicability.

Assuming the compiler has identified f and i as potential targets for the transformation, the condition under which the transformation can be applied is that, within the body of f, there exists an operation $x \leftarrow g(v_0, \ldots, v_n, i)$, where $v_0, \ldots, v_n$ are constants. It must be also possible to define a recurrence-based transformation to replace repeated evaluations of g with an incremental update. Thus, a recurrence computation of g must be defined as:

$$g(v_0, \ldots, v_n, i_{n+1}) = g(v_0, \ldots, v_n, i_n + \delta) = T(g(v_0, \ldots, v_n, c_n), v_0, \ldots, v_n, \delta)$$

where T is a suitable operation.

As an example, if $g(c, i) = c^i$ with c constant, then T can be defined as $x_n \times c$ where $x_n = g(c, i_n)$.

Thus, given the original computation:

$$x_n \leftarrow g(v_0, \ldots, v_n, i_n)$$

it can be transformed by replacing the direct computation of g with the application of T, as long as the value of the previous computation of g is preserved:

$$\begin{aligned} x_0 &\leftarrow g(v_0, \ldots, v_n, i_0) \\ x_{n+1} &\leftarrow T(x_n, v_0, \ldots, v_n, \delta) \end{aligned}$$

where x must be a static variable – i.e., its contents must survive between different invocations of the same function.

3.1 Limitations

If subprogram f has other call sites that do not participate in the same counting, the a copy of f, f' must be generated. While this may appear as a contrived case, code generated by model-based tools may reuse the same block (for which code is generated in the form of a subprogram) across both related and unrelated calls. However, at the model level, the presence or absence of relation between the different call is usually detectable – whereas this might not be the case after the code is generated.

As mentioned above, the identification of the behaviour of parameter i might be difficult to perform at the level of the compiler for a general purpose language. However, the designer of an embedded system would often know such opportunities, and could add hints to the source code. Even in a proprietary model-based system, such a hint could be embedded in a comment or other metadata that is not discarded before code generation, and then detected by a suitably modified compiler.

The transformation also has the same limitations as the underlying strength reduction – e.g., even if the relation between g and T is mathematically correct, it might incur in inaccuracies due to floating point rounding errors. This, however, is generally accepted in compilers (e.g., the `-ffast-math` optimizations in `gcc`).

3.2 Performance Issues

The proposed transformation is in principle as effective as intra-procedural strength reduction, but suffers from an added effect, due to the nature of the stored temporary value. While in intra-procedural strength reduction the temporary value can be stored in a register, this is generally not possible in our case (except under the assumption of inter-procedural register allocation, which is not generally performed by the compiler). Instead, the value is stored as a static variable, thus requiring access to the memory. The cost of this access depends on the data type and the microprocessor architecture – it is minimal if the data size is the same or smaller than the word size, but can be larger otherwise. In this case, there may be interactions with transformations such as precision tuning, which alter the data type and size [21, 9].

Furthermore, the performance relation between the two operations g and T may be complex to understand. This is generally not the case in RISC architectures, where instructions have generally simpler semantics and performance are somewhat easier to understand, but it may easily happen in CISC architectures. However, since the vast majority of embedded software runs on RISC-based microcontrollers, this is a relatively minor issue.

Still, we investigate experimentally both issues in the Section 4.

3.3 Generalization to variable δ

While the standard strength reduction is applied with constant values of δ, it is possible to generalize it to some extent. However, this requires T to be designed for the corresponding g. For this purpose, we take into account the power function as g. We can perform a rewrite according to the fragment reported in Listing 1, which performs the strength reduced version of the power function only when the current parameter i_{n+1} is within $[-3, +3]$ of i_n.

Listing 1 Strength reduction applied to the pow function, with support for non-unitary induction variable steps.

```
DTYPE pow_sred(int i){
    static DTYPE x;
    static int old_i;
    if (i==0) { x=1; old_i=0; }
    switch (i-old_i) {
        case 0: return x;
        case 1: x*=C; break;
        case 2: x*=C*C; break;
        case 3: x*=C*C*C; break;
        case -1: x/=C; break;
        case -2: x/=C*C; break;
        case -3: x/=C*C*C; break;
      default : x=pow(C,i);
    }
    old_i=i;
    return x;
}
```

Intuitively, this transformation works as long as consecutive calls have values that are close – i.e., if the step in the induction variable is small. We will show this dependence in Section 4.3.1. In principle, the code could be adapted for larger steps, but the beneficial effects will be soon lost, since extending the range of supported steps increases the code size.

4 Experimental Evaluation

In this section we perform an experimental evaluation of the proposed transformation. The goal is to understand the opportunities for speedups compared to code optimized with existing transformations (i.e., those enabled by the -O3 flag), as well as to explore the interaction with precision tuning and different architectural models.

4.1 Benchmark Definition

To test the proposed transformation, we employ three kernels, representative of computations involving arithmetic operations of significant cost (powers and multiplications), both on floating point and integer numbers. Specifically, the three kernels model a sine wave generation, an amplitude modulation with exponential decay, and a cryptographic key evolution.

Sine wave generation. This kernel generates a sinusoidal waveform employing a tabulated sin function driven by a phase Φ computed as the product of a constant phase increment δ_Φ and the time step i.

$$\begin{aligned} \Phi &\leftarrow i \times \delta_\Phi \\ w &\leftarrow sin(\Phi) \end{aligned}$$

The computation is performed using double or single precision floating point numbers (we test both cases), and a 32-bit integer counter.

Amplitude modulation with exponential decay. This kernel applies an exponentially decaying envelope to a sine wave carrier, modelling a variety of fade-out controls:

$$\begin{aligned} \Phi &\leftarrow \Phi + 2\pi \times f \times \tfrac{i}{f_s} \\ \Phi &\leftarrow \Phi - 2\pi \, if \, \Phi \geq 2\pi \\ x &\leftarrow 100 \times 0.95^i \times sin(\Phi) \end{aligned}$$

where Φ is the phase, i is the induction variable representing discrete time steps, f is the carrier frequency and f_s is the sample rate. The operation to which strength reduction is applied is the power 0.95^i. The computation is performed using double or single precision floating point numbers, and a 32-bit integer counter.

Cryptographic key evolution. This kernel employs an integer exponentiation to derive a new key at every step i, starting from an initial prime p and the base key k_0.

$$k_{i+1} \leftarrow k_0 \oplus p^i \, mod \, 0xFFFFFFFF$$

This computation is performed employing a 32-bit integer counter and 64-bit integers for intermediate values.

Each kernel is deployed as a function, where i is one of the parameters. The function is repeatedly called from the main loop, with values of i increasing by 1 at each call.

Strength reduction is applied employing static variables to hold the partial computation, as described in Section 3.

4.2 Experimental Setup

For our experiments, we use a range of different platforms, both embedded and general purpose. While the general purpose platforms are not as interesting as embedded ones from the application perspective, they still provide some useful insights in understanding the behaviour of the transformation.

4.2.1 ARMv8 and x86_64 platforms

For ARM, the benchmarks have been compiled using `gcc` 12.2.0 (with target triplet arm-linux-gnueabihf and -O3 optimization level) on a Broadcom ARMv8 CPU with Cortex-A72 cores, mounted on a Raspberry PI 4 platform with the Raspbian operating system.

For x86_64, the benchmarks have been compiled using `gcc` 9.4.0 (with target triplet x86_64-linux-gnu and -O3 optimization level) on a Intel(R) Core(TM) i7-8750H CPU, mounted on a laptop with the Ubuntu operating system.

Each benchmark runs the main loop multiple 10000 times, to minimize the impact of initialization effects and other disturbances on the time measurement. All benchmarks have been run 36 times, reporting values for the mean case, to further remove disturbances.

Results are reported in seconds, measured as the user time with the Linux `time` utility.

4.2.2 AVR microcontrollers

To provide insights on the behaviour of the optimization on smaller microcontrollers, the benchmarks have been also compiled and simulated on the *Simulavr* simulator[1]. Simulavr is instruction-accurate and cycle accurate, and was successfully used as the starting point for tasks that require significant accuracy, such as side channel analysis [20].

For this experiment, we have employed `avr-gcc` version 5.4.0, targeting two cores from the "classic" AVR2 family (at90s4433) and the "enhanced" AVR5 (atmega328) family[2]. In the case of AVR devices, i was set to run in the $[0, 1000)$ range, and the outer loop is run only once, since the simulator does not suffer from disturbances.

Results are reported in simulated clock cycles from the start of the program to the reaching of the program exit point.

4.3 Experimental Results and Analysis

Table 1 reports the results for the ARMv8 platform. It can be seen that a speedup is produced in all cases. While the general expectation would be to see some further improvement when using smaller data types, this is not the case for ARMv8, since this is a 64-bit architecture. Reducing the data type size only adds some cast operations, which degrade the performance in the case of the amplitude modulation benchmark. `gcc` does not support quadruple precision floating points on ARMv8, which would likely show a degradation of performance due to the increased access to memory-stored static variables.

Table 2 reports the results for the Intel x86_64 platform. This platform offers the opportunity to also compare the use of long double. The interesting result here is the major slowdown obtained with double precision floating point compared to both `long double` and single precision `float`. A more in-depth analysis is required, but preliminarily the additional costs can be attributed to the memory moves which are performed here using mostly `movq` and `movsd`, whereas the single precision `float` version employs mostly `movss`.

[1] `http://www.nongnu.org/simulavr/`
[2] The compiler version employed is the one shipped with Simulavr.

Table 1 Experimental results on ARMv8, with precision tuning (double vs single precision floating point).

Benchmark	Precision	Baseline	Strength Reduction	Speedup
Sine Wave	Double	1.19s	1.15s	1.03
Amplitude modulation	Double	5.32s	2.03s	2.67
Sine Wave	Float	1.19s	1.15s	1.03
Amplitude modulation	Float	5.43s	2.32s	2.34
Key evolution	Int64	3.48s	0.36s	9.67

Table 2 Experimental results on x86_64, with precision tuning (quadruple vs double vs single precision floating point).

Benchmark	Precision	Baseline	Strength Reduction	Speedup
Sine Wave	Long Double	0.83s	0.89s	0.93
Amplitude modulation	Long Double	3.21s	1.64s	1.95
Sine Wave	Double	1.09s	1.08s	1.01
Amplitude modulation	Double	2.78s	8.60s	0.32
Sine Wave	Float	1.07s	1.08s	0.99
Amplitude modulation	Float	2.92s	1.37s	2.13
Key evolution	Int64	1.40s	0.16s	9.37

Table 3 shows the results for the two AVR platforms, which are essentially the same in terms of achievable speedups. The interesting point is that the speedup is below 1 (thus, a slowdown) for the sine wave generation example, where the target operation is a multiplication. This slowdown is driven by the need to store and load the 4-word floating point temporary needed for storing the old Φ, thus demonstrating the negative impact of memory-stored static variables.

Table 3 Experimental results on AVR (single precision only), reported in clock cycles.

Benchmark	CPU model	Baseline	Strength Reduction	Speedup
Sine Wave	atmega328	9995	12748	0.78
Amplitude modulation	atmega328	720350	232804	3.09
Key evolution	atmega328	588796	64788	9.08
Sine Wave	at90s4433	9790	12843	0.76
Amplitude modulation	at90s4433	1337724	441186	3.03
Key evolution	at90s4433	1160673	233912	4.96

4.3.1 Analysis of the generalized strength reduction

To understand the dependence of the generalized version of the transformation introduced in Section 3.3, we run on the ARMv8 platform a simple loop performing 0.95^v for 1M randomly generated values. The values v change with δ that are generated uniformly in a range $[-x, x+1]$, i.e., $v_{n+1} \leftarrow v_n + choice([-x, x+1])$, thus leading a slowly increasing value of v. Smaller ranges lead to higher likelihood of falling into one of the optimized cases, whereas larger ranges will lead to higher chances of taking the defaults, non-optimized case. The loop is further iterated 10k times, accumulating the total value to amplify execution times and errors (which in the end are negligible). Latency is measured with the standard C `time` library.

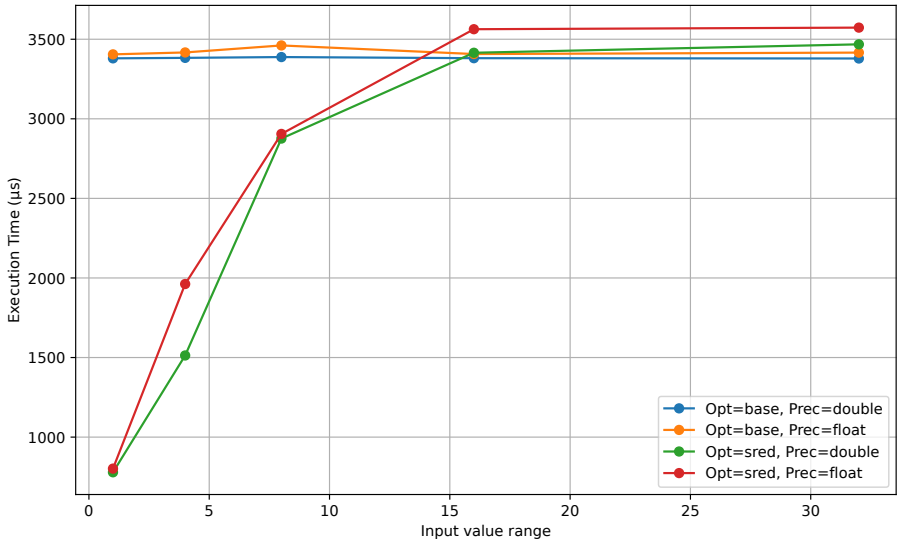

Figure 1 Variation of the execution time as a function of the input value range. The plot shows how the generalized strength reduction (Opt=sred) effectiveness depends on the careful identification of input range variability, whereas the non-optimized code (Opt=base) does not exhibit this dependency.

The graph in Figure 1 reports the execution time (in μs), as a function of the value of the input range extreme x. As expected, the non-optimized version (base) does not depend on the increment value range, whereas the optimized version behaves increasingly better as the increment range shrinks. The data type precision does not impact significantly in this case.

5 Conclusions

In this work, we explored the opportunity of applying the strength reduction transformation in an inter-procedural way in the context of embedded systems. The research is motivated by the need to optimize code that is automatically generated from proprietary model-based and visual programming tools, which often produce redundant code. We show, through an experimental analysis on synthetic benchmarks and a range of hardware platforms, how the proposed transformation can obtain significant speedups (3 to 9 $\times$ in strength reduction of power operations), but also some limitations, particularly the difficulty to predict performance in CISC architectures and interactions with precision tuning.

The latter issue, together with the identification of input value ranges, may benefit from integration with automated precision tuning tools at compiler level such as TAFFO [7, 17], leveraging either value range analysis [8] or profiling [10].

Future research will aim at exploring the impact on actual code generated from model-based tools, as well as integration in such a toolchain. Furthermore, a tradeoff comparison with an approach based on function inlining would be helpful to better understand the limitations of the proposed approach. Finally, strength reduction does affect the type of operations that are performed, so in the case of application to cryptographic operations, one

must also consider whether there is an impact on timing, to ensure that no side-channel information leakage is added. More in general, the interactions with compiler transformations aiming at implementation security are non-trivial [2].

References

1 Giovanni Agosta, Emanuele Baldino, Francesco Casella, Stefano Cherubin, Alberto Leva, Federico Terraneo, et al. Towards a high-performance modelica compiler. In *Proceedings of the 13th International Modelica Conference*, pages 313–320, 2019. doi:10.3384/ecp19157313.

2 Giovanni Agosta, Alessandro Barenghi, and Gerardo Pelosi. Compiler-based techniques to secure cryptographic embedded software against side-channel attacks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(8):1550–1554, 2019. doi:10.1109/TCAD.2019.2912924.

3 Deniz Akdur. Skills gaps in the industry: Opinions of embedded software practitioners. *ACM Transactions on Embedded Computing Systems (TECS)*, 20(5):1–39, 2021. doi:10.1145/3463340.

4 Deniz Akdur, Bilge Say, and Onur Demirörs. Modeling cultures of the embedded software industry: feedback from the field. *Software and Systems Modeling*, 20(2):447–467, 2021. doi:10.1007/s10270-020-00810-9.

5 David F Bacon, Susan L Graham, and Oliver J Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys (CSUR)*, 26(4):345–420, 1994. doi:10.1145/197405.197406.

6 Jiahong Bi, Guilherme Korol, and Jeronimo Castrillon. Leveraging the mlir infrastructure for the computing continuum. In *CPS Workshop 2024*, September 2024. URL: https://ceur-ws.org/Vol-3960/short5.pdf.

7 Daniele Cattaneo, Michele Chiari, Giovanni Agosta, and Stefano Cherubin. Taffo: The compiler-based precision tuner. *SoftwareX*, 20:101238, 2022. doi:10.1016/j.softx.2022.101238.

8 Daniele Cattaneo, Michele Chiari, Nicola Fossati, Stefano Cherubin, and Giovanni Agosta. Architecture-aware precision tuning with multiple number representation systems. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 673–678. IEEE, 2021. doi:10.1109/DAC18074.2021.9586303.

9 Stefano Cherubin and Giovanni Agosta. Tools for reduced precision computation: a survey. *ACM Computing Surveys (CSUR)*, 53(2):1–35, 2020. doi:10.1145/3381039.

10 Lev Denisov, Gabriele Magnani, Daniele Cattaneo, Giovanni Agosta, and Stefano Cherubin. The impact of profiling versus static analysis in precision tuning. *IEEE Access*, 12:69475–69487, 2024. doi:10.1109/ACCESS.2024.3401831.

11 Jack Ganssle. *The art of designing embedded systems*. Newnes, 2008.

12 W.H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Transactions on Software Engineering*, SE-3(3):243–250, 1977. doi:10.1109/TSE.1977.231133.

13 Yew Ho Hee, Mohamad Khairi Ishak, Mohd Shahrimie Mohd Asaari, and Mohamad Tarmizi Abu Seman. Embedded operating system and industrial applications: a review. *Bulletin of Electrical Engineering and Informatics*, 10(3):1687–1700, 2021.

14 Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004. doi:10.1109/CGO.2004.1281665.

15 Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE, 2021. doi:10.1109/CGO51591.2021.9370308.

16 Peter Liggesmeyer and Mario Trapp. Trends in embedded software engineering. *IEEE Software*, 26(3):19–25, 2009. doi:10.1109/MS.2009.80.

17 Gabriele Magnani, Daniele Cattaneo, Michele Chiari, Giovanni Agosta, et al. The impact of precision tuning on embedded systems performance: A case study on field-oriented control. In *Proceedings of PARMA-DITAM 2021 (OASICS)*, volume 88, pages 1–13, 2021. doi:10.4230/OASIcs.PARMA-DITAM.2021.3.

18 John A Stankovic. Real-time and embedded systems. *ACM Computing Surveys (CSUR)*, 28(1):205–208, 1996. doi:10.1145/234313.234400.

19 Theodoros Theodoridis, Tobias Grosser, and Zhendong Su. Understanding and exploiting optimal function inlining. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 977–989, 2022. doi:10.1145/3503222.3507744.

20 Nikita Veshchikov and Sylvain Guilley. Use of simulators for side-channel analysis. In *2017 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 104–112, 2017. doi:10.1109/EuroSPW.2017.59.

21 Yutong Wang and Cindy Rubio-González. Predicting performance and accuracy of mixed-precision programs for precision tuning. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–13, 2024. doi:10.1145/3597503.3623338.