CoSense: Compiler Optimizations using Sensor Technical Specifications

Pei Mu
University of Edinburgh
United Kingdom
pei.mu@ed.ac.uk

Nikolaos Mavrogeorgis
University of Edinburgh
United Kingdom
nikos.mavrogeorgis@ed.ac.uk

Christos Vasiladiotis
University of Edinburgh
United Kingdom
c.vasiladiotis@ed.ac.uk

Vasileios Tsoutsouras
University of Cambridge
United Kingdom
vt298@cam.ac.uk

Orestis Kaparounakis
University of Cambridge
United Kingdom
ok302@cam.ac.uk

Phillip Stanley-Marbell
University of Cambridge
United Kingdom
phillip.stanley-marbell@eng.cam.ac.uk

Antonio Barbalace
University of Edinburgh
United Kingdom
antonio.barbalace@ed.ac.uk

Abstract

Embedded systems are ubiquitous, but in order to maximize their lifetime on batteries there is a need for faster code execution – i.e., higher energy efficiency, and for reduced memory usage. The large number of sensors integrated into embedded systems gives us the opportunity to exploit sensors' technical specifications, like a sensor's value range, to guide compiler optimizations for faster code execution, small binaries, etc.

We design and implement such an idea in CoSense, a novel compiler (extension) based on the LLVM infrastructure, using an existing domain-specific language (DSL), Newton, to describe the bounds of and relations between physical quantities measured by sensors. CoSense utilizes previously unexploited physical information correlated to program variables to drive code optimizations. CoSense computes value ranges of variables and proceeds to overload functions, compress variable types, substitute code with constants and simplify the condition statements.

We evaluated CoSense using several microbenchmarks and two real-world applications on various platforms and CPUs. For microbenchmarks, CoSense achieves 1.18× geomean speedup in execution time and 12.35% reduction on average in binary code size with 4.66% compilation time overhead on x86, and 1.23× geomean speedup in execution time and 10.95% reduction on average in binary code size with 5.67% compilation time overhead on ARM. For real-world applications, CoSense achieves 1.70× and 1.50× speedup in execution time, 12.96% and 0.60% binary code reduction, 9.69% and 30.43% lower energy consumption, with a 26.58% and 24.01% compilation time overhead, respectively.

CCS Concepts: • **Software and its engineering** → **Compilers**; • **Computer systems organization** → *Embedded systems.*

Keywords: embedded systems, sensors, interval arithmetic, value interval propagation, compiler optimizations

ACM Reference Format:
Pei Mu, Nikolaos Mavrogeorgis, Christos Vasiladiotis, Vasileios Tsoutsouras, Orestis Kaparounakis, Phillip Stanley-Marbell, and Antonio Barbalace. 2024. CoSense: Compiler Optimizations using Sensor Technical Specifications. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction (CC '24), March 2–3, 2024, Edinburgh, United Kingdom.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3640537.3641576

CC '24, March 2–3, 2024, Edinburgh, United Kingdom

ACM ISBN 979-8-4007-0507-6/24/03
https://doi.org/10.1145/3640537.3641576

1 Introduction

Embedded systems are everywhere, from wearables to personal electronic devices – such as smartwatches, to appliances, industrial control systems, etc. Modern embedded systems integrate an increasing number of sensors, and in most cases do computations based on the sensed quantities, other than eventually logging such quantities.

While the computational capability of embedded devices drastically improved in the last decade, largely driven by the widespread adoption of mobile phones, IoT gadgets, etc., embedded devices are still characterized by low-power consumption, to maximize their lifetime on batteries. At the same time, the memory on such devices may be limited, due to cost or miniaturization (space/volume) considerations.

The reduction of power consumed can be addressed almost at any level of the hardware and software stack: from digital logic gating to supervisor software (e.g., firmware), from control algorithms to compiler optimizations. Memory utilization can be minimized via compiler optimizations or using memory compression. However, memory compression is not convenient in embedded systems as it consumes additional power.

In this paper, we focus on compiler optimizations aiming to reduce execution time and consequently energy consumption and memory utilization of embedded devices integrating sensors.

Key Idea. Our idea is to leverage sensor specifications, or physical characteristics, to drive compiler optimizations for improved performance and reduced code size. Each sensor is used to measure diverse physical quantities with a priori known value range(s), resolution(s), etc., therefore we can use the information in the compilation process to guide optimizations, like pruning the control flow, which won't affect the correctness and accuracy.

CoSense. We implemented our key idea in CoSense, an LLVM compiler extension that employs the Newton language [44], a DSL, which describes sensor technical specifications. CoSense implements Value Range Propagation that identifies the data flow at the LLVM intermediate representation (IR) level and propagates ranges across all functions of a program, including arithmetic operations. New follow-up LLVM passes use the output of the Value Range Propagation to: i) overload functions (i.e., specialization), ii) compress data types, iii) prune control flow, and iv) substitute constants.

We evaluated CoSense on different ARM CPU microarchitectures (including Cortex M4, and M0) and platforms (devboard and embedded) as well as an x86 machine using widely-adopted microbenchmarks, such as EEMBC and CHStone, but also real-world applications, including a Madgwick Filter [17] on the Warp board [53] and the Inertial Measurement Units (IMUs) driver [7] on a Crazyflie quadrotor platform [34]. We show that on microbenchmarks, CoSense achieves a geomean execution time speedup of 1.18× on x86 and of 1.23× on ARM, with 12.35% average binary size reduction on x86 and 10.95% on ARM. At a minimal extra compilation time of about 4.66% on x86 and 5.67% on ARM. We noticed that CoSense can offer more than 2× speedup without any negative performance impact in all cases. Indeed, the benefits introduced by this technique are highly related to each sensor's specification. With real-world applications, we observed 1.7× speedup and 13.0% binary code reduction on Warp, 1.5× speedup and 0.6% binary code reduction on the Crazyflie, and respectively 9.69% and 30.43% lower energy consumption.

Contributions. To the best of our knowledge, CoSense is the first proposal describing a compiler (extension) to couple code in widely-used programming languages (C/C++) with sensor technical specifications described via a DSL, Newton, and using such specifications for *automatic* code optimizations. CoSense is implemented within LLVM, operates at the IR level – supporting all languages supported by LLVM . It has been fully tested with C/C++ programs and the source code is open-source [1][47]. We make the following *key contributions*:

- Introduces and empirically validates the idea of using sensor technical specifications of their physical environment to drive compiler optimizations at module-level;

[1]https://github.com/systems-nuts/CoSense

```
bmx055: sensor (bmx055xAcceleration: acceleration[0],
                bmx055yAcceleration: acceleration[1],
                bmx055zAcceleration: acceleration[2],
                bmx055Temperature: temperature) = {
  # acceleration range in 3-axis (x, y, z) space
  range bmx055xAcceleration == [-16 g, 16 g],
  range bmx055yAcceleration == [-16 g, 16 g],
  range bmx055zAcceleration == [-16 g, 16 g],
  # temperature range
  range bmx055Temperature == [-40 C, 85 C]}
```

Figure 1. Newton DSL [44] specification of the BOSCH BMX055 sensor unit [6] with 4x signals. The `range` keyword defines a value range with its unit (e.g., gravitational acceleration `g`) for each signal (3x acceleration, 1x temperature).

- Proposes new algorithms for analyzing the range of *bitwise binary* operators, the *modulo* operator, and composite types to enable value interval propagation program-wide;
- CoSense, a prototype implementation built on top of LLVM, utilizing the Newton DSL, introducing *new optimization passes* that exploit sensor technical specifications;
- Evaluation of the CoSense prototype using microbenchmarks and real-world applications on several ARM and x86 platforms, assessing CoSense benefits and overheads.

The rest of the paper is structured as follows: Section 2 provides background information, Sections 3 and 4 introduce CoSense's design principles, design, and implementation; Section 5 evaluates CoSense, Section 6 contrasts CoSense with related works, Sections 7 and 8 summarize and conclude.

2 Background

2.1 Sensor DSL

The Newton DSL describes physical invariants for sensor data in a structured manner [44]. In Newton, sensors are first-class entities, allowing their declaration using signals and their properties. A sensor definition contains technical specifications about a sensor such as range, uncertainty, accuracy, precision, and operational information (e.g., how to read data from a sensor). Figure 1 shows a specification excerpt in the Newton DSL of a commercial *sensor*, the Bosch BMX055 [6], describing the value range of the sensors' data.

The machine-readable format of the sensor's specifications is intended to be used with analysis and transformation tools. A first such application has been the open-source Newton compiler which is used to synthesize the relation of multi-dimensional streams of sensor data [56]. In this paper, we explore how the use of this physical information in the form of sensor specifications within a general-purpose compiler can lead to novel or guide existing compiler optimizations.

2.2 Interval Arithmetic

Interval arithmetic is the body of mathematical techniques that aims to provide ways of calculating the upper and lower bounds of a calculation's range over one or more dimensions (i.e., variables). While initially applied to the estimation of rounding errors in floating-point calculations, it has later been employed in numerical optimization, program analysis and compiler optimizations [33, 38, 46, 48, 49]. More recently, it

Table 1. Interval arithmetic of binary operations supported by CoSense as defined in prior work [48].

Expr	Interval Range
$c=a+b$	$[\underline{A}+\underline{B},\overline{A}+\overline{B}]$
$c=a-b$	$[\underline{A}-\overline{B},\overline{A}-\underline{B}]$
$c=a\times b$	$[\min(x),\max(x)]$, $x\in\{\underline{A}\underline{B},\underline{A}\overline{B},\overline{A}\underline{B},\overline{A}\overline{B}\}$
$c=a\div b$	$[\min(x),\max(x)]$, $x\in\{\underline{A}\div\underline{B},\underline{A}\div\overline{B},\overline{A}\div\underline{B},\overline{A}\div\overline{B}\}$ or $[-\inf,\inf]$ if $b=0$
$c=a\ll b$	$[\min(x),\max(x)]$, $x\in\{\underline{A}\ll\underline{B},\underline{A}\ll\overline{B},\overline{A}\ll\underline{B},\overline{A}\ll\overline{B}\}$
$c=a\gg b$	$[\min(x),\max(x)]$, $x\in\{\underline{A}\gg\underline{B},\underline{A}\gg\overline{B},\overline{A}\gg\underline{B},\overline{A}\gg\overline{B}\}$

has been implemented in deep neural network (DNN) compilers [20, 26, 51].

Previous work has defined interval arithmetic, but in short, given the intervals of two operands, a and b, which are $A=[\underline{A},\overline{A}]$ and $B=[\underline{B},\overline{B}]$, where A ranges from $\underline{A}$ to $\overline{A}$ included and B ranges from $\underline{B}$ to $\overline{B}$ included, when in a binary operation, $C=[\underline{C},\overline{C}]$ be its resulting range and $\diamond$ be one of the binary operators: $\{+,-,\times,\div,\%,\ll,\gg,\wedge,\vee,\oplus\}$. We denote these operations as:

$$c=a\diamond b : a\in A, b\in B, c\in C \quad (1)$$

Table 1 recaps known interval arithmetic operators from previous work. We added support for more operators in Section 4.1.

2.3 Compiler Optimizations

Over decades of compiler research, several code optimizations have been developed that aim to simplify computation and code by reasoning about the values of program variables. Modern compiler frameworks (LLVM, GCC, etc.) have several transformations that exploit such information, such as instruction and control flow graph (CFG) simplifications (e.g., LLVM's `instcombine`, `simplifycfg`), constant hoisting, constant merging, strength reduction, etc.

In addition, language extensions (intrinsics) have been introduced to assist the compiler with extra information about the value of a variable (e.g., `__builtin_assume`) [31]. However, providing and maintaining information about the values of program variables for the compiler to utilize during optimization *is tedious, error-prone, and not portable, requiring developer time (which is a cost).*

Therefore, our work seeks solutions for automatically generating, maintaining, and propagating (physical) information about the variables starting from a high-level description of the (sensor) platform and its application.

3 Design Principles and Approach

Our key idea is to use domain-specific information, i.e., sensor technical specifications – like value range or resolution, to enable compiler optimizations that are tailored to an individual embedded platform (i.e., device configuration) and its application. Therefore, we propose that given an embedded platform, the developer should collect its technical specifications in a DSL that will be given as input to the compiler together with the application source code. The compiler will exploit such specifications to expand the breadth of possible optimizations and generate program binaries that are faster, smaller, etc. Moreover, this enables the same source code to be deployed on different platforms while being automatically optimized for each platform sensor's specification, avoiding costly developer performance tuning or intrinsics rewriting.

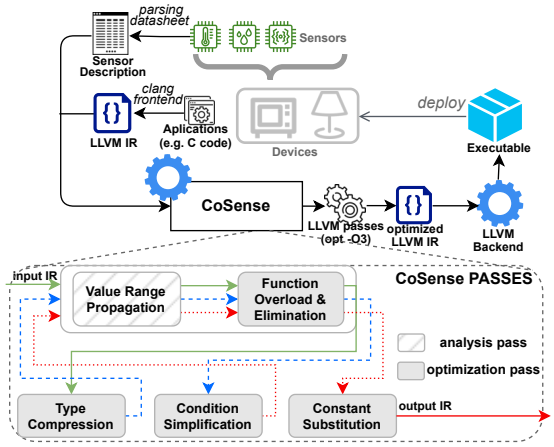


Figure 2. High-level overview of CoSense in the embedded application development lifecycle. The analysis pass combines the IR with the sensor DSL specifications and the optimization passes produce an optimized embedded application.

```
1 typedef double lm35aTemperature;  /* given [-55, 150] */
2
3 double toFahrenheit(lm35aTemperature c) {
4   double f = c * 1.8 + 32;
5   return
        f; /* from CoSense's Value Interval => [-67, 302] */
6 }
```

Figure 3. User-defined temperature unit conversion in C using the Newton type name for the `double` native data type. CoSense exploits this type of information to perform its analyses and optimizations. Assuming we are using an LM35A sensor (Table 5) whose Celsius temperature ranges in $[-55°C, 150°C]$, CoSense's *Value Range Propagation* analysis is able to propagate this information to Fahrenheit units.

Typically, at the source code level, a programming language does not offer any information about a variable, except for its data type (e.g., `int`, `float`, `char`). Language extensions or runtime systems attempt to augment variables with additional information by using annotations or profiling respectively. Herein, we start from the sensors' technical specifications (as DSL) and we combine those with an application source code. Specifically, this paper focuses on sensors' value range (i.e., upper and lower bounds), and we associate each sensor's range to variables that host data coming directly or indirectly from sensors by propagating value ranges based on operations on variables. An example of this combination is in Figure 3, where the developer was only required to redefine the native data types (line 1) to types defined in a DSL.This augments each variable with additional information on its data at compile time. Such information can be used by classical compiler optimization passes (e.g., branch optimization), or new ones.

4 Design and Implementation

CoSense implements the above design principles and consists of a compiler analysis stage and a series of optimizing transformations operating at the IR level. It expects an application source in IR, and a DSL description of sensors' specifications. Figure 2 illustrates a high-level overview of our design, placing CoSense within the embedded software development lifecycle and showing its main compilation components. CoSense is based on the LLVM ecosystem, and uses Newton DSL. However, our design can be implemented within another compiler and using another DSL. We built our analysis and optimization passes out of the LLVM source tree, and invoked them using Newton as a driver. We target C source code.

CoSense uses `clang` to convert C source code to LLVM IR code. Then, the Newton compiler processes the physical information specification and stores it in memory – then recorded in *llvm.debug.value*. Our LLVM passes query such information and optimize the LLVM IR code as described hereafter.

Firstly, CoSense's value range propagation (Section 4.1), which is a data-flow analysis, propagates the range information to each (reachable) variable using a Newton type via the program's use-def chains and CFG. The outcome of the range propagation is first exploited by two *new* optimization passes: Function Overload (Section 4.2), Type Compression (Section 4.3); and then by two *classical* compiler optimization passes that we extended: branch elimination by our Condition Simplification (Section 4.4), and Constant Substitution (Section 4.5). Note that range propagation and function overload passes are re-called after each other pass to ensure full code coverage.

4.1 Value Range Propagation

CoSense propagates value ranges across different variables and across functions using data-flow analysis, while supporting most LLVM IR instructions. Specifically, CoSense supports all variable-related IR instructions in LLVM, including unary/binary operations, `call`, `getelementptr`, `load`, `store`, instructions, `phi` node, and all the type conversion instructions. CoSense visits each IR instruction individually and propagates the range information from the source operands to the destination, based on the instruction, and if mathematic, on interval arithmetic. In Figure 3 the information about the temperature range in degrees Celsius is converted to equivalent in Fahrenheit: CoSense is using interval arithmetic for the multiplication and addition operations associated with the Celsius variable (i.e., `c`) as described in Section 2.

Supported Arithmetic Operations. We introduced support in LLVM for interval arithmetic of the binary operators addition, subtraction, multiplication, division, and bit-shift, which are well known in the literature, see Section 2. CoSense extends that with support for: (a) bitwise binary operations (`AND`: $\wedge$, `OR`: $\vee$, `XOR`: $\oplus$), (b) the modulo (or remainder) operation. We describe these below together with how we handle composite types, and how we propagate ranges across functions.

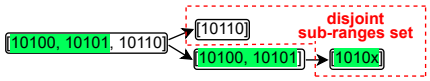


Figure 4. CoSense's calculation of the disjoint set for the range of $[20,22]$ ($[10100, 10101, 10110]$ when expanded). It merges the 10100 and 10101 values into $1010x$, where x can be either 0 or 1.

Bitwise Binary Operations. CoSense extends interval arithmetic to support bitwise binary operations because those are commonly used sensor applications. In fact, picking a microbenchmark from Section 5, `float64_add`, we found that there are 9 AND, 15 OR and 4 XOR operations. If the interval arithmetic of such operations is not supported, then the value range propagation will block, and CoSense will not be able to deduce the range of values that are defined from the bitwise operation result onward – losing optimization opportunities.

The result of a binary bitwise operation on value ranges can be brute-force calculated by executing the specific operation on all combinations of value pairs from each range – which is computationally expensive. CoSense introduces a faster algorithm that splits each value range in sub-ranges based on values' binary representation (base-2). In base-2, n-bit numbers $b_{n-1}...b_0$, 2^k consecutive values in a range that starts on an even value are characterized by the exact same $n-k$ most significant bits and varying k less significant bits. We represent such ranges as $b_{n-1}...b_k x...x$, with k trail x, where x stands for both 0 and 1. Each value range can be broken down into multiple such disjoint sub-ranges (disjoint union set [30]), and Figure 4 illustrates an example of how the value range $[20,22]$ is split into sub-ranges set. The specific bitwise binary operations need only to be executed on all combinations of sub-ranges pairs treating unknown bits as 1 when it tries to find the maximum result, and as 0 to find the minimum.

Modulo Operation. CoSense extends range analysis to support the modulo operation of the LLVM IR [2]. Similarly to the case of bitwise binary operations, this is necessary because broadly used by sensors' applications, and again the range information can be propagated to the subsequent operators for optimizations' wider code coverage. CoSense supports signed and unsigned dividends and divisors. This method is based on the properties of the modulo operator shown in Equation (2), where $mod(x,y)$ stands for $x \bmod y$.

$$\begin{cases} \bmod(-x,y) \equiv -\bmod(x,y) \\ \bmod([\underline{X},\overline{X}],[\underline{Y},\overline{Y}]) \equiv \bmod([\underline{X},\overline{X}],[-\overline{Y},-\underline{Y}]) \end{cases} \quad (2)$$

The general algorithm for the modulo operator is given in Algorithm 1, which shows the sub-ranges that can be optimized. Apart from these, CoSense performs a brute force algorithm as shown in Algorithm 2.

When one of the two value range operands is *degenerate*, i.e., of the form $[a,a]$, where a is the operand value, the modulo

operation between two value ranges is simplified, as shown in Table 2, where each line of a category represents a case in a C-like switch statement.

Algorithm 1: Unified Algorithm for Remainder

```
Function ModRange(X,X̄,Y,Ȳ):
    if X̄ < 0 then
        return −ModRange(−X̄,−X,Y,Ȳ);
    else if X̄ < 0 then
        [pos,p̄os] = ModRange(0,X̄,Y,Ȳ);
        [neg,n̄eg] = ModRange(X,−1,Y,Ȳ));
        return [min(pos,neg),max(p̄os,n̄eg)];
    else if Ȳ ≤ 0 then
        return ModRange(X,X̄,−Ȳ,−Y);
    else if Y ≤ 0 then
        return ModRange(X,X̄,1,max(−Y,Ȳ));
    else if X̄ − X ≥ Ȳ then
        return [0,Ȳ − 1]
    else if X̄ − X ≥ Y then
        [split,split] = ModRange(X,X̄,X̄ − X+1,Ȳ);
        return [min(0,split),max(X̄ − X − 1,split)];
    else if Y > X̄ then
        return [X,X̄];
    else if Ȳ > X̄ then
        return [0,X̄];
    else
        return GetAllValuesBetween[X,X̄,Y,Ȳ];
    end
End Function
```

Algorithm 2: Get All Values Between Operands

```
Function GetAllValuesBetween(X,X̄,Y,Ȳ):
    for i ∈ [X,X̄] do
        for j ∈ [Y,Ȳ] do
            temp = mod (i,j);
            min_value = min(temp,min_value);
            max_value = max(temp,max_value);
        end
    end
    return [min_value,max_value];
End Function
```

Composite Types. CoSense does not just support LLVM basic data types like *int*, *float*, and *double*, but also composite types like *array*, *structure*, and *union* when their range information is provided sensors' technical specifications. For arrays with the same type, CoSense only checks the first element; for structures with different types, CoSense saves the types of all elements; for union types, according to the combination of *bitcast*, *getelementptr*, and *store* instructions in LLVM IR, CoSense analyze its element range through a *map* that contains the value information.

Function Calls. CoSense propagates the variables' value range across all functions of a module. For externally defined functions, e.g., library functions, CoSense can also propagate the value ranges, if the function implementation is available.

Table 2. CoSense's interval analysis of the modulo operation for degenerate operands (i.e., value ranges of the form $[a,a]$).

Category	Range of mod(x,y)
$\underline{X}=\overline{X}$	
$\underline{Y}>X$	$[X,X]$
$\overline{Y}>X$	$[0,X]$
$\underline{Y}\geq X\div 2+1$	$[\ \mathrm{mod}\ (X,\overline{Y}),\ \mathrm{mod}\ (X,\underline{Y})\ ]$
$\overline{Y}\geq X\div 2+1$	$[\min(\ \mathrm{mod}\ ([X,X],[\underline{Y},X\div 2+1])),$ $\mathrm{mod}\ (X,X\div 2+1)\ ]$
Otherwise	$[\min(\ \mathrm{mod}\ (X,Y)),\max(\ \mathrm{mod}\ (X,Y))\ ]$
$\underline{Y}=\overline{Y}$	
$\lvert Y\rvert > \overline{X}-\underline{X}$ and $\mathrm{mod}\ (\underline{X},Y)\leq \mathrm{mod}\ (\overline{X},Y)$	$[\ \mathrm{mod}\ (\underline{X},Y),\ \mathrm{mod}\ (\overline{X},Y)\ ]$
Otherwise	$[0,\lvert Y\rvert-1]$

In general, the same function may be called from different locations of a program with different arguments, which may be characterized by different value ranges. Hence, a function cannot be blindly optimized for a single (set of) value range(s). Section 4.2 proposes function overloading to address this problem.

4.2 Range-Guided Function Overload

As already observed, a function is often called from different program's call sites with different arguments – hence, value ranges. Inspired by techniques like devirtualization, and programming languages that natively support function overloading (see C++ [55]), CoSense uses the inferred ranges for the function's arguments to emit specialized functions for each call site. Each specialized function can be further optimized by using the propagated range information.

In Figure 5, the Value Range Propagation deduces that the range of the function parameter of `foo` (top subfigure) is $[-16,16]$. CoSense will: i) clone function `foo`, ii) rename it to `foo_rng16` using the range bounds, and iii) replace the appropriate `foo` call site with a call to `foo_rng16`. Then, `foo_rng16` is further optimized by the Condition Simplification (Section 4.4), since for the call site in Figure 5(c) the value of parameter `a` is certain to be less than 20.

Duplication of overloaded functions is avoided using a hash map to elide them once the transformation has processed all code modules. Indeed, range-guided function overloading may cause an increase in the binary size, which we evaluate in Section 5. Therefore, our function overload optimization can be enabled or disabled via a compiler argument in CoSense, `-Os` disables the optimization that otherwise is enabled.

4.3 Type Compression

Tailoring the size of data types to the exact ranges they will take during execution is a critical optimization to reduce memory consumption, improving cache locality and speedup execution time. This is enabled by CoSense's Value Range Propagation and implemented in its *Type Compression* pass

```
// in [-16 g, 16 g] from Newton spec
typedef double bmx055xAcceleration;

double foo(double a) { return (a > 20) ? a + 100 : a * 100; }
```

(a)

```
void bar(bmx055xAcceleration a)
{
  ...

  double r = foo(a);
  ...
}
```

(b)

```
double foo_rng16(double x){...}

void bar(bmx055xAcceleration a)
{
  ...
  // x in interval [-16, 16]
  double r = foo_rng16(a);
  ...
}
```

(c)

Figure 5. Resulting code (bottom right) of CoSense's Range-Guided Function Overload when applied on original code (top and bottom left) using a sensor-based type and an associated called function `foo`. Using the value range of type, CoSense creates `foo_rng16`, a specialized function overload, and replaces calls to it wherever that range is in effect.

that is partly inspired by compressed instruction set architectures (ISAs).

CoSense internally uses double-precision floating point numbers to represent value ranges since it is convenient and adequate for data originating from physical systems via the Newton DSL. Further, it keeps track of the sign, and the relation to the type used by the infrastructure used (i.e., now LLVM). Type bookkeeping between CoSense and the surrounding infrastructure is unavoidable since traditional languages (source, IR) are strongly tied with a specific type system.

Type Compression operates on each instruction in the use-def chain that has value range information as follows: i) find smallest common data type of operand and result types, ii) insert appropriate casts, iii) merge redundant casts, iv) adjust types for sign, and v) adjust types for instruction data layout and alignment. Note that we do not alter the original instructions, but add cast operations instead. This allows for easy fallback to the original code when it is not possible to apply type compression. Table 3 shows which parts of an LLVM instruction are used in the first step of Type Compression. Merging redundant casts occurs by following the use-def chain and is necessary since cast insertion takes into account instruction-local information, thus potentially introducing spurious casts.

4.4 Condition Statements Simplification

Using the value range information, CoSense examines all branch conditions (i.e., `if/else` statements) and determines whether the result of the condition can be established statically. Subsequently, CoSense eliminates the redundant not-taken branches and the basic blocks associated with it by extending the classic branch elimination pass in LLVM. In turn, this enables the CFG simplification and other dead code elimination (DCE)-based compiler passes. A simple example of this behaviour, taken from the EEMBC Coremark-Pro benchmark [14], is presented in Figure 6.

Table 3. LLVM instructions whose types participate in Type Compression when value range information is available.

LLVM Inst	Result	Operand	LLVM Inst	Result	Operand
Phi	✓	✓	Load	✓	
Alloca	✓	✓	Store		✓
Binary	✓	✓	Switch		✓
Unary	✓		Return		✓
Cast	✓		Function call	✓	
GetElementPtr	✓		System call		

```
1  ...
2  %9 = and i32 %8, 2147483647 ; %9 in [1079274518, 1080017492]
3  ...
4  ; delete '%86 = ashr i32 %9, 20' and replace with '1029'
5  ...
6  %91 = ashr i32 %90, 20      ; %91 in [-1030, -1026]
7  %92 = and i32 %91, 2047     ; %92 in [1018, 1022]
8  ; rewrite '%93 = sub nsw i32 %86, %92'
9  %93 = sub nsw i32 1029, %92 ; %93 in [7, 11]
10 ; delete
       '%94 = icmp sgt i32 %93, 16', which is always 'false'
11 ; delete 'br i1 %94, label %95, label %119'
12 ; and remove the 'IF block %95'
13 119:                        ; ELSE block
14 ...
```

Figure 6. CoSense-enabled Dead Code Elimination (DCE) with Condition Statements Simplification and Constant Substitution determines the value interval bounds of `%86` (line 4 and 8) are equal to $[1029, 1029]$ and the result of `icmp` (line 10) will always be `false`. CoSense eliminates the associated instructions/variables/blocks.

Table 4. Functions extracted from the EEMBC and CHStone benchmark suites and used in our evaluation.

Benchmark Suite	Functions
EEMBC	`e_{exp, log, j0, y0, acosh, rem_pio2}, sincosf`
CHStone	`float64_{add, div, mul}`

4.5 Constant Substitution

Operations on variables on which CoSense maintains value range information can lead to trivial ranges where the lower and upper bounds are equal, i.e. a constant. In Figure 6 introduced above, the virtual register `%86` in line 4 results as a constant value where a value range "collapses" after a bitwise shift operation and CoSense is able to substitute the variable with a constant value.

5 Empirical Evaluation

We evaluate CoSense to assess its potential in speeding up execution time, reducing binary size, reducing energy consumption on microbenchmarks and real-world applications. Moreover, we analyzed potential (compilation) overheads.

5.1 Hardware and Software Setup

Hardware. We built and evaluated CoSense on different hardware, including one laptop, a development board, and two

embedded platforms. The laptop is based on the x86 ISA, while the others are based on ARM. The embedded platforms are the Crazyflie quadrotor [34] and the Warp sensor platform [53], including an ultra-low-power microcontroller. CoSense supports any other ISA targeted by LLVM. Evaluating on x86 allowed us to explore our approach on different ISA.

The technical specifications of such hardware follow:

- x86 (laptop): Dell G5 5500 (12 cores Intel i7-10750H, 2.60GHz), with 16GB of RAM.
- ARM (dev-board): ROC-RK3328-CC firefly (quad-Core Cortex-A53 64-bit processor, 1.50GHz), with 4GB of RAM.
- Crazyflie's ARM microcontroller unit (MCU): Cortex-M4 based microcontroller, 168MHz, with 192kb SRAM and 1Mb program flash memory.
- Warp's ARM MCU: Cortex-M0+ based microcontroller, 48MHz, with 2KB SRAM and 32 KB flash.

Although the first three platforms accommodate floating-point units (FPUs), the Warp platform, being ultra-low-power, requires software floating-point (*softfloat*) support. Hence, in the evaluation, we include use cases of *softfloat* libraries.

Hardwdare Sensors. Both embedded platforms have 3-axis accelerometers and gyroscopes. The Crazyflie platform has also a pressure sensor, but we do not present optimizations to the code that uses this sensor because they were marginal.

Software. The x86 laptop and the ARM dev-board use Linux Ubuntu 22.04.1 LTS. CoSense capitalizes on and extends the Newton [44] compiler, while the analysis and optimization passes are implemented on top of LLVM (version 13.0.1) [41]. CoSense implements 5 LLVM passes in about 7200 lines of code (LoC).

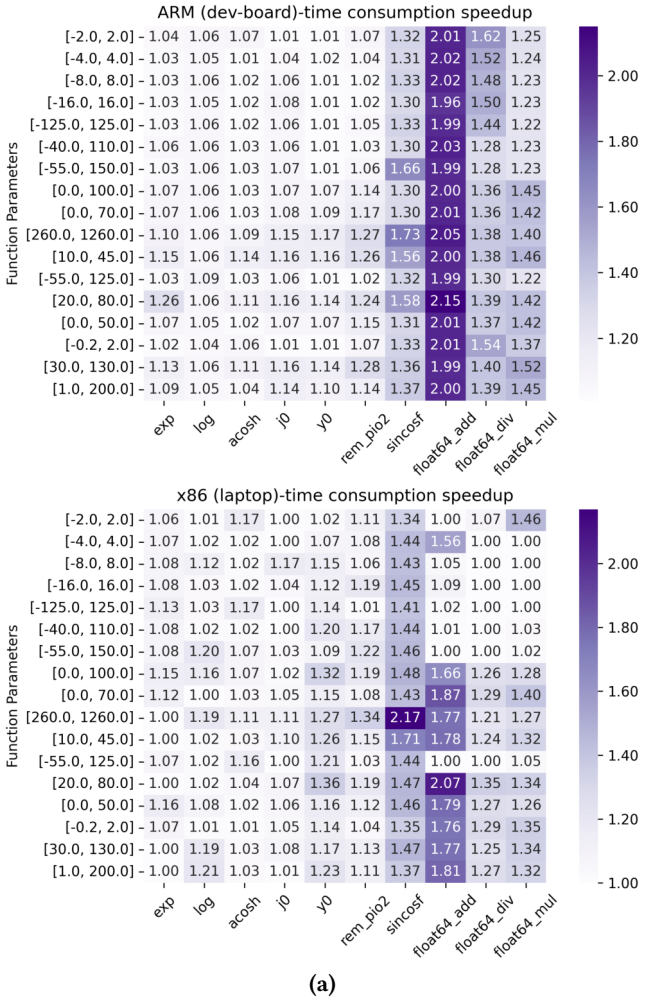
Software Configuration. CoSense operates on unoptimized LLVM IR in static single-assignment (SSA) form with full debug information obtained using clang (`-O0, -mem2reg`). After performing our analyses and transformations, we hand over the modified IR back to LLVM where it is compiled with `-O3`. CoSense is also compatible with `-Os and -LTO`.

5.2 Benchmarks

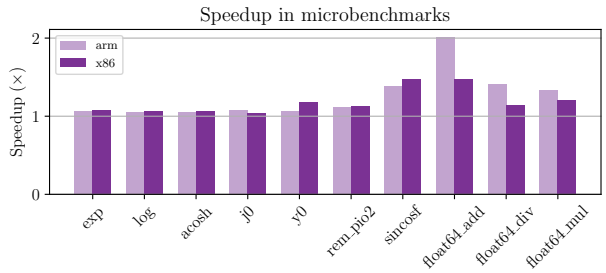
We used microbenchmarks and two real-world applications.

Microbenchmarks. The microbenchmarks comprise mathematical functions extracted from the EEMBC and CHStone benchmark suites [28, 36]. We chose common mathematical functions from EEMBC and software floating-point (FP) arithmetic with double precision from CHStone, see Table 4, which are widely used by the embedded systems community – thus, an important optimization target. Each function is statically linked to a "driver program" in order to evaluate CoSense's impact on code size. For accurate time measurements, we follow guidelines in [1]. In fact, we used `clock_gettime()` [4] to time 50,000 executions of each function ($stddev \leq 1\%$).

Since the optimizations of CoSense do not only depend on the target function but also on the value range of its parameters – including the range span, we evaluate with multiple



(a)

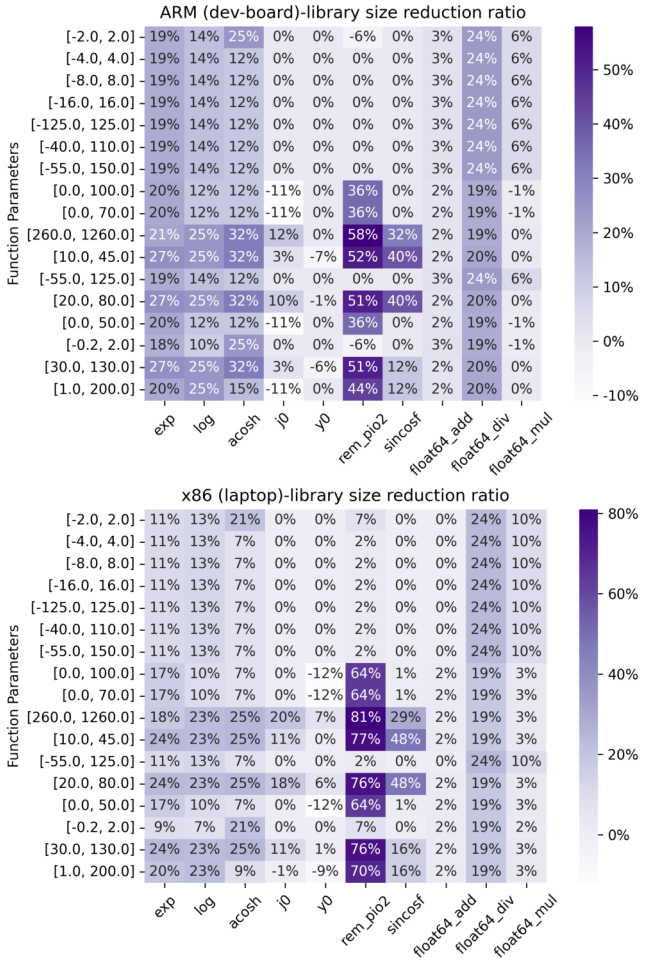


(b)

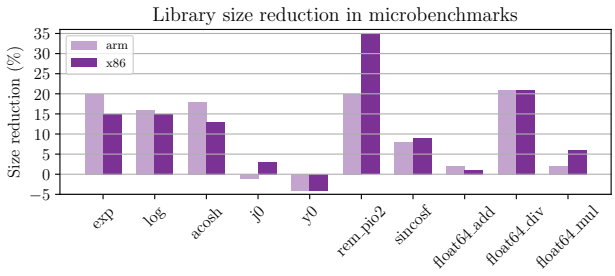
Figure 7. Heatmaps (top) and average speedup (bottom) on the ARM dev-board and x86 laptop.

value ranges. To get a realistic set of different such ranges, we examine several widely-used off-the-shelf sensors [3, 5, 6, 8–12, 15] and glean multiple real-world value ranges. Table 5 presents the different value ranges we use as reference in our experiments, along with their quantities and physical units.

Real-world Applications. We evaluate CoSense on two real-world embedded platforms: Crazyflie and Warp board.



(a)



(b)

Figure 8. Heatmaps (top) and average(bottom) library size reduction on the ARM dev-board and x86 laptop.

For easy-to-understand experiments, for each platform, we optimize a single computationally intensive module only, while leaving the rest of the code unmodified. The computationally intensive part for the Warp firmware [13] is the Madgwick state estimation filter [17], whereas for the Crazyflie we optimize the inertial measurement unit (IMU) sensor driver module [7]. On the Crazyflie, we measured time using the *DWT Program Counter Sample Register*, while on the Warp board,

Table 5. Physical quantities of popular sensors used in our microbenchmarks. Note the same physical quantity in a sensor can have different resolutions (e.g., BMX055), shown in parentheses.

Sensor	Quantity	Range	Unit
BMX055	Accelerometer (0.98 mg/LSB)	[-2, 2]	g
BMX055	Accelerometer (1.95 mg/LSB)	[-4, 4]	g
BMX055	Accelerometer (3.91 mg/LSB)	[-8, 8]	g
BMX055	Accelerometer (7.81 mg/LSB)	[-16, 16]	g
BMX055	Gyroscope	[-125, 125]	$^\circ/s$
LM35C	Temperature	[-40, 110]	$^\circ C$
LM35A	Temperature	[-55, 150]	$^\circ C$
LM35D	Temperature	[0, 100]	$^\circ C$
LM35DH	Temperature	[0, 70]	$^\circ C$
LPS25H	Pressure	[260, 1260]	hPa
MAX31820	Temperature (0.5 $^\circ C$)	[10, 45]	$^\circ C$
MAX31820	Temperature (2 $^\circ C$)	[-55, 125]	$^\circ C$
D11	Humidity	[20, 80]	%RH
D11	Temperature	[0, 50]	$^\circ C$
LMP92064	Voltage	[-0.2, 2]	V
PCE-353	Sound	[30, 130]	dB
LLS05-A	Light	[1, 200]	Lux

Table 6. Physical quantities of sensors used in real-world applications of our evaluation. Warp board uses BMX055 and Crazyflie uses MPU-9250.

Sensor	Quantity	Range	Unit
BMX055	Accelerometer	[-16, 16]	g
BMX055	Gyroscope	[-2000, 2000]	$^\circ/s$
BMX055	Magnetometer (x, y axis)	[-1300, 1300]	μT
BMX055	Magnetometer (z axis)	[-2500, 2500]	μT
MPU-9250	Accelerometer	[-16, 16]	g
MPU-9250	Gyroscope	[-2000, 2000]	$^\circ/s$
MPU-9250	Magnetometer	[-4800, 4800]	μT

we used the *SysTick* timer, timer interrupts have been disabled to avoid interference in measurements. We repeated each measurement 50 times on each platform achieving $stddev \leq 1\%$.

The on-board measurement of the energy consumption of low-power (battery-operated) embedded devices is tricky because the action of measuring the power affects their energy consumption. Hence, we substitute their battery with a power supply with logging capabilities. Specifically, we use the Monsoon Power Monitor 2405 [19] to log power consumption. Also, for evaluation purposes, we modify the real-world applications code replacing their main (infinite) control loop with a fixed number of iterations – e.g., 10,000, then shut down the embedded device.

As an additional input to CoSense we use the BMX055 Newton specifications for Warp and MPU-9250 for Crazyflie. For ranges and units of the physical quantities see Table 6.

5.3 Evaluation of Microbenchmarks

We used microbenchmarks to evaluate CoSense's execution time speedup, binary shrinkage, compilation time overhead.

Execution Time. Heatmaps in Figure 7(a) report for each microbenchmark the execution time speedup varying the

```
libc_sincosf(lps25hPressure y, ...) {
    ...
    else if (abstop12(y) < abstop12(120.0f)) {
        x = reduce_fast(x, p, &n);
        ...
    }
    ...
}
```

Figure 9. Code extract from the `sincosf` function of the CHStone benchmark for which CoSense eliminates the `else if` branch, yielding a high execution speedup.

parameter range on x86 (laptop) and on ARM (dev-board). In Figure 7(b) we compare averages between ISAs.

First, we observe that all functions exhibit a speedup for most of the parameter ranges on both architectures, with only a few functions not demonstrating any speedup for some of the ranges. For example, the ARM version of `float64_add` shows speedup for every parameter range given, whereas the x86 version of `float64_div` for the range $[-4.0, 4.0]$ does not exhibit any speedup. Hence, we conclude that the speedup is highly dependent on the input function.

Second, we observe that some ranges yield better speedups for every function than all others. For instance, the range [260.0, 1260.0] leads to the highest speedups on average in every function and in both architectures. Therefore, it seems that when the range is known to take only positive and/or relatively high values, the functions in question can be more effectively optimized. To illustrate this point, we examine the source code of `sincosf`, which demonstrates its highest speedup for the range of [260.0, 1260.0] for both architectures. We show a code extract of `sincosf` in Figure 9. For the range [260.0, 1260.0], our compiler proves that the range of `abstop12(y)` will always be higher than the range of `abstop12(120.0f)`, eliminating the `else if` branch. This is highly beneficial in this case, because it eliminates the calls to `abstop` inside the condition of this branch, contributing to the overall speedup, compared to other ranges that did not eliminate this branch.

Third, we observe that the length of the range does not impact the optimization result as much as the limits of the range. We conclude that most of the optimization of these microbenchmarks is harvested through knowing the exact limits of the range, rather than how broad this range is.

Binary Size. The code size reduction results of every microbenchmark and parameter range combination are shown in Figure 8(a) – trends are similar to the speedup case. Averages among ISAs are reported in Figure 8(b).

The main differences are the functions `j0` and `y0`, where CoSense leads to a small code size increase in some ranges. In these cases, there are several function calls in the source code, so the function overload mechanism adds several of their optimized instances, which results in adding more lines of code than those reduced by all other CoSense optimizations overall. Nevertheless, as explained in Section 4.2, function overloading can be disabled in these cases.

Table 7. Compilation time overhead of microbenchmarks in Table 4 on x86 (laptop) and ARM (dev-board).

Test case	Function calls	LoC of IR	x86_64 overhead	aarch64 overhead
e_exp	0	443	1.89%	2.13%
e_log	0	444	1.96%	2.09%
e_acosh	0	213	1.45%	1.70%
e_j0	2	856	2.83%	3.30%
e_y0	5	1202	4.21%	5.19%
e_rem_pio2	1	2219	13.72%	19.42%
sincosf	23	652	4.22%	6.97%
float64_add	74	1522	18.79%	21.91%
float64_div	33	2148	9.83%	11.30%
float64_mul	24	1597	7.09%	7.92%

Table 8. CoSense's performance improvement on real-world applications for execution time speedup and binary size reduction. The optimizations are applied to the modules in the second column which constitutes a computationally intensive part of the applications.

Application	Module	Speedup	Binary Size Reduction	CompTime Overhead
Warp	Madgwick Filter	1.7x	13.0%	26.58%
Crazyflie	IMU Driver	1.5x	0.6%	24.01%

Compilation Time. We evaluated the compilation time overhead introduced by CoSense on microbenchmarks, and we found the overhead mostly proportional to the lines of IR code or the number of function calls. Also, the overhead is higher on ARM. Results are shown in Table 7: almost every microbenchmark shows that the longer the IR code, the higher the compilation time overhead – this is expected because we traverse the entire IR code. The only exception comes from *float64_add*, which does 74 function calls. Since CoSense iterates through every function call and may overload it in the function overload optimization (Section 4.2), the compilation overhead is also proportional to the number of function calls.

5.3.1 Performance Improvements with Support for Bitwise Binary Operations.

We evaluate the execution time speedup and binary code size reduction of five microbenchmarks that contain bitwise binary operations in Figure 10 with and without CoSense's support for such operations on ARM and x86. Supporting bitwise binary operations always leads to speedups. However, the size of the binaries is only reduced for `sincosf`, `float64_add`, and `float64_div`. This is because of our overload function optimization, which can be disabled when compiling to reduce the binary size.

5.3.2 Performance Improvements with Support for Modulo Operation.

We tested the performance of the `arm_cmplx_mag_q15.c` from the CMSIS DSP Software Library [16] because it includes the modulo operation and it is used by both our real-world applications, We tested for different value ranges. We got up to 8.5× speedup (range [1024,1054]) when

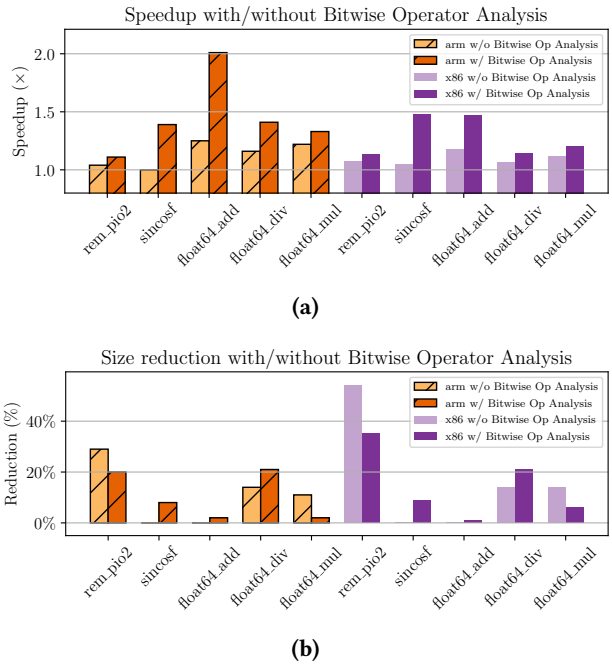


Figure 10. Performance comparison with and without bitwise operator range analysis support (higher is better). The dashed and plain bars show the comparison on ARM and x86, respectively. The speedups are always better when supporting the bitwise operators, but the size reductions are slightly impacted in `sincosf`, `float64_add`, and `float64_div`, due to the function overload optimization.

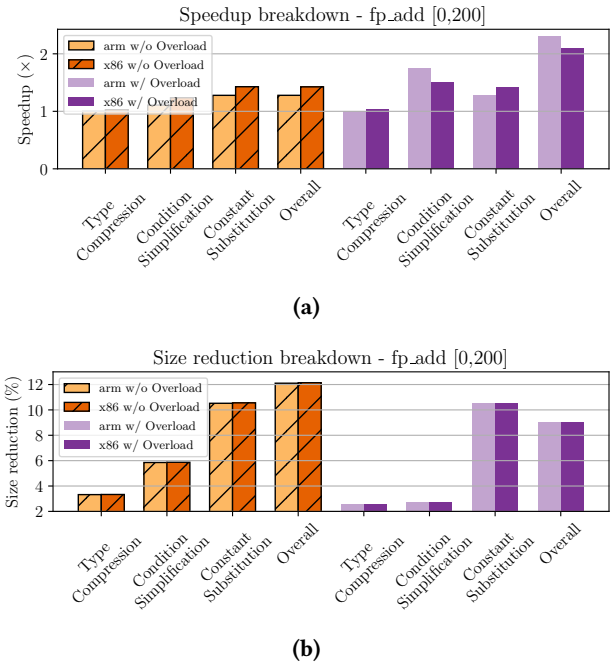


Figure 11. Speedup and size reduction with each CoSense's optimization for *float64_add* (soft-float) (higher is better). Dashed bars show the performance of each optimization individually. Plain bars also include function overloading.

the modulo operation is supported versus when it is not supported.

5.3.3 Performance Breakdown per Optimization.

To examine the effect of range-based function overloading on the benefit of the other optimizations presented, we break down the performance of each optimization individually with and without function overloading. Figure 11(a) and Figure 11(b) showcase this breakdown for the *float64_add* soft-float microbenchmark, assumming the range [0,200], when run on ARM and x86. The *float64_add* function is one the most commonly used, whereas the range [0,200] is representative of the readings of many sensors.

For both architectures, we observe in Figure 11(a) that combining each optimization with the function overloading improves the speedup for each optimization up to 1.5x, e.g., in the case of condition statements simplification, and up to more than double when all optimizations are applied. Using different versions of functions maximizes the benefit obtained from each optimization individually, since it allows for the most specific ranges to be used per function clone, depending on the real parameters, rather than just optimizing the function based on the more general ranges of the formal parameters.

However, we observe that applying the function overloading reduces the benefit of code size. For example, as shown in Figure 11(b), the type compression size reduction drops to almost 2%, from 6%, while the overall size reduction from all optimizations is reduced from 12% to around 9%. Nevertheless, there is still a size reduction and not an increase in size. Overall, as mentioned in Section 4.2, function overloading could offer a tradeoff to the developers who, with the help of a compiler flag, could choose between maximizing speedup or balancing between speedup and size reduction.

5.4 Evaluation of Real-World Applications

Table 8 shows CoSense's execution time speedup, binary size reduction and compilation time overhead for Warp and for Crazyflie. We observe significant speedups for each application, whereas the binary size reductions are notable only for Warp. Finally, the compilation time overhead is around 25%.

CoSense effectively reduces the execution time when applied to the computationally intensive code of each application. On the other hand, since we do not optimize the whole codebase, we only get a significant binary size reduction in the case of the smaller Warp firmware [13], where the optimized module has a comparable size to the sum of the rest of the code. Similarly to what was observed with microbenchmarks, the compilation time overhead is proportional to the lines of IR code (1823 for Warp's Madgwick Filter, and 4614 for Crazyflie's IMU Driver) or the number of function calls (11 for Warp's Madgwick Filter, and 9 for Crazyflie's IMU Driver).

Energy Consumption. CoSense saves up to 9.69% energy on Warp and 30.43% on Crazyflie. The reduction in energy

consumption is mainly due to a shorter execution time, and not because of lower power drawn (which we didn't observe). In fact, shorter execution time equal less energy to complete a specific task. Anyway, CoSense can potentially further save energy due to reducing data bus activity and increasing the cache efficiency that benefits from the Type Compression optimization in Section 4.3, which we couldn't fully assess[2].

6 Related Work

Exploiting Physical Information. Using physical information to enhance the type system of a programming language is not a new concept [21, 32, 39] though it has not caught on lately. Software solutions exist for most popular programming languages [18, 22, 23, 35, 50, 52]. Obtaining variable range through profiling [25, 27, 27, 29, 40, 45, 57] can also partially achieve similar effects to CoSense. However, CoSense can get leverage more accurate ranges as input, as well as information like accuracy error, and there is no need for tedious multiple runs for profiling.

Information (Range) Propagation. To the best of our knowledge, Harrison's work [37] is the first introducing range analysis and propagation, which (only) eliminates dead code. Blume and Eigenmann [24] introduce symbolic range propagation and related optimization for Parallelizing Compilers, but the idea cannot be generalized to general-purpose compilers. In addition, LLVM implements similar techniques for loop parallelization, loop strength reduction, etc. CoSense is general-purpose – it targets all variables, not just variables as loop indexes, and it is not limited to dead code elimination – it supports several other optimizations.

Range Information. Compiler intrinsics, which are hand-written by the programmer, can constrain the range value of a variable. However, first, we found that assumption propagation is limited (i.e., the latest LLVM compiler framework does not handle `GetElementPtr` and `Cast` instructions or cross-function call boundaries). CoSense provides a more powerful propagation strategy, overcoming these limitations. Second, programmers need to manually annotate the source code with the corresponding assumptions and maintain their validity over the software lifetime (accounting for hardware deployment changes, etc). On the other hand, CoSense exploits the sensor DSL which the programmer can concisely and independently modify. Third, this assumption intrinsics takes effect during the code generation at the compiler backend. CoSense operates at the IR level, exploiting by more comprehensive and generic compiler optimizations.

Bitwidth Minimization. Both *Bitwise* [54] and *Minibit* [43] use variables' range information to minimize the variable's bitwidth in the case of a compiler and an FPGA respectively. While similar to CoSense in using value range propagation and similar to CoSense's Type Compression in minimizing the bitwidth, those are limited to certain code patterns, not general-purpose like CoSense. Plus they cannot support other optimizations.

[2]Both the Warp and the Crazyflie have very limited performance monitoring units (PMUs), which hinders an accurate assessment of buses and caches.

7 Discussion

Integrating CoSense into existing embedded systems is not automatic, but requires (minimal) programmer intervention. Given a platform, the programmer has to come up with a specification for each sensor, then he/she has to match the relevant sensors with the variables in their source code by providing type definitions similar to Figure 5(a). Note that in most of the cases, specifications need not be written from scratch as these can be shipped by the manufacturer or shared as open-source code. In the future, specifications may be machine extracted from sensors datasheet, making CoSense automatic.

To ensure a program will not introduce bugs by erroneous sensor specifications, CoSense retains assertions and provides optional logging to ease debugging.

The MLIR [42] project is a possible future host for CoSense by creating a new MLIR dialect for physical computation incorporating the design and implementation of Newton [44].

8 Conclusion

We introduced CoSense, the first attempt to incorporate sensor physical information into a state-of-the-practice compiler to enable novel and existing compiler optimizations. CoSense extends LLVM and uses the Newton programming language as a DSL for sensor technical specifications, which are propagated along an entire program, including among arithmetic operations. CoSense focuses on value ranges, and extends the state-of-the-art on interval arithmetic by supporting bitwise binary operations and the modulo operation. Moreover, it introduces several optimization passes that exploit the value range.

For microbenchmarks, CoSense achieves a 1.18× geomean (up to 2.17×) speedup in execution time and 12.35% reduction on average in binary code size on x86, and a 1.23× geomean (up to 2.15×) speedup in execution time and 10.95% reduction on average in binary code size on ARM, with a compilation overhead proportional to the lines of IR and the number of function calls. For real-world applications (Warp, Crazyflie), CoSense achieves 1.70× and 1.50× speedup in execution time, 13.0% and 0.6% binary code reduction respectively, and up to 30.43% lower energy consumption.

Acknowledgments

This work was largely supported by the UK EPSRC under the grant EP/V004654/1.

References

[1] 2003. *LLVM Benchmarking Tips.* Retrieved 15-10-2023 from https://llvm.org/docs/Benchmarking.html

[2] 2003. *LLVM Language Reference Manual.* Retrieved 03-05-2023 from https://llvm.org/docs/LangRef.html

[3] 2005. *SENBA OPTICAL ELECTRONIC LLS05-A Linear Light Sensor.* Retrieved 03-05-2023 from https://www.ledsales.com.au/pdf/lightsensor.pdf

[4] 2011. *GNU Libc Time Types.* Retrieved 15-10-2023 from https://www.gnu.org/software/libc/manual/html_node/Time-Types.html

[5] 2013. *maxim integrated 1-Wire Ambient Temperature Sensor.* Retrieved 03-05-2023 from https://cdn.sparkfun.com/datasheets/Sensors/Temp/MAX31820.pdf

[6] 2014. *Bosch BMX055 datasheet.* Retrieved 03-03-2023 from https://www.mouser.com/datasheet/2/783/BST_BMX055_DS000-1509552.pdf

[7] 2014. *Crazyflie sensor fusion.* Retrieved 22-3-2023 from https://github.com/bitcraze/crazyflie-firmware/blob/master/src/hal/src/sensors_bmi088_bmp388.c

[8] 2014. *STMicroelectronics LPS25H MEMS pressure sensor.* Retrieved 03-05-2023 from https://www.mouser.com/datasheet/2/783/BST_BMX055_DS000-1509552.pdf

[9] 2014. *Texas Instruments LMP92064 Precision Low-Side, 125-kSps Simultaneous Sampling.* Retrieved 03-05-2023 from https://www.ti.com/lit/ds/symlink/lmp92064.pdf

[10] 2015. *PCE Instruments PCE-353 LEQ Sound Level Meter Datasheet.* Retrieved 03-05-2023 from https://hub-4.com/files/document/4377/datasheetclass2soundlevelmeterpce353en.pdf

[11] 2016. *InvenSense MPU-9250 Product Specification Revision 1.1.* Retrieved 03-05-2023 from https://invensense.tdk.com/wp-content/uploads/2015/02/PS-MPU-9250A-01-v1.1.pdf

[12] 2017. *Texas Instruments LM35 1C high voltage analog temperature sensor.* Retrieved 03-05-2023 from https://www.ti.com/product/LM35#params

[13] 2018. *Warp Board Firmware.* Retrieved 22-3-2023 from https://github.com/physical-computation/Warp-firmware

[14] 2019. *e_rem_pio2.c from EEMBC coremark-pro benchmark.* Retrieved 22-3-2023 from https://github.com/eembc/coremark-pro/blob/main/mith/al/fdlibm/e_rem_pio2.c#LL132C1-L135C61

[15] 2021. *DHT 11 Humidity and Temperature Sensor.* Retrieved 03-05-2023 from https://www.mouser.com/datasheet/2/758/DHT11-Technical-Data-Sheet-Translated-Version-1143054.pdf

[16] 2022. *CMSIS DSP Software Library.* Retrieved 15-10-2023 from https://www.keil.com/pack/doc/CMSIS/DSP/html/arm__cmplx__mag__q15_8c.html

[17] 2022. *Fusion.* Retrieved 30-9-2022 from https://github.com/xioTechnologies/Fusion/

[18] 2022. *units: A domain-specific type system for dimensional analysis.* Retrieved 15-10-2023 from https://hackage.haskell.org/package/units

[19] 2023. *Monsoon Power Monitor Python Library.* Retrieved 15-10-2023 from https://github.com/msoon/PyMonsoon#usage-notes

[20] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 265–283.

[21] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L Steele Jr, Sam Tobin-Hochstadt, Joao Dias, Carl Eastlund, et al. 2005. The Fortress language specification. *Sun Microsystems* 139, 140 (2005), 116.

[22] Eric Allen, David Chase, Victor Luchangco, Jan-Willem Maessen, and Guy L Steele Jr. 2004. Object-oriented units of measurement. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 384–403. https://doi.org/10.1145/1028976.1029008

[23] Geoffrey Biggs and Bruce MacDonald. 2005. A design for dimensional analysis in robotics. In *Third International Conference on Computational Intelligence, Robotics and Autonomous Systems, Singapore*. Citeseer.

[24] William Blume and Rudolf Eigenmann. 1995. Symbolic range propagation. In *Proceedings of 9th International Parallel Processing Symposium*. IEEE, 357–363. https://doi.org/10.1109/TSE.1977.231133

[25] Brad Calder, Peter Feller, and Alan Eustace. 1997. Value profiling. In *Proceedings of 30th Annual International Symposium on Microarchitecture*. IEEE, 259–269. https://doi.org/10.1109/MICRO.1997.645816

[26] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[27] W Chen, R Bringmann, S Mahlke, Sadun Anik, Tokuzo Kiyohara, N Warter, D Lavery, W M Hwu, R Hank, and J Gyllenhaal. 1993. Using profile information to assist advanced compiler optimization and scheduling. In *Languages and Compilers for Parallel Computing: 5th International Workshop New Haven, Connecticut, USA, August 3–5, 1992 Proceedings 5*. Springer, 31–48. https://doi.org/10.1007/3-540-57502-2_38

[28] Embedded Microprocessor Benchmark Consortium et al. 2008. EEMBC benchmark suite.

[29] Thomas M Conte, Burzin A Patel, Kishore N Menezes, and J Stan Cox. 1996. Hardware-based profiling: An effective technique for profile-driven optimization. *International Journal of Parallel Programming* 24 (1996), 187–206. https://doi.org/10.1007/BF03356747

[30] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, 3rd Edition.* MIT Press. http://mitpress.mit.edu/books/introduction-algorithms

[31] Timur Doumler. 2022. Portable assumptions. (2022).

[32] A Eliasen. 2004. Frink-A Language for Understanding the Physical World. In *Lightweight Languages 2004 (LL4) Conference, Massachusetts*.

[33] Mohammad Ali Ghodrat, Tony Givargis, and Alex Nicolau. 2008. Control flow optimization in loops using interval analysis. In *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*. 157–166. https://doi.org/10.1145/1450095.1450120

[34] Wojciech Giernacki, Mateusz Skwierczyński, Wojciech Witwicki, Paweλ Wroński, and Piotr Kozierski. 2017. Crazyflie 2.0 quadrotor as a platform for research and education in robotics and control engineering. In *2017 22nd International Conference on Methods and Models in Automation and Robotics (MMAR)*. IEEE, 37–42. https://doi.org/10.1109/MMAR.2017.8046794

[35] Adam Gundry. 2015. A typechecker plugin for units of measure: Domain-specific constraint solving in GHC Haskell. *ACM SIGPLAN Notices* 50, 12 (2015), 11–22. https://doi.org/10.1145/2887747.2804305

[36] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii. 2008. Chstone: A benchmark program suite for practical c-based high-level synthesis. In *2008 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 1192–1195. https://doi.org/10.1109/ISCAS.2008.4541637

[37] William H. Harrison. 1977. Compiler analysis of the value ranges for variables. *IEEE Transactions on software engineering* 3 (1977), 243–250.

[38] Chenyi Hu. [n. d.]. Interval Computing and Information Technology. ([n. d.]).

[39] Andrew Kennedy. 2009. Types for units-of-measure: Theory and practice. In *Central European Functional Programming School*. Springer, 268–305. https://doi.org/10.1007/978-3-642-17685-2_8

[40] Hyesoon Kim, José A Joao, Onur Mutlu, and Yale N Patt. 2007. Profile-assisted compiler support for dynamic predication in diverge-merge processors. In *International Symposium on Code Generation and Optimization (CGO'07)*. IEEE, 367–378. https://doi.org/10.1109/CGO.2007.31

[41] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86. https://doi.org/10.1109/CGO.2004.1281665

[42] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A compiler infrastructure for the end of Moore's law. *arXiv preprint arXiv:2002.11054* (2020). https://doi.org/10.48550/arXiv.2002.11054

[43] Dong-U Lee, Altaf Abdul Gaffar, Oskar Mencer, and Wayne Luk. 2005. Minibit: bit-width optimization via affine arithmetic. In *Proceedings*

of the 42nd annual Design Automation Conference. 837–840.

[44] Jonathan Lim and Phillip Stanley-Marbell. 2018. Newton: A language for describing physics. *arXiv preprint arXiv:1811.04626* (2018). https://doi.org/10.48550/arXiv.1811.04626

[45] Jin Lin, Tong Chen, Wei-Chung Hsu, Pen-Chung Yew, Roy Dz-Ching Ju, Tin-Fook Ngai, and Sun Chan. 2003. A compiler framework for speculative analysis and optimizations. *ACM SIGPLAN Notices* 38, 5 (2003), 289–299. https://doi.org/10.1145/780822.781164

[46] SK Mahato and AK Bhunia. 2006. Interval-arithmetic-oriented interval computing technique for global optimization. *Applied Mathematics Research Express* 2006 (2006), 69642.

[47] Pei Mu. 2024. CoSense Artifact. https://doi.org/10.5281/zenodo.10577943

[48] Tsuneo Nakanishi, Kazuki Joe, Constantine D Polychronopoulos, and Akira Fukuda. 1999. The modulo interval: A simple and practical representation for program analysis. In *1999 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 91–96. https://doi.org/10.1109/PACT.1999.807422

[49] Tsuneo Nakanishit and Akira Fukudat. 2001. Modulo interval arithmetic and its application to program analysis. *IPSJ Journal* (2001).

[50] Grant W Petty. 2001. Automated computation and consistency checking of physical dimensions and units in scientific programs. *Software: Practice and Experience* 31, 11 (2001), 1067–1076. https://doi.org/10.1002/spe.401

[51] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* 48, 6 (2013), 519–530. https://doi.org/10.1145/2499370.2462176

[52] Grigore Rosu and Feng Chen. 2003. Certifying measurement unit safety policy. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.* IEEE, 304–309. https://doi.org/10.1109/ASE.2003.1240326

[53] P. Stanley-Marbell and M. Rinard. 2020. Warp: A Hardware Platform for Efficient Multi-Modal Sensing with Adaptive Approximation. *IEEE Micro* 40, 1 (Jan 2020), 57–66. https://doi.org/10.1109/MM.2019.2951004

[54] Mark William Stephenson. 2000. *Bitwise: Optimizing bitwidths using data-range propagation.* Ph. D. Dissertation. Massachusetts Institute of Technology.

[55] Bjarne Stroustrup. 1984. Operator overloading in C++. In *Conference on System Implementation Languages: Experience & Assessment ('84 Proceedings)*.

[56] Vasileios Tsoutsouras, Sam Willis, and Phillip Stanley-Marbell. 2021. Deriving equations from sensor data using dimensional function synthesis. *Commun. ACM* 64, 7 (2021), 91–99. https://doi.org/10.1145/3358218

[57] John Whaley. 2000. A portable sampling-based profiler for Java virtual machines. In *Proceedings of the ACM 2000 conference on Java Grande*. 78–87.

Received 25-OCT-2023; accepted 2023-12-23