CoSense: Compiler Optimizations using Sensor Technical Specifications

Citation for published version:

Digital Object Identifier (DOI):
10.1145/3640537.3641576

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction (CC ‘24), March 2–3, 2024, Edinburgh, UK

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
CoSense: Compiler Optimizations using Sensor Technical Specifications

Pei Mu
University of Edinburgh
peimue@ed.ac.uk

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

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

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

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

Phillip Stanley-Marbell
University of Cambridge
philipp.stanley-marbell@eng.cam.ac.uk

Antonio Barbalace
University of Edinburgh
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 $1.18 \times$ geometric speedup in execution time and $12.35\%$ reduction on average in binary code size with $4.66\%$ compilation time overhead on x86, and $1.23 \times$ geometric 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 $1.70 \times$ and $1.50 \times$ 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:

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 (dev-board 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 Warps board [53] and the Inertial Measurement Units (IMUs) driver [7] on a Crazyflie quadrotor platform [34]. We show that on microbenchmarks, CoSense achieves a geometric 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 Warps, 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 [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;

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 multidimensional 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

1https://github.com/systems-nuts/CoSense
Table 1. Interval arithmetic of binary operations supported by CoSense as defined in prior work [48].

<table>
<thead>
<tr>
<th>Expr</th>
<th>Interval Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c = a + b$</td>
<td>$[A + B, \bar{a} + \bar{B}]$</td>
</tr>
<tr>
<td>$c = a - b$</td>
<td>$[\bar{A} - B, \bar{a} - \bar{B}]$</td>
</tr>
<tr>
<td>$c = a \times b$</td>
<td>$[\min(a, \max(x)), \max(x)]$, $x \in (A \times \overline{B}, \overline{A} \times B)$</td>
</tr>
<tr>
<td>$c = a \div b$</td>
<td>$[\min(x), \max(x)]$, $x \in (A \div \overline{B}, \overline{A} \div B)$</td>
</tr>
<tr>
<td>$c = a &amp; b$</td>
<td>$[\min(x), \max(x)]$, $x \in (A &amp; \overline{B}, \overline{A} &amp; B)$</td>
</tr>
<tr>
<td>$c = a | b$</td>
<td>$[\min(x), \max(x)]$, $x \in (A | \overline{B}, \overline{A} | B)$</td>
</tr>
</tbody>
</table>

Table 1 recap's 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.

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.

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.

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 $[-55^\circ C, 150^\circ C]$, CoSense’s Value Range Propagation analysis is able to propagate this information to Fahrenheit units.

```c
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  }
```
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: ∧, OR: ∨, XOR: ⊕), (b) the modulo (or remainder) operation. We describe these below together with how we handle composite types, and how we propagate ranges across functions.

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 \( b_{n-1} \ldots b_0, 2^k \) 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 \( b_{n-1} \ldots b_k x \ldots x \), 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 \( x \mod y \).

\[
\begin{align*}
\mod(x, y) & \equiv - \mod(x, y) \\
\mod([x], [y]) & \equiv \mod([x], [-y])
\end{align*}
\]

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

\begin{figure*}[h]
\centering
\includegraphics[width=\textwidth]{figure4.png}
\caption{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.}
\end{figure*}
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

```plaintext
Function ModRange([X, X, Y, Y]):
    if X < 0 then
        return ModRange([-X, X, Y, Y]);
    else if X < 0 then
        [pos, neg] = ModRange(0, X, Y, Y);
        return [min(pos, neg), max(pos, neg)];
    else if Y < 0 then
        return ModRange(X, X, -Y, -Y);
    else if Y < 0 then
        return ModRange(X, X, 1, max(-Y, Y));
    else if X - X ≥ Y then
        return (0, Y - 1);
    else if X - X ≥ Y then
        [split, split] = ModRange(X, X, X, X, Y, Y);
        return [min(0, split), max(X - X - 1, split)];
    else if Y > X then
        return X, X;
    else if Y > X then
        return (0, X);
    else
        return GetAllValuesBetween(X, X, Y, Y);
end
End Function
```

**Algorithm 2:** Get All Values Between Operands

```plaintext
Function GetAllValuesBetween([X, X, Y, Y]):
    for i ∈ [X, X] do
        for j ∈ [Y, Y] 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>
<thead>
<tr>
<th>Category</th>
<th>Range of mod(x,y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>X=X</td>
<td>[X,X]</td>
</tr>
<tr>
<td>Y&gt;X</td>
<td>[0,X]</td>
</tr>
<tr>
<td>Y ≥ X+2+1</td>
<td>[mod(X,Y), mod(X,Y)]</td>
</tr>
<tr>
<td>Y ≥ X+2+1</td>
<td>[min(mod(0,(X,Y),X,Y+2+1))], mod(X,Y+2+1)]</td>
</tr>
<tr>
<td>Otherwise</td>
<td>[min(mod(X,Y),max(mod(X,Y)))]</td>
</tr>
</tbody>
</table>

\[ Y=\bar{Y} \]

\[ |Y| > |X| - |X| \]

\[ \text{mod}(X,Y) \leq \text{mod}(\bar{X},\bar{Y}) \]

\[ \text{mod}(X,Y) \]

\[ [0,|Y|-1] \]

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.
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.

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

---

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

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

double foo_rng16(double x) {
    // x in interval [-16, 16]
    return a > 20 ? a + 100 : a = 100;
}

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

double r = foo(a);

double r = foo_rng16(a);

(a)

(b)

(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.

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

<table>
<thead>
<tr>
<th>Inst</th>
<th>Result</th>
<th>Operand</th>
<th>Inst</th>
<th>Result</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phi</td>
<td>✓</td>
<td>✓</td>
<td>Load</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Alloca</td>
<td>✓</td>
<td>✓</td>
<td>Store</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Binary</td>
<td>✓</td>
<td>✓</td>
<td>Switch</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Unary</td>
<td>✓</td>
<td>✓</td>
<td>Return</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Cast</td>
<td>✓</td>
<td>✓</td>
<td>Function call</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>GetElementPtr</td>
<td>✓</td>
<td>✓</td>
<td>System call</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Benchmark Suite</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>EEMBC</td>
<td>e_{exp, log, j0, y0, acosh, rem_pio2}, sincosf</td>
</tr>
<tr>
<td>CHStone</td>
<td>float64_{add, div, mul}</td>
</tr>
</tbody>
</table>
```
CoSense: Compiler Optimizations using Sensor Technical Specifications

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.

**Hardware 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 (stdev <= 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 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.
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 \( \leq 1\% \).

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
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.

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).

<table>
<thead>
<tr>
<th>Test case</th>
<th>Function calls</th>
<th>LoC of IR</th>
<th>x86 64</th>
<th>aarch64 overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>e_exp</td>
<td>443</td>
<td>1.89%</td>
<td>2.13%</td>
<td></td>
</tr>
<tr>
<td>e_log</td>
<td>444</td>
<td>1.96%</td>
<td>2.09%</td>
<td></td>
</tr>
<tr>
<td>e_acosh</td>
<td>0</td>
<td>1.45%</td>
<td>1.70%</td>
<td></td>
</tr>
<tr>
<td>e_j0</td>
<td>2</td>
<td>2.83%</td>
<td>3.30%</td>
<td></td>
</tr>
<tr>
<td>e_y0</td>
<td>1202</td>
<td>4.21%</td>
<td>5.19%</td>
<td></td>
</tr>
<tr>
<td>e_rem_pio2</td>
<td>2219</td>
<td>13.72%</td>
<td>19.42%</td>
<td></td>
</tr>
<tr>
<td>sincosf</td>
<td>23</td>
<td>4.22%</td>
<td>6.97%</td>
<td></td>
</tr>
<tr>
<td>float64_add</td>
<td>74</td>
<td>18.79%</td>
<td>21.91%</td>
<td></td>
</tr>
<tr>
<td>float64_div</td>
<td>33</td>
<td>9.83%</td>
<td>11.30%</td>
<td></td>
</tr>
<tr>
<td>float64_mul</td>
<td>24</td>
<td>7.09%</td>
<td>7.92%</td>
<td></td>
</tr>
</tbody>
</table>

### 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.

<table>
<thead>
<tr>
<th>Application Module</th>
<th>Speedup</th>
<th>Binary Size Reduction</th>
<th>CompTime Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warp</td>
<td>1.7x</td>
<td>13.8%</td>
<td>26.58%</td>
</tr>
<tr>
<td>Crazyflie IMU Driver</td>
<td>1.5x</td>
<td>0.6%</td>
<td>24.01%</td>
</tr>
</tbody>
</table>

**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 armcmplx_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 8.5x speedup (range [1024,1054]) when
shows that Figure 4.2 and Figure 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, assuming 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.

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, 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 handwritten 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.

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 6.0% 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


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


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


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