Journal of Systems Architecture 168 (2025) 103546

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Bit-level compiler optimization for ultra low-power embedded systems

Seonyeong Heo [a], Jiho Kim [b], Woohyeop Im [a], Jiyun Moon [a], Daehee Jang [a,*]

[a] School of Computing, Kyung Hee University, 1732 Deogyeong-daero, Giheung-gu, Yongin-si, 17104, Gyeonggi-do, Republic of Korea
[b] School of Computer Science, Georgia Institute of Technology, 225 North Avenue NW, Atlanta, 30332, GA, United States

ARTICLE INFO

Keywords:
Ultra low-power system
Compiler optimization
Embedded system
Static analysis
Bit-level analysis

ABSTRACT

Achieving ultra low-power consumption is essential for embedded systems deployed in harsh environments, such as space and deep sea locations, where energy resources are scarce and physical accessibility is limited. Typically, these systems employ ultra low-power microcontrollers that operate on narrow data widths of 8 or 16 bits at the microarchitecture level. If software developers do not carefully consider the data widths during programming, the resulting programs may be suboptimally optimized for these ultra low-power systems. To address this issue and enable more efficient low-power computing, this work proposes a novel optimizing compiler that supports bit-level analyses and transformations. The proposed compiler analyzes how each individual bit of a data item is utilized within a program to determine its optimal width. Consequently, the proposed compiler reduces unnecessary data movements and computational overhead on ultra low-power processors. This work implements the prototype compiler on top of the LLVM compiler framework and evaluates the performance impact of the optimized embedded applications with a processor simulator.

1. Introduction

Ultra low-power embedded systems are critical for applications that operate in remote or inaccessible environments where energy availability is highly constrained [1], such as deep-sea sensor networks, satellite system in space, environment monitoring systems, and wildlife trackers. These systems typically rely on small battery or limited energy harvesting, which makes energy efficiency not just a performance concern but a fundamental requirement for long-term deployment and maintenance cost reduction [2]. To meet such constraints, those systems generally adopt ultra low-power microcontrollers that can operate at the microwatts level [3,4].

Ultra low-power processors often utilize narrow data paths, 8-bit or 16-bit widths [5,6], rather than the more common 32-bit or 64-bit widths used in general-purpose processors. Using a narrower data path helps reduce switching activity, shortens critical paths, and limits memory access overhead, resulting in significant energy savings. However, when programs are written in high-level languages like C, they often rely on standard data types (e.g., `int`) assuming wider data widths. As a result, the compiled binaries may include unnecessary computations, type promotions, and memory accesses.

Manually tuning the data widths can alleviate these inefficiencies, but it is labor-intensive, error-prone, and not scalable for frequently changing applications. Then, automatic compiler optimizations can provide a more scalable and convenient solution by automatically transforming code to better exploit hardware characteristics. Traditional compiler optimizations, however, typically operate at the level of bytes or words and lack the granularity needed to take advantage of narrow data paths. These coarse-grained analyses may overlook optimization opportunities that exist when only a subset of bits within a variable are actually needed or used. In ultra low-power systems, such missed opportunities can accumulate into a nontrivial power budget overhead.

To address the limitation, this work proposes a novel optimizing compiler that analyzes data usage at the level of bits. The proposed compiler performs a static analysis, called bit usage analysis, to identify the usage of each bit of a data value by analyzing bit-level instructions (e.g., `shift` and bitwise `and`) if possible. Based on the bit usage analysis, the compiler determines the minimum required data width and transforms the code accordingly. The transformation can eliminate unnecessary data movements and bit extensions when only a smaller-width computation is sufficient. In the way, the compiler can enhance both performance and resource efficiency of embedded applications, in which bit-level instructions are frequently used.

Fig. 1 illustrates an example of the proposed bit-level compiler optimization assuming 8-bit AVR architecture. In the example, the upper 10 bits of the 16-bit data item `x` are masked by the bitwise `and` operation, and then the upper bits are not used afterward. However, the processor would load all the 16 bits because it is not aware of

* Corresponding author.
E-mail address: daehee87@khu.ac.kr (D. Jang).

https://doi.org/10.1016/j.sysarc.2025.103546
Received 28 April 2025; Received in revised form 7 August 2025; Accepted 11 August 2025
Available online 8 September 2025

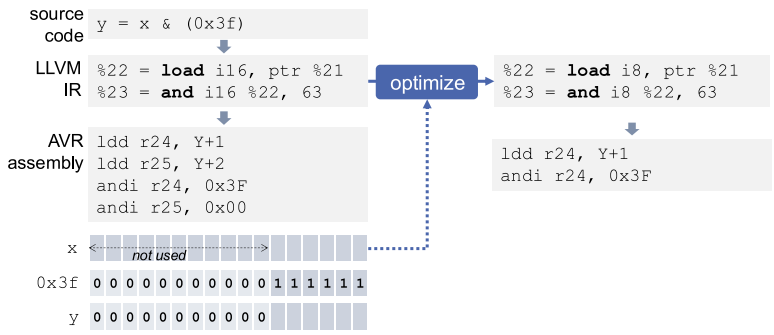

Fig. 1. An example of the proposed bit-level compiler optimization. The example assumes that the target system is based on an AVR architecture, which uses 8-bit word size and the data size of x is 16 bits.

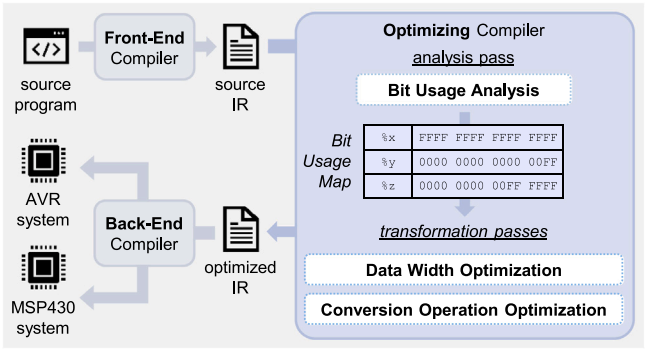

Fig. 2. Overview of the proposed optimizing compiler.

its bit-level usage. It leads to unnecessary data movements and bit-level operations, consuming additional clock cycles. To reduce the undesirable overhead, the compiler statically analyzes the bit usage of data items and adjusts the data width in the IR level. Then, instead of loading the entire bits, it optimizes the source code to load only the lower 8-bits (see Fig. 2).

This work implements the prototype optimizing compiler on top of the LLVM compiler framework for compatibility with existing development toolchains. This work evaluates the proposed compiler using a processor simulator that models 8-bit AVR architecture. With the simulator, this work uses the MiBench benchmark suite [7], which include typical embedded workloads, compiling its benchmarks with and without the proposed optimizations. The evaluation results show that the compiler can statically analyze the bit usage of data items and optimize the data width of data items.

Through the evaluation results, this work demonstrates that bit-level compiler optimizations can improve the alignment between software behavior and hardware capabilities for ultra low-power embedded systems. By bridging the gap between high-level programming abstractions and low-level hardware efficiency, this work contributes a practical and scalable approach to optimizing embedded software at compile time without runtime overhead.

The contributions of this work are:

- This work presents a novel optimizing compiler for ultra low-power embedded system, which provide bit-level static analysis and transformation.
- This work conducts a comprehensive use case analysis of the proposed compiler optimization method with existing embedded benchmark suites.
- This work implements the proposed method on top of the LLVM compiler infrastructure and evaluation of the method with a processor simulation.

2. Background & Motivation

See Fig. 3.

2.1. Ultra low-power embedded systems

Ultra low-power embedded systems are designed to operate with minimal energy consumption, making them ideal for battery-powered and energy-harvesting applications such as wearable devices, remote sensors, and medical implants. These systems typically employ ultra low-power microcontrollers like AVR architecture illustrated in Fig. 3, which integrates essential components including arithmetic logic units, general-purpose registers, timers, and communication interfaces within a compact, power-efficient design. They provide key features like sleep modes, watchdog timers, and minimal clock usage to minimize power consumption during idle periods, ensuring longer operational lifetimes on limited power sources.

These ultra-low power embedded systems often use narrow data widths, such as 8-bit or 16-bit data paths, to reduce both dynamic and static power consumption. For example, in the figure, the data bus of the architecture operates on 8-bit data as it is sufficient for many control-oriented tasks while significantly lowering switching activity and silicon area. Narrow data widths simplify circuit complexity, reduce memory footprint, and allow for more aggressive voltage scaling, which can contribute to extended battery life in resource- and energy-constrained environments.

Although ultra-low power embedded systems may use narrow data paths like 8 or 16 bits, they often need to handle larger data types commonly used in general-purpose programming languages, such as 32-bit or 64-bit integers and floating-point values. In these cases, the system emulates wider data operations by performing multiple sequential operations on narrower data chunks. For example, a 32-bit addition might be implemented as a sequence of four 8-bit additions with carry propagation between them. Therefore, the selection of data widths in a high-level program can affect the overall performance of the program, resulting in a different number of instructions.

2.2. Applications for embedded systems

Embedded applications are software designed to perform dedicated functions within an embedded system, often under strict resource constraints such as limited memory, processing power, and energy consumption. These applications are typically developed for real-time responsiveness, reliability, and efficiency, tightly coupled with the hardware they operate on. As embedded applications are directly programmed into target embedded systems, it is important to optimize them considering low-level hardware features.

One high-level characteristic of embedded applications is that bit-level operations are widely used in the applications due to their computational efficiency and direct hardware relevance. Bit-level operations

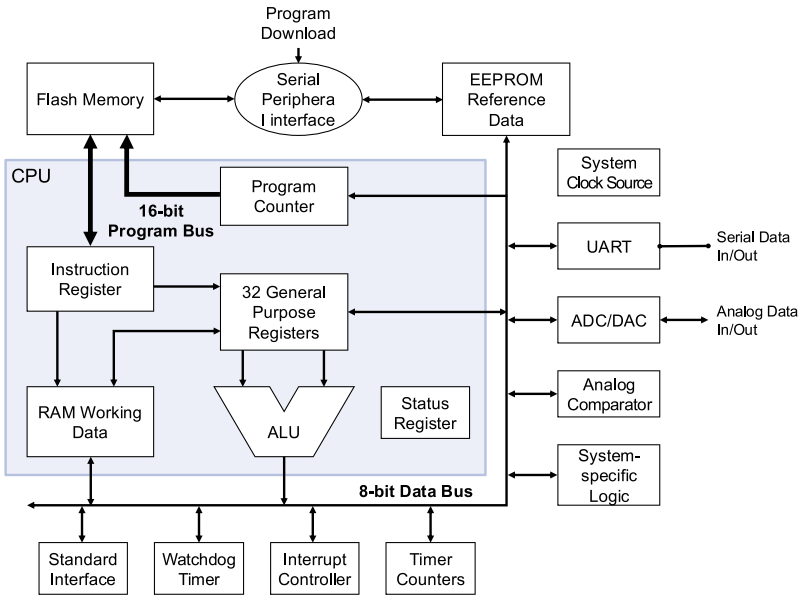

Fig. 3. The example architecture of an ultra lower-power embedded system with narrow data width.

```
#include "proto.h"

void gsm_encode P3((s, source, c), gsm s, gsm_signal *source,
    gsm_byte *c)
{
  word  LARc[8], Nc[4], Mc[4], bc[4], xmaxc[4], xmc[13*4];

  Gsm_Coder(s, source, LARc, Nc, bc, Mc, xmaxc, xmc);

  *c++ =   ((GSM_MAGIC & 0xF) << 4)
         | ((LARc[0] >> 2) & 0xF);
  *c++ =   ((LARc[0] & 0x3) << 6)
         | (LARc[1] & 0x3F);
  *c++ =   ((LARc[2] & 0x1F) << 3)
         | ((LARc[3] >> 2) & 0x7);
  *c++ =   ((LARc[3] & 0x3) << 6)
         | ((LARc[4] & 0xF) << 2)
         | ((LARc[5] >> 2) & 0x3); d
```

Fig. 4. The example code of an embedded application called `gsm` from the MiBench benchmark suite. In the example, the data type `word` is a 16-bit data type.

such as bitwise `and`, bitwise `or`, and `shift` allow manipulating individual bits of data, which is essential for tasks like setting or clearing flags, configuring hardware registers, and handling I/O at the bit level. These operations are computationally inexpensive and fast, making them ideal for optimizing performance in resource-constrained environments.

However, not all bits in every data are always fully utilized because it is common to pack multiple small fields into bytes to save space. Fig. 4 shows a code snippet from an example embedded application called `gsm` that performs speech compression. The `gsm_encode` function encodes speech parameters into a byte stream. Each parameter occupies a specific number of bits, and bitwise operations are used to combine them compactly. For example, `(LARc[0] » 2) & 0xF` extracts only 4 bits of the value to store in a byte. The packing process may leave certain bits unused or reserved for alignment or future use. This implies that it may not be necessary to load all the bits at a certain time.

2.3. LLVM compiler infrastructure

LLVM [8] is a modular, reusable compiler infrastructure designed to support the development of modern programming languages and sophisticated code transformations. It provides a set of libraries and tools that enable source code to be compiled into an intermediate representation (IR), which can then be analyzed, optimized, and translated into machine code for various architectures. Thanks to its language-agnostic design, LLVM is used as the backend for compilers like Clang (for C/C++), and it is widely adopted in academia, industry, and embedded toolchains.

The pass infrastructure of LLVM is a core framework that allows developers to implement modular analysis and transformation algorithms for code optimization, known as *passes*. Fig. 5 briefly illustrate the concept of the pass infrastructure. Passes can perform a variety of optimizations on the LLVM IR, such as dead code elimination, constant propagation, loop unrolling, or custom static analysis. The pass infrastructure enables fine-grained control over program optimization and facilitates the implementation of custom compiler extensions.

3. Compiler design

This work presents a novel compiler for analyzing and optimizing the bit-level data usage of source program. The compiler conducts a static analysis of how each bit of data is used in the program. Based on the analysis results, the compiler transforms the program to use the optimal data width and remove unnecessary conversion operations.

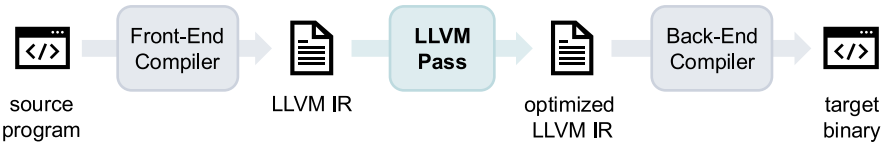

Fig. 5. Concept of LLVM pass infrastructure. The source program is optimized through LLVM passes in the IR level.

3.1. Bit usage analysis

The compiler examines if each bit of a *value* is used or not. For the analysis, the compiler identifies *source* and *sink* instructions in a function.

- A *source* instruction becomes the source of a data value. It may load a value from the memory or allocate a memory space to store a value. For example, in LLVM IR, `load` and `alloca` (which allocates memory on the stack frame) can be source instructions.
- A *sink* instruction becomes the sink of a data value, at which the data value is finally used. It may return no value or may use a value for comparison. For example, in LLVM IR, `store` and `icmp` can be sink instructions.

The compiler tracks the bit usage of each value along the paths from its source instruction to sink instructions. From each source instruction, the compiler follows the def-use (definition and usage) chains until it reaches the sink instructions. Then, starting from the sink instructions, the compiler updates the bit usage of each data value until it reaches back to the source instruction.

Note that the compiler targets only instructions with the integer types (e.g., $i16$, $i32$ in LLVM IR). In general, bit masking operations are applied to integer values, rather than to floating-point values. Moreover, low-power embedded systems often lack floating-point units in hardware and therefore tend to avoid floating-point computations due to computational overhead. As a result, the bit-level usage of floating-point values is rarely analyzable or unnecessary for low-power embedded applications.

The compiler can statically determine the bit usage of a data value for specific bit-level operations.

- **Masking**: A bitwise `and` operation with a constant can mask partial bits of a data value. For example, with `x & 0x7F`, the compiler can infer only the lower `7` bits of `x` is used if the instruction is the only user of `x`.
- **Shift**: A `shift` operation may drop upper or lower bits of a data value. For example, with `x` $\gg$ `16`, the compiler can infer the lower 16 bits of `x` is dropped by the instruction.

Algorithm 1: Bit Usage Analysis

Input: Target instruction I
Output: Bit usage map M

```
if I is already in M or a sink instruction then
    return
end
if I is not an integer type then
    return
end
b ← 0
w ← the data width of I
for every user U of the instruction I do
    if U is a zext, sext, or trunc instruction then
        b_u ← Get the bit usage of U
        b ← b | b_u & {(1 ≪ w) − 1}
    else if U is a bitwise instruction then
        o_1 ← the first operand of U
        o_2 ← the second operand of U
        b_u ← Get the bit usage of U
        if U is an and instruction ∧ o_2 is a constant then
            b ← b | (b_u & o_2)
        else
            b ← b | b_u
        end
    else if U is a constant shift instruction then
        b_u ← Get the bit usage of U
        b_u ← Shift the bit usage b_u according to U
        b ← b | b_u
    else
        b ← b | {(1 ≪ w) − 1)}
    end
end
M[I] ← b
```

Algorithm. Algorithm 1 describes the overall algorithm of the bit usage analysis for a given instruction I. The compiler stores the analysis result in the bit usage map M, of which key becomes an instruction and value becomes a bit mask that indicates if each bit of the result is used or not. Note that the compiler applies the algorithm to each source instruction in the target function. The algorithm begins by checking if the instruction has already been analyzed or the instruction is a sink instruction. If so, it immediately returns because it is unnecessary to perform the analysis. If not, it initializes the bit usage value b and obtains the data width w of I.

For each instruction U that uses the result of I (i.e., the *value* of I), it updates b based on the instruction type of U. If U is a `zext` (zero extension), `sext` (sign extension), or `trunc` (truncation) instruction, which is used to change the data type and the data width accordingly, it recursively computes the bit usage of the user. Then, the bit usage of the user is directly combined with the bitwise `or` operation regarding the current data width using bit masking. As extension or truncation does not change the bit usage of the value but the data width, the compiler keeps the result as it is.

If U is a bitwise instruction, it first checks if the first operand is a bitwise `and` instruction and the second operand is a constant. If so, the algorithm applies a bitwise `and` with the constant to narrow down the bit usage. If a bitwise `and` operation is applied with the constant, it implies a masking operation that may zero out some bits. Then, the compiler marks the bits by applying the same mask to the bit usage of the user b_u. Otherwise, it combines its usage.

If U is a constant shift operation, the bit usage is adjusted (shifted) accordingly before being combined. If the user is a shift left instruction, the bit usage of the user has to be shifted to the right with the same amount. For example, if the user is a shift left instruction with the shift amount of 2 and the bit usage of the user is 11000000_2, the compiler will compute the shifted bit usage as 00110000_2. Note that the compiler applies the logical shift right operation, as the bits truncated by the shift left operation cannot be used afterward.

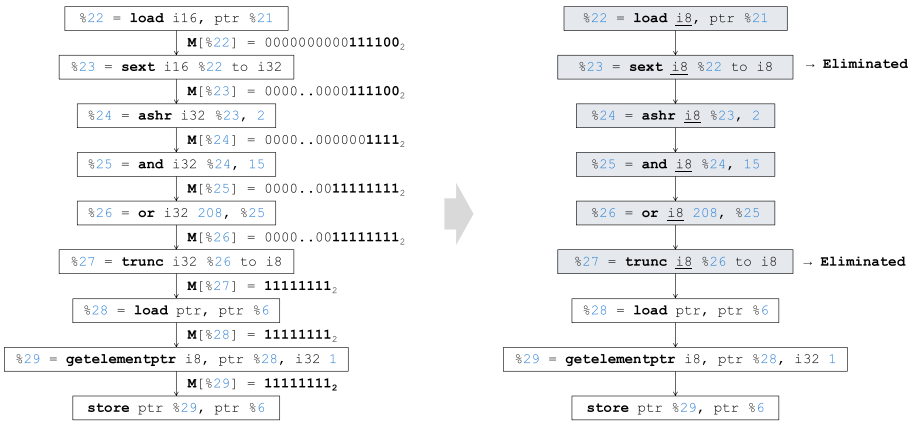

Fig. 6. An example of bit-level analysis and transformation for a sample IR snippet generated from the `gsm` benchmark.

If none of these special cases apply, it assumes all bits are used and sets b accordingly. Then, the compiler stops following the def-use chain for the bit usage analysis because it reaches the sink instruction.

Example. Fig. 6 illustrates an example of how the bit usage analysis algorithm works on a sequence of LLVM IR instructions. The analysis analyzes the bit usage of source instructions from the sink instructions backwards through the instruction dependencies. In the example, the value %27 is a truncated 8-bit value derived from %26. As the value %27 is 8 bits wide, the 32-bit bit usage of %26 is masked with 8 bits only. As %26 is the result of the bitwise `or` instruction, the bit usage of %26 must be preserved for the bit usage of %25. Thus, %25 has the same bit usage with %26.

Next, %25 is the result of the bitwise `and` instruction between %24 and the constant 15 (which is 00001111 in binary). According to the algorithm, the compiler masks the bit usage of %24 to only the lower 4 bits, meaning only bits 0 to 3 of %24 are actually needed. %24 is the result of an arithmetic right shift with %23 by 2. Then, the compilers determines that bits 2 to 5 of %23 are required because the lower 2 bits are dropped by the shift right operation. Next, %23 is the result of a sign extension with %22, which means the relevant bits in %23 come directly from %22. Since bits 2 to 5 of %23 are needed, those same bits must be marked as used in %22, which is a 16-bit loaded value. Thus, the bit usage map for %22 ends up marking bits 2 through 5 as used.

Overall, the example demonstrates how the compiler identifies the bit usage of each value that are necessary for producing the final result. The compiler uses the results of the bit usage analysis (i.e., bit usage map) for transformations, to optimize the source program targeting ultra low-power systems that use narrow data paths.

3.2. Data width optimization

As one of the bit-level optimizations, the compiler adjusts the data width of each value based on the bit usage analysis. If only part of bits are used, it is unnecessary to load or allocate the entire bits. For example, if only the lower 8 bits of a 32-bit value are used, the program may load only one byte of the data instead of loading the entire four bytes. The compiler optimizes the data widths of values when applicable.

The data width optimization can enhance the performance of the source program in various aspects. First, since ultra low-power microcontrollers may use the word size smaller than 32 bits, the optimization will decrease instruction count by eliminating redundant operations for wide data widths. Second, the optimization will help better utilize the memory bus by removing unnecessary memory accesses.

Algorithm. Algorithm 2 briefly describes the data width optimization. Given a target instruction I and a bit usage map M, the compiler first checks if I is included in M. If not, no optimization is performed as it means that the compiler cannot analyze its bit usage statically. If I is present in the bit usage map, the compiler retrieves its bit usage information b from $M[I]$. If I is a source instruction, it generates a narrower version of the instruction that preserves only the required bytes indicated by the most significant bit in b, and replaces I with the new instruction I^*.

If I is a binary instruction, the algorithm first compares the data widths of its two operands. If they differ, it inserts appropriate zero-extension instructions to align operand widths. Otherwise, it replaces the original instruction with a narrower equivalent derived from b. After optimizing I, the algorithm recursively applies the same transformation to all user instructions of I, propagating the data width

Algorithm 2: Data Width Optimization

Input: Target instruction I
Bit usage map M

```
if I is not in M then
    return
end
b ← M[I]
if I is a source instruction then
    I* ← Generate a new instruction with a narrow width
      based on b
    Replace I with I*
else if I is a binary instruction then
    o1 ← the first operand of I
    o2 ← the second operand of I
    if the data widths of o1 and o2 differ then
        Insert a zext instruction for the value with a narrower
          width
    end
    if the data width of I differs from o1 and o2 then
        I* ← Generate a new instruction with a narrow width
          based on b
        Replace I with I*
    end
end
if the data width of I is changed then
    for every user U of the instruction I do
        Apply the data width optimization for U
    end
end
```

Algorithm 3: Conversion Operation Optimization

```
Input: Target function F
       Bit usage map M
for every instruction I in F do
    if I is neither an ext nor a trunc instruction then
        continue
    end
    o ← the operand of the instruction
    if the source and target data types of I are the same then
        Replace all usages of I with o
        Erase I from F
    end
    else if I is an ext instruction ∧ I has one user then
        U ← the user of the instruction
        if U is a trunc instruction then
            if the source type of I is equal to the target type of U
             then
                Replace all usages of U with o
                Erase I and U from F
            end
        end
    end
end
```

optimization throughout the program. This optimization enables compilers to cut back unnecessary bit widths, improving memory efficiency and reducing computation cost.

Example. Fig. 6 shows how the compiler optimizes the data width of each instruction. As only the bits 2 through 5 of %22 are used, the compiler replaces the load instruction with the narrower type of 8-bit integer. Then, the following instructions are updated accordingly. As the data type of %27 is not changed, the compiler stops the transformation at %27. Note that the compiler will move on to the next source instruction afterward.

3.3. Conversion operation optimization

After applying the data width optimization, the compiler tries to remove unnecessary bit-level operations, extension or truncation. As the compiler adjusts the data widths of bit-level instructions and inserts zero extension instructions to match the data widths, there might be unnecessary extensions or truncations in the source program. For example, in Fig. 6, the sign extension and truncation instructions are no longer needed. Therefore, the compiler identifies and eliminates unnecessary instructions to save more clock cycles for the execution.

Algorithm. Algorithm 3 briefly describes the data width optimization. The algorithm iterates over each instruction I in a given function F, checking whether I is a zext instruction. If so, it first ensures that I has only a single user. If it has multiple users, the optimization is skipped to preserve correctness. If there is only one user U, and U is a trunc instruction, the algorithm further checks whether the source type of the zext instruction matches the target type of the trunc instruction. When these conditions are satisfied, the intermediate zext and trunc operations form a redundant type conversion that can be eliminated. The algorithm replaces all uses of U with the original operand o of the zext, then removes both I and U from the function.

4. Use cases

This work explores two additional use cases of the proposed compiler optimization, where bit-level analysis can be beneficial, to demonstrate its practical usefulness and broader applicability.

4.1. Efficient fault-tolerant computing

Reliability is a critical requirement for embedded systems, particularly when the systems operate in harsh environments where transient faults frequently occur [9]. For example, on-board computers in a satellite often encounter soft errors, such as single-event upsets (SEUs), flipping a single bit of data in a storage, due to strong cosmic radiation. Although soft errors do not cause permanent physical damage, their impact on data integrity can disrupt system functionality, potentially leading to incorrect computations or system failures.

Typically, fault-tolerant computing systems rely upon hardware-level solutions due to the considerable runtime overhead of software-level solutions. Fault-tolerant hardware components, such as fault-tolerant processors and memories, are designed to detect and correct soft errors autonomously, ensuring data integrity and system reliability without software modifications.

However, fault-tolerant processors might suffer from unnecessary error detection and correction as they are not aware of the high-level data usage. As error detection and correction are typically done when data is loaded, if soft errors are detected in unused bits, the processor would try to correct the error in the bits, flushing the cache line and executing the instruction again as illustrated in Fig. 7. Then, the processor consumes extra cycles for error correction even though it is not necessary.

The proposed bit-level optimization can alleviate those unnecessary overheads by loading the bits that are actually used. Then, the error detection and correction will be performed on the data that are actually used in the program. Here, note that the processor must support narrow data manipulation (error detection and correction) to obtain actual performance gains. With the hardware support, the optimizing compiler can provide more efficient fault-tolerant computing.

4.2. Bit-banding optimization

Another compelling application of the bit usage analysis is bit-banding optimization, a hardware feature supported by ARM Cortex-M processors (e.g., ARM Cortex-M4). Bit-banding enables direct, atomic accesses to individual bits in memory by mapping each bit to a dedicated alias address. When the bit usage analysis reveals that only a single bit of a memory-mapped variable is read or written (e.g., through bitwise operations with constants), standard memory operations that involve load–modify–store sequences can be replaced with bit-band accesses. This transformation may reduce memory traffic, improve atomicity, and lower instruction count.

For example, if only a single bit of a static variable is used in a function, the compiler may place the variable in the bit-band region (e.g., `.bitband_sram`) and transform the corresponding operations into bit-band memory accesses. Fig. 8 illustrates a more concrete optimization in a high-level representation for clarity. In the example, the `check_flag` function reads the Nth bit of the static variable `flag`. Without the bit banding, it may load the flag value and apply shift and masking operations to check the specific bit. On the other hand, with bit-banding optimization, it can check the bit with a single load instruction from the corresponding bit-band alias address.

5. Evaluation

5.1. Experimental setup

This work implements the proposed optimizing compiler on top of the LLVM compiler infrastructure (Version: 18.1.0rc). With the LLVM pass framework, this work develops three analysis and transformation passes: `BitUsageAnalysis`, `DataWidthOpt`, and `ConvOpOpt`. The optimization passes can be applied using the `opt` tool of LLVM. This work applies the optimization passes to six benchmarks, `bitcount`, `dijkstra`, `sha`, `adpcm`, `gsm`, and `crc32`, from the MiBench

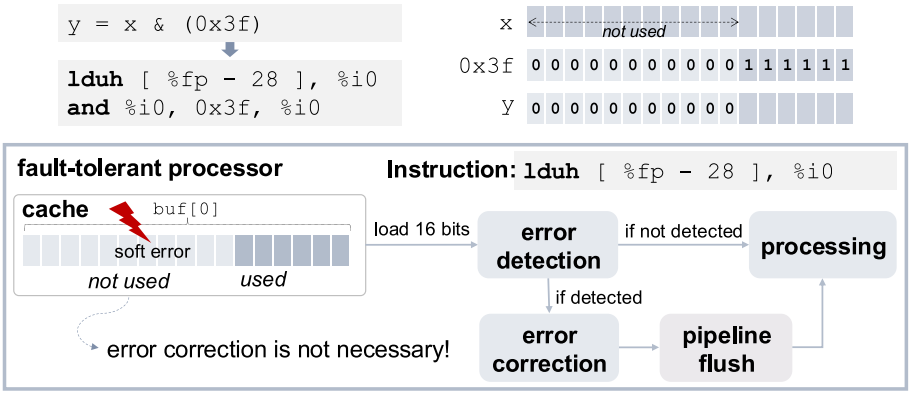

Fig. 7. An example of an unnecessary error correction on a fault-tolerant processor. Even though soft error occurs in the unused bit, the fault-tolerant processor would correct the soft error unnecessarily.

```c
static int flag;

void check_flag() {
  if (flag & (1 << N)) {
    // do something
  }
}
```

(a) Original Code

```c
static int flag __attribute__
((section(".bitband_sram")));

void check() {
  if (*BITBAND(flag, N)) {
    // do something
  }
}
```

(b) Optimized Code

Fig. 8. High-level representation of a bit-banding optimization for an example code. Here, `N` is a constant integer and `BITBAND` indicates a macro that computes the bit-band alias address.

Table 1
The percentages of the IR instructions that have partial bit usage and the total number of AVR instructions and the number of AVR memory access instructions in each executable binary. Note that `adpcm` includes separate binaries for encryption and decryption.

Benchmark		bitcount	dijkstra	sha	adpcm enc/dec	gsm enc	crc32
Partial bit usage		12.1%	0.0%	4.93%	5.76%	17.6%	0.0%
Original	Total	1663	1642	2146	1276/1276	13,851	211
	Memory	258	333	529	216/215	2547	22
Optimized	Total	1660	1642	2146	1435/1435	13,298	211
	Memory	258	333	529	242/241	2520	22
Difference	Total	0.18%	0.0%	0.0%	−12.5%/−12.5%	3.99%	0.0%
	Memory	0.0%	0.0%	0.0%	−12.0%/−12.1%	1.06%	0.0%

suite [7]. This work uses the `-O2` optimization flag to generate the IR code and binary of each benchmark. The other benchmarks are excluded for evaluation as they are unfortunately too large for AVR-based systems, which have a small register file and limited program storage. This work verifies the correctness of the implementation using the sample inputs given by the benchmark suite. With the benchmarks, this work evaluates the compiler optimization both statically and dynamically. For static evaluation, this work measures the number of IR instructions in each executable binary whose bit usage is analyzable and optimizable using the proposed compiler. In addition, this work counts the number of AVR instructions to quantify how much the compiler can reduce the number of instructions; also evaluating the compilation overhead of the proposed optimization to show its feasibility, measured on a desktop with an Intel Core i7-13700F CPU and 32 GB memory.

For dynamic evaluation, this work builds a simulation environment based on QEMU, targeting an AVR-based MCU (ATmega2560). With the emulator, this work accurately measures the number of instructions actually executed at run-time (via dynamic binary instrumentation) and compares the results with the original and optimized benchmarks. In addition, this work measures the execution time of the commercial AVR MCU board, Arduino Mega 2560, which uses the same ATmega2560 chip used in the emulator.

5.2. Results

Instruction Count: This work first reports the number of instructions for each benchmark, counted at compile time. Table 1 presents the percentage of (analyzable) IR instructions with partial bit usage and the total number of AVR instructions in each compiled binary, both before and after applying the proposed optimization. The results show that the `bitcount`, `adpcm`, `sha`, and `gsm` benchmarks contain IR instructions with partial bit usage, with `gsm` reaching up to 17.6%.

While the proposed compiler reduces the total instruction count for benchmarks such as `bitcount` and `gsm`, the instruction count of `adpcm` increases after optimization. It is because the proposed optimizations enable other compiler optimizations such as loop unrolling in the case of `adpcm`. Such optimizations may increase the static instruction count but can improve run-time performance, as reflected in the dynamic instruction count.

Table 2
The total number of instructions and the number of memory access instructions executed at run-time with the AVR processor emulator.

Benchmark		bitcount	adpcm enc	adpcm dec	gsm enc
Original	Total	12,898,386	323,439	877,826	7,011,975
	Memory	195,677	29,266	45,699	595,278
Optimized	Total	12,717,413	306,026	734,143	7,009,047
	Memory	195,677	29,272	45,699	594,382
Difference	Total	−1.42%	−5.69%	−19.6%	−0.04%
	Memory	0%	0.02%	0%	−0.15%

Table 3
The execution time of the original and optimized executable binaries of each benchmark on the commercial AVR board.

Benchmark	bitcount	adpcm enc	adpcm dec	gsm enc
Original	902,288 µs	22,936 µs	61,504 µs	2,269,296 µs
Optimized	890,908 µs	21,968 µs	53,512 µs	2,268,464 µs
Difference	−1.28%	−4.41%	−14.9%	−0.04%

In addition, the sha benchmark shows no difference in static instruction count despite the presence of partial bit usage, as the compiler fails to find sequences of optimizable instructions. For instance, reducing the data width of a single load instruction may not be beneficial if it requires additional extension instructions to match the data widths with its users. These results indicate that the bit usage analysis and transformation are effective, particularly when consecutive instructions exhibit partial bit usage.

Furthermore, although the proposed compiler mainly targets memory instructions, many memory instructions are eliminated during backend compilation. As the proposed compiler optimizes other instructions associated with the memory instructions, it can still achieve performance improvements exploiting partial bit usage.

Table 2 shows the number of instructions executed during the run-time of each application. The benchmarks bitcount, adpcm enc, adpcm dec, and gsm achieve reductions in the number of dynamically executed instructions on the simulated AVR processor. The other benchmarks exhibit no change in static instruction count, and thus are omitted from the table. Interestingly, although the adpcm benchmark shows the largest increase in static memory instruction count, it achieves the greatest run-time improvement, with reductions of 5.69% and 19.6% in dynamic instruction count.

These results confirm that the proposed bit-level optimizations can successfully decrease dynamic execution cost, particularly when the data width of memory accesses can be optimized. In contrast, benchmarks like dijkstra, sha, and crc32, which exhibit limited bit-level optimization opportunities at compile time, show no measurable impact at run-time. This correlation reinforces the effectiveness of the proposed bit usage analysis in identifying and optimizing instructions that can benefit from data width reduction.

Execution Time: This work also measures the actual execution time of each benchmark on a commercial AVR-based board. Table 3 summarizes the execution time of each benchmark in microseconds. The observed trends are consistent with the simulation results: the adpcm dec benchmark shows the greatest reduction in execution time, while the gsm enc benchmark shows the smallest.

Compilation Overhead: This work evaluates the compilation overhead of the proposed optimizations by comparing the compilation times of the original and optimized executable binaries. Fig. 9 shows the total compilation time of each benchmark, with and without optimizations. On average, the proposed optimizations incur only an 8.8% overhead compared to the original compilation time. For the largest benchmark, gsm enc, the entire optimization process takes only 250 ms. These results demonstrate that the proposed bit-level optimizations are practical and introduce small compilation overhead.

6. Related work

6.1. Compilers for low-power embedded systems

Compilers for low-power embedded systems have been studied in various ways, including hardware-aware optimizations and memory management.

Some work focuses on selecting the best compiler flags or optimizations for target embedded systems. Kyriakos et al. [10] show that skipping certain standard optimizations can improve both performance and energy efficiency on ARM Cortex-M4 systems. Sachan and Ghoshal [11] use machine learning to select compiler flags based on power profiles, achieving over 18% energy savings. Ni et al. [12] apply genetic algorithms to optimize compiler flags on Raspberry Pi 4, improving performance by 19% on average. Peker and Ozturk [13] propose a fast method to find optimal compiler settings in under 100 ms.

Koutsoumpas et al. [14] proposes a constrained-based compiler for energy harvesting applications, which considers energy availability during compilation to optimize the energy efficiency of software code running on small computing devices. The proposed compiler identifies computations that may be performed ahead of time and optimizes the precomputation policy to match the intermittent power supply while satisfying system requirements.

Manjunath and Baunach [15] propose a novel static analysis framework that enables performance analysis and verification of manually implemented low-level RTOS code against internal hardware effects. The proposed framework is built on top of an existing WCET static analysis tool called OTAWA to analyze the compiled low-level code and extract intermediate results of the WCET analysis.

For embedded systems, memory optimizations have also been considered important. Zhang et al. [16] propose tunable cache configurations, while Lee and Kim [17] minimizes write buffer activity to reduce power consumption. CLAP [18] improves DRAM prefetching efficiency, and Macho [19] ensures cache reliability near threshold voltage. Rouf and Kim [20] use pipeline redundancy for control-flow protection.

For specific target workloads, such as artificial intelligence and sensor data processing, Petruccelli [21] proposes to replace multiply-and-accumulate operations with shift-add, improving energy use by 31%. Mu et al. [22] use domain-specific languages to describe the bounds of and relations between physical quantities measured by sensors and optimize target application based on the information. MANIC [23] introduces a low-power vector-dataflow architecture. Recent work explores using large language models for code optimization [24].

Empirical studies also show compiler optimizations can affect battery life. Fernandes et al. [25] find that dynamic voltage and frequency scaling (DVFS) combined with compiler tuning can lead to 4% additional energy savings. Compiler autotuning frameworks like Milepost GCC [26] and COBAYN [27] demonstrate practical energy gains.

However, many of these approaches are coarse-grained and overlook bit-level redundancies in data movement. On the other hand, this work addresses this gap by statically analyzing the bit usage to minimize data widths, specially targeting ultra-low power processors operating on narrow data paths.

Similar to this work, some existing work studies bit width-aware compilation based on the motivation that using narrow-width data can reduce switching activity and redundant computation [28]. Zhang et al. [29] apply bit-level optimization to FPGA synthesis, achieving up to 30% logic reduction.

6.2. Bit-level compiler analysis

Some existing work [30,31] conducts bit-level compiler analysis to examine reliability against soft errors or track bit values at static time.

Ko et al. [30] propose Bit-level Error Coalescing (BEC) analysis to improve reliability against soft errors by tracking the semantic effects of

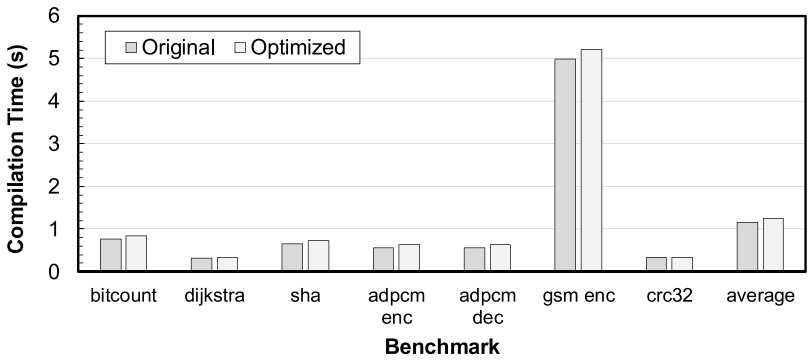

Fig. 9. Total compilation times of the original and optimized binaries for each benchmark.

individual bit corruptions. The proposed analysis enables fault injection pruning and vulnerability-aware instruction scheduling on top of the LLVM compiler framework. While their method also performs bit-level analysis, it focuses on reliability, whereas this work targets code size and memory efficiency in constrained embedded systems. This work applies bit usage analysis to enable bit-width reduction considering narrow data paths.

The LLVM compiler framework [8] includes a built-in analysis called *KnownBits* [31], which tracks known zero and one bits in values, mainly used for constant folding and instruction selection. Since its fundamental goal is to track bit-level *values*, it cannot track bit-level *usage*. For example, when a value is loaded, it can only tell that all bits are unknown, without analyzing the further usage of the bits through the data flow. That is, unknown bits cannot be assumed to be used, as their usage depends on how the value is consumed in subsequent computations. Therefore, the analysis cannot provide precise information about bit-level usage. In contrast, the proposed analysis propagates bit usage across the dataflow graph, identifies dead bits, and enables optimizations like data width reduction, which is not possible with the KnownBits analysis. That is, the target of the proposed analysis is fundamentally different from that of KnownBits, enabling different types of optimizations.

7. Conclusion

Ultra low-power embedded systems generally employ microcontrollers that operate on data widths of 8 or 16 bits at the microarchitecture level. If software developers do not carefully consider the data widths during programming, the resulting programs may be suboptimally optimized for these ultra low-power systems. To address this issue and enable more efficient low-power computing, this work proposes novel bit-level compiler optimizations. These optimizations analyze how each individual data bit is utilized within a program to determine the optimal operation width. Consequently, the proposed compiler reduces unnecessary data movements and computational overhead on ultra low-power processors. This compiler approach has been implemented using the LLVM compiler framework and evaluated through simulations on a processor simulator.

CRediT authorship contribution statement

Seonyeong Heo: Writing – review & editing, Writing – original draft, Validation, Software, Methodology, Investigation, Conceptualization. **Jiho Kim:** Software. **Woohyeop Im:** Validation. **Jiyun Moon:** Writing – original draft. **Daehee Jang:** Writing – review & editing, Supervision, Project administration.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Daehee Jang reports financial support was provided by Korea Ministry of Science and ICT. Daehee Jang reports financial support was provided by Korea Defense Acquisition Program Administration. Daehee Jang reports financial support was provided by Kyung Hee University - Global Campus. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported in part by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. RS-2023-00266615, RS-2025-02214497, RS-2024-00337703), KRIT (Korea Research Institute for defense Technology planning and advancement) grant funded by Defense Acquisition Program Administration (DAPA) (KRIT-CT-24-001); and in part by a grant from Kyung Hee University in 2023 (KHU-20230880).

Data availability

No data was used for the research described in the article.

References

[1] E. Villarino, Design and implementation of low-power microcontrollers for embedded systems in IoT applications, J. Electr. Electron. Syst. 13 (5) (2024) 140, http://dx.doi.org/10.37421/2332-0796.2024.13.140.

[2] R.M. Santos, J. Santos, J.D. Orozco, Power saving and fault-tolerance in real-time critical embedded systems, J. Syst. Archit. 55 (2) (2009) 90–101, http://dx.doi.org/10.1016/j.sysarc.2008.09.001.

[3] R. Immonen, T. Hämäläinen, Tiny machine learning for resource-constrained microcontrollers, J. Sens. 2022 (1) (2022) 7437023, http://dx.doi.org/10.1155/2022/7437023, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1155/2022/7437023. URL https://onlinelibrary.wiley.com/doi/abs/10.1155/2022/7437023.

[4] D. Yadav, B. Raj, B. Raj, Design and simulation of low power microcontroller for internet of things applications, Sens. Lett. 18 (2020) 401–409, http://dx.doi.org/10.1166/sl.2020.4241.

[5] Y. Xu, X. Wang, Y. Chen, et al., An ultra-low-power embedded processor with variable micro-architecture, Micromachines 12 (3) (2021) 314, http://dx.doi.org/10.3390/mi12030314, URL https://www.ncbi.nlm.nih.gov/pmc/articles/PMC8000853/.

[6] M. Aldea Rivas, H. Perez Tijero, Leveraging real-time and multitasking ada capabilities to small microcontrollers, J. Syst. Archit. 94 (2019) 32–41, http://dx.doi.org/10.1016/j.sysarc.2019.02.015.

[7] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, R.B. Brown, Mibench: A free, commercially representative embedded benchmark suite, in: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop, WWC '01, IEEE Computer Society, USA, 2001, pp. 3–14.

[8] C. Lattner, V. Adve, LLVM: A compilation framework for lifelong program analysis & transformation, in: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization, CGO '04, IEEE Computer Society, USA, 2004, p. 75.
[9] J. Gutiérrez-Zaballa, K. Basterretxea, J. Echanobe, Evaluating single event upsets in deep neural networks for semantic segmentation: An embedded system perspective, J. Syst. Archit. 154 (2024) 103242, http://dx.doi.org/10.1016/j.sysarc.2024.103242.
[10] K. Georgiou, C. Blackmore, S. Xavier-de Souza, K. Eder, Less is more: Exploiting the standard compiler optimization levels for better performance and energy consumption, in: Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems, SCOPES '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 35–42, http://dx.doi.org/10.1145/3207719.3207727.
[11] A. Sachan, B. Ghoshal, Learning based compilation of embedded applications targeting minimal energy consumption, J. Syst. Archit. 116 (C) (2021) http://dx.doi.org/10.1016/j.sysarc.2021.102116.
[12] Y. ni, X. Du, Y. Yuan, R. Xiao, G. Chen, Tsoa: a two-stage optimization approach for GCC compilation options to minimize execution time, Autom. Softw. Eng. 31 (2024) http://dx.doi.org/10.1007/s10515-024-00437-w.
[13] M. Peker, O. Ozturk, Fast compiler optimization flag selection, in: Proceedings of the 34th International Workshop on Rapid System Prototyping, RSP '23, Association for Computing Machinery, New York, NY, USA, 2024, pp. 1–5, http://dx.doi.org/10.1145/3625223.3649273.
[14] Y. Li, C. Wang, Constraint based compiler optimization for energy harvesting applications, in: K. Ali, G. Salvaneschi (Eds.), 37th European Conference on Object-Oriented Programming, ECOOP 2023, in: Leibniz International Proceedings in Informatics (LIPIcs), vol. 263, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2023, pp. 16:1–16:29, http://dx.doi.org/10.4230/LIPIcs.ECOOP.2023.16, URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2023.16.
[15] V. Manjunath, M. Baunach, A framework for static analysis and verification of low-level RTOS code, J. Syst. Archit. 154 (2024) 103220, http://dx.doi.org/10.1016/j.sysarc.2024.103220.
[16] C. Zhang, F. Vahid, W. Najjar, A highly configurable cache architecture for embedded systems, in: Proceedings of the 30th Annual International Symposium on Computer Architecture, ISCA '03, Association for Computing Machinery, New York, NY, USA, 2003, pp. 136–146, http://dx.doi.org/10.1145/859618.859635.
[17] J. Lee, S. Kim, An energy-delay efficient 2-level data cache architecture for embedded system, in: Proceedings of the 2009 ACM/IEEE International Symposium on Low Power Electronics and Design, ISLPED '09, Association for Computing Machinery, New York, NY, USA, 2009, pp. 343–346, http://dx.doi.org/10.1145/1594233.1594318.
[18] Y. Lee, S. Kim, CLAP: Clustered look-ahead prefetching for energy-efficient DRAM system, IEEE Trans. Very Large Scale Integr. Syst. 24 (5) (2016) 1770–1782, http://dx.doi.org/10.1109/TVLSI.2015.2488282.
[19] T. Mahmood, S. Hong, S. Kim, Ensuring cache reliability and energy scaling at near-threshold voltage with macho, IEEE Trans. Comput. 64 (6) (2015) 1694–1706, http://dx.doi.org/10.1109/TC.2014.2339813.
[20] M.A. Rouf, S. Kim, Low-cost control flow protection via available redundancies in the microprocessor pipeline, IEEE Trans. Very Large Scale Integr. Syst. 23 (1) (2015) 131–141, http://dx.doi.org/10.1109/TVLSI.2013.2297573.
[21] A. Petruccelli, Strength Reduction Techniques in Compilers for Optimizing Inference on Edge Devices (Ph.D. thesis), uppsala university, 2025, URL https://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-551565.
[22] P. Mu, N. Mavrogeorgis, C. Vasiladiotis, V. Tsoutsouras, O. Kaparounakis, P. Stanley-Marbell, A. Barbalace, Cosense: Compiler optimizations using sensor technical specifications, in: Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction, in: CC 2024, Association for Computing Machinery, New York, NY, USA, 2024, pp. 73–85, http://dx.doi.org/10.1145/3640537.3641576.
[23] G. Gobieski, A. Nagi, N. Serafin, M.M. Isgenc, N. Beckmann, B. Lucia, MANIC: A vector-dataflow architecture for ultra-low-power embedded systems, in: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '52, Association for Computing Machinery, New York, NY, USA, 2019, pp. 670–684, http://dx.doi.org/10.1145/3352460.3358277.
[24] H. Peng, A. Gupte, N.J. Eliopoulos, C.C. Ho, R. Mantri, L. Deng, W. Jiang, Y.-H. Lu, K. Läufer, G.K. Thiruvathukal, J.C. Davis, Large language models for energy-efficient code: Emerging results and future directions, 2024, arXiv:2410.09241. URL https://arxiv.org/abs/2410.09241.
[25] I. Sofianidis, V. Konstantakos, S. Nikolaidis, Reducing energy consumption in embedded systems applications, Technologies 13 (2) (2025) http://dx.doi.org/10.3390/technologies13020082, URL https://www.mdpi.com/2227-7080/13/2/82.
[26] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. O'Boyle, J. Thomson, M. Toussaint, C. Williams, Using machine learning to focus iterative optimization, in: International Symposium on Code Generation and Optimization, CGO'06, 2006, p. 11, http://dx.doi.org/10.1109/CGO.2006.37, pp.–305.
[27] A.H. Ashouri, G. Mariani, G. Palermo, E. Park, J. Cavazos, C. Silvano, COBAYN: Compiler autotuning framework using Bayesian networks, ACM Trans. Arch. Code Optim. 13 (2) (2016) http://dx.doi.org/10.1145/2928270.
[28] G. Suzuki, T. Watanabe, S. Moriguchi, Mruby on resource-constrained low-power coprocessors of embedded devices, in: Proceedings of the 21st ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, in: MPLR 2024, Association for Computing Machinery, New York, NY, USA, 2024, pp. 41–47, http://dx.doi.org/10.1145/3679007.3685064.
[29] J. Zhang, Z. Zhang, S. Zhou, M. Tan, X. Liu, X. Cheng, J. Cong, Bit-level optimization for high-level synthesis and FPGA-based acceleration, in: Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA '10, Association for Computing Machinery, New York, NY, USA, 2010, pp. 59–68, http://dx.doi.org/10.1145/1723112.1723124.
[30] Y. Ko, B. Burgstaller, BEC: Bit-level static analysis for reliability against soft errors, in: 2024 IEEE/ACM International Symposium on Code Generation and Optimization, CGO, IEEE Computer Society, Los Alamitos, CA, USA, 2024, pp. 283–295, http://dx.doi.org/10.1109/CGO57630.2024.10444844.
[31] Known bits analyiss – LLVM documentation, 2025, https://llvm.org/docs/GlobalISel/KnownBits.html. (Accessed 15 April 2025).

Seonyeong

Jiho

Woohyeop

Daehee