A Compiler-Driven Approach for Static Dependency Injection in Embedded Software

Anonymous Author(s)

Abstract

Writing embedded software is challenging due to microcontroller's limitations in processing power, memory, and storage, and to address these limitations the resulting code is frequently tightly coupled to hardware and difficult to maintain. While object-oriented and generic programming offer improved abstractions and cohesion, traditional low-level implementation of polymorphism often introduces runtime overhead that hinders their adoption. This paper proposes a novel compiler-driven dependency injection (DI) technique, implemented as language features in a prototype language, that turns the compiler aware of dependency bindings. By leveraging abstractions for hardware components and peripherals, the compiler resolves dependencies at compile time and replaces bound interfaces with concrete implementations. We compared five implementations of a breakout game written with object-oriented language features, C++20 concepts, and our proposed method. Results show that the proposed language features enable clear interface definitions and centralized binding configurations, enhancing maintainability and portability. Our compiler-driven approach enables optimizations beyond interface boundaries, improving code inlining, constant propagation, interprocedural optimization, and dead code elimination, producing a 41.9% smaller and up to 73.3% faster firmware than equivalent versions with compile-time injection.

Keywords: Compiler-driven dependency injection; Embedded software; Hardware Abstraction; Optimization

1 Introduction

Embedded systems pose unique challenges to software design due to the inherent resource constraints of microcontrollers, which include limited memory, storage, and processing power, as well as application-specific constraints such as security, energy consumption, weight, and cost limitations [3]. Frequently, supporting multiple hardware architectures is also a firmware requirement or becomes a necessity when designing new product versions with upgraded hardware. These constraints require efficient software, typically tightly integrated with hardware and with low-level configuration of microcontroller units (MCU) using preprocessor macros and other low-level features of programming languages such as C and C++ [22].

While these methods of coping with hardware heterogeneity and product requirements are functional and widely used, they often lead to reduced software cohesion, and the reliance on preprocessor macros tightly couples the code to the hardware architecture, obscuring its structure and making it harder to maintain and debug. That is particularly true when targeting multiple platforms, as the platform-specific details increase, resulting in maintenance challenges, higher complexity, and decreased portability [7, 15, 21].

In turn, principles from object-oriented programming (OOP) such as encapsulation, inheritance, and polymorphism improve software cohesion and reduce coupling by promoting a clear separation between hardware-specific code and application logic through standardized interfaces. However, the standard low-level implementation of these principles provokes additional code size and performance overhead due to dynamic dispatch mechanisms, such as virtual function tables (vtables) and supporting structures [2]. To address these inefficiencies, generic programming and metaprogramming are options available in languages such as C++ and Rust, which provide flexible and type-safe abstractions, enabling static polymorphism and not incurring runtime overhead [10]. Nevertheless, their use increases the complexity of code maintenance and sometimes introduces challenges related to code size (code or template bloat) – a significant constraint in embedded system development due to the monomorphization technique employed when compiling generic or metaprogramming code [1].

Dependency Injection (DI) [19], often overlooked in embedded system software, is a technique commonly used in object-oriented programming to separate the concern of how dependencies are provided to a component from the component's core logic. The key advantages of DI are enhanced modularity, testability, and flexibility, which result in easier maintenance in general and can enable portability for hardware or peripheral changes independently of the core application logic in embedded software. However, as DI frameworks are built on top of OOP, they share the same disadvantages discussed for dynamic dispatch. Additionally, the need for runtime resolution of dependencies introduces performance overhead due to the use of Run-Time Type Information (RTTI or reflection). Furthermore, DI containers can complicate dependency tracking, making it more challenging to debug the firmware due to reduced predictability, especially when managing low-level hardware-software interactions.

However, recent advances in DI frameworks sacrifice flexibility for runtime performance by resolving dependencies at compile time, thus eliminating the need for reflection and reducing dynamic dispatch (e.g., the Dagger 2 framework [12] for Java). With compile-time DI, dependencies are resolved statically before program execution, eliminating the mentioned runtime overhead. That makes DI more suitable for resource-constrained environments and can potentially avoid the disadvantages of using preprocessor macros or platform-specific code, leading to more maintainable and portable firmware.

In this paper, we further expand compile-time DI, evaluating a novel compiler-driven dependency injection approach for embedded systems. In addition to generating code that instantiates and injects dependencies at compile-time, we also make the compiler frontend aware of the bindings and able to replace bound interfaces entirely with their concrete implementations. This deeper integration enables the compiler's intermediate code generation to optimize beyond opaque interface boundaries, significantly increasing the number of static calls and improving the efficiency of optimization passes such as inlining [5], constant propagation [23], interprocedural optimization, and dead code elimination, resulting in smaller binaries and faster execution – critical for embedded system development. We argue that this approach also simplifies hardware abstraction, improves early error detection, and enhances maintainability compared to object-oriented programming using vtables and generic programming/metaprogramming using C++20 concepts.

The remainder of this paper is organized as follows. In Section 2, we discuss runtime and compile-time dependency injection as well as their limitations regarding embedded systems constraints. In Section 3, we introduce the syntax of the language features that support our approach through an easy-to-follow example. Section 4 describes the compiler-driven DI and discusses our implementation choices. Section 5 begins the evaluation, comparing five versions of a breakout game concerning its software quality attributes, followed by a performance evaluation in Section 6. Finally, we present our conclusion and propose future work in Section 7.

2 Dependency Injection

Suppose that in the source code of a firmware, we have a class representing an MCU hardware with inner classes abstracting each of its digital ports. Each of these subclasses implements a `digitalport` interface that has methods to set the port direction (input/output) and to get or set its value. A digital port then becomes a component that other classes for hardware peripherals can depend on. For example, a class (or driver) for an SPI display can depend on ports for tasks such as selecting, resetting, and sending data or commands to draw pixels in the display panel. The specific ports used to interconnect these two hardware components vary significantly across different hardware designs or MCUs. The process of mapping, in software, the bindings of the circuit board becomes increasingly challenging as the firmware increases support for products from distinct vendors (3D printer firmware, for example [4]).

Dependency injection (DI) [19] is a design pattern built on top of OOP that implements the principle of inversion of control. Given a set of binding rules, a DI framework implements an external entity responsible for providing the dependencies required by a software component. In the example above, a set of rules in the application configuration determines what digital ports to bind to the SPI display without hard-coding it in the display class. This way, the display class becomes modular and can even be tested with mocked ports when unit testing the software. Thus, by decoupling component configuration from implementation logic, DI promotes modularity, testability, and reuse [9, 19].

While most popular DI frameworks target general-purpose environments with dynamic runtime support, such as Java's Spring [8] and Google's Guice [16], embedded systems demand a more static and predictable approach due to limited resources, the lack or the burden of RTTI, and tight performance constraints. However, advances in DI frameworks trade runtime flexibility for performance by resolving dependencies statically at compile time, eliminating the need for RTTI and reducing dynamic dispatch (e.g., the Dagger 2 framework [12] for Java and Boost-ext.di for C++ [11]), making them more suitable for resource-constrained environments.

In general, dependency injection frameworks implement a model for binding known as the single binding per interface (SBI), where only one concrete interface implementation is bound to a specific interface. That becomes cumbersome when multiple components implement the same interface, as in the above example, where all ports implement the `digitalport` interface. To conform to the SBI model and properly bind dependencies to components in such scenarios, the best practices recommend using role-specific interfaces, type aliases, or named binding, each with its disadvantages and trade-offs. The most naive solution, role-specific interfaces, can cause code duplication, interface bloat, and reduce reusability when designing a standard library for hardware abstraction. In turn, type aliases (e.g., `using spi_reset_port = digitalport` in C++) introduce new names for already existing types with no type distinction or safety enforcement. Finally, named bindings, implemented using language annotations, can lead to string-based errors and ambiguity as the set of names increases and becomes difficult to maintain.

At their core, these mechanisms try to overcome the challenge of implementing dependency injection without relying on RTTI or with the absence of specific language constructs to bind concrete components directly to class fields (or constructor parameters) that already have meaningful names. However, achieving this level of integration using actual language features is challenging and can be made safer, simpler, and more efficient if supported at the language level and integrated into the compiler frontend.

Believing that language-level support for dependency injection can address the challenges related to virtual dispatch (as discussed in the introduction) and the aforementioned limitations of DI in embedded systems, we propose a compile-driven dependency injection feature in a prototype language, Robotics Language [6], and implemented the lowering in its compiler, (robcmp).[1] To reduce the burden of implementing and experimenting with such a design, we leverage the LLVM compiler infrastructure [13], which provides a flexible and reusable set of compiler and toolchain technologies, including a rich intermediate representation (IR).

3 A Blinking LED Example

This section introduces the language features and minimal standard library support for compiler-driven dependency injection through an easy-to-understand LED blinking firmware. The following sections will expand this further and present implementation details. The hardware for the example application is composed of an MCU and an LED connected to one of its digital ports. The purpose of the software is to set up the MCU and to blink the LED at 500 ms intervals.

The code in Listing 1 defines the interface that abstracts some standard routines of an MCU. It has three methods: `wait_ms` for delaying execution, `clock` to retrieve the MCU clock speed, and `set_interruptions` for enabling or disabling interrupts. Typical MCU peripherals should be abstracted as well, as presented in Listing 2. The code defines interfaces for digital and analog ports, including methods for getting or setting their value, as well as modifying the port direction (input or output).

[1]The Robotics Language is developed as part of research projects at Universidade Federal de Jataí and is used in the Compiler course of the Computer Science Bachelor's program.

Listing 1. An example MCU interface

```
interface mcu {
  // delay ms miliseconds
  void wait_ms(uint16 ms);
  // enable or disable MCU interruptions
  void set_interruptions(bool enabled);
  // get the MCU clock speed
  uint32 clock();
  ...
}
```

Listing 2. Interfaces for digital and analog ports

```
enum portmode { input = 0, output = 1}

interface digitalport {
  void mode(portmode m);
  void set(bool v);
  bool get();
}

interface analogport {
  void mode(portmode m);
  void set(uint16 v);
  uint16 get();
}
```

A concrete type (or class) can implement the `mcu` and `digitalport` interfaces, as shown in Listing 3. The `avr5mcu` type implements the `mcu` interface, defining the `clock` and `set_interruptions` methods as well as the interface implementations for each of the MCU ports. We use a specific syntax for inner classes that embeds interface implementation, increasing cohesion. The implementation, as hierarchical fields of the type, encapsulates the configuration of I/O operations while maintaining a structural and semantic bond to the parent MCU type, which will ease the specification of binding rules for compiler-assisted dependency injection. The listing shows the implementation of two ports as examples: `b0` and `b5` of an AVR MCU [14], both controlled by `ddrb` and `portb` registers (another source file, generated from a System View Description file

Listing 3. AVR MCU partial implementation

```
type avr5mcu implements mcu {

  uint32 clock() { return 16E6; }

  void set_interruptions(bool enabled) {
    if enabled { asm "sei"; }
    else { asm "cli"; }
  }

  // subtype implementing the digitalport
     interface for the MCU B0 port
  b0 implements digitalport {
    void mode(portmode m) { ddrb.b0 = m; }
    void set(bool v) { portb.b0 = v; }
    bool get() { return portb.b0; }
  }
  // implementation of the MCU B5 port
  b5 implements digitalport {
    void mode(portmode m) { ddrb.b5 = m; }
    void set(bool v) { portb.b5 = v; }
    bool get() { return portb.b5; }
  }
  //...
}
```

(SVD or ATDF) provided by the MCU manufacturer, defines the structure and addresses of these registers). Due to space constraints, we omit the other digital MCU ports, though we implement them similarly.

The code in Listing 4 presents the traditional LED blink program for microcontrollers, written in a hardware-agnostic way using the previously defined interfaces. This separation of application logic from hardware details promotes portability and maintainability across different embedded platforms. The variables `mmcu` and `led` (lines 2 and 4) are declared using the interface types (`mcu` and `digitalport`, respectively). The language's type inference will determine their types from the expressions on the right-hand side. What appears to be an interface constructor call is a binding point that will receive a concrete implementation through injection at compile time. The `main` function initializes the LED port mode to output mode (line 7) and enters a loop that toggles the LED state every 500 milliseconds by calling the `wait_ms` method of the MCU interface.

Concrete implementations of `mcu` and `digitalport` are bound to the global variables `mmcu` and `led`, respectively, by a hardware specification source file. Listing 5 provides a binding example that makes use of the previously defined `avr5mcu` type and binds a single global instance to it. When targeting this particular MCU, the build process includes this file. To separate hardware specifics from the application logic, each MCU architecture supported by a product can have its binding file. As changes to the hardware configuration do not imply changes to the application source code, this modular approach encourages code reuse and improves maintainability.

Listing 4. A hardware-agnostic LED blink example.

```
 1 // an mcu implementation will be bond here
 2 mmcu = mcu();
 3 // the firmware needs a digital port
 4 led = digitalport();
 5
 6 int16 main() {
 7   led.mode(portmode.output);
 8   loop {
 9     led.set(true);      // turn on the LED
10     mmcu.wait_ms(500);
11     led.set(false);     // turn off the LED
12     mmcu.wait_ms(500);
13   }
14 }
```

Listing 5. DI code for binding an AVR MCU in the LED blink example.

```
1 bind avr5mcu to mmcu {
2   bind b5 to led;
3 }
```

4 Compiler-Driven Dependency Injection

In our prototype language, there are two binding points for concrete types. First, an interface can be the type of variables in the global scope (such as the `mmcu` and `led` variables shown in Listing 4). Although an interface is abstract and not instantiable, the expression on the right-hand side establishes the semantic type needed in the symbols table, enabling the compiler to enforce a binding rule that specifies a concrete type implementing the interface. The compiler then generates code for instantiating and binding the type into that variable at the program start (Listing 5).

The second binding point is at the fields of a type, as shown in Listing 6. The example `game` type uses a `display` interface for the `gdisplay` field (line 2). In this case, the `gdisplay` field will be implicitly initialized with the appropriate concrete type instance, `ssd1306`, whenever a `game` instance is created, as defined by the `bind` rule (line 5).

The `bind` rule's syntax can accommodate multiple bindings (line 6 in Listing 6 binds an instance of `other` to both `x` and `w`) and binding of inner classes of the instance (line 7 binds the `f1` field of `other` to `y` and `z`). Also, any missing rule for a defined binding point is reported early by the compiler. Furthermore, when binding a concrete type to these points, the compiler visits the Abstract Syntax Tree (AST) and replaces the interface type with the concrete type.

To support interface-based polymorphism, we use a switch-based dynamic dispatch [2]. Instead of generating vtables, the compiler adds a unique ID field for each type that implements an interface and uses it to dispatch a method call to the correct method implementation. Specifically, the compiler generates a function for each interface method in the format defined in Listing 7. The ID of the instance (`this.id`) is read (line 2), and a case statement is chosen based on the ID of each concrete type implementation (x, y, or z in the listing, lines 3–5). A default case (line 6) halts the MCU when the ID is not a concrete implementation of

Listing 6. Binding points in the prototype language.

```
 1 type game {
 2     gdisplay = display();
 3 }
 4 // sample code that will bind the concrete ssd1306
       display to game.gdisplay (DI code)
 5 bind ssd1306 to game.gdisplay;
 6
 7 // extended syntax of the bind statement
 8 bind other to x, w {
 9     f1 to y, z;
10 }
```

Listing 7. Interface switch-based template for dynamic dispatch using the id field.

```
return_type interface_name.method_name(this) {
    switch (this.id) {
        case x: return x_type.method_name(this);
        case y: return y_type.method_name(this);
        case z: return z_type.method_name(this);
        default: halt();
    }
}
```

the interface, preventing memory corruption or control flow hijacks.

A disadvantage of the switch-based dynamic dispatch method is that it requires a monolithic build, forcing that all classes are known at compile-time[2] to generate the switch. This requirement prevents adding new types or updating implementations after compilation (dynamic linking), a feature needed in systems designed to dynamically load new behavior (e.g., plugins or drivers). Despite these scenarios, for MCUs and embedded software, the firmware is often rebuilt and deployed as a complete image, making a monolithic build acceptable.

Although the method seems inefficient at first, Bauer and Rossow [2] showed it reduces binary size, can improve runtime performance, and is a complete protection to mitigate vtable hijacking attacks (such as the COOP attack [17]).

Additionally, to reduce code size and memory usage, as well as to increase runtime performance, we add passes in the semantic analysis that remove the ID field of types that are the only concrete implementation of a specific interface and mark the dispatch function as inline (i.e., devirtualization). Also, observing that dense packs of numbers in switch statements can be lowered by more efficient constructions than sparse sets (jumptables, in constant-time *vs* compare and branch, in linear-time) [18, Chap. 6], we assign consecutive numbers to the ID field of concrete types of an interface. This way, they appear as dense packs in the switch statement of the dispatch function.

5 Case Study: A breakout game

To evaluate the proposed compiler-driven dependency injection and present a minimal accompanying standard library, we developed a breakout game as a case study.

Breakout is a classic arcade game in which the player controls a paddle to bounce a ball upward toward a wall of bricks. The objective is to break all the bricks by hitting them with the ball, which rebounds off the paddle, walls, and bricks. If the ball falls past the paddle at the bottom of the screen, the player loses the game. Figure 1 shows two screenshots of the game: one at the beginning of a level and another after some bricks have been broken. In our implementation, the game starts with five full rows of bricks. After breaking all the bricks of a level, the game generates another level with a random pattern of missing bricks and reduces the size of the paddle. The number of missing bricks increases progressively, and the paddle size reduces for a total of 30 levels.

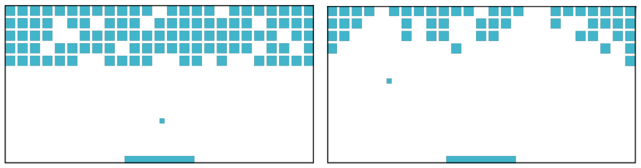

Figure 1. Breakout game screenshots: the start of a level (left) and after some bricks have been broken (right). The paddle is at the bottom center and the smaller square represents the ball.

The game's source code was designed to be abstract and decoupled from the hardware (underlying MCU, its peripherals, and the display), enabling portability with minimal modifications. The architecture defines interfaces for core hardware components: the MCU (Listing 1), display, databus, and digital ports (Listing 2). The display interface is built over a generic canvas abstraction, which renders to an off-screen buffer. Concrete implementations were developed for the ATmega328P MCU [14] (Listing 3) and for an SSD1306 SPI-based display [20]. There are concrete implementations of the databus interface for both SPI and UART serial communication protocols. At the core of the application, a game class encapsulates the logic and interprets input signals (via UART) to control the paddle's movement. The architecture of interfaces and types allows easily modifying the SPI display with a corresponding one using I2C protocol, rewiring digital ports, or even replacing the MCU: the same logic runs across different hardware configurations with no changes in the core logic.

We implemented five versions of the game: *rob*, using the prototype language with compiler-driven DI; *vtable*, using C++ OOP, with interfaces implemented as structs with virtual methods, generic programming in strategically chosen places (templates for repeated code), and hard-coded dependencies; *vtabledi*, same as *vtable* but using a compile-time DI framework [11] instead of hard-coded dependencies; *concept*, using C++20 concepts instead of virtual dispatch and hard-coded dependencies; and *conceptdi*, same as *concept* but using compile-time DI. In the following, we

[2]This requirement can be postponed to link time by enumerating types in the linker, as done by Bauer and Rossow [2].

discuss some challenges related to software quality attributes, such as cohesion and maintainability for each version. We choose the Boost-ext.di framework [11] for implementing compile-time DI in the C++ versions. The framework itself is implemented in a unique header file, without dependencies, can be built without exceptions, and supports the binding of C++ concepts. The complete source code for each game version is available at https://github.com/thborges/sblp2025.

Listing 8 shows the `digitalport` interface implementation for each C++ breakout game version. For reference, the implementation for the *rob* version is in Listing 2. C++ does not have interfaces, but they can be implemented as structs (or classes) with virtual methods (lines 2–6). The `=0` indicates a pure virtual function without implementation, making the struct abstract (uninstantiable). Derived classes override these functions and enforce the compiler to use vtables. In contrast, lines 9–14 present the C++20 concept implementation. Distinctly, the concept is only a compile-time constraint that specifies the requirements a type must satisfy to serve as a template parameter. The dependent concrete type `avr5spi`, in lines 17–20, is a template class that requires its dependency (`reset_port`) to satisfy the `digitalport` concept. Any overhead in size or performance comes from the template usage in the concrete type and not from the concept itself. Note that the `avr5spi` definition does not declare that it implements a databus concept; it is necessary as a template parameter in the dependent class. Despite the verbosity of the C++ language, which impacts readability and maintainability, both in the virtual struct with the dispensable `virtual` and `=0`, and more pronounced in the concept version,[3] the constructs are equivalent in functionality. In the prototype language, the use of the `interface` keyword (Listing 2) indicates the nature of the construct, dispensing additional symbols.

Listing 9 shows the setup of the binding rules for *vtabledi* and *conceptdi*. The injector for *vtabledi* version, lines 2–11, presents the aforementioned use of named bindings (`nm_uart`, `nm_display`, `dp_ss`, and `avr5_ss`). These names have to be shared between all classes that use them (the main app and the ssd1306 display classes), which reveals as a potential source of name collisions when using libraries provided by distinct vendors or ambiguity if distinctiveness of names is enforced, as exemplified in lines 8 and 9, two names for the same concrete class `avr5mcu_b2`. These aspects reduce modularity and extensibility.

The concept-based injector for the *conceptdi* version, lines 14–26 of Listing 9, presents a template-heavy configuration that nests templates for the display type (line 15) and the SPI databus (line 14), with additional parameters for three digital ports and a framebuffer. Although the `using` keyword (C++11) allows a split definition, maintaining such code is

Listing 8. C++ interface implementation using abstract struct and concepts.

```
1  // virtual digitalport for vtable versions
2  struct digitalport {
3      virtual void mode(port_mode m) = 0;
4      virtual void set(bool v) = 0;
5      virtual bool get() = 0;
6  };
7
8  // concept based digitalport
9  template<typename T>
10 concept digitalport = requires(T obj) {
11     { obj.mode(port_mode{}) } -> same_as<void>;
12     { obj.set(bool{}) } -> same_as<void>;
13     { obj.get() } -> same_as<bool>;
14 };
15
16 // use of the concept in a concrete type
17 template<digitalport dp>
18 class avr5spi {
19     dp& reset_port;
20     ...
21 };
```

Listing 9. Injectors for each DI-enabled version of breakout game.

```
1  // injector for the vtabledi version
2  auto breakout_injector = make_injector(
3    bind<mcu>.to<avr5mcu>(),
4    bind<buffer8>.to<ssd1306_framebuffer>(),
5    bind<display>.to<ssd1306>(),
6    bind<databus>.named(nm_uart).to<avr5_uart0>(),
7    bind<databus>.named(nm_display).to<avr5_spi>(),
8    bind<digitalport>.named(dp_ss).to<avr5mcu_b2>(),
9    bind<digitalport>.named(avr5_ss).to<avr5mcu_b2
       >()
10   ... // 5 more bindings for b0, b1, b3 -- b5
11 };
12
13 // injector for the conceptdi version
14 using spi_t = avr5_spi<avr5mcu_b3, avr5mcu_b4,
       avr5mcu_b5, avr5mcu_b2>;
15 using display_t = ssd1306<spi_t, avr5mcu_b1,
       avr5mcu_b0, avr5mcu_b2, avr5mcu,
       ssd1306_framebuffer>;
16
17 auto breakout_injector = make_injector(
18   bind<c_mcu>.to<avr5mcu>(),
19   bind<c_databus_uart0>.to<avr5_uart0>(),
20   bind<c_buffer8>.to<ssd1306_framebuffer>(),
21   bind<c_databus_display>.to<spi_t>(),
22   bind<c_display>.to<display_t>(),
23   bind<c_digitalport_b2>.to<avr5mcu_b2>(),
24   bind<c_avr5_ss>.to<avr5mcu_b2>()
25   ... // 5 more bindings for b0, b1, b3 -- b5
26 };
27
28 // injector for the rob version
29 bind avr5mcu to mcu {
30   bind b0 to ssd1306.reset;
31   bind b1 to ssd1306.datacmd;
32   bind b2 to ssd1306.select;
33   bind uart0 to dbus_uart;
34   bind spi to dbus_display, ssd1306.dbus;
35 };
```

[3] The concept is a powerful construct of C++20 that can be used in many other compile-time checks other than the one shown here.

challenging and heavily depends on understanding the subjacent templates (e.g., the order of template parameters). Another problem is the ambiguity caused by the repeated use of the `avr5mcu_b2` port both in the type definition (lines 14 and 15) and the bindings (lines 23 and 24). Furthermore, the compiler error messages often will not aid in diagnosing issues effectively. Thus, we consider it low in maintainability and readability.

Finally, lines 29–35 in Listing 9 show the injector for the *rob* version. The use of inner classes allows a concise and centralized expression of binding rules, mapping the relationships between high-level components (e.g., the display to data buses) and low-level MCU resources (e.g., digital ports, UART0, and SPI), which improves readability and enhances modularity. Moreover, inner classes prevent unnecessary exposure of hardware-specific details: for instance, ports `b3`, `b4`, and `b5`, which are fixed for the SPI peripheral in the AVR5 platform, are kept internal and hidden from the top-level configuration. We believe that this approach reduces cognitive overhead and the risk of misconfiguration, improving the maintainability and portability of the overall design.

6 Performance and Size Evaluation

We evaluated the impact of each game version on the firmware size, instruction count, and runtime performance. The C++ source code was built with `clang++` version 19.1.2 using `-Oz`, `-fno-exceptions`, `-ffunction-sections`, `-fno-rtti`, `-fdata-sections`, and `-lto=thin` as build options. The compiler of the prototype language used the same LLVM 19.1.2 backend and equivalent build options.

The resulting firmware sizes are shown in Table 1. The firmware built for *rob* is 4234 bytes (41.9%) smaller than *vtabledi*, and 3492 bytes (37.3%) smaller than *conceptdi*. These are the three versions with DI support. The use of the DI framework added 3222 bytes in the *concept* version and 1406 bytes in *vtable*, a significant amount of memory for an MCU. *Rob* size is also smaller compared to *vtable* (32.5%) and to *concept* (4.4%), which shows the effectivity of the concepts feature regarding not increasing the binary size. The main cause for the larger size of *vtable* is the use of vtables itself, which increases call overhead, producing significant stack manipulation and reducing the efficiency of some compiler optimization passes (inlining, constant propagation, DCE, IPO). Both *concept* and *rob* firmware lack vtable structures, and their reduced size difference stems from the *concept's* reliance on generic programming and minor variations in compiler inlining and optimizations.

To measure performance, we disabled the game-over feature so that when the ball hits the bottom edge, the game continues (as if the paddle was there). We also let the game run at full speed, without time constraints imposed at normal execution. This way, we can run all game levels and measure the time needed to complete the game (t), how many times the frame was updated (u), and, finally, the number of frames per second fps=u/t. We captured the start and the end of the game execution through the `uart0` port.

The results are shown in Table 1, under the fps column. The *rob* implementation was 73.3% faster than *vtables*; *vtabledi* runs 4.4% more frames per second than *vtable*, followed by *conceptdi* (41%) and *concept* (41.8%). The reason for the difference in performance is the reduction in the total number of instructions (simplification of critical paths) and the use of simpler, faster instructions. The rob version reduced costly operations such as indirect calls (`icall`, 3 clock cycles) and performed significantly fewer memory and stack operations, such as `movw`, `ldd`, `ld`, `push`, and `pop`. The vtable version, by contrast, introduces extra overhead through indirect calls and increased memory manipulation, while the concept version, though avoiding `icall`, still exhibits more stack manipulation than rob. In summary, compile-time dependency injection exposed better inlining and optimization opportunities that remain hidden in the C++ code.

We investigated the reason behind this result by disassembling the firmware ELF binaries with `avr-objdump -D`. By using a custom Python script, we counted how many times each instruction mnemonic appeared in the disassembled code. The results are shown in Table 2. *Rob* firmware has 1298 fewer instructions than *vtable*, representing a 32% decrease in instruction count. Compared to concept, rob also presented 116 fewer instructions (4%). The most reduced instructions compared to vtable are memory-related operations (`movw`: −421, `ldd`: −286, `ld`: −129) and call stack manipulations (`push`: −87, `pop`: −87), reflecting the elimination of vtable lookups and reduced call overhead. These instructions are also the most

Table 1. Size (in bytes) and performance measured as fps of each breakout game implementation.

version	**size**	**Δ%**	**fps**	**Δ%**
vtabledi	10108	–	2925	+4.4
vtable	8702	-13.9	2801	–
conceptdi	9366	-7.3	3948	+41.0
concept	6144	-39.2	3972	+41.8
rob	5874	-41.9	4855	+73.3

Table 2. Instruction count comparison for the breakout game firmware. The hidden rows have $-20 \leq \Delta \leq 9$.

		C++			
Instr.	**rob**	**vtables**	**Δ**	**concepts**	**Δ**
movw	173	594	-421	309	-136
ldd	73	359	-286	114	-41
ld	8	137	-129	35	-27
pop	87	174	-87	132	-45
push	87	174	-87	132	-45
icall	0	63	-63	0	0
lds	13	68	-55	46	-33
ret	65	118	-53	72	-7
mov	141	190	-49	161	-20
add	86	126	-40	109	-23
sbci	71	107	-36	87	-16
adc	109	140	-31	123	-14
subi	60	89	-29	69	-9
eor	102	126	-24	115	-13
or	11	33	-22	23	-12
sbc	25	46	-21	45	-20
		...			
in	66	57	9	28	38
out	77	66	11	39	38
dec	29	14	15	14	15
ldi	213	195	18	146	67
cpi	76	55	21	48	28
sts	25	2	23	2	23
adiw	50	22	28	18	32
std	236	129	107	102	134
	2762	4060	-1298	2878	-116

reduced ones when compared with concept. Notably, `icall` instructions were eliminated (*vtable*: 63, *concept* and *rob*: 0), demonstrating the effectiveness of replacing runtime indirection with concepts and compiler-driven dependency injection. Although some instructions, such as `std` (+134) and `ldi` (+67), increased in rob, they reflect specific instance bindings at initialization adopted by the language and were compensated by consequent optimizations.

7 Conclusion and Future Work

In this paper, we addressed the challenges of developing maintainable and highly optimized software for resource-constrained embedded systems. Traditional object-oriented approaches often introduce unwanted runtime overheads due to their low-level implementation of polymorphism (vtables with dynamic dispatch). While alternative language features, such as generic programming (templates) and C++20 concepts, do eliminate dynamic dispatch in favor of static dispatch, they often introduce challenges regarding readability and maintainability due to their verbosity and complex metaprogramming constructs.

To overcome these limitations, we proposed a novel compiler-driven dependency injection approach deeply integrated into a prototype language. The design shifts the responsibility of dependency resolution from runtime or library-based metaprogramming at compile-time to the language frontend and compiler intermediate representation. The strategic binding points and a concise bind syntax allowed the compiler to perform static dependency injection by replacing interface types with concrete implementations in the Abstract Syntax Tree (AST).

Our comprehensive case study, a breakout game implemented in five distinct versions (including C++ OOP, C++20 Concepts, and boost-ext.di frameworks), provides empirical evidence of the language syntax adherence to the targeted application domain. The *rob* implementation consistently shows smaller firmware sizes (e.g., 41.9% smaller than *vtabledi*) and higher runtime performance (e.g., 73.3% faster than *vtable*). A disassembly analysis revealed the underlying reasons: compiler-driven DI eliminated costly runtime indirections, such as vtable lookups and icall instructions, and exposed opportunities for optimization passes, reducing memory and stack operations.

Beyond the quantitative gains, language features such as the bind statement and inner classes improve maintainability, enhance early error detection for missing or incorrect dependencies, and reduce cognitive overhead by providing a coherent, language-native mechanism for managing component relationships.

Looking forward, our work opens some avenues for future research. We plan to explore the expansion of the language features to support more complex dependency patterns and how to incorporate other binding scopes, such as transient, feature-specific, or thread-local dependencies. Furthermore, we aim to evaluate our compiler-driven paradigm on a broader range of embedded software to further validate its benefits across diverse hardware platforms.

References

[1] Hudson Ayers, Evan Laufer, Paul Mure, Jaehyeon Park, Eduardo Rodelo, Thea Rossman, Andrey Pronin, Philip Levis, and Johnathan Van Why. 2022. Tighten rust's belt: shrinking embedded Rust binaries. In *Proceedings of the 23rd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems* (San Diego, CA, USA) *(LCTES 2022)*. Association for Computing Machinery, New York, NY, USA, 121–132. doi:10.1145/3519941.3535075

[2] Markus Bauer and Christian Rossow. 2021. NoVT: Eliminating C++ Virtual Calls to Mitigate Vtable Hijacking.

In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. 650–666. doi:10.1109/EuroSP51992.2021.00049

[3] Giorgio Buttazzo. 2006. Research trends in real-time computing for embedded systems. *SIGBED Rev.* 3, 3 (July 2006), 1–10. doi:10.1145/1164050.1164052

[4] Marlin Community. 2025. Marlin Firmware. https://marlinfw.org/docs/basics/introduction.html. Accessed: 2025-04-27.

[5] Thaís Damásio, Vinícius Pacheco, Fabrício Goes, Fernando Pereira, and Rodrigo Rocha. 2021. Inlining for Code Size Reduction. In *Proceedings of the 25th Brazilian Symposium on Programming Languages* (Joinville, Brazil) *(SBLP '21)*. Association for Computing Machinery, New York, NY, USA, 17–24. doi:10.1145/3475061.3475081

[6] Thiago Borges de Oliveira. 2023. Robotis Language: A prototype language for embedded systems with compile-time dependency injection. https://github.com/thborges/robcmp. Accessed: 2025-06-08.

[7] Michael D. Ernst, Greg J. Badros, and David Notkin. 2002. An Empirical Analysis of C Preprocessor Use. *IEEE Trans. Softw. Eng.* 28, 12 (Dec. 2002), 1146–1170. doi:10.1109/TSE.2002.1158288

[8] Martin Fowler. 2004. *Inversion of Control Containers and the Dependency Injection pattern.* martinfowler.com. Accessed: 2025-06-08..

[9] Vahid Garousi, Michael Felderer, and Feyza Nur Kılıçaslan. 2019. A survey on software testability. *Information and Software Technology* 108 (2019), 35–64. doi:10.1016/j.infsof.2018.12.003

[10] Marcell Ferenc Juhász. 2023. *Modern C++ in embedded systems-Utilization of modern C++ language features for creating zero-overhead firmware architectures for resource-constrained embedded systems.* Ph. D. Dissertation. Institute of Computer Technology, Technische Universität Wien.

[11] Kris Jusiak. 2018. boost-ext.di: C++14 Dependency Injection Library. https://github.com/boost-ext/di. Accessed: 2025-05-20.

[12] Gregory Kick. 2025. Dagger Framework. https://dagger.dev. Accessed: 2025-04-27.

[13] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO)*. 75–86. doi:10.1109/CGO.2004.1281665

[14] Microchip Technology Inc. 2015. ATmega328P Datasheet: 8-bit AVR Microcontroller with 32K Bytes In-System Programmable Flash. https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf. Accessed: 2025-06-08.

[15] Brent Pappas and Paul Gazzillo. 2024. Semantic Analysis of Macro Usage for Portability. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3597503.3623323

[16] Robbie Vanbrabant Schaefer. 2008. *Google Guice: Agile Lightweight Dependency Injection Framework.* Apress.

[17] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *2015 IEEE Symposium on Security and Privacy*. 745–762. doi:10.1109/SP.2015.51

[18] Michael L. Scott. 2016. *Programming Language Pragmatics* (4 ed.). Morgan Kaufmann, Boston, MA.

[19] Mark Seemann and Steven van Deursen. 2019. *Dependency Injection Principles, Practices, and Patterns.* Manning Publications.

[20] Solomon Systech Limited. 2008. SSD1306: 128 x 64 Dot Matrix OLED/PLED Segment/Common Driver with Controller. https://cdn-shop.adafruit.com/datasheets/SSD1306.pdf. Accessed: 2025-04-02.

[21] Bjarne Stroustrup. 1994. *The Design and Evolution of C++.* Addison-Wesley.

[22] Steven Varoumas, Basile Pesin, Benoît Vaugon, and Emmanuel Chailloux. 2023. Programming microcontrollers through high-level abstractions: The OMicroB project. *Journal of Computer Languages* 77 (2023), 101228. doi:10.1016/j.cola.2023.101228

[23] Mark N. Wegman and Frank Kenneth Zadeck. 1985. Constant propagation with conditional branches. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (New Orleans, Louisiana, USA) *(POPL '85)*. Association for Computing Machinery, New York, NY, USA, 291–299. doi:10.1145/318593.318659