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

Thiago Borges de Oliveira – thborges@ufj.edu.br
Ariadne de Andrade Costa – ariadne.costa@ufj.edu.br

Instituto de Ciências Exatas e Tecnológicas (ICET)
Universidade Federal de Jataí (UFJ)

SBLP 2025, Recife, Brazil

Outline

Challenges in Embedded Software Development

- Loosely coupled and maintainable code is desired
- Resource-constrained MCUs: memory, storage, performance
- Application-specific constraints: security, energy, weight, and cost limitations
- Portability across hardware platforms often needed
- Efficient software $\rightarrow$ tightly coupled with hardware using low-level language features

Object-oriented programming (OOP)

- Cohesion, low coupling and maintainability: separated hardware-specific and application code using encapsulation, inheritance, and polymorphism
  - vtables: Additional code size and performance overhead due to dynamic dispatch mechanisms
- Static polymorphism: generic programming and metaprogramming, not incurring runtime overhead
  - increases the complexity of code maintenance and sometimes introduces challenges related to code size (code or template bloat)

Dependency Injection (DI)

- Often overlooked in embedded system software
- Separate the concern of how dependencies are provided to a component from the component's core logic
- Enhanced modularity, testability, and flexibility
- Interfaces: DI frameworks are built on top of OOP
- Share the disadvantages of dynamic dispatch

Runtime vs Compile-Time

Runtime DI

- Flexible: dynamic implementations
- Dependencies injected at execution
- Runtime overhead due to RTTI
- Challenges to debug due to reduced predictability

Compile-Time DI

- No flexibility at runtime
- Code generation resolves dependencies
- Uses generics/templates
- No RTTI
- Near zero runtime overhead

In both, compiler is unaware of the concrete types associated with interfaces

Compiler-driven dependency injection

- Although software quality attributes are increased, interface boundaries hinder code optimization
  - Indirect calls
  - Inlining, constant propagation
  - Interprocedural optimization
  - Dead code elimination
- **Compiler**-driven approach: the frontend is aware of dependencies
- How much does substituting interfaces with concrete implementations in the AST improve code optimization?

MCU and DigitalPort interfaces

In the prototype Robotics Language[1], built on top of the LLVM backend:

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

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

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

[1] Available at github.com/thborges/robcmp

LED blinking

The main program:

```
// an mcu implementation will be bond here
mmcu = mcu();
// the firmware needs a digital port
led = digitalport();

int16 main() {
  led.mode(portmode.output);
  loop {
    led.set(true);      // turn on the LED
    mmcu.wait_ms(500);
    led.set(false);     // turn off the LED
    mmcu.wait_ms(500);
  }
}
```

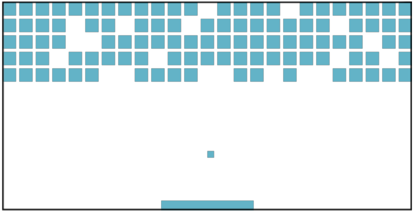
The injection file:

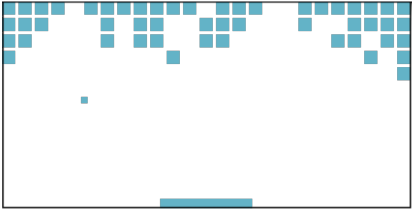
```
bind avr5mcu to mmcu {
  bind b5 to led;
}
```

Case study: Breakout game

Core hardware components:
- the MCU (ATmega328P, tested with STM32)
- digital ports
- SPI display (SSD1306)
- Databus (SPI and UART)

Software interfaces:
- Canvas abstraction, which renders to an off-screen buffer
- Display driver which uses the canvas





Case study: Breakout game

- Game versions[2]:
  - *vtable*, using C++ with virtual methods
  - *vtabledi*, using compile-time DI
  - *concept*, using C++ 20 concepts
  - *conceptdi*, using compile-time DI
  - *rob* (proposed approach)
- Target: ATmega328P + SSD1306 display
- clang++ version 19.1.2 using `-Oz`, `-fno-exceptions`, `-ffunction-sections`, `-fno-rtti`, `-fdata-sections`, and `-lto=thin`
- Metrics: firmware size, runtime fps, instruction count.

[2] Available at https://github.com/thborges/sblp2025.

Software quality attributes: maintainability

```
// injector for the conceptdi version
using spi_t = avr5_spi<avr5mcu_b3, avr5mcu_b4, avr5mcu_b5,
    avr5mcu_b2>;
using display_t = ssd1306<spi_t, avr5mcu_b1, avr5mcu_b0,
    avr5mcu_b2, avr5mcu, ssd1306_framebuffer>;

auto breakout_injector = make_injector(
  bind<c_mcu>.to<avr5mcu>(),
  bind<c_databus_uart0>.to<avr5_uart0>(),
  bind<c_buffer8>.to<ssd1306_framebuffer>(),
  bind<c_databus_display>.to<spi_t>(),
  bind<c_display>.to<display_t>(),
  bind<c_digitalport_b2>.to<avr5mcu_b2>(),
  bind<c_avr5_ss>.to<avr5mcu_b2>()
  ... // 5 more bindings for b0, b1, b3 -- b5
};
```

```
// injector for the rob version
bind avr5mcu to mcu {
  bind b0 to ssd1306.reset;
  bind b1 to ssd1306.datacmd;
  bind b2 to ssd1306.select;
  bind uart0 to dbus_uart;
  bind spi to dbus_display, ssd1306.dbus;
};
```

Firmware size results

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

version	**size**	$\Delta\%$	**fps**	$\Delta\%$
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

Size: Significant stack manipulation and reduced efficiency of compiler optimization passes (inlining, constant propagation, DCE, IPO)

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

version	**size**	$\Delta\%$	**fps**	$\Delta\%$
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

fps: Reduction in the total number of instructions (simplification of critical paths), use of simpler, faster instructions, and fewer memory and stack operations

Instruction count comparison

		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

...

Instruction count comparison

		C++			
Instr.	**rob**	**vtables**	**Δ**	**concepts**	**Δ**
		...			
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

Conclusion and Future Work

- Developing maintainable and optimized software for resource-constrained embedded systems is challenging
- Compiler-driven DI consistently shows smaller, faster firmware, outperforming C++ OOP and concept-based DI
- Future work:
  - Evaluation on a broader range of embedded software
  - More benchmarks on diverse platforms
  - Support for advanced dependency patterns (transient, thread-local, ...)

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

Thiago Borges de Oliveira – thborges@ufj.edu.br
Ariadne de Andrade Costa – ariadne.costa@ufj.edu.br

Instituto de Ciências Exatas e Tecnológicas (ICET)
Universidade Federal de Jataí (UFJ)

SBLP 2025, Recife, Brazil

Concrete MCU implementation

```
type avr5mcu implements mcu {

  uint32 clock() { return 16E6; }

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

  b5 implements digitalport {
    void mode(portmode m) { ddrb.b5 = m; }
    void set(bool v) { portb.b5 = v; }
    bool get() { return portb.b5; }
  }
  //...
}
```

Binding points in the prototype language

```
type game {
    gdisplay = display();
}

// sample code that will bind the concrete ssd1306 display to
    game.gdisplay (DI code)
bind ssd1306 to game.gdisplay;

// extended syntax of the bind statement
bind other to x, w {
    f1 to y, z;
}
```

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();
    }
}
```