MoonBit: A Language and Toolchain

Designed for tooling and large scale collaboration

- Hongbo Zhang @ MoonBit

- Bisheng Meetup

About Me

Hongbo Zhang @bobzhang1988

Been passionate about creating programming languages and developer tools for 20 years:

- Wukong DSL (Bachelor's thesis, 2009, Tsinghua + MSR)
- Fan (Master's thesis, UPenn PLClub), bootstrapped as its own meta-language
- OCaml (Core contributor)
- BuckleScript/ReScript (known as ReasonML) (Creator)
- Flow (Core contributor)

Outline

- Why MoonBit?
    - Our long-term vision and mid-term goals
- What is MoonBit?
    - Designed for tooling
    - A tour of the language
        - Data-oriented design
        - Efficient functional style
        - Checked effects
    - Multiple backends
- AI-assisted programming (skipped in this talk due to time constraint)

Start of MoonBit, October 2022

- An opportunity to build a complete new language from scratch with a *team*
  - A significant advantage given that BuckleScript began as a hobby project
- Long-term thinking: What's the *next big thing* in programming?
  - Incremental delivery is *needed*

The Next Big Thing?

- Cloud computing? (WASM - 2017)
    - A prime opportunity for new languages (comparative advantages)
- AI coding? (ChatGPT - Nov 2022)
    - Not hype—it's real and increasingly practical today
    - Open question: Which languages and tools are AI-friendly?
        - Languages easy for static analysis => Easier for LLMs?
        - Should we build IDEs optimized for LLMs rather than human programmers?
        - How to enable concurrent AI-based IDEs?

Long-term Vision: Large-scale AI-assisted Programming from the ground up

- IDEs built for hundreds of AI programmers to collaborate on large projects
- AI programming requires fundamentally re-architecting the entire development toolchain(not just IDE, VCS, etc)
- Humans and AI agents working together fluidly on the same codebase

From Vision to Reality

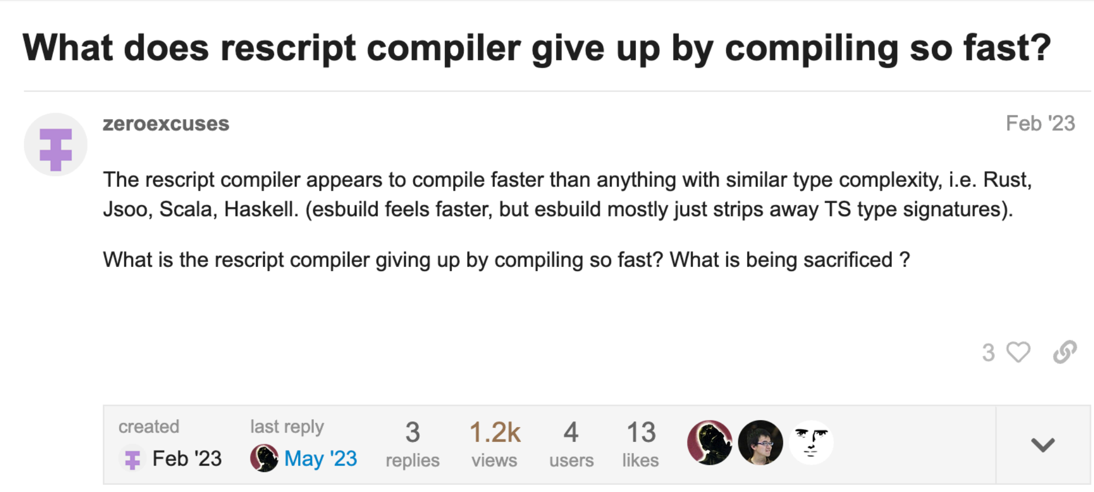
What Is MoonBit?

- An integrated development platform with a comprehensive toolchain

  Editing, debugging, building, testing, coverage, packaging, AI integration, etc.

- Designed from the ground up to prioritize:
  - AI productivity and reliability
  - Tooling excellence
  - Fast feedback loop

MoonBit: Heavily Influenced by ReScript(OCaml), Go, and Rust



What does rescript compiler give up by compiling so fast?

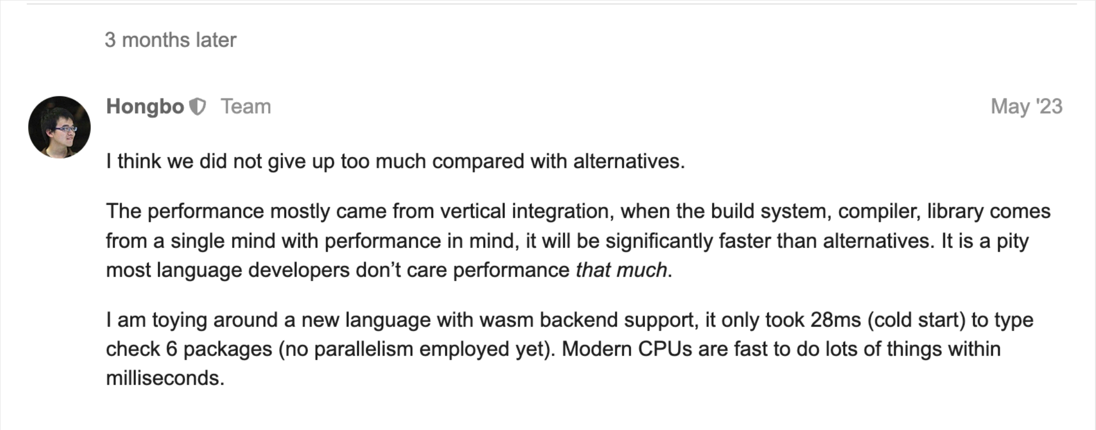
zeroexcuses Feb '23

The rescript compiler appears to compile faster than anything with similar type complexity, i.e. Rust, Jsoo, Scala, Haskell. (esbuild feels faster, but esbuild mostly just strips away TS type signatures).

What is the rescript compiler giving up by compiling so fast? What is being sacrificed ?

3

created Feb '23 | last reply May '23 | 3 replies | 1.2k views | 4 users | 13 likes



3 months later

Hongbo Team May '23

I think we did not give up too much compared with alternatives.

The performance mostly came from vertical integration, when the build system, compiler, library comes from a single mind with performance in mind, it will be significantly faster than alternatives. It is a pity most language developers don't care performance *that much*.

I am toying around a new language with wasm backend support, it only took 28ms (cold start) to type check 6 packages (no parallelism employed yet). Modern CPUs are fast to do lots of things within milliseconds.

- Fast feedback loop is key to developer experience

MoonBit: Heavily Influenced by ReScript(OCaml), Go, and Rust

- Go's philosophy: *less is more*
  - Simplicity matters
  - From the **user** point of view v.s. from the **implementation** point of view
- Focus on tooling

MoonBit: Heavily Influenced by ReScript(OCaml), Go, and Rust

- Rust's *good* parts with the borrow checker(*opt-in* vs *opt-out*)
  - Such complexity isn't necessary for everyday programming 😄

Mid-term Goals: incremental delivery

MoonBit gained its first commercial users in 2023. Why?

- MoonBit's first primary focus: WebAssembly platform
    - An opportunity to significantly outperform existing solutions
    - The smallest output size
        - Efficient dead code elimination
    - Performance comparable to Rust

Entering into other domains after the success on WebAssembly platform

MoonBit Prioritizes Tooling

- MoonBit is a *new language* designed for efficient tooling and static analysis (lessons learned from ReScript)
    - Co-designed with the IDE (available in the initial release, August 18, 2023)
    - The whole IDE running in the browser without server-side containers
    - Mario **Demo**(https://www.moonbitlang.cn/gallery/mario/)
- Fast static analysis and IDE services:
    - Parallel and incremental design
        - Parallel lexing/parsing and type checking
        - All phases are fault tolerant(IDE shares the same code with compiler)

More Tooling:

- Integrated testing and coverage
- Developer-friendly testing: the community-driven core library has 93% test coverage
- Tests and documentation are first-class citizens in MoonBit's ecosystem

More Tooling:

- Documentation-oriented programming
  "Literate programming done right"
- Documentation is treated as code, with type-checking and verification
- This slide is type checked
- **Demo** (show the source of this slide)

More Tooling:

- Out-of-the-box GUI debugging (with sourcemap support)
- Support for JavaScript, WebAssembly, and native targets
- LLDB-based debugger coming this month

Demo (tour.moonbitlang.com)

- Integrated workflow of testing, coverage and debugging
- tour.moonbitlang.com
    - Interactive learning with live tracing, debugging, and testing

Language Tour

What's the Language Like?

Rust's selective "good" parts without the borrow checker

```
traits, enum, pattern matching, generics...
```

Beyond that, we focus on data-oriented programming and checked effects (work in progress)

Data-Oriented: ADT and Derivable Data Types

```
pub enum JsonValue {
  Null
  True
  False
  Number(Double)
  String(String)
  Array(Array[JsonValue])
  Object(Map[String, JsonValue])
} derive(Eq, Show)  // <-- Automatic derivation of traits
```

Data-Oriented: Pattern Matching Over JSON

```
fn process(value : Json) -> Unit {
  match value {
    {
      "headers": [{ "name": String(a), .. },  ..,  { "name": String(b), .. }],
      ..
    } if a is [.."PREFIX", ..rest] && rest == b => println("same name")
    // pattern match over map, array, json, string
    { "body": { "name": String(s), .. }, .. } => println(s)
    x => println(x) // Exhaustive matching required
  }
}
```

- Native support for pattern matching over JSON with exhaustive checking
- Pattern can be nested, composed with `is` expression

Data-Oriented: Unicode-safe Pattern Matching Over Strings

```
fn is_palindrome(s : @string.View) -> Bool {
  loop s {
    [] | [_] => true // Empty or single character strings are palindromes
    [first, .. rest, last] =>
      if first == last {
        continue rest
      } else {
        false
      }
  }
}
test {
  inspect(is_palindrome("😃heh😃"), content="true")
}
```

- Unicode-safe processing

Data-Oriented: UTF-8 Decoding

```
pub fn decode_utf8(bytes: @bytes.View) -> String {
  let sb = StringBuilder::new()
  loop bytes {
    [0x00..=0x7F as b, .. next] => {
      ... // 1-byte sequence (ASCII): 0xxxxxxx
      continue next
    }
    [0xC0..=0xDF as b1, 0x80..=0xBF as b2, .. next] => {
      ... // 2-byte sequence: 110xxxxx 10xxxxxx
      continue next
    }
    [0xE0..=0xEF as b1, 0x80..=0xBF as b2, 0x80..=0xBF as b3, .. next] => {
      ... // 3-byte sequence: 1110xxxx 10xxxxxx 10xxxxxx
      continue next
    }
    [ 0xF0..=0xF7 as b1, 0x80..=0xBF as b2, 0x80..=0xBF as b3, 0x80..=0xBF as b4,
      .. next,
    ] => {
      ... // 4-byte sequence: 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
      continue next
    }
    [_, .. next] => continue next // Invalid sequence - skip one byte and continue
    [] => ()
  }
  sb.to_string()
}
```

Expression-Oriented: Modular, Easy to Reason About, Composable

```python
def find(seq, target):
  found = False
  for i, value in enumerate(seq):
    if value == target:
      found = True
      break
  if found:
    return i
  else:
    return -1
```

```
fn find[A: Eq](seq: Array[A], target: A) -> Int? {
  for i, item in seq {
    if item == target {
      break Some(i)  // break with payload
    }
  } else {  // 'else' clause for loops!
    None    // Optional type for cleaner API
  }
}
test {
  inspect(find([1, 2, 3], 2), content="Some(1)")
}
```

Even For-Loops Are Functional Expressions (No Mutation Needed)

```
test {
  let array = [1, 2, 3]
  let mut sum = 0 // local mutation
  for i = 0; i < array.length(); i = i + 1 {
    sum += array[i]  // Mutation
  }
}
```

```
test {
  let array = [1, 2, 3]
  let sum = for i = 0, sum = 0 {
    if i < array.length() {
      continue i + 1, sum + array[i] // State passing
    } else {
      break sum // break with payload
    }
  }

}
```

- Easier static analysis for bounds checking

Checked Effects System

Checked Effects: Exceptions

```
fn div(x: Int, y: Int) -> Int! {  // '!' indicates effect (default to Error)
    if y == 0 {
        fail("division by zero") // fail rendered with underlying
    }
    x / y
}
```

- Explicit error checking
- Everything is an expression (no return keyword needed)
- Static control flow with checked exceptions

Checked Effects: Exceptions

```
test {
  let (x1, x2, y1, y2) = (1, 2, 3, 0)
  try {
    let a = div(x1, x2) // IDE rendered _
    let b = div(y1, y2)
    println(a + b)
  } catch {
    err => println(err)
  }
}
```

- Error handling is very fast (implemented via goto, no heap allocation)
- Warnings on unused try blocks
- Type-safe error handling

Checked Effects: Async (Work in Progress)

```
async fn fetch_url(url: String) -> String {  // 'Async' effect
  ...
}
async fn fetch_all(urls: Array[String]) -> Array[String]{
  let results = []
  for url in urls {
    let result = fetch_url(url)  // Effect propagation
    results.push(result)
  }
  results
}
```

- Compiled with continuations (normal and error continuations)
- Shipped the latest release (experimental)
- Exploring structured concurrency in future releases
- Cross-backend support via virtual packages planned

Effect polymorphism

```
fn Array::pmap[A, B](data: Self[A], f: (A) -> B?Error) -> Self[B]?Error {
  let result = []
  for item in data {
    result.push(f(item))
  }
  result
}
```

```
test {
  let v = [1, 2, 3]
  inspect(v.pmap(fn { x => x + 1 }), content="[2, 3, 4]")
  inspect(
    try? v.pmap(fn { x => if x > 2 { raise Failure("too large") } else { x + 1 } }),
  content=
    #|Err("Failure(too large)")

  )
}
```

More to Explore of the language

- https://docs.moonbitlang.com
- Multiple paradigm support with data-oriented focus:
  - Data-oriented
  - Limited Object-oriented
  - Efficient functional
  - Limited imperative

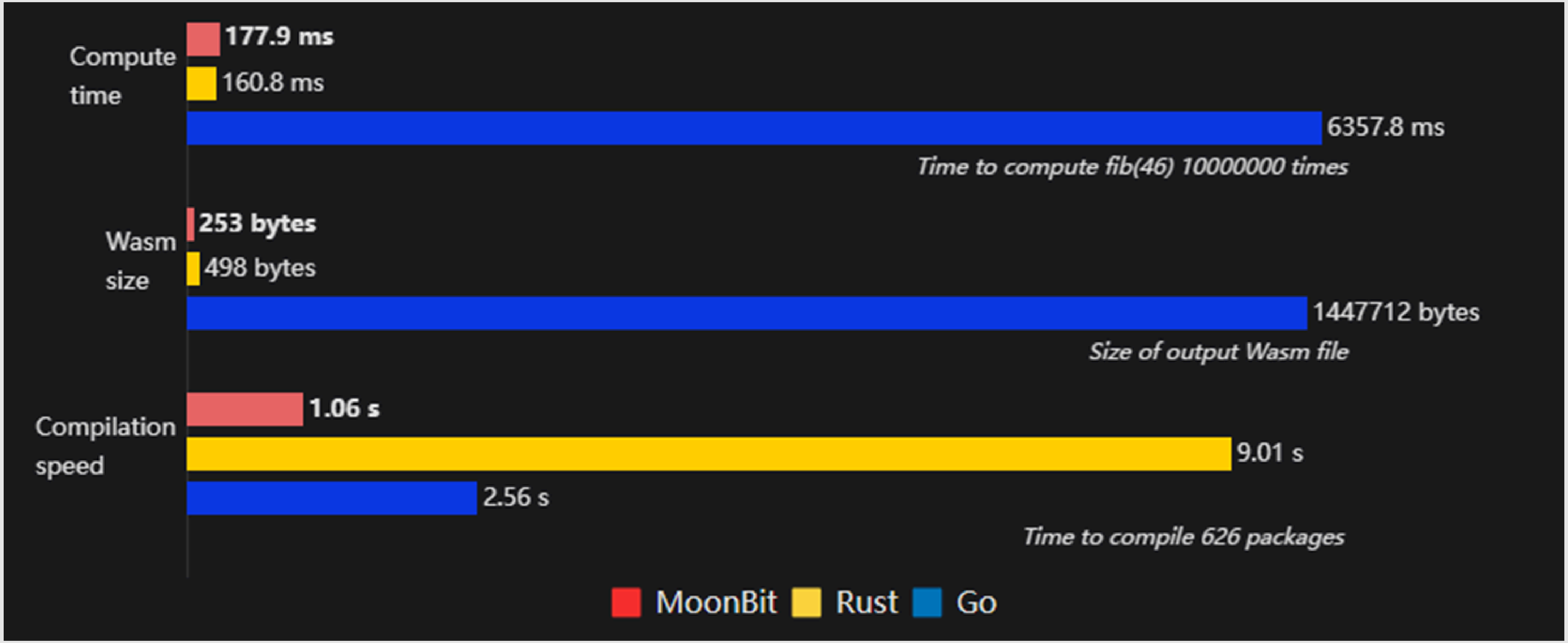
From One Language to Many Targets

Multiple Backends

- Multi-backend support for diverse industrial applications:
    - WebAssembly (1.0, 2.0) backends, with/without GC
        - WebAssembly Component Model support
    - JavaScript backend
        - Performance exceeding hand-tuned JavaScript
    - Native backends
        - LLVM backend
        - C backend

WebAssembly Backend

- MoonBit generates highly optimized Wasm code



JavaScript Backend

- MoonBit outperforms hand-written JavaScript in many cases

```
data.flatMap(c => c.members)
    .filter(it => it.gender)
    .map(it => Math.min(100, it.score + 5))
    .map(it => grade(it))
    .filter(it => it === 'A')
    .reduce((acc, _) => acc + 1, 0);
```

JS style iteration

```
data.iter()
    .flat_map(fn { c => c.members.iter() })
    .filter(fn { r => r.gender })
    .map(fn { r => min(100, r.score + 5) })
    .map(fn { r => grade(r) })
    .filter(fn { g => g == "A" })
    .fold(fn { c, _ => c + 1 }, 0)
```

MoonBit style iteration (26x faster)

Native Backends

- Compile to C for microcontrollers
  - Ideal for embedded and resource-constrained environment
- Compile using LLVM IR for better optimizations and debugger support
  - Full native performance for desktop and server applications
- Compile using MoonSSA IR: *planned for the future*
  - Completly self hosted without relying on LLVM or C
  - MoonBit is ideal for writing compilers

The Status of MoonBit Optimizations

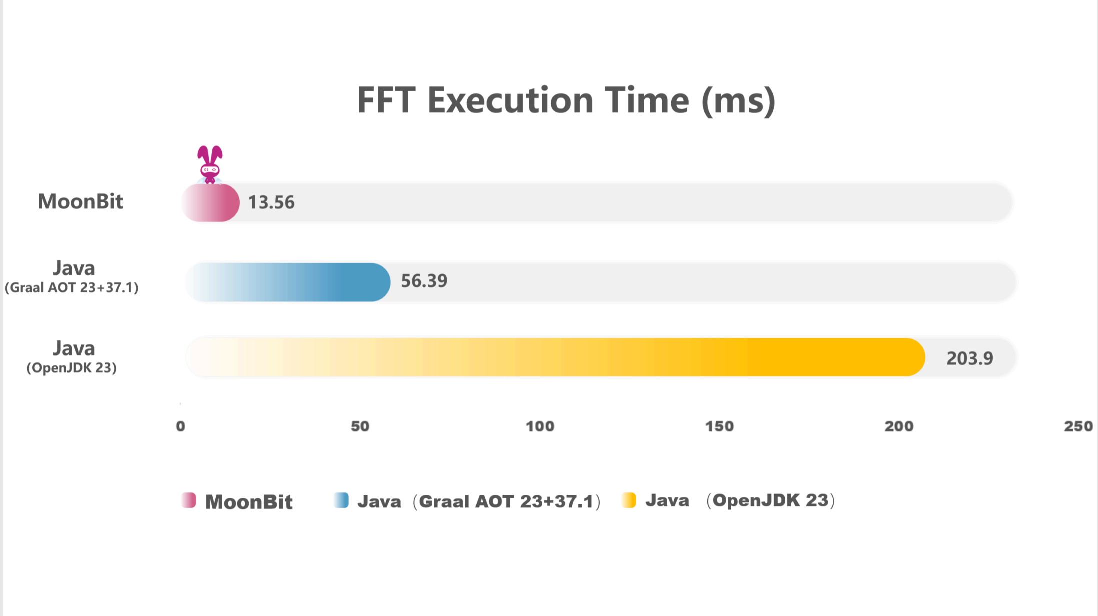
- Many low-hanging fruits remain, but performance is already *excellent*
- Closures are heavily optimized away, especially for hot paths
- Whole program compilation:
    - Memory layout is crucial
    - Optimizations like unboxed characters (T? unboxed)
    - Carefully designed to *compile fast*

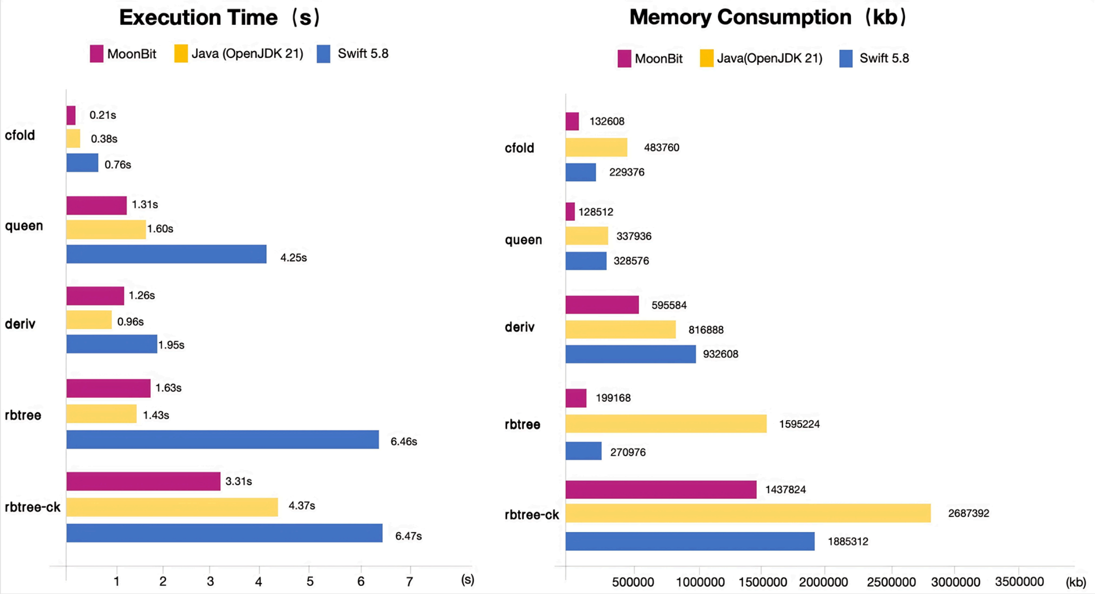
Optimization Examples:

```
fn sum(x : @list.T[Int]) -> Int {
  let mut sum = 0
  x.each(fn { i => sum += i }) // Higher-order function with closure
  // No heap allocations
  // Competitive with hand-optimized C in many benchmarks

  sum
}
```

Native Backends Performance





LLDB Support (Coming Next Month)

2025/06/18 beta