2022 | OpenInfra Days China
中国开源基础设施活动日

Will D be a better system programming language?

李枫 Koo

独立开发者(Indie developer)

Who Am I

- The main translator of the book «Gray Hat Hacking The Ethical Hacker's Handbook, Fourth Edition» (ISBN：9787302428671) & «Linux Hardening in Hostile Networks, First Edition» (ISBN: 9787115544384)

- Pure software developmement for ~15 years(~11 years on Mobile Dev)
- Actively participate in various activities of the open source community
- https://github.com/XianBeiTuoBaFeng2015/MySlides/tree/master/Conf
  https://github.com/XianBeiTuoBaFeng2015/MySlides/tree/master/LTS
- Recently, focus on infrastructure of Cloud/Edge Computing, AI, Virtualization, Program Runtimes, Network, 5G, RISC-V, EDA…

Agenda

I. Background
- System Programming
- Tech Stack

II. Design philosophy
- Overview
- Features

III. LDC
- Overview
- Internals

IV. D for Linux Kernel
- Latest Trends
- D for a @safer Linux Kernel
- DPP

V. DHDL
- Overview

VI. D-based DSLs

- Overview
- Vox

VII. Pros & Cons

- Overview
- Performance
- Ecosystem
- D 3.0

VIII.My ideas

- Ideas

IX. Wrap-up

I. Background

1) System programming

- https://en.wikipedia.org/wiki/Systems_programming

Systems programming, or **system programming**, is the activity of programming[1] computer system software. The primary distinguishing characteristic of systems programming when compared to application programming is that application programming aims to produce software which provides services to the user directly (e.g. word processor), whereas systems programming aims to produce software and software platforms which provide services to other software, are performance constrained, or both (e.g. operating systems, computational science applications, game engines, industrial automation, and software as a service applications).[1]

Systems programming requires a great degree of hardware awareness. Its goal is to achieve efficient use of available resources, either because the software itself is performance critical or because even small efficiency improvements directly transform into significant savings of time or money.

Overview [edit]

The following attributes characterize systems programming:

- The programmer can make assumptions about the hardware and other properties of the system that the program runs on, and will often exploit those properties, for example by using an algorithm that is known to be efficient when used with specific hardware.
- Usually a low-level programming language or programming language dialect is used so that:
  - Programs can operate in resource-constrained environments
  - Programs can be efficient with little runtime overhead, possibly having either a small runtime library or none at all
  - Programs may use direct and "raw" control over memory access and control flow
  - The programmer may write parts of the program directly in assembly language
- Often systems programs cannot be run in a debugger. Running the program in a simulated environment can sometimes be used to reduce this problem.

Systems programming is sufficiently different from application programming that programmers tend to specialize in one or the other.[*citation needed*]

In systems programming, often limited programming facilities are available. The use of automatic garbage collection is not common and debugging is sometimes hard to do. The runtime library, if available at all, is usually far less powerful, and does less error checking. Because of those limitations, monitoring and logging are often used; operating systems may have extremely elaborate logging subsystems.

Implementing certain parts in operating systems and networking requires systems programming, for example implementing paging (virtual memory) or a device driver for an operating system.

History [edit]

Originally systems programmers invariably wrote in assembly language. Experiments with hardware support in high level languages in the late 1960s led to such languages as PL/S, BLISS, BCPL, and extended ALGOL for Burroughs large systems. Forth also has applications as a systems language. In the 1970s, C became widespread, aided by the growth of Unix. More recently a subset of C++ called Embedded C++ has seen some use, for instance it is used in the I/O Kit drivers of macOS.[2]

- …

1.1 System programming language

- https://en.wikipedia.org/wiki/System_programming_language

A **system programming language** is a programming language used for system programming; such languages are designed for writing system software, which usually requires different development approaches when compared with application software. Edsger Dijkstra refers to these language as **Machine Oriented High Order Languages**, or **mohol**.[1]

General-purpose programming languages tend to focus on generic features to allow programs written in the language to use the same code on different platforms. Examples of such languages include ALGOL and Pascal. This generic quality typically comes at the cost of denying direct access to the machine's internal workings, and this often has negative effects on performance.

System languages, in contrast, are designed not for compatibility, but for performance and ease of access to the underlying hardware while still providing high-level programming concepts like structured programming. Examples include SPL and ESPOL, both of which are similar to ALGOL in syntax but tuned to their respective platforms. Others are cross-platform but designed to work close to the hardware, like BLISS, JOVIAL and BCPL.

Some languages straddle the system and application domains, bridging the gap between these uses. The canonical example is C, which is used widely for both system and application programming. Some modern languages also do this such as Rust and Swift.

Features [edit]

In contrast with application languages, system programming languages typically offer more-direct access to the physical hardware of the machine: an archetypical system programming language in this sense was BCPL. System programming languages often lack built-in input/output (I/O) facilities because a system-software project usually develops its own I/O mechanisms or builds on basic monitor I/O or screen management facilities. The distinction between languages used for system programming and application programming became blurred over time with the widespread popularity of PL/I, C and Pascal.

Major languages [edit]

Language [hide]	Originator	Birth date	Influenced by	Used for
ESPOL	Burroughs Corporation	1961	ALGOL 60	MCP
PL/I	IBM, SHARE	1964	ALGOL, FORTRAN, some COBOL	Multics
PL/S	IBM	1960's	PL/I	OS/360
PL360	Niklaus Wirth	1968	ALGOL 60	ALGOL W
Pascal	Niklaus Wirth	1970	ALGOL W	Apollo Computer Aegis, Apple MacApp, UCSD p-System
BLISS	Carnegie Mellon University	1970	ALGOL-PL/I[5]	VMS (portions)
Language for Systems Development (LSD)	R. Daniel Bergeron, et.al. (Brown University)	1971	PL/I	
C	Dennis Ritchie	1972	BCPL, B (programming language)	Most operating system kernels, including Unix-like systems
NEWP	Burroughs	1970' s	ESPOL, ALGOL	MCP
PL/8	IBM	1970's	PL/I	AIX
PL-6	Honeywell, Inc.	1970's	PL/I	CP-6
SYMPL	CDC	1970's	JOVIAL	NOS subsystems, most compilers, FSE editor
C++	Bjarne Stroustrup	1979	C, Simula	GUI applications (Qt, Windows, etc), games (Unreal Engine)
Ada	Jean Ichbiah, S. Tucker Taft	1983	ALGOL 68, Pascal, C++, Eiffel	Embedded systems, OS kernels, compilers, games, simulations, CubeSat, air traffic control, avionics
D	Digital Mars	2001	C++	Multiple domains
Nim	Andreas Rumpf	2006	Python, Ada, Lisp, Oberon, C++, Modula-3, Object Pascal	Games, compilers, OS kernels, app development, embedded systems, etc.
Go	Google	2009	Python, dislike of C++, some syntax from Pascal	Docker, Podman
Rust	Mozilla Research[6]	2010	C++, Haskell, Erlang, Ruby	Servo, Redox OS
Swift	Apple Inc.	2014	C, Objective-C, D, Rust	macOS, iOS, watchOS, and tvOS app development [a]

- ...

1.1.1 Go

- https://en.wikipedia.org/wiki/Go_(programming_language)

Go is a statically typed, compiled programming language designed at Google[10] by Robert Griesemer, Rob Pike, and Ken Thompson.[11] It is syntactically similar to C, but with memory safety, garbage collection, structural typing,[5] and CSP-style concurrency.[12] It is often referred to as **Golang** because of its former domain name, `golang.org`, but its proper name is Go.[13]

There are two major implementations:

- Google's self-hosting[14] "gc" compiler toolchain, targeting multiple operating systems and WebAssembly.[15]
- gofrontend, a frontend to other compilers, with the *libgo* library. With GCC the combination is gccgo;[16] with LLVM the combination is gollvm.[17][a]

A third-party source-to-source compiler, GopherJS,[19] compiles Go to JavaScript for front-end web development.

History [edit]

Go was designed at Google in 2007 to improve programming productivity in an era of multicore, networked machines and large codebases.[20] The designers wanted to address criticism of other languages in use at Google, but keep their useful characteristics:[21]

- Static typing and run-time efficiency (like C)
- Readability and usability (like Python or JavaScript)[22]
- High-performance networking and multiprocessing

Its designers were primarily motivated by their shared dislike of C++.[23][24][25]

Go was publicly announced in November 2009,[26] and version 1.0 was released in March 2012.[27][28] Go is widely used in production at Google[29] and in many other organizations and open-source projects.

In November 2016, the Go and Go Mono fonts were released by type designers Charles Bigelow and Kris Holmes specifically for use by the Go project. Go is a humanist sans-serif resembling Lucida Grande, and Go Mono is monospaced. Both fonts adhere to the WGL4 character set and were designed to be legible with a large x-height and distinct letterforms. Both Go and Go Mono adhere to the DIN 1450 standard by having a slashed zero, lowercase `l` with a tail, and an uppercase `I` with serifs.[30][31]

In April 2018, the original logo was replaced with a stylized GO slanting right with trailing streamlines. (The Gopher mascot remained the same.[32])

Mascot of Go programming language is a Gopher shown above.

Generics [edit]

The lack of support for generic programming in initial versions of Go drew considerable criticism.[33] The designers expressed an openness to generic programming and noted that built-in functions *were* in fact type-generic, but are treated as special cases; Pike called this a weakness that might at some point be changed.[34] The Google team built at least one compiler for an experimental Go dialect with generics, but did not release it.[35]

In August 2018, the Go principal contributors published draft designs for generic programming and error handling and asked users to submit feedback.[36][37] However, the error handling proposal was eventually abandoned.[38]

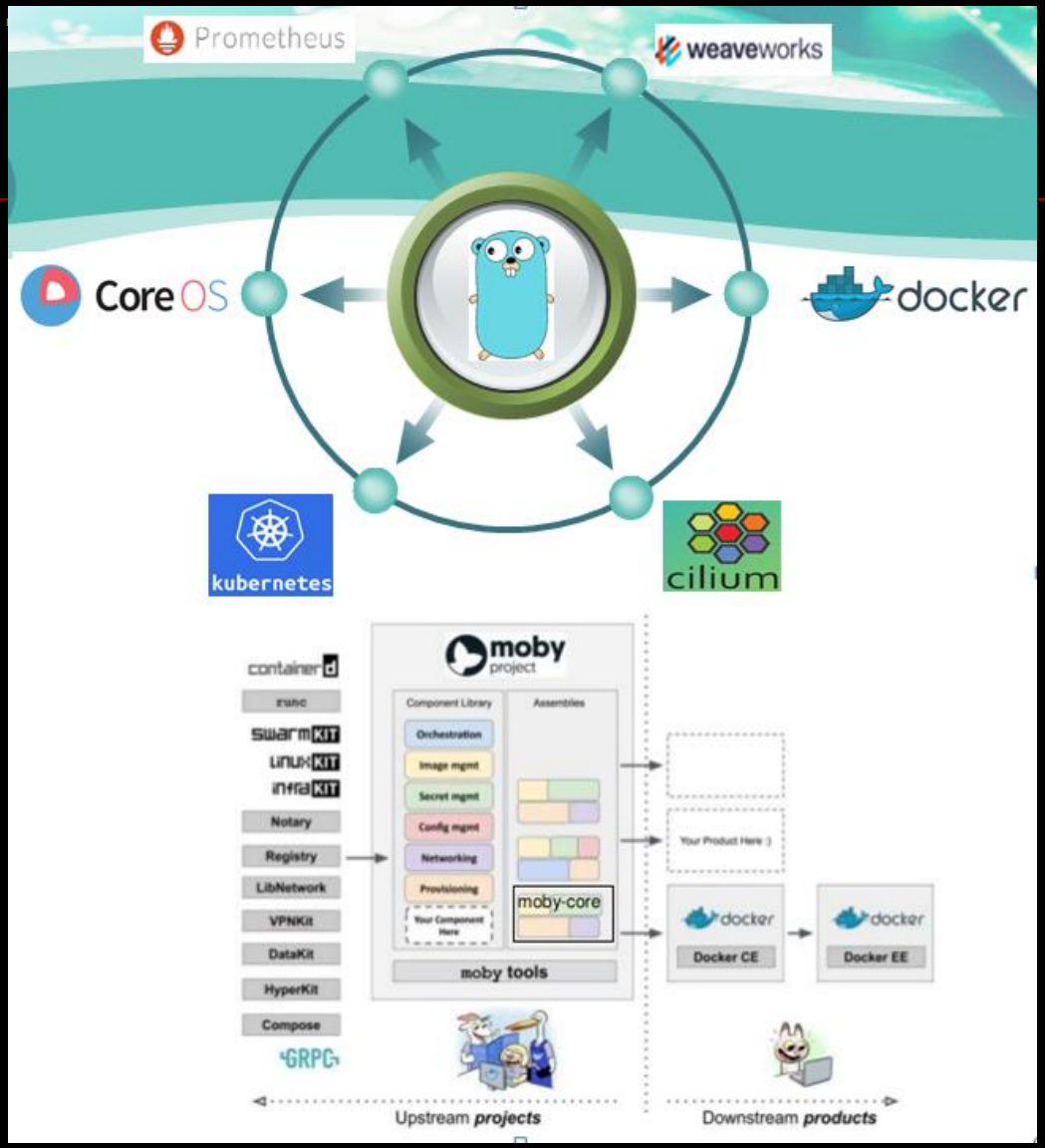
In June 2020, a new draft design document[39] was published that would add the necessary syntax to Go for declaring generic functions and types. A code translation tool, *go2go*, was provided to allow users to try the new syntax, along with a generics-enabled version of the online Go Playground.[40]

Generics were finally added to Go in version 1.18.[41]

Paradigm	Multi-paradigm: concurrent imperative, object-oriented[1][2]
Designed by	Robert Griesemer Rob Pike Ken Thompson
Developer	The Go Authors[3]
First appeared	November 10, 2009; 12 years ago
Stable release	1.18.4[4] / 12 July 2022; 12 days ago
Typing discipline	Inferred, static, weak, structural,[5][6] nominal
Implementation language	Go, Assembly language (gc); C++ (gofrontend)
OS	DragonFly BSD, FreeBSD, Linux, macOS, NetBSD, OpenBSD,[7] Plan 9,[8] Solaris, Windows
License	3-clause BSD[3] + patent grant[9]
Filename extensions	.go
Website	go.dev
Major implementations	
gc, gofrontend	
Influenced by	
C, Oberon-2, Limbo, Active Oberon, communicating sequential processes, Pascal, Oberon, Smalltalk, Newsqueak, Modula-2, Alef, APL, BCPL, Modula, occam	
Influenced	
Odin, Crystal, Zig	

- https://github.com/avelino/awesome-go
- ...

Go-based Cloud Ecosystem



...

1.1.2 Rust

1.1.2.1 Overview

- https://en.wikipedia.org/wiki/Rust_(programming_language)

Rust is a multi-paradigm, general-purpose programming language. Rust emphasizes performance, type safety, and concurrency.[9][10][11] Rust enforces memory safety—that is, that all references point to valid memory—without requiring the use of a garbage collector or reference counting present in other memory-safe languages.[11][12] To simultaneously enforce memory safety and prevent concurrent data races, Rust's *borrow checker* tracks the object lifetime and variable scope of all references in a program during compilation.[13] Rust is popular for systems programming[11] but also offers high-level features including functional programming constructs.[14]

Software developer Graydon Hoare designed Rust while working at Mozilla Research in 2006.[15] Mozilla officially sponsored the project in 2009, and the designers refined the language while writing the Servo experimental browser engine[16] and the Rust compiler. Rust's major influences include SML, OCaml, C++, Cyclone, Haskell, and Erlang.[4] Since the first stable release in January 2014, Rust has been adopted by companies including Amazon, Discord, Dropbox, Facebook (Meta), Google (Alphabet), and Microsoft.

Rust has been noted for its growth as a newer language[10][17] and has been the subject of academic programming languages research.[18][19][20][11]

The official Rust logo

Paradigms	Multi-paradigm: concurrent, functional, generic, imperative, structured
Designed by	Graydon Hoare
First appeared	July 7, 2010; 12 years ago
Stable release	1.62.1[1] / July 19, 2022; 7 days ago
Typing discipline	Affine, inferred, nominal, static, strong
Implementation language	Rust
Platform	Cross-platform[2][note 1]
OS	Cross-platform[2][note 2]
License	MIT and Apache 2.0 (dual-licensed) [3]
Filename extensions	`.rs`, `.rlib`
Website	www.rust-lang.org
Influenced by	Alef, C#, C++, Cyclone, Erlang, Haskell, Limbo, Newsqueak, OCaml, Ruby, Scheme, Standard ML, Swift[4][note 3]
Influenced	Carbon, Crystal, Idris[5], Spark[6], Swift[7], Project Verona[8], Zig

- History

Early origins (2006–2012) [edit]

Rust grew out of a personal project begun in 2006 by Mozilla employee Graydon Hoare. Mozilla began sponsoring the project in 2009 and officially announced the project in 2010.[15][21] During the same year, work had shifted from the initial compiler written in OCaml to a self-hosting compiler based on LLVM written in Rust. The new Rust compiler, named `rustc`, successfully compiled itself in 2011.[22] The first numbered pre-alpha version of the compiler, Rust 0.1, was released in January 2012.[23]

Mozilla Foundation headquarters in Mountain View, California

Evolution (2013–2019) [edit]

Rust's type system, changed considerably between versions 0.2, 0.3, and 0.4 of Rust. Version 0.2 introduced classes for the first time,[24] and version 0.3 added destructors and polymorphism through the use of interfaces.[25] In Rust 0.4, traits were added as a means to provide inheritance; interfaces were unified with traits and removed as a separate feature. Classes were also removed and replaced by a combination of implementations and structured types.[26] Along with conventional static typing, before version 0.4, Rust also supported typestate analysis through contracts. It was removed in release 0.4, though the same functionality can be achieved by leveraging Rust's type system.[27]

In January 2014, the editor-in-chief of *Dr. Dobb's Journal*, Andrew Binstock, commented on Rust's chances of becoming a competitor to C++ in addition to the languages D, Go, and Nim (then Nimrod). According to Binstock, while Rust was "widely viewed as a remarkably elegant language", adoption slowed because it repeatedly changed between versions.[28] The first stable release, Rust 1.0, was announced on May 15, 2015.[29][30]

Mozilla Layoffs and Rust Foundation (2020–present) [edit]

In August 2020, Mozilla laid off 250 of its 1,000 employees worldwide as part of a corporate restructuring caused by the long-term impact of the COVID-19 pandemic.[31][32] The team behind Servo, a browser engine written in Rust, was completely disbanded. The event raised concerns about the future of Rust, as some members of the team were active contributors to Rust.[33] In the following week, the Rust Core Team acknowledged the severe impact of the layoffs and announced that plans for a Rust foundation were underway. The first goal of the foundation would be to take ownership of all trademarks and domain names, and take financial responsibility for their costs.[34]

On February 8, 2021, the formation of the Rust Foundation was announced by its five founding companies (AWS, Huawei, Google, Microsoft, and Mozilla).[35][36] In a blog post published on April 6, 2021, Google announced support for Rust within Android Open Source Project as an alternative to C/C++.[37][38]

According to the Stack Overflow Developer Survey in 2022, 9% of respondents have recently done extensive development in Rust.[39] The survey has additionally named Rust the "most loved programming language" every year from 2016 to 2022 (inclusive), a ranking based on the number of current developers who express an interest in continuing to work in the same language.[40][note 4] In 2022, Rust tied with Python for "most wanted technology" with 18% of developers not currently working in Rust expressing an interest in doing so.[39][41]

- https://foundation.rust-lang.org/

- https://www.rust-lang.org/

Why Rust?

Performance

Rust is blazingly fast and memory-efficient: with no runtime or garbage collector, it can power performance-critical services, run on embedded devices, and easily integrate with other languages.

Reliability

Rust's rich type system and ownership model guarantee memory-safety and thread-safety — enabling you to eliminate many classes of bugs at compile-time.

Productivity

Rust has great documentation, a friendly compiler with useful error messages, and top-notch tooling — an integrated package manager and build tool, smart multi-editor support with auto-completion and type inspections, an auto-formatter, and more.

Build it in Rust

In 2018, the Rust community decided to improve programming experience for a few distinct domains (see the 2018 roadmap). For these, you can find many high-quality crates and some awesome guides on how to get started.

Command Line

Whip up a CLI tool quickly with Rust's robust ecosystem. Rust helps you maintain your app with confidence and distribute it with ease.

BUILDING TOOLS

WebAssembly

Use Rust to supercharge your JavaScript, one module at a time. Publish to npm, bundle with webpack, and you're off to the races.

WRITING WEB APPS

Networking

Predictable performance. Tiny resource footprint. Rock-solid reliability. Rust is great for network services.

WORKING ON SERVERS

Embedded

Targeting low-resource devices? Need low-level control without giving up high-level conveniences? Rust has you covered.

STARTING WITH EMBEDDED

- https://www.memorysafety.org/
- ...

Official Compiler(base on LLVM)

- https://github.com/rust-lang/rust

```
[mydev@fedora rust-master]$ tree -L 1 .
.
├── Cargo.lock
├── Cargo.toml
├── CODE_OF_CONDUCT.md
├── compiler
├── config.toml.example
├── configure
├── CONTRIBUTING.md
├── COPYRIGHT
├── library
├── LICENSE-APACHE
├── LICENSE-MIT
├── README.md
├── RELEASES.md
├── rustfmt.toml
├── src
├── triagebot.toml
└── x.py
```

```
[mydev@fedora rust-master]$ find . -name "*.git"
./.git
./library/backtrace/.git
./library/stdarch/.git
./library/stdarch/crates/intrinsic-test/acle/.git
./src/doc/book/.git
./src/doc/edition-guide/.git
./src/doc/embedded-book/.git
./src/doc/nomicon/.git
./src/doc/reference/.git
./src/doc/rust-by-example/.git
./src/doc/rustc-dev-guide/.git
./src/llvm-project/.git
./src/tools/cargo/.git
./src/tools/miri/.git
./src/tools/rls/.git
./src/tools/rust-installer/.git
[mydev@fedora rust-master]$
```

```
[mydev@fedora rust-master]$ tree -L 1 ./compiler
./compiler
├── rustc
├── rustc_apfloat
├── rustc_arena
├── rustc_ast
├── rustc_ast_lowering
├── rustc_ast_passes
├── rustc_ast_pretty
├── rustc_attr
├── rustc_borrowck
├── rustc_builtin_macros
├── rustc_codegen_cranelift
├── rustc_codegen_gcc
├── rustc_codegen_llvm
├── rustc_codegen_ssa
├── rustc_const_eval
├── rustc_data_structures
├── rustc_driver
├── rustc_error_codes
├── rustc_error_messages
├── rustc_errors
├── rustc_expand
├── rustc_feature
├── rustc_fs_util
├── rustc_graphviz
├── rustc_hir
├── rustc_hir_pretty
├── rustc_incremental
├── rustc_index
├── rustc_infer
├── rustc_interface
├── rustc_lexer
├── rustc_lint
├── rustc_lint_defs
├── rustc_llvm
├── rustc_log
├── rustc_macros
├── rustc_metadata
├── rustc_middle
├── rustc_mir_build
├── rustc_mir_dataflow
├── rustc_mir_transform
├── rustc_monomorphize
├── rustc_parse
├── rustc_parse_format
├── rustc_passes
├── rustc_plugin_impl
├── rustc_privacy
├── rustc_query_impl
├── rustc_query_system
├── rustc_resolve
├── rustc_save_analysis
├── rustc_serialize
├── rustc_session
├── rustc_smir
├── rustc_span
├── rustc_symbol_mangling
├── rustc_target
├── rustc_traits
├── rustc_trait_selection
├── rustc_typeck
├── rustc_type_ir
└── rustc_ty_utils
```

Rust for Cloud Native

- https://github.com/awesome-rust-cloud-native/awesome-rust-cloud-native

Applications and Services

- apache/incubator-teaclave: open source universal secure computing platform, making computation on privacy-sensitive data safe and simple
- bottlerocket-os/bottlerocket: an operating system designed for hosting containers
- containers/krunvm: manage lightweight VMs created from OCI images
- containers/youki: a container runtime written in Rust
- datafuselabs/datafuse: A Modern Real-Time Data Processing & Analytics DBMS with Cloud-Native Architecture, built to make the Data Cloud easy
- firecracker-microvm/firecracker: secure and fast microVMs for serverless computing
- infinyon/fluvio: Cloud-native real-time data streaming platform with in-line computation capabilities
- krustlet/krustlet: Kubernetes Rust Kubelet
- kube-rs/controller-rs: a Kubernetes example controller
- kube-rs/version-rs: example Kubernetes reflector and web server
- kubewarden/policy-server: webhook server that evaluates WebAssembly policies to validate Kubernetes requests
- linkerd/linkerd2-proxy: a purpose-built proxy for the Linkerd service mesh
- openebs/mayastor: A cloud native declarative data plane in containers for containers
- rancher-sandbox/lockc: eBPF-based MAC security audit for container workloads
- tikv/tikv: distributed transactional key-value database
- tremor-rs/tremor-runtime: an event processing system that supporting complex workflows such as aggregation, rollups, an ETL language, and a query language
- valeriansaliou/sonic: fast, lightweight & schema-less search backend
- WasmEdge/WasmEdge: WasmEdge is a high-performance WebAssembly (Wasm) Virtual Machine (VM) runtime, which enables serverless functions to be embedded into any software platform; from cloud's edge to SaaS to automobiles

Libraries

- CNI Plugins: crate/framework to write CNI (container networking) plugins in Rust (includes a few custom plugins as well)
- containers/libkrun: a dynamic library providing Virtualization-based process isolation capabilities
- kube-rs/kube-rs: Kubernetes Rust client and async controller runtime
- qovery/engine: Qovery Engine is an open-source abstraction layer library that turns easy apps deployment on AWS, GCP, Azure, and other Cloud providers in just a few minutes
- open-telemetry/opentelemetry-rust: OpenTelemetry is a set of APIs, SDKs, tooling and integrations that are designed for the creation and management of telemetry data such as traces, metrics, and logs.

- ...

CHINA OpenInfra Days

GCC for Rust

- https://rust-gcc.github.io/

GCC Front-End For Rust

This is a full alternative implementation of the Rust language on top of GCC with the goal to become fully upstream with the GNU toolchain.

As this is a front-end project, the compiler will gain full access to all of GCC's internal middle-end optimization passes which are distinct from LLVM. For example, users of this compiler can expect to use the familiar -O2 flags to tune GCC's optimizer. Going forward, we will be happy to see more LLVM vs GCC graphs in respect to compilation speed, resulting code size and performance.

The project is still in an early phase with the goal to compile the offical Rust test suite. There are no immediate plans for a borrow checker as this is not required to compile rust code and is the last pass in the RustC compiler. This can be handled as a separate project when we get to that point.

https://github.com/Rust-GCC

- https://github.com/rust-lang/rustc_codegen_gcc

This is a GCC codegen for rustc, which means it can be loaded by the existing rustc frontend, but benefits from GCC: more architectures are supported and GCC's optimizations are used.

Despite its name, libgccjit can be used for ahead-of-time compilation, as is used here.

was merged into upstream Rust

https://blog.antoyo.xyz/rustc_codegen_gcc-progress-report-10

- Rust frontend approved for GCC

https://lwn.net/Articles/900721/

- ...

1.1.1.2 MiniRust

- https://github.com/RalfJung/minirust

  A precise specification for "Rust lite / MIR plus

  MiniRust is the cornerstone of my vision for a normative specification of Rust semantics. It is an idealized MIR-like language with the purpose of serving as a "core language" of Rust. This is part of a larger story whose goal is to precisely specify the operational behavior of Rust, i.e., the possible behaviors that a Rust program might have when being executed: the behavior of a Rust program is defined by first translating it to MiniRust (which is outside the scope of this repository), and then considering the possible behaviors of the MiniRust program as specified in this document. That translation does a *lot* of work; for example, traits and pattern matching are basically gone on the level of MiniRust. On the other hand, MiniRust is concerned a lot with details such as the exact evaluation order, data representations, and precisely what is and is not Undefined Behavior.

  To separate the complexities of memory from the semantics of MiniRust statements and expressions, we introduce the MiniRust *memory interface*: think of memory as implementing some trait; MiniRust semantics is generic over the actual implementation of that trait. The interface between the MiniRust language (specified in `lang`) and its memory model (specified in `mem`) is *untyped and byte-oriented* (but "bytes" are a bit more complex than you might expect). For now, we only define the memory interface, but do not give an implementation. Even without deciding what exactly the final memory model will look like, we can answer a surprising amount of interesting questions about Rust semantics!

  On the MiniRust language side, the most important concept to understand is that of a *value* and how it relates to *types*. Values form a high-level, structural view of data (e.g. mathematical integers); types serve to relate values with their low-level byte-oriented representation. Types are essentially just parameters attached to certain operations to define the (de)serialization format. Well-formedness of a MiniRust program ensures that expressions and statements satisfy some basic typing discipline, but MiniRust is by design *not* type-safe.

- https://www.ralfj.de/blog/2022/08/08/minirust.html
- ...

Miri

- https://github.com/rust-lang/miri/

An interpreter for Rust's mid-level intermediate representation

An experimental interpreter for Rust's mid-level intermediate representation (MIR). It can run binaries and test suites of cargo projects and detect certain classes of undefined behavior, for example:

- Out-of-bounds memory accesses and use-after-free
- Invalid use of uninitialized data
- Violation of intrinsic preconditions (an `unreachable_unchecked` being reached, calling `copy_nonoverlapping` with overlapping ranges, ...)
- Not sufficiently aligned memory accesses and references
- Violation of *some* basic type invariants (a `bool` that is not 0 or 1, for example, or an invalid enum discriminant)
- **Experimental**: Violations of the Stacked Borrows rules governing aliasing for reference types
- **Experimental**: Data races

On top of that, Miri will also tell you about memory leaks: when there is memory still allocated at the end of the execution, and that memory is not reachable from a global `static`, Miri will raise an error.

Miri supports almost all Rust language features; in particular, unwinding and concurrency are properly supported (including some experimental emulation of weak memory effects, i.e., reads can return outdated values).

You can use Miri to emulate programs on other targets, e.g. to ensure that byte-level data manipulation works correctly both on little-endian and big-endian systems. See cross-interpretation below.

Cross-interpretation: running for different targets

Miri can not only run a binary or test suite for your host target, it can also perform cross-interpretation for arbitrary foreign targets: `cargo miri run --target x86_64-unknown-linux-gnu` will run your program as if it was a Linux program, no matter your host OS. This is particularly useful if you are using Windows, as the Linux target is much better supported than Windows targets.

You can also use this to test platforms with different properties than your host platform. For example `cargo miri test --target mips64-unknown-linux-gnuabi64` will run your test suite on a big-endian target, which is useful for testing endian-sensitive code.

- Languages

Roadmap

- https://blog.rust-lang.org/inside-rust/2022/04/04/lang-roadmap-2024.html

Our focus for Rust 2024 is to **scale empowerment** in many different ways. As we grow, we face increasing challenges in how we can scale the ways in which we empower people to an increasing number of people. This roadmap presents three general themes we plan to focus on:

- **Flatten the (learning) curve**: scaling to new users and new use cases
  - Make Rust more accessible to new and existing users alike, and make solving hard problems easier.
- **Help Rust's users help each other**: scaling the ecosystem
  - Empower library authors so they can---in turn---empower their users.
- **Help the Rust project scale**: scaling the project
  - Develop processes to scale to the needs and use cases of a growing number of users; evaluate and finish projects we've started.

…

- https://blog.rust-lang.org/inside-rust/2022/02/22/compiler-team-ambitions-2022.html
- …

1.1.3 Miscs

1.1.3.1 Carbon

- https://github.com/carbon-language/carbon-lang

  An experimental successor to C++

Fast and works with C++

- Performance matching C++ using LLVM, with low-level access to bits and addresses
- Interoperate with your existing C++ code, from inheritance to templates
- Fast and scalable builds that work with your existing C++ build systems

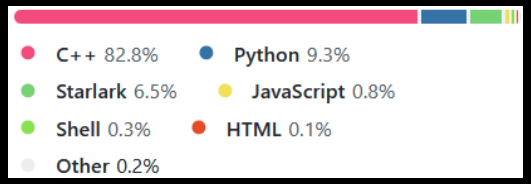
Modern and evolving

- Solid language foundations that are easy to learn, especially if you have used C++
- Easy, tool-based upgrades between Carbon versions
- Safer fundamentals, and an incremental path towards a memory-safe subset

Welcoming open-source community

- Clear goals and priorities with robust governance
- Community that works to be welcoming, inclusive, and friendly
- Batteries-included approach: compiler, libraries, docs, tools, package manager, and more

- Languages



- https://github.com/carbon-language/carbon-lang/blob/trunk/docs/project/roadmap.md

Why build Carbon?

C++ remains the dominant programming language for performance-critical software, with massive and growing codebases and investments. However, it is struggling to improve and meet developers' needs, as outlined above, in no small part due to accumulating decades of technical debt. Incrementally improving C++ is extremely difficult, both due to the technical debt itself and challenges with its evolution process. The best way to address these problems is to avoid inheriting the legacy of C or C++ directly, and instead start with solid language foundations like modern generics system, modular code organization, and consistent, simple syntax.

Existing modern languages already provide an excellent developer experience: Go, Swift, Kotlin, Rust, and many more. **Developers that *can* use one of these existing languages *should*.** Unfortunately, the designs of these languages present significant barriers to adoption and migration from C++. These barriers range from changes in the idiomatic design of software to performance overhead.

Carbon is fundamentally **a successor language approach**, rather than an attempt to incrementally evolve C++. It is designed around interoperability with C++ as well as large-scale adoption and migration for existing C++ codebases and developers. A successor language for C++ requires:

- **Performance matching C++**, an essential property for our developers.
- **Seamless, bidirectional interoperability with C++**, such that a library anywhere in an existing C++ stack can adopt Carbon without porting the rest.
- **A gentle learning curve** with reasonable familiarity for C++ developers.
- **Comparable expressivity** and support for existing software's design and architecture.
- **Scalable migration**, with some level of source-to-source translation for idiomatic C++ code.

With this approach, we can build on top of C++'s existing ecosystem, and bring along existing investments, codebases, and developer populations. There are a few languages that have followed this model for other ecosystems, and Carbon aims to fill an analogous role for C++:

- JavaScript → TypeScript
- Java → Kotlin
- C++ → ***Carbon***

Language Goals

We are designing Carbon to support:

- Performance-critical software
- Software and language evolution
- Code that is easy to read, understand, and write
- Practical safety and testing mechanisms
- Fast and scalable development
- Modern OS platforms, hardware architectures, and environments
- Interoperability with and migration from existing C++ code

While many languages share subsets of these goals, what distinguishes Carbon is their combination.

We also have explicit *non-goals* for Carbon, notably including:

- A stable application binary interface (ABI) for the entire language and library
- Perfect backwards or forwards compatibility

https://github.com/carbon-language/carbon-lang/blob/trunk/docs/project/goals.md

Generics

Carbon provides a **modern generics system** with checked definitions, while still **supporting opt-in templates for seamless C++ interop**. Checked generics provide several advantages compared to C++ templates:

- **Generic definitions are fully type-checked**, removing the need to instantiate to check for errors and giving greater confidence in code.
  - Avoids the compile-time cost of re-checking the definition for every instantiation.
  - When using a definition-checked generic, usage error messages are clearer, directly showing which requirements are not met.
- **Enables automatic, opt-in type erasure and dynamic dispatch** without a separate implementation. This can reduce the binary size and enables constructs like heterogeneous containers.
- **Strong, checked interfaces** mean fewer accidental dependencies on implementation details and a clearer contract for consumers.

Without sacrificing these advantages, **Carbon generics support specialization**, ensuring it can fully address performance-critical use cases of C++ templates. For more details about Carbon's generics, see their design.

In addition to easy and powerful interop with C++, Carbon templates can be constrained and incrementally migrated to checked generics at a fine granularity and with a smooth evolutionary path.

- **https://github.com/carbon-language/carbon-lang/blob/trunk/docs/design/generics**

Memory safety

Safety, and especially memory safety, remains a key challenge for C++ and something a successor language needs to address. Our initial priority and focus is on immediately addressing important, low-hanging fruit in the safety space:

- Tracking uninitialized states better, increased enforcement of initialization, and systematically providing hardening against initialization bugs when desired.
- Designing fundamental APIs and idioms to support dynamic bounds checks in debug and hardened builds.
- Having a default debug build mode that is both cheaper and more comprehensive than existing C++ build modes even when combined with Address Sanitizer.

Once we can migrate code into Carbon, we will have a simplified language with room in the design space to add any necessary annotations or features, and infrastructure like generics to support safer design patterns. Longer term, we will build on this to introduce **a safe Carbon subset**. This will be a large and complex undertaking, and won't be in the 0.1 design. Meanwhile, we are closely watching and learning from efforts to add memory safe semantics onto C++ such as Rust-inspired lifetime annotations.

- ...

1.1.3.2 Nim

- https://en.wikipedia.org/wiki/Nim_(programming_language)

Nim is a general-purpose, multi-paradigm, statically typed, compiled systems programming language,[9] designed and developed by a team around Andreas Rumpf. Nim is designed to be "efficient, expressive, and elegant",[10] supporting metaprogramming, functional, message passing,[7] procedural, and object-oriented programming styles by providing several features such as compile time code generation, algebraic data types, a foreign function interface (FFI) with C, C++, Objective-C, and JavaScript, and supporting compiling to those same languages.

Description [edit]

Nim was created to be a language as fast as C, as expressive as Python, and as extensible as Lisp.

Nim is statically typed.[11] It supports compile-time metaprogramming features such as syntactic macros and term rewriting macros.[12] Term rewriting macros enable library implementations of common data structures, such as bignums and matrices, to be implemented efficiently, as if they were built-in language facilities.[13] Iterators are supported and can be used as first class entities,[12] as can functions, allowing for the use of functional programming methods. Object-oriented programming is supported by inheritance and multiple dispatch. Functions can be generic, they can be overloaded, and generics are further enhanced by Nim's support for type classes. Operator overloading is also supported.[12] Nim includes tunable automatic garbage collection based on deferred reference counting with cycle detection, which can also be turned off altogether.[14]

> [Nim] ... presents a most original design that straddles Pascal and Python and compiles to C code or JavaScript.[15]
>
> — Andrew Binstock, editor-in-chief of Dr. Dobb's Journal, 2014

As of October 2021, Nim compiles to C, C++, JavaScript, and Objective-C.[16]

The Nim crown logo

Paradigms	Multi-paradigm: compiled, concurrent, procedural, imperative, functional, object-oriented, meta
Designed by	Andreas Rumpf
Developer	Nim Lang Team[1]
First appeared	2008; 14 years ago
Stable release	1.6.6[2] / 5 May 2022; 2 months ago
Typing discipline	Static,[3] strong,[4] inferred, structural
Scope	Lexical
Implementation language	Nim (self-hosted)
Platform	IA-32, x86-64, ARM, Aarch64, RISC-V, PowerPC ...[5]
OS	Cross-platform[6]
License	MIT[7][8]
Filename extensions	.nim, .nims, .nimble
Website	nim-lang.org
Influenced by	
Ada, Modula-3, Lisp, C++, Object Pascal, Python, Oberon, Rust	

Hello world [edit]

The "Hello, World!" program in Nim:

```
echo("Hello, world!")
# Procedures can be called with no parentheses
echo "Hello, World!"
```

Another version of making a "Hello World" is...

```
stdout.write("Hello, world!\n")
```

Factorial [edit]

Program to calculate the factorial of a positive integer using the iterative approach:

```
import strutils

var n = 0
try:
  stdout.write "Input positive integer number: "
  n = stdin.readline.parseInt
except ValueError:
  raise newException(ValueError, "You must enter a positive number")

var fact = 1
for i in 2..n:
  fact = fact * i

echo fact
```

Using the module math from Nim's standard library:

```
import math
echo fac(x)
```

- **https://nim-lang.org/**

Efficient, expressive, elegant

Nim is a statically typed compiled systems programming language. It combines successful concepts from mature languages like Python, Ada and Modula.

CHINA OpenInfra Days

Efficient

» Nim generates native dependency-free executables, not dependent on a virtual machine, which are small and allow easy redistribution.

» The Nim compiler and the generated executables support all major platforms like Windows, Linux, BSD and macOS.

» Nim's memory management is deterministic and customizable with destructors and move semantics, inspired by C++ and Rust. It is well-suited for embedded, hard-realtime systems.

» Modern concepts like zero-overhead iterators and compile-time evaluation of user-defined functions, in combination with the preference of value-based datatypes allocated on the stack, lead to extremely performant code.

» Support for various backends: it compiles to C, C++ or JavaScript so that Nim can be used for all backend and frontend needs.

Expressive

» Nim is self-contained: the compiler and the standard library are implemented in Nim.

» Nim has a powerful macro system which allows direct manipulation of the AST, offering nearly unlimited opportunities.

Elegant

» Macros cannot change Nim's syntax because there is no need for it — the syntax is flexible enough.

» Modern type system with local type inference, tuples, generics and sum types.

» Statements are grouped by indentation but can span multiple lines.

```
import std/strformat

type
  Person = object
    name: string
    age: Natural # Ensures the age is positive

let people = [
  Person(name: "John", age: 45),
  Person(name: "Kate", age: 30)
]

for person in people:
  # Type-safe string interpolation,
  # evaluated at compile time.
  echo(fmt"{person.name} is {person.age} years old")


# Thanks to Nim's 'iterator' and 'yield' constructs,
# iterators are as easy to write as ordinary
# functions. They are compiled to inline loops.
iterator oddNumbers[Idx, T](a: array[Idx, T]): T =
  for x in a:
    if x mod 2 == 1:
      yield x

for odd in oddNumbers([3, 6, 9, 12, 15, 18]):
  echo odd


# Use Nim's macro system to transform a dense
# data-centric description of x86 instructions
# into lookup tables that are used by
# assemblers and JITs.
import macros, strutils

macro toLookupTable(data: static[string]): untyped =
  result = newTree(nnkBracket)
  for w in data.split(';'):
    result.add newLit(w)

const
  data = "mov;btc;cli;xor"
  opcodes = toLookupTable(data)

for o in opcodes:
  echo o
```

1.1.3.3 Zig

- https://en.wikipedia.org/wiki/Zig_(programming_language)

Zig is an imperative, general-purpose, statically typed, compiled system programming language designed by Andrew Kelley.[3][4] The language is designed for "robustness, optimality and maintainability",[5][6] supporting compile-time generics, reflection and evaluation, cross-compilation and manual memory management.[7] A major goal of the language is to improve upon the C language,[8][9] while also taking inspiration from Rust,[10][11] among others. Zig has many features for low-level programming, notably: packed structs (structs without padding between fields), arbitrary width integers[12] and multiple pointer types.[13]

The stage 1 compiler is written in Zig and C++, using LLVM 13[14] as a back-end,[15][16] supporting many of its native targets.[17] The compiler is open source under the MIT License.[18] The Zig compiler exposes the ability to compile C and C++ similarly to Clang with the commands "zig cc" and "zig c++",[19] providing many headers including libc and libcxx for many different platforms, allowing Zig's cc and c++ sub-commands to act as cross compilers out of the box.[20][21]

Zig development is funded by the Zig Software Foundation (ZSF), a non-profit corporation with Andrew Kelley as president, which takes in donations and hires multiple full-time employees.[22][23][24]

ZIG

Paradigms	Multi-paradigm: imperative, concurrent, procedural, functional
Designed by	Andrew Kelley
First appeared	8 February 2016; 6 years ago[1]
Preview release	0.9.1[2] / 15 February 2022; 5 months ago
Typing discipline	Static, strong, inferred, structural, generic
Platform	x86-64, ARM, MIPS, IA-32, WebAssembly, RISC-V
OS	Cross-platform
License	MIT License
Filename extensions	.zig, .zir
Website	ziglang.org
Influenced by	
C, C++, LLVM IR, Go, Rust, JavaScript	

Hello World [edit]

```
const std = @import("std");

pub fn main() !void {
    const stdout = std.io.getStdOut().writer();
    try stdout.print("Hello, {}!\n", .{"world"});
}
```

Generic linked list [edit]

```
pub fn main() void {
    var node = LinkedList(i32).Node {
        .prev = null,
        .next = null,
        .data = 1234,
    };

    var list = LinkedList(i32) {
        .first = &node,
        .last = &node,
        .len = 1,
    };
}

fn LinkedList(comptime T: type) type {
    return struct {
        pub const Node = struct {
            prev: ?*Node,
            next: ?*Node,
            data: T,
        };

        first: ?*Node,
        last:  ?*Node,
        len:   usize,
    };
}
```

https://ziglang.org/

Zig is a general-purpose programming language and toolchain for maintaining **robust**, **optimal** and **reusable** software.

⚡ A Simple Language

Focus on debugging your application rather than debugging your programming language knowledge.

- No hidden control flow.
- No hidden memory allocations.
- No preprocessor, no macros.

⚡ Comptime

A fresh approach to metaprogramming based on compile-time code execution and lazy evaluation.

- Call any function at compile-time.
- Manipulate types as values without runtime overhead.
- Comptime emulates the target architecture.

⚡ Maintain it with Zig

Incrementally improve your C/C++/Zig codebase.

- Use Zig as a zero-dependency, drop-in C/C++ compiler that supports cross-compilation out-of-the-box.
- Leverage `zig build` to create a consistent development environment across all platforms.
- Add a Zig compilation unit to C/C++ projects; cross-language LTO is enabled by default.

```
const std = @import("std");
const json = std.json;
const payload =
    \\{
    \\    "vals": {
    \\        "testing": 1,
    \\        "production": 42
    \\    },
    \\    "uptime": 9999
    \\}
;
const Config = struct {
    vals: struct { testing: u8, production: u8 },
    uptime: u64,
};
const config = x: {
    var stream = json.TokenStream.init(payload);
    const res = json.parse(Config, &stream, .{});
    // Assert no error can occur since we are
    // parsing this JSON at comptime!
    break :x res catch unreachable;
};
pub fn main() !void {
    if (config.vals.production > 50) {
        @compileError("only up to 50 supported");
    }
    std.log.info("up={d}", .{config.uptime});
}
```

Project Bun

- https://bun.sh/

Bun is a fast all-in-one JavaScript runtime

Bundle, transpile, install and run JavaScript & TypeScript projects — all in Bun. Bun is a new JavaScript runtime with a native bundler, transpiler, task runner and npm client built-in.

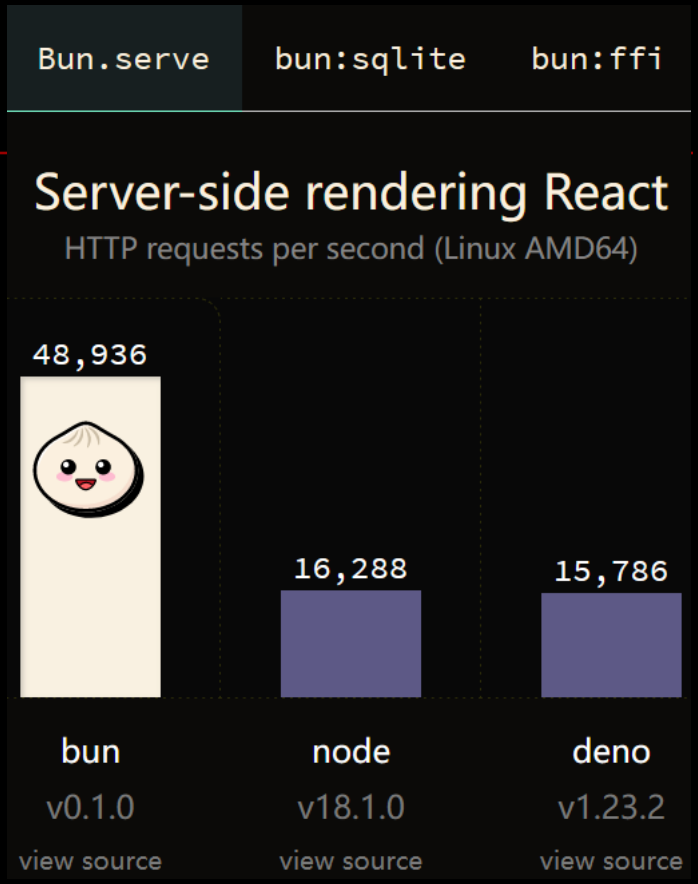
How does Bun work?

Bun.js uses the JavaScriptCore engine, which tends to start and perform a little faster than more traditional choices like V8. Bun is written in ZIG, a low-level programming language with manual memory management.

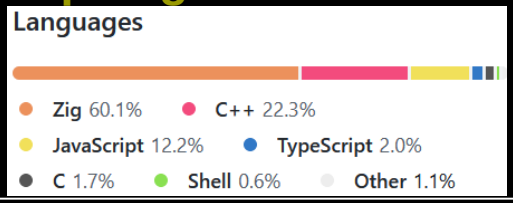
Most of Bun is written from scratch including the JSX/TypeScript transpiler, npm client, bundler, SQLite client, HTTP client, WebSocket client and more.

Why is Bun fast?

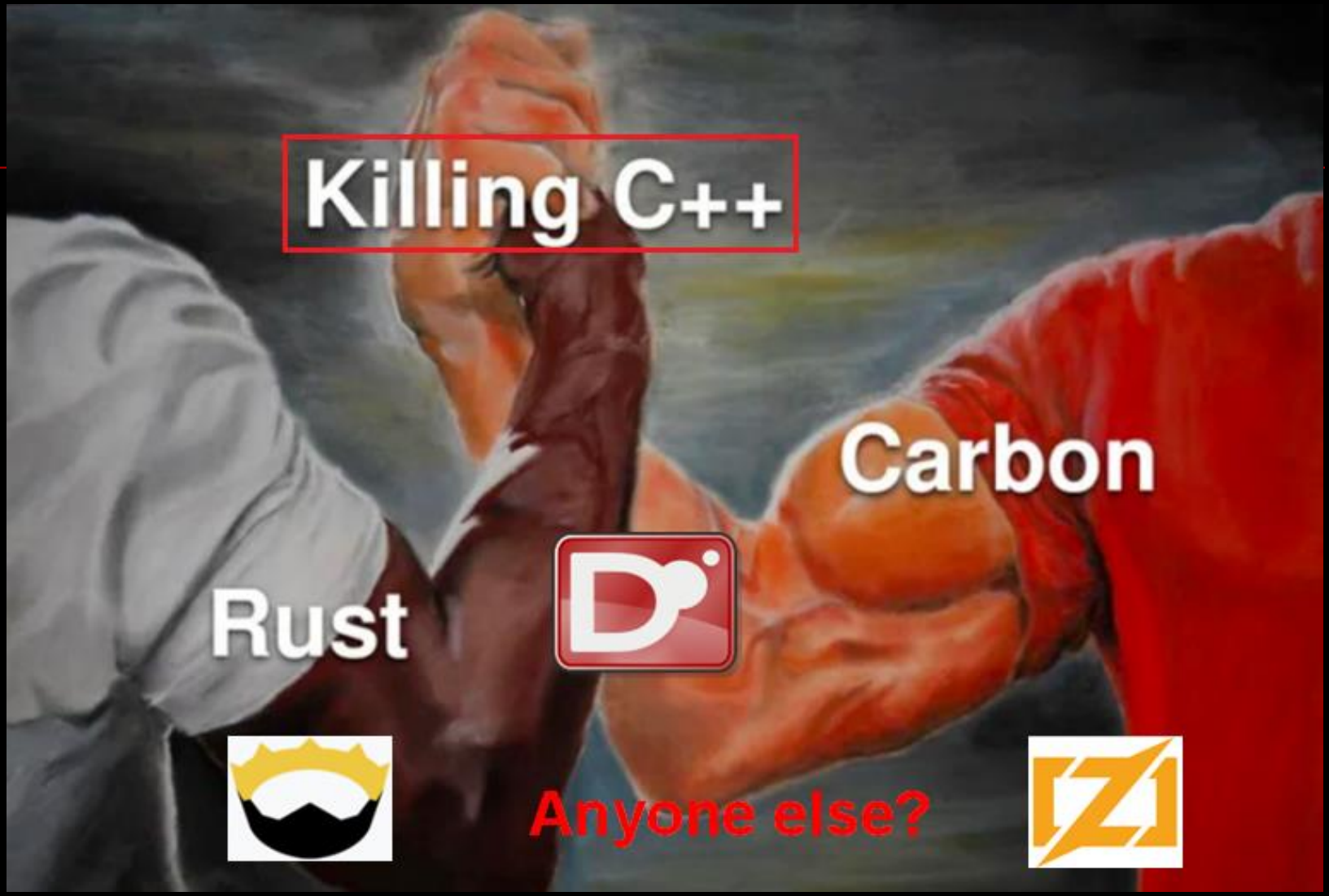
An enormous amount of time spent profiling, benchmarking and optimizing things. The answer is different for every part of Bun, but one general theme: ZIG's low-level control over memory and lack of hidden control flow makes it much simpler to write fast software. Sponsor the Zig Software Foundation.



- https://github.com/oven-sh/bun



1.1.3.4 Who is the murderer?



2) Tech Stack

- Native programming language
- Managed programming language
- Dynamic programming language

- …

Runtime

- https://en.wikipedia.org/wiki/Low-level_programming_language
- https://en.wikipedia.org/wiki/Assembly_language
- https://en.wikipedia.org/wiki/Inline_assembler
- https://en.wikipedia.org/wiki/Machine_code

Runtime

- https://en.wikipedia.org/wiki/Runtime_system
- https://en.wikipedia.org/wiki/Register_machine
- https://en.wikipedia.org/wiki/Stack_machine
- https://en.wikipedia.org/wiki/Intermediate_representation
- https://en.wikipedia.org/wiki/Bytecode
- https://en.wikipedia.org/wiki/Compiler
- https://en.wikipedia.org/wiki/Just-in-time_compilation
- https://en.wikipedia.org/wiki/Ahead-of-time_compilation
- https://en.wikipedia.org/wiki/Source-to-source_compiler
- https://en.wikipedia.org/wiki/Interpreter_(computing)
- https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)
- …
- https://en.wikipedia.org/wiki/Polyglot_(computing)

Polyglot — use the best tool for the right jobs: high performance, scripting, web, functional programming, etc

The most popular Polyglot Runtimes: GraalVM, .Net, Wasm…

2.1 Domain-Specific Language(DSL)

Overview

- https://en.wikipedia.org/wiki/Domain-specific_language

A **domain-specific language** (**DSL**) is a computer language specialized to a particular application domain. This is in contrast to a general-purpose language (GPL), which is broadly applicable across domains. There are a wide variety of DSLs, ranging from widely used languages for common domains, such as HTML for web pages, down to languages used by only one or a few pieces of software, such as MUSH soft code. DSLs can be further subdivided by the kind of language, and include domain-specific *markup* languages, domain-specific *modeling* languages (more generally, specification languages), and domain-specific *programming* languages. Special-purpose computer languages have always existed in the computer age, but the term "domain-specific language" has become more popular due to the rise of domain-specific modeling. Simpler DSLs, particularly ones used by a single application, are sometimes informally called **mini-languages**.

The line between general-purpose languages and domain-specific languages is not always sharp, as a language may have specialized features for a particular domain but be applicable more broadly, or conversely may in principle be capable of broad application but in practice used primarily for a specific domain. For example, Perl was originally developed as a text-processing and glue language, for the same domain as AWK and shell scripts, but was mostly used as a general-purpose programming language later on. By contrast, PostScript is a Turing-complete language, and in principle can be used for any task, but in practice is narrowly used as a page description language.

Domain-specific language topics [edit]

External and Embedded Domain Specific Languages [edit]

DSLs implemented via an independent interpreter or compiler are known as *External Domain Specific Languages*. Well known examples include LaTeX or AWK. A separate category known as *Embedded (or Internal) Domain Specific Languages* are typically implemented within a host language as a library and tend to be limited to the syntax of the host language, though this depends on host language capabilities.[1]

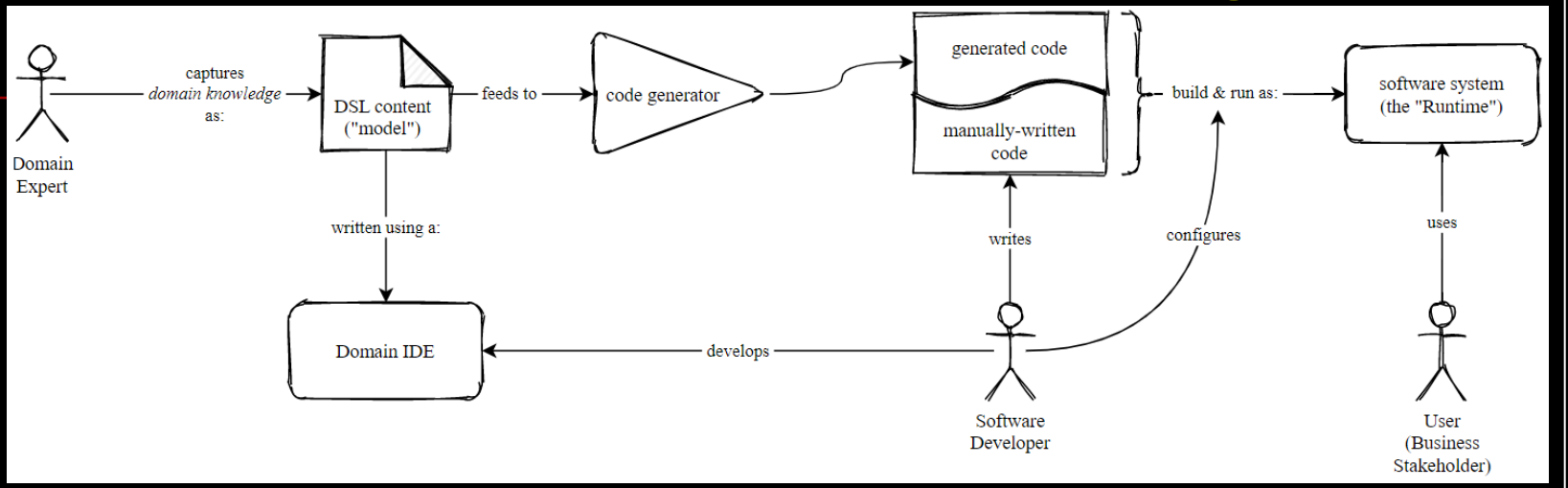
Usage patterns [edit]

There are several usage patterns for domain-specific languages:[2][3]

- Processing with standalone tools, invoked via direct user operation, often on the command line or from a Makefile (e.g., grep for regular expression matching, sed, lex, yacc, the GraphViz toolset, etc.)
- Domain-specific languages which are implemented using programming language macro systems, and which are converted or expanded into a host general purpose language at compile-time or realtime
- **embedded domain-specific language** (**eDSL**),[4] implemented as libraries which exploit the syntax of their host general purpose language or a subset thereof while adding domain-specific language elements (data types, routines, methods, macros etc.). (e.g. jQuery, React, Embedded SQL, LINQ)
- Domain-specific languages which are called (at runtime) from programs written in general purpose languages like C or Perl, to perform a specific function, often returning the results of operation to the "host" programming language for further processing; generally, an interpreter or virtual machine for the domain-specific language is embedded into the host application (e.g. format strings, a regular expression engine)
- Domain-specific languages which are embedded into user applications (e.g., macro languages within spreadsheets) and which are (1) used to execute code that is written by users of the application, (2) dynamically generated by the application, or (3) both.

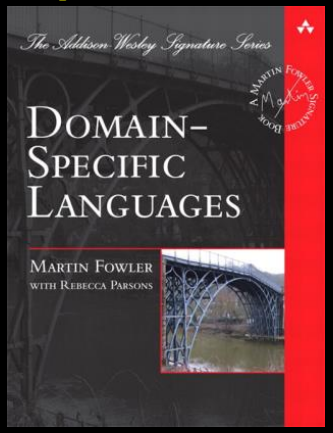
Many domain-specific languages can be used in more than one way.[*citation needed*] DSL code embedded in a host language may have special syntax support, such as regexes in sed, AWK, Perl or JavaScript, or may be passed as strings.

- https://www.jetbrains.com/mps/concepts/domain-specific-languages/
- The DSL-based (“model-driven”) approach to developing SW systems



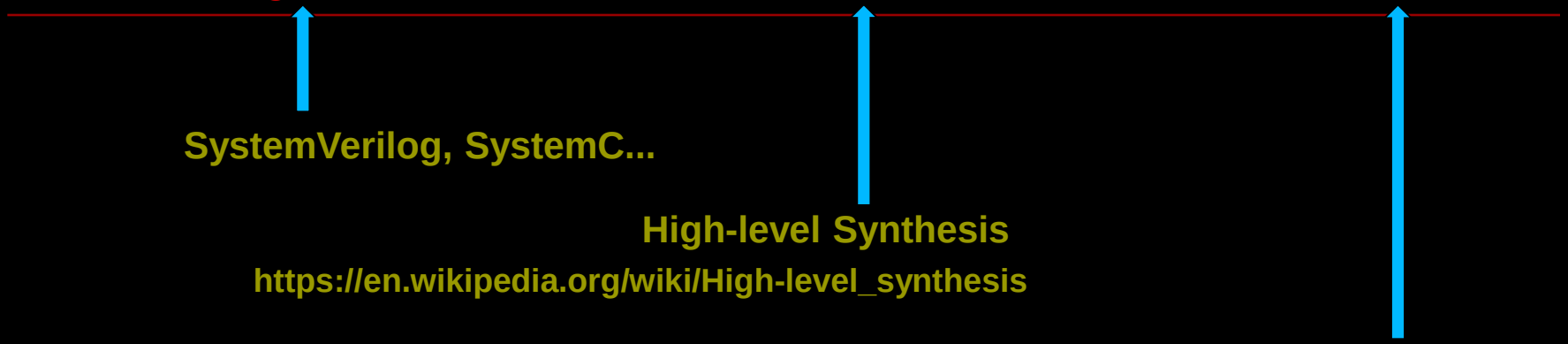
Source: https://freecontent.manning.com/the-what-and-why-of-domain-specific-lanugages/

- https://martinfowler.com/dsl.html



2.2 Evolution of HDLs

- https://en.wikipedia.org/wiki/Hardware_description_language
- https://hdl.github.io/awesome/items/
- Verilog/VHDL → HLS → eDSL



Typical eDSLs

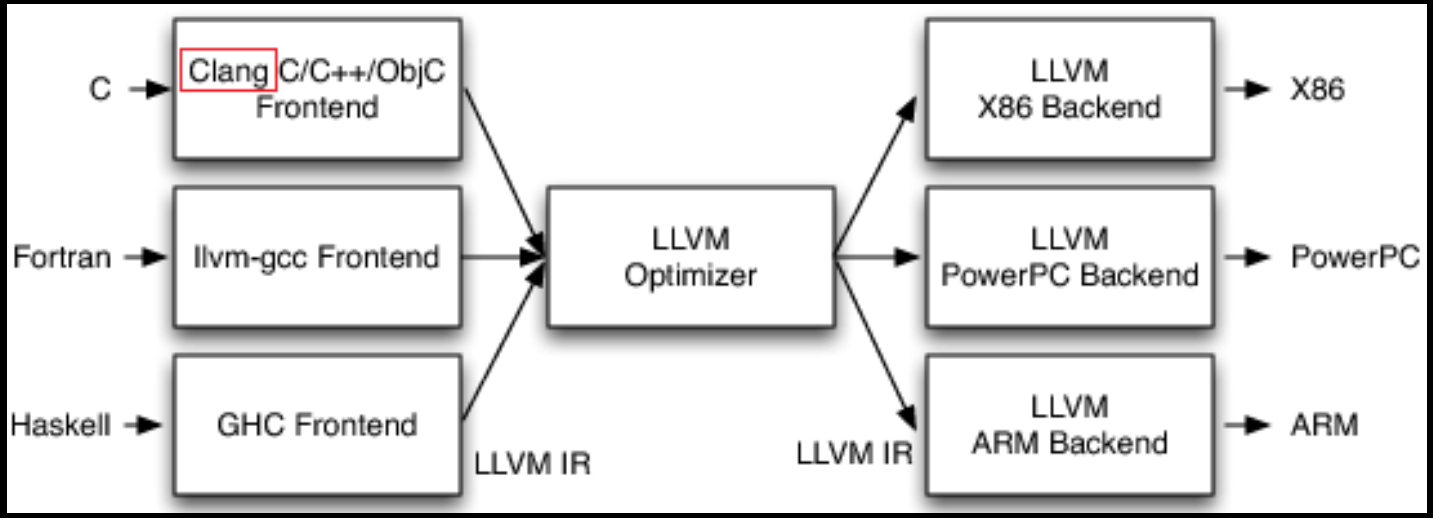
- Haskell as host
  Bluespec, Clash...
- Scala as host
  Chisel, SpinalHDL...
- Python as host
  Amaranth, FHDL, PyGears...
- Finally convert to Verilog or VHDL
  https://en.wikipedia.org/wiki/Source-to-source_compiler

2.3 LLVM

- https://en.wikipedia.org/wiki/LLVM

LLVM is a set of compiler and toolchain technologies,[5] which can be used to develop a front end for any programming language and a back end for any instruction set architecture. LLVM is designed around a language-independent intermediate representation (IR) that serves as a portable, high-level assembly language that can be optimized with a variety of transformations over multiple passes.[6]

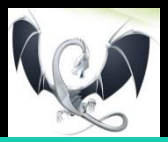
- https://llvm.org
- LLVM's Implementation of the Three-Phase Design



Source: http://www.aosabook.org/en/llvm.html

- http://clang.llvm.org/
- https://llvm.org/docs/
- https://www.llvm.org/ProjectsWithLLVM/
- …

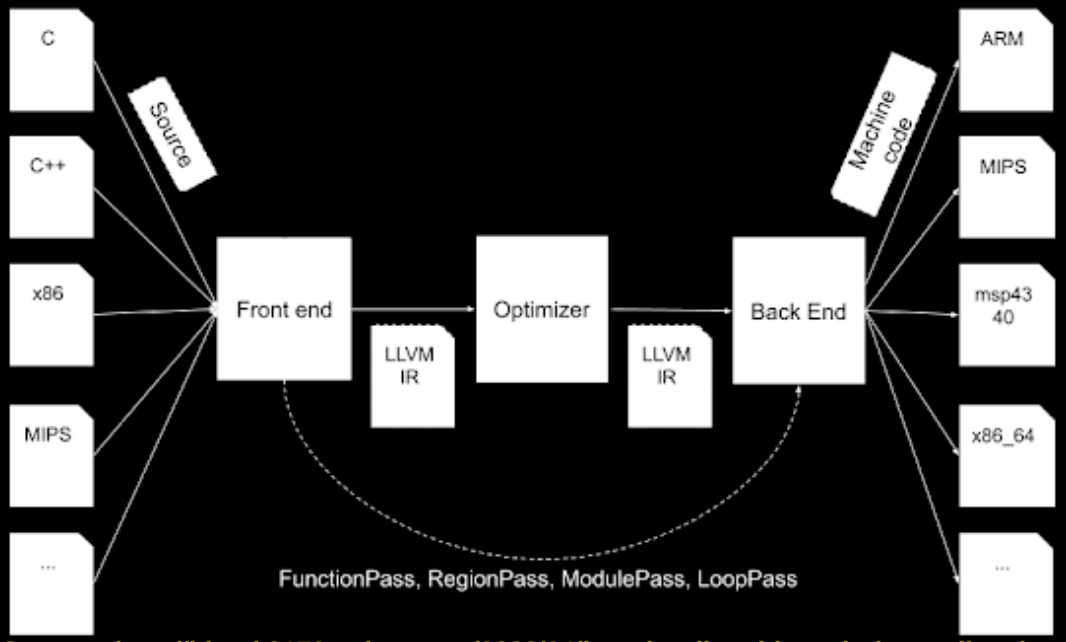
LLVM vs GCC



GPL v3	UIUC, MIT
Front-end: CC1 / CPP	Front-end: Clang
ld.bfd / ld.gold	lld / mclinker
gdb	lldb
as / objdump	MC layer
glibc	llvm-libc?
libstdc++	libc++
libsupc++	libc++abi
libgcc	libcompiler-rt
libgccjit	libLLVMMCJIT
. . .	ORC JIT, Coroutines, Clangd, libclc, Falcon...

- http://lld.llvm.org/
- https://llvm.org/docs/Proposals/LLVMLibC.html

2.3.1 LLVM IR



Source: http://blog.k3170makan.com/2020/04/learning-llvm-i-introdution-to-llvm.html

The core of LLVM is the intermediate representation (IR), a low-level programming language similar to assembly. IR is a strongly typed reduced instruction set computing (RISC) instruction set which abstracts away most details of the target. For example, the calling convention is abstracted through *call* and *ret* instructions with explicit arguments. Also, instead of a fixed set of registers, IR uses an infinite set of temporaries of the form %0, %1, etc. LLVM supports three equivalent forms of IR: a human-readable assembly format, an in-memory format suitable for frontends, and a dense bitcode format for serializing. A simple "Hello, world!" program in the IR format:[32]

```
@.str = internal constant [14 x i8] c"hello, world\0A\00"

declare i32 @printf(i8*, ...)

define i32 @main(i32 %argc, i8** %argv) nounwind {
entry:
    %tmp1 = getelementptr [14 x i8], [14 x i8]* @.str, i32 0, i32 0
    %tmp2 = call i32 (i8*, ...) @printf( i8* %tmp1 ) nounwind
    ret i32 0
}
```

- https://llvm.org/docs/LangRef.html
- Bitcode (https://llvm.org/docs/BitCodeFormat.html)

MLIR

- **https://mlir.llvm.org/**
  Multi-Level Intermediate Representation

The MLIR project is a novel approach to building reusable and extensible compiler infrastructure. MLIR aims to address software fragmentation, improve compilation for heterogeneous hardware, significantly reduce the cost of building domain specific compilers, and aid in connecting existing compilers together.

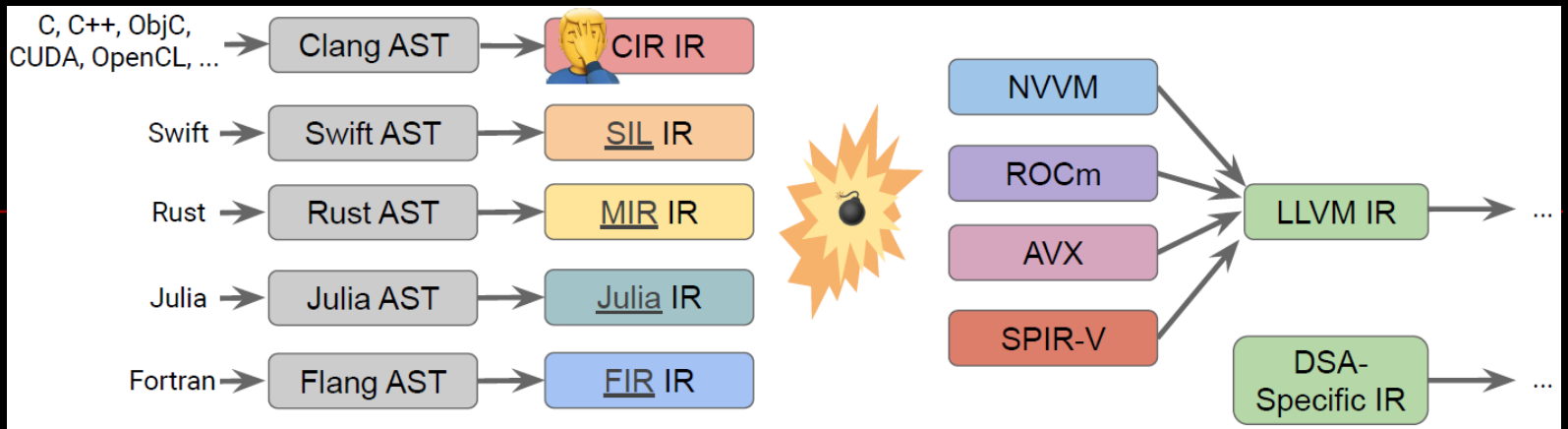
- **Motivation**

MLIR is intended to be a hybrid IR which can support multiple different requirements in a unified infrastructure. For example, this includes:

- The ability to represent dataflow graphs (such as in TensorFlow), including dynamic shapes, the user-extensible op ecosystem, TensorFlow variables, etc.
- Optimizations and transformations typically done on such graphs (e.g. in Grappler).
- Ability to host high-performance-computing-style loop optimizations across kernels (fusion, loop interchange, tiling, etc.), and to transform memory layouts of data.
- Code generation “lowering” transformations such as DMA insertion, explicit cache management, memory tiling, and vectorization for 1D and 2D register architectures.
- Ability to represent target-specific operations, e.g. accelerator-specific high-level operations.
- Quantization and other graph transformations done on a Deep-Learning graph.
- Polyhedral primitives.
- Hardware Synthesis Tools / HLS.

- **...**

From LLVM to MLIR

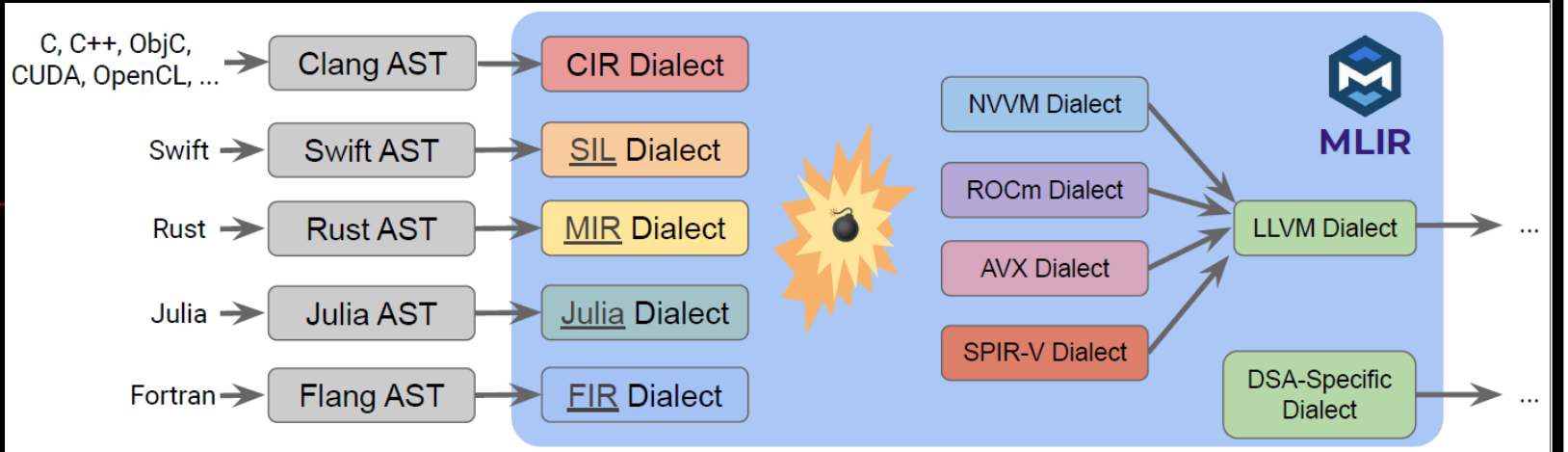


- Different back-ends demand customized IR for optimization
- DSAs even cannot use LLVM for generating back-end codes and demand their own IR for code generation

Severe Fragmentation: IRs have different implementations and "frameworks"

Source: https://hanchenye.com/assets/pdf/Gatech_CIRCT_Slides_Hanchen.pdf

MLIR: “Meta IR” and Compiler Infrastructure



MLIR is a **“Meta IR”** and **compiler infrastructure** for:

- Design and implement **dialect**
- Optimization and transform inside of a **dialect**
- Conversion between different **dialects**
- Code generation of **dialect**

Source: https://hanchenye.com/assets/pdf/Gatech_CIRCT_Slides_Hanchen.pdf

CIRCT

- https://circt.llvm.org/
  Circuit IR Compilers and Tools.
- Motivation

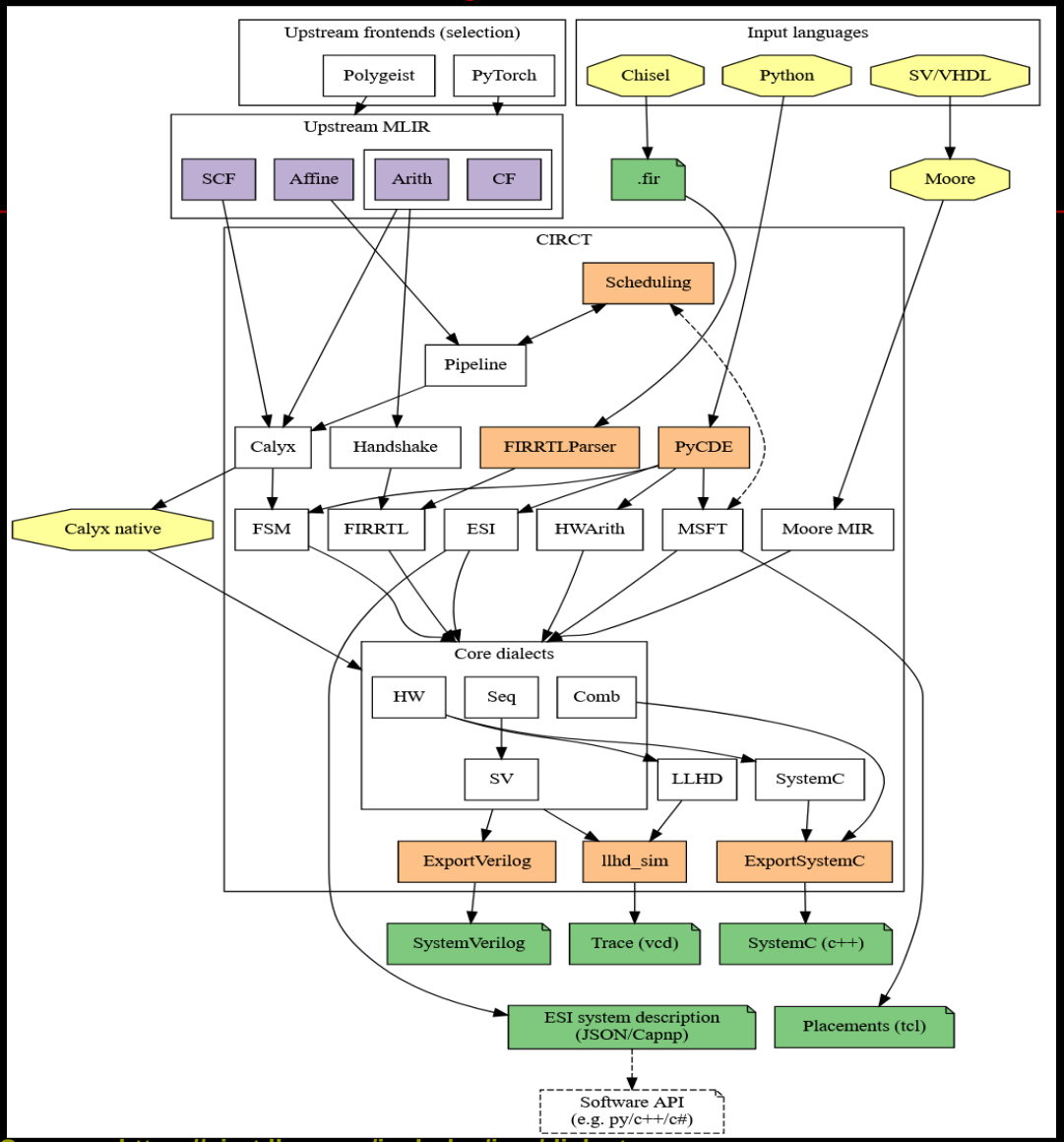
The EDA industry has well-known and widely used proprietary and open source tools. However, these tools are inconsistent, have usability concerns, and were not designed together into a common platform. Furthermore these tools are generally built with Verilog (also VHDL) as the IRs that they interchange. Verilog has well known design issues, and limitations, e.g. suffering from poor location tracking support.

The CIRCT project is an (experimental!) effort looking to apply MLIR and the LLVM development methodology to the domain of hardware design tools. Many of us dream of having reusable infrastructure that is modular, uses library-based design techniques, is more consistent, and builds on the best practices in compiler infrastructure and compiler design techniques.

By working together, we hope that we can build a new center of gravity to draw contributions from the small (but enthusiastic!) community of people who work on open hardware tooling. In turn we hope this will propel open tools forward, enables new higher-level abstractions for hardware design, and perhaps some pieces may even be adopted by proprietary tools in time.

- https://circt.llvm.org/docs/Charter/
- https://mlir.llvm.org/
- https://circt.org/perf/
- …

dialects and how they interact



Source: https://circt.llvm.org/includes/img/dialects.svg

ClangIR (CIR)

- https://clangir.org/
  A new IR for Clang.
- https://discourse.llvm.org/t/rfc-an-mlir-based-clang-ir-cir/63319

Hello Clang and MLIR folks, this RFC proposes CIR 176, a new IR for Clang.

TL;DR — We have been working on an MLIR based IR for Clang, currently called CIR (ClangIR, C/C++ IR, name-it). It's open source 212 by inception and we'd love to upstream it sooner rather than later. Our current (and initial) goal is to provide a framework for improved diagnostics for modern C++, meaning better support for coroutines and checks for idiomatic uses of known C++ libraries. Design has grown out of implementing a lifetime analysis/checker 26 pass for CIR 33, based on the C++ lifetime safety paper 99. C++ high level optimizations and lowering to LLVM IR are highly desirable but are a secondary goal for now - unless, of course, we get early traction and interested members in the community to help :).

Motivation

In general, Clang's AST is not an appropriate representation for dataflow analysis and reasoning about control flow. On the other hand, LLVM IR is too low level — it exists at a point in which we have already lost vital language information (e.g. scope information, loop forms and type hierarchies are invisible at the LLVM level), forcing a pass writer to attempt reconstruction of the original semantics. This leads to inaccurate results and inefficient analysis - not to mention the Sisyphean maintenance work given how fast LLVM changes. Clang's CFG is supposed to bridge this gap but isn't ideal either: a parallel lowering path for dataflow diagnostics that (a) is discarded after analysis, (b) has lots of known problems (checkout Kristóf Uman's great survey 39 regarding "dataflowness") and (c) has testing coverage for CFG pieces not quite up to LLVM's standards.

We also have the prominent recent success stories of Swift's SIL and Rust's HIR and MIR. These two projects have leveraged high level IRs to improve their performance and safety. We believe CIR could provide the same improvements for C++.

- https://llvm.github.io/clangir/
- https://github.com/llvm/clangir
- ...

2.4 RISC-V

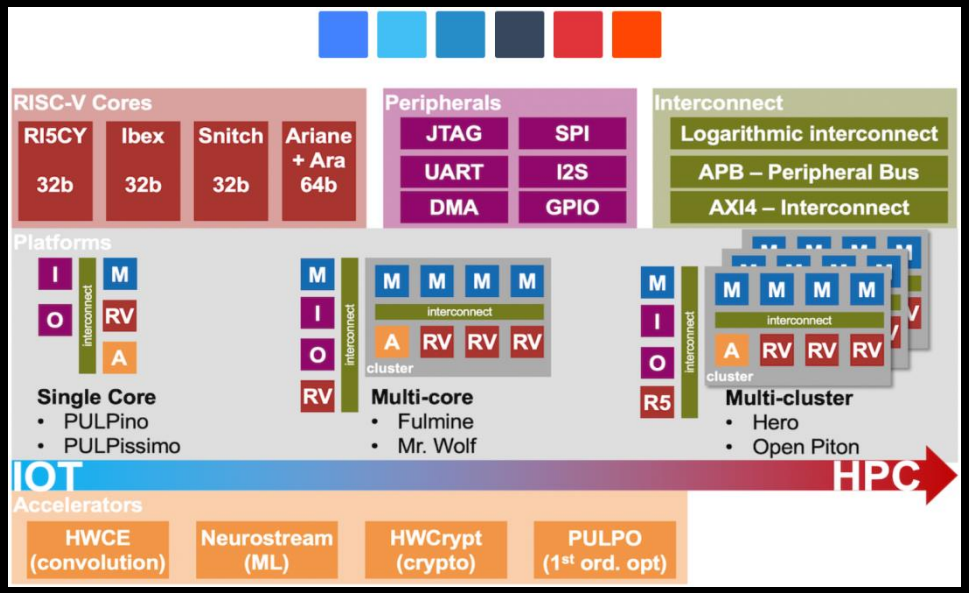
- https://en.wikipedia.org/wiki/RISC-V
- https://riscv.org/

Development

- https://riscv.org/exchange/
- https://github.com/riscv/riscv-isa-manual

Cores & SoC

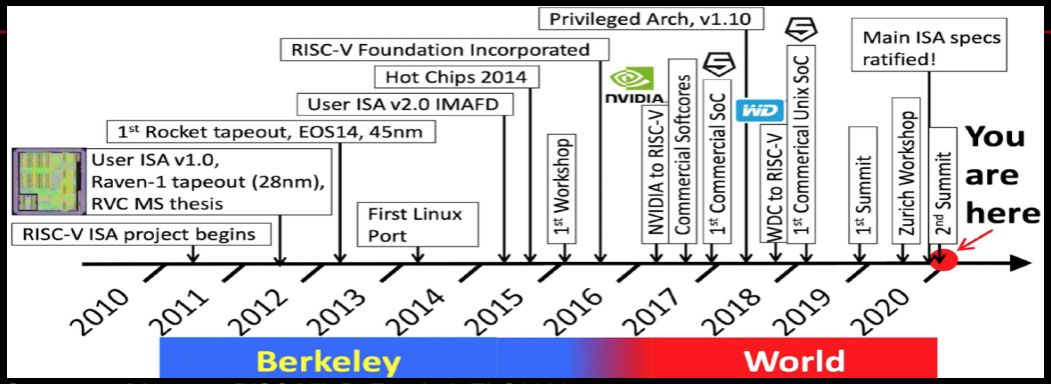
- https://github.com/riscv/riscv-cores-list
- https://riscv.org/exchange/cores-socs/
-



Source: https://fuentitech.com/what-is-the-risc-v-ecosystem/19227/

Ecosystem

- https://riscv.org/exchange/
- https://community.riscv.org/events/details/risc-v-foundation-bay-area-risc-v-group-presents-2021-risc-v-ecosystem-updates/
-



Source: "Linux on RISC-V", D. Fustimi, ELC2020

- **Incredible industry progress**

- The European Processor Initiative finalized the first version of its **RISC-V accelerator architecture** and will deliver test chip in 2021.
- The RIOS Lab announced PicoRio, an affordable **RISC-V open source small-board computer** available in 2021.
- Imperas announced first **RISC-V verification reference model with UVM encapsulation**.
- Seagate announced **hard disk drive controller** with high-performance RISC-V CPU.
- GreenWaves **ultra-low power GAP9 hearables platform** enabling scene-aware and neural network-based noise reduction.
- Alibaba unveiled RV64GCV core in its Xuantie 910 processor for **cloud and edge servers**.
- Microchip released the first **SoC FPGA development kit** based on the RISC-V ISA.
- Andes released **superscalar multicore and L2 cache controller** processors.
- StarFive released the world's first **RISC-V AI visual processing platform**
- SiFive unveiled world's fastest development board for **RISC-V Personal Computers**.
- Micro Magic announced an incredibly **fast 64-bit RISC-V core** achieving 5GHz and 13,000 CoreMarks at 1.1V.

Source: "RISC-V: The Open Era of Computing", Calista Redmond, The Linux Foundation Spring Member Meeting 2021.

A New Golden Age for Computer Architecture

- https://cacm.acm.org/magazines/2019/2/234352-a-new-golden-age-for-computer-architecture/fulltext

CHINA OpenInfra Days

“the Linux of the chip world”

- https://www.zdnet.com/article/risc-v-the-linux-of-the-chip-world-is-starting-to-produce-technological-breakthroughs/

2.5 HW/SW Co-design

University of Victoria

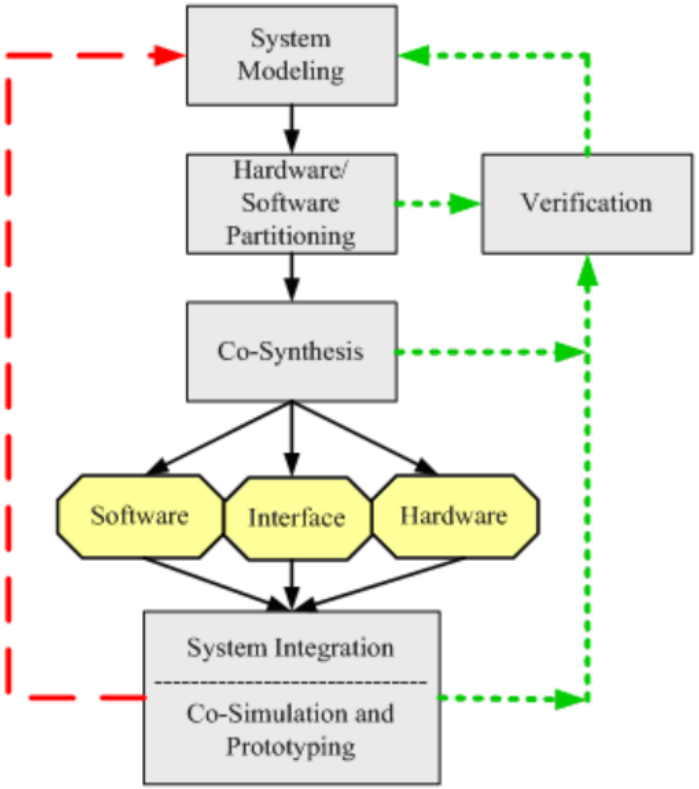
- https://webhome.cs.uvic.ca/~mserra/HScodesign.html
-

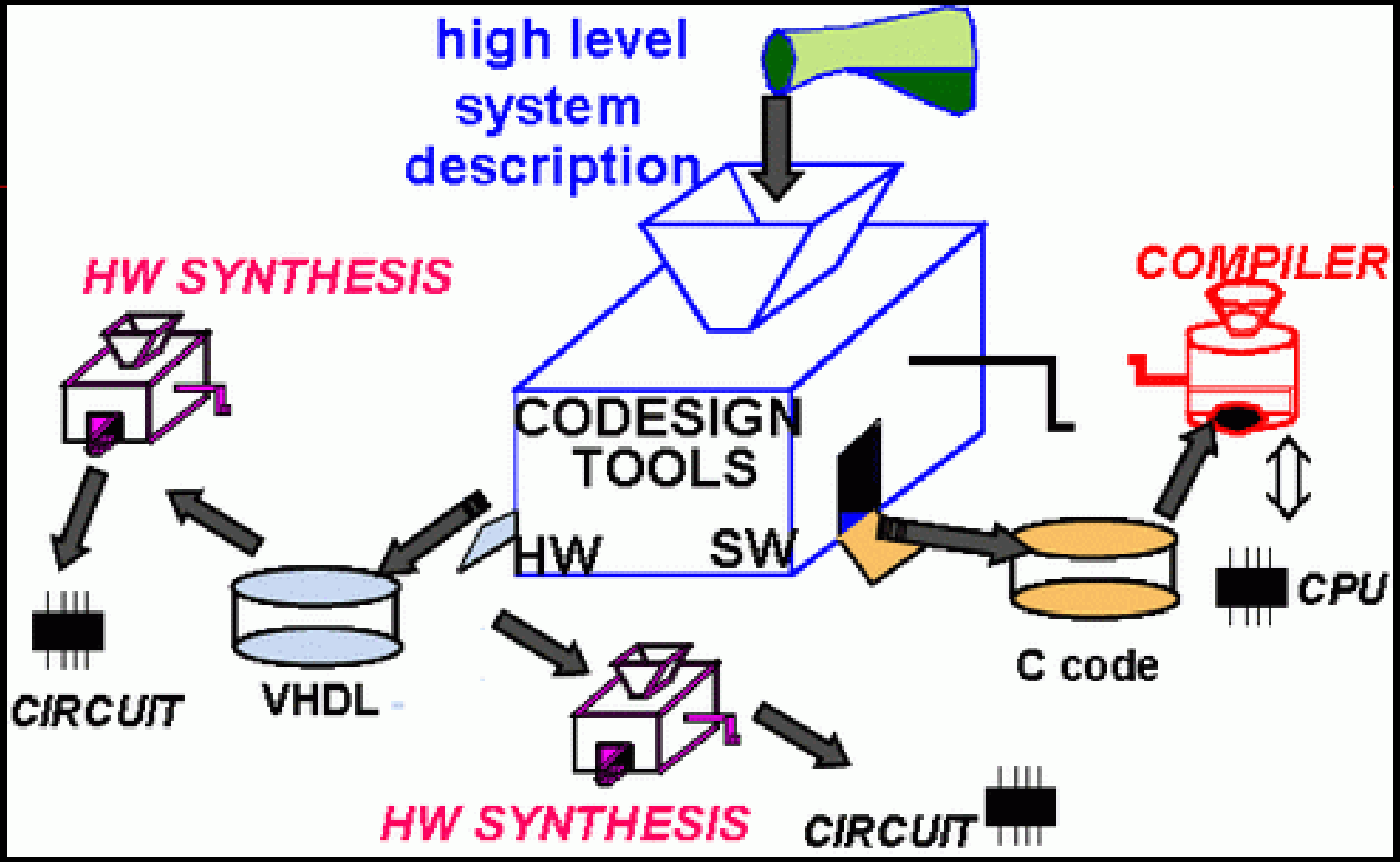
The co-design of HW/SW systems may be viewed as composed of four main phases as illustrated in in the side diagram:

1. Modeling
2. Partitioning
3. Co-Synthesis
4. C-Simulation

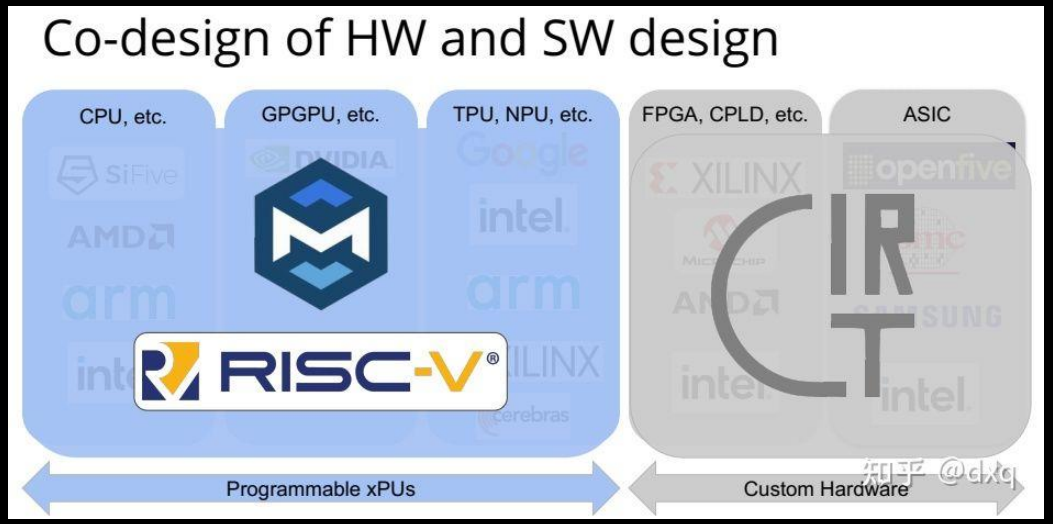
Modeling involves specifying the concepts and the constraints of the system to obtain a refined specification. This phase of the design also specifies a software and hardware model. The first problem is to find a suitable specification methodology for a target system. Some researchers favour a formal language which can yield provably-correct code. But most prefer a hardware-type language (e.g., VHDL, Verilog), a software-type language (C, C++, Handel-C, SystemC), or other formalism lacking a hardware or software bias (such as Codesign Finite State Machines). There are three different paths the modeling process can take, considering its starting point:

- An existing software implementation of the problem.
- An existing hardware, for example a chip, is present.
- None of the above is given, only specifications leaving an open choice for a model.





The Golden Age of Compiler Design in an Era of HW/SW Co-design



- https://zhuanlan.zhihu.com/p/367035973 //看Chris Lattner在ASPLOS演讲有感

II. Design philosophy

1) Overview

- https://en.wikipedia.org/wiki/D_(programming_language)

D, also known as **dlang**, is a multi-paradigm system programming language created by Walter Bright at Digital Mars and released in 2001. Andrei Alexandrescu joined the design and development effort in 2007. Though it originated as a re-engineering of C++, D is a profoundly different language —features of D can be considered streamlined and expanded-upon ideas from C++,[10] however D also draws inspiration from other high-level programming languages, notably Java, Python, Ruby, C#, and Eiffel.

D combines the performance and safety of compiled languages with the expressive power of modern dynamic and functional[11] programming languages. Idiomatic D code is commonly as fast as equivalent C++ code, while also being shorter.[12] The language as a whole is not memory-safe[13] but includes optional attributes designed to guarantee memory safety of either subsets of or the whole program.[14]

Type inference, automatic memory management and syntactic sugar for common types allow faster development, while bounds checking, design by contract find bugs earlier at runtime and a concurrency-aware type system catches bugs at compile time.[15]

Hello World

```d
import std.stdio;

void main()
{
    writeln("Hello, world!");
}
```

```d
// Sort lines
import std.stdio, std.array, std.algorithm;

void main()
{
    stdin
        .byLineCopy
        .array
        .sort!((a, b) => a > b) // descending order
        .each!writeln;
}
```

Count frequencies of character pairs — your code here

```d
void main()
{
    import std.stdio : writefln;
    // An associative array mapping pairs of characters to integers
    int[char[2]] aa;
    auto arr = "ABBBA";

    // Iterate over all pairs in the string
    // ['A', 'B'], ['B', 'B'], ..., ['B', 'A']
    foreach (i; 0 .. arr.length - 1)
    {
        // String slicing doesn't allocate a copy
        char[2] pair = arr[i .. i + 2];
        // count occurrences
        aa[pair]++;
    }
    foreach (key, value; aa)
        writefln("key: %s, value: %d", key, value);
}
```

Paradigm	Multi-paradigm: functional, imperative, object-oriented
Designed by	Walter Bright, Andrei Alexandrescu (since 2007)
Developer	D Language Foundation
First appeared	8 December 2001; 20 years ago[1]
Stable release	2.099.1[2] / 7 April 2022; 3 months ago
Typing discipline	Inferred, static, strong
OS	FreeBSD, Linux, macOS, Windows
License	Boost[3][4][5]
Filename extensions	.d[6][7]
Website	dlang.org
Major implementations	
DMD (reference implementation), GCC, GDC, LDC, SDC	
Influenced by	
C, C++, C#, Eiffel,[8] Java, Python	
Influenced	
Genie, MiniD, Qore, Swift,[9] Vala, C++11, C++14, C++17, C++20, Go, C#, and others.	

- https://dlang.org/spec/spec.html
- https://en.wikibooks.org/wiki/D_Programming

Major Design Goals

Everything in designing a language is a tradeoff. Keeping some principles in mind will help to make the right decisions.

1. Enable writing fast, effective code in a straightforward manner.
2. Make it easier to write code that is portable from compiler to compiler, machine to machine, and operating system to operating system. Eliminate undefined and implementation defined behaviors as much as practical.
3. Provide syntactic and semantic constructs that eliminate or at least reduce common mistakes. Reduce or even eliminate the need for third party static code checkers.
4. Support memory safe programming.
5. Support multi-paradigm programming, i.e. at a minimum support imperative, structured, object oriented, generic and even functional programming paradigms.
6. Make doing things the right way easier than the wrong way.
7. Have a short learning curve for programmers comfortable with programming in C, C++ or Java.
8. Provide low level bare metal access as required. Provide a means for the advanced programmer to escape checking as necessary.
9. Be compatible with the local C application binary interface.
10. Where D code looks the same as C code, have it either behave the same or issue an error.
11. Have a context-free grammar. Successful parsing must not require semantic analysis.
12. Easily support writing internationalized applications - Unicode everywhere.
13. Incorporate Contract Programming and unit testing methodology.
14. Be able to build lightweight, standalone programs.
15. Reduce the costs of creating documentation.
16. Provide sufficient semantics to enable advances in compiler optimization technology.
17. Cater to the needs of numerical analysis programmers.
18. Obviously, sometimes these goals will conflict. Resolution will be in favor of usability.

Source: https://dlang.org/overview.html

Designed by Experts

- https://en.wikipedia.org/wiki/Walter_Bright
  http://digitalmars.com/

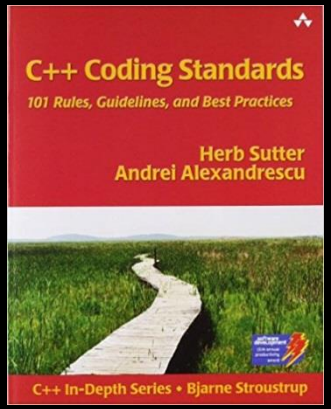
Digital Mars D compiler
Digital Mars C compiler
Digital Mars C++ compiler

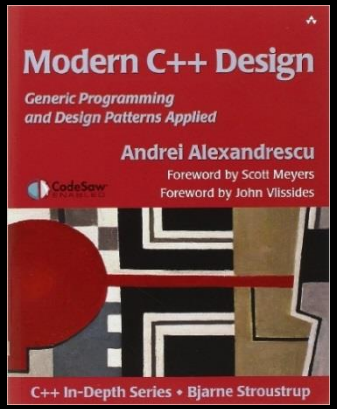
D

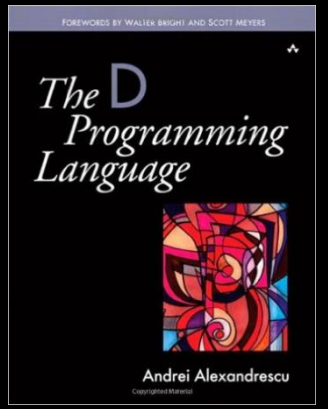
Programming Language

Specification

- https://en.wikipedia.org/wiki/Andrei_Alexandrescu
  http://erdani.org/







Ranking

- https://www.tiobe.com/tiobe-index/ (Jul 2022)

Jul 2022	Jul 2021	Change	Programming Language	Ratings	Change
1	3	↑	Python	13.44%	+2.48%
2	1	↓	C	13.13%	+1.50%
3	2	↓	Java	11.59%	+0.40%
4	4		C++	10.00%	+1.98%
5	5		C#	5.65%	+0.82%
6	6		Visual Basic	4.97%	+0.47%
7	7		JavaScript	1.78%	-0.93%
8	9	↑	Assembly language	1.65%	-0.76%
9	10	↑	SQL	1.64%	+0.11%
10	16	⇈	Swift	1.27%	+0.20%
11	8	↓	PHP	1.20%	-1.38%
12	13	↑	Go	1.14%	-0.03%

...

24	Lua	0.55%
25	Prolog	0.54%
26	COBOL	0.53%
27	Julia	0.49%
28	D	0.43%
29	Rust	0.42%
30	Ada	0.38%
31	Lisp	0.33%
32	Dart	0.31%
33	Haskell	0.29%
34	Scala	0.28%
35	Kotlin	0.28%

- http://redmonk.com (Jan 2022)

https://redmonk.com/rstephens/2022/03/28/top-20-jan-2022/

…

2) Features

D was designed with lessons learned from practical C++ usage, rather than from a purely theoretical perspective. Although the language uses many C and C++ concepts, it also discards some, or uses different approaches (and syntax) to achieve some goals. As such, it is not source compatible (nor does it aim to be) with C and C++ source code in general (some simpler code bases from these languages might by luck work with D, or require some porting). D has, however, been constrained in its design by the rule that any code that was legal in both C and D should behave in the same way.

D gained some features before C++, such as closures, anonymous functions, compile-time function execution, ranges, built-in container iteration concepts and type inference. D adds to the functionality of C++ by also implementing design by contract, unit testing, true modules, garbage collection, first class arrays, associative arrays, dynamic arrays, array slicing, nested functions, lazy evaluation, scoped (deferred) code execution, and a re-engineered template syntax. C++ multiple inheritance was replaced by Java-style single inheritance with interfaces and mixins. On the other hand, D's declaration, statement and expression syntax closely matches that of C++.

D retains C++'s ability to perform low-level programming including inline assembler, which typifies the differences between D and application languages like Java and C#. Inline assembler lets programmers enter machine-specific assembly code within standard D code, a method used by system programmers to access the low-level features of the processor needed to run programs that interface directly with the underlying hardware, such as operating systems and device drivers, as well as writing high-performance code (i.e. using vector extensions, SIMD) that is hard to generate by the compiler automatically.

D supports function overloading and operator overloading, as well as dynamic arrays and associative arrays by default. Symbols (functions, variables, classes) can be declared in any order - forward declarations are not required. Similarly imports can be done almost in any order, and even be scoped (i.e. import some module or part of it inside a function, class or unittest only). D has built-in support for documentation comments, allowing automatic documentation generation.

In D, text character strings are just arrays of characters, and arrays in D are bounds-checked, unlike those in C++.[16] Specific operators for string handling exist, visually distinct from mathematical corellaries. D has first class types for complex and imaginary numbers, and evaluates expressions involving such types, efficiently.[17]

Source: https://en.wikipedia.org/wiki/D_(programming_language)

Features To Leave Behind

- Support for 16 bit computers. No consideration is given in D for mixed near/far pointers and all the machinations necessary to generate good 16 bit code. The D language design assumes at least a 32 bit flat memory space and supports 64 bit as well.
- Mutual dependence of compiler passes. Having a clean separation between the lexer, parser, and semantic passes makes for an easier to implement and understand language. This means no user-defined tokens and no user-defined syntax.
- Macro systems. Macros are an easy way for users to add very powerful features. Unfortunately, the complexity is pushed off on to the user trying to understand the macro usage, and the result is rarely worth it and not justifiable.
- C/C++ source code compatibility. The use of the preprocessor has made this difficult to achieve without inevitably requiring manual touch up. It's best to leave this to external tools.
- Multiple inheritance. It's a complex feature of debatable value. It's very difficult to implement in an efficient manner, and compilers are prone to many bugs in implementing it. Nearly all the value of MI can be handled with single inheritance coupled with interfaces and aggregation. What's left does not justify the weight of MI implementation.

Source: https://dlang.org/overview.html

...

Comparison

- https://digitalmars.com/d/2.0/comparison.html

D Language Feature Comparison Table

Feature	
Garbage Collection	Yes
Functions	
Function delegates	Yes
Function overloading	Yes
Out function parameters	Yes
Nested functions	Yes
Function literals	Yes
Closures	Yes
Typesafe variadic arguments	Yes
Lazy function argument evaluation	Yes
Compile time function evaluation	Yes
Arrays	
Lightweight arrays	Yes
Resizeable arrays	Yes
Built-in strings	Yes
Array slicing	Yes
Array bounds checking	Yes
Array literals	Yes
Associative arrays	Yes
Strong typedefs	Yes
String switches	Yes
Aliases	Yes

OOP	
Object Oriented	Yes
Multiple Inheritance	No
Interfaces	Yes
Operator overloading	Yes
Modules	Yes
Dynamic class loading	No
Nested classes	Yes
Inner (adaptor) classes	Yes
Covariant return types	Yes
Properties	Yes
Performance	
Inline assembler	Yes
Direct access to hardware	Yes
Lightweight objects	Yes
Explicit memory allocation control	Yes
Independent of VM	Yes
Direct native code gen	Yes

Generic Programming	
Class Templates	Yes
Function Templates	Yes
Implicit Function Template Instantiation	Yes
Partial and Explicit Specialization	Yes
Value Template Parameters	Yes
Template Template Parameters	Yes
Variadic Template Parameters	Yes
Template Constraints	Yes
Mixins	Yes
static if	Yes
is expressions	Yes
typeof	Yes
foreach	Yes
Implicit Type Inference	Yes
Reliability	
Contract Programming	Yes
Unit testing	Yes
Static construction order	Yes
Guaranteed initialization	Yes
RAII (automatic destructors)	Yes
Exception handling	Yes
Scope guards	Yes
try-catch-finally blocks	Yes
Thread synchronization primitives	Yes

Compatibility	
C-style syntax	Yes
Enumerated types	Yes
Support all C types	Yes
80 bit floating point	Yes
Complex and Imaginary	Yes
Direct access to C	Yes
Use existing debuggers	Yes
Struct member alignment control	Yes
Generates standard object files	Yes
Macro text preprocessor	No
Other	
Conditional compilation	Yes
Unicode source text	Yes
Documentation comments	Yes

- https://github.com/phillvancejr/Cpp-Go-Zig-Odin
- ...

2.1 Paradigms

D supports five main programming paradigms:

- concurrent (actor model)
- object-oriented
- imperative
- functional
- metaprogramming

2.1.1 Imperative

Imperative programming in D is almost identical to that in C. Functions, data, statements, declarations and expressions work just as they do in C, and the C runtime library may be accessed directly. On the other hand, some notable differences between D and C in the area of imperative programming include D's `foreach` loop construct, which allows looping over a collection, and nested functions, which are functions that are declared inside another and may access the enclosing function's local variables.

```
import std.stdio;

void main() {
    int multiplier = 10;
    int scaled(int x) { return x * multiplier; }

    foreach (i; 0 .. 10) {
        writefln("Hello, world %d! scaled = %d", i, scaled(i));
    }
}
```

Source: https://en.wikipedia.org/wiki/D_(programming_language)

2.1.2 Object-oriented

Object-oriented programming in D is based on a single inheritance hierarchy, with all classes derived from class Object. D does not support multiple inheritance; instead, it uses Java-style interfaces, which are comparable to C++'s pure abstract classes, and mixins, which separates common functionality from the inheritance hierarchy. D also allows the defining of static and final (non-virtual) methods in interfaces.

Interfaces and inheritance in D support covariant types for return types of overridden methods.

D supports type forwarding, as well as optional custom dynamic dispatch.

Classes (and interfaces) in D can contain invariants which are automatically checked before and after entry to public methods, in accordance with the design by contract methodology.

Many aspects of classes (and structs) can be introspected automatically at compile time (a form of reflection using `type traits`) and at run time (RTTI / `TypeInfo`), to facilitate generic code or automatic code generation (usually using compile-time techniques).

Source: https://en.wikipedia.org/wiki/D_(programming_language)

2.1.3 Functional

D supports functional programming features such as function literals, closures, recursively-immutable objects and the use of higher-order functions. There are two syntaxes for anonymous functions, including a multiple-statement form and a "shorthand" single-expression notation:[12]

```
int function(int) g;
g = (x) { return x * x; }; // longhand
g = (x) => x * x;          // shorthand
```

There are two built-in types for function literals, `function`, which is simply a pointer to a stack-allocated function, and `delegate`, which also includes a pointer to the surrounding environment. Type inference may be used with an anonymous function, in which case the compiler creates a `delegate` unless it can prove that an environment pointer is not necessary. Likewise, to implement a closure, the compiler places enclosed local variables on the heap only if necessary (for example, if a closure is returned by another function, and exits that function's scope). When using type inference, the compiler will also add attributes such as `pure` and `nothrow` to a function's type, if it can prove that they apply.

Other functional features such as currying and common higher-order functions such as map, filter, and reduce are available through the standard library modules `std.functional` and `std.algorithm`.

```
import std.stdio, std.algorithm, std.range;

void main()
{
    int[] a1 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
    int[] a2 = [6, 7, 8, 9];

    // must be immutable to allow access from inside a pure function
    immutable pivot = 5;

    int mySum(int a, int b) pure nothrow // pure function
    {
        if (b <= pivot) // ref to enclosing-scope
            return a + b;
        else
            return a;
    }

    // passing a delegate (closure)
    auto result = reduce!mySum(chain(a1, a2));
    writeln("Result: ", result); // Result: 15

    // passing a delegate literal
    result = reduce!((a, b) => (b <= pivot) ? a + b : a)(chain(a1, a2));
    writeln("Result: ", result); // Result: 15
}
```

Alternatively, the above function compositions can be expressed using Uniform function call syntax (UFCS) for more natural left-to-right reading:

```
    auto result = a1.chain(a2).reduce!mySum();
    writeln("Result: ", result);

    result = a1.chain(a2).reduce!((a, b) => (b <= pivot) ? a + b : a)();
    writeln("Result: ", result);
```

Source: https://en.wikipedia.org/wiki/D_(programming_language)

Tutorials

- https://tour.dlang.org/tour/en/gems/functional-programming
- https://dlang.org/phobos/std_functional.html
- https://garden.dlang.io/
- …

2.1.4 Concurrent

- **Parallelism**

Parallel programming concepts are implemented in the library, and do not require extra support from the compiler. However the D type system and compiler ensure that data sharing can be detected and managed transparently.

```d
import std.stdio : writeln;
import std.range : iota;
import std.parallelism : parallel;

void main()
{
    foreach (i; iota(11).parallel) {
        // The body of the foreach loop is executed in parallel for each i
        writeln("processing ", i);
    }
}
```

`iota(11).parallel` is equivalent to `std.parallelism.parallel(iota(11))` by using UFCS.

The same module also supports `taskPool` which can be used for dynamic creation of parallel tasks, as well as map-filter-reduce and fold style operations on ranges (and arrays), which is useful when combined with functional operations. `std.algorithm.map` returns a lazily evaluated range rather than an array. This way, the elements are computed by each worker task in parallel automatically.

```d
import std.stdio : writeln;
import std.algorithm : map;
import std.range : iota;
import std.parallelism : taskPool;

/* On Intel i7-3930X and gdc 9.3.0:
 * 5140ms using std.algorithm.reduce
 * 888ms using std.parallelism.taskPool.reduce
 *
 * On AMD Threadripper 2950X, and gdc 9.3.0:
 * 2864ms using std.algorithm.reduce
 * 95ms using std.parallelism.taskPool.reduce
 */
void main()
{
  auto nums = iota(1.0, 1_000_000_000.0);

  auto x = taskPool.reduce!"a + b"(
      0.0, map!"1.0 / (a * a)"(nums)
  );

  writeln("Sum: ", x);
}
```

Source: https://en.wikipedia.org/wiki/D_(programming_language)

Concurrency

Concurrency is fully implemented in the library, and it does not require support from the compiler. Alternative implementations and methodologies of writing concurrent code are possible. The use of D typing system does help ensure memory safety.

```
import std.stdio, std.concurrency, std.variant;

void foo()
{
    bool cont = true;

    while (cont)
    {
        receive( // Delegates are used to match the message type.
            (int msg) => writeln("int received: ", msg),
            (Tid sender) { cont = false; sender.send(-1); },
            (Variant v) => writeln("huh?") // Variant matches any type
        );
    }
}

void main()
{
    auto tid = spawn(&foo); // spawn a new thread running foo()

    foreach (i; 0 .. 10)
        tid.send(i);   // send some integers

    tid.send(1.0f);    // send a float
    tid.send("hello"); // send a string
    tid.send(thisTid); // send a struct (Tid)

    receive((int x) => writeln("Main thread received message: ", x));
}
```

Source: https://en.wikipedia.org/wiki/D_(programming_language)

APIs

- core.Thread
- std.concurrency.spawn
- std.parallelism.task / taskPool
- vibe.d.runTask
- etc.

Source: https://skoppe.github.io/dconf-2022/8

- **Fiber**

https://tour.dlang.org/tour/en/multithreading/fibers
https://dlang.org/phobos/core_thread_fiber.html
https://dlang.org/library/core/thread/fiber/fiber.html
http://ddili.org/ders/d.en/fibers.html

```
import core.thread;

/* This is the fiber function that generates each element and
 * then sets the 'ref' parameter to that element. */
void fibonacciSeries(ref int current) {                    // (1)
    current = 0;    // Note that 'current' is the parameter
    int next = 1;

    while (true) {
        Fiber.yield();                                     // (2)
        /* Next call() will continue from this point */    // (3)

        const nextNext = current + next;
        current = next;
        next = nextNext;
    }
}

void main() {
    int current;                                           // (1)
                        // (4)
    Fiber fiber = new Fiber(() => fibonacciSeries(current));

    foreach (_; 0 .. 10) {
        fiber.call();                                      // (5)

        import std.stdio;
        writef("%s ", current);
    }
}
```

2.1.5 Metaprogramming

Metaprogramming is supported through templates, compile-time function execution, tuples, and string mixins. The following examples demonstrate some of D's compile-time features.

Templates in D can be written in a more imperative style compared to the C++ functional style for templates. This is a regular function that calculates the factorial of a number:

```
ulong factorial(ulong n) {
    if (n < 2)
        return 1;
    else
        return n * factorial(n-1);
}
```

Here, the use of `static if`, D's compile-time conditional construct, is demonstrated to construct a template that performs the same calculation using code that is similar to that of the function above:

```
template Factorial(ulong n) {
    static if (n < 2)
        enum Factorial = 1;
    else
        enum Factorial = n * Factorial!(n-1);
}
```

In the following two examples, the template and function defined above are used to compute factorials. The types of constants need not be specified explicitly as the compiler infers their types from the right-hand sides of assignments:

```
enum fact_7 = Factorial!(7);
```

This is an example of compile-time function execution (CTFE). Ordinary functions may be used in constant, compile-time expressions provided they meet certain criteria:

```
enum fact_9 = factorial(9);
```

The `std.string.format` function performs `printf`-like data formatting (also at compile-time, through CTFE), and the "msg" pragma displays the result at compile time:

```
import std.string : format;
pragma(msg, format("7! = %s", fact_7));
pragma(msg, format("9! = %s", fact_9));
```

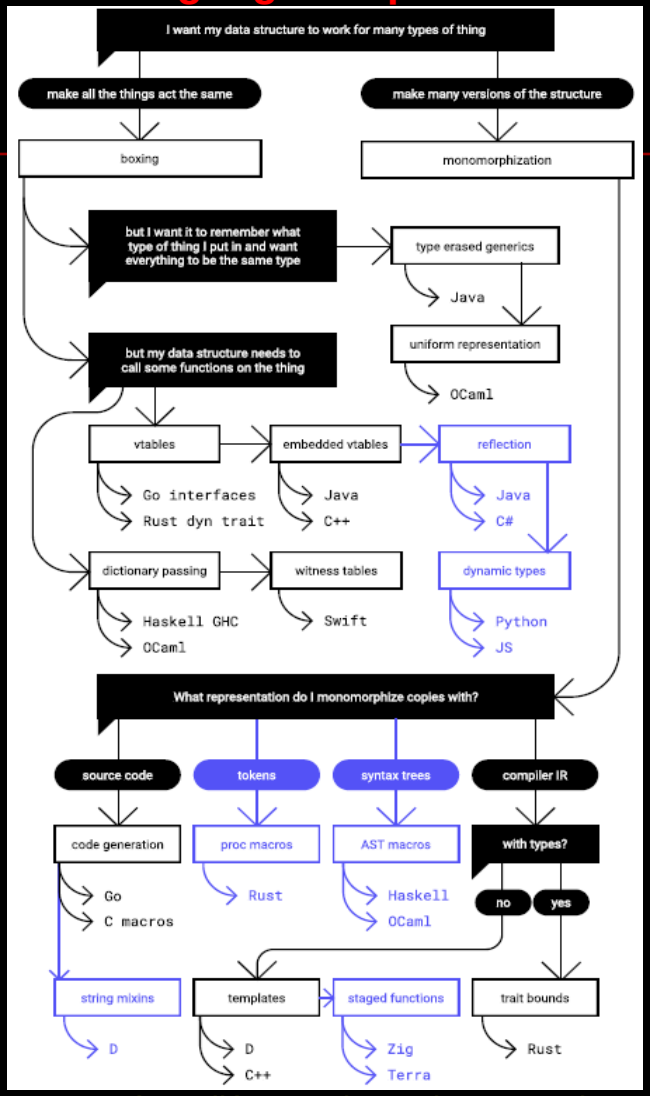
String mixins, combined with compile-time function execution, allow for the generation of D code using string operations at compile time. This can be used to parse domain-specific languages, which will be compiled as part of the program:

```
import FooToD; // hypothetical module which contains a function that parses Foo source code
               // and returns equivalent D code
void main() {
    mixin(fooToD(import("example.foo")));
}
```

Source: https://en.wikipedia.org/wiki/D_(programming_language)

Comparison

- How Languages Implement Generics and Extensions to Metaprogramming



Source: https://thume.ca/assets/postassets/generics/flowchart.pdf

2.2 Memory management

Memory is usually managed with garbage collection, but specific objects may be finalized immediately when they go out of scope. This is what the majority of programs and libraries written in D use.

In case more control over memory layout and better performance is needed, explicit memory management is possible using the overloaded operators `new` and `delete`, by calling C's malloc and free directly, or implementing custom allocator schemes (i.e. on stack with fallback, RAII style allocation, reference counting, shared reference counting). Garbage collection can be controlled: programmers may add and exclude memory ranges from being observed by the collector, can disable and enable the collector and force either a generational or full collection cycle.[18] The manual gives many examples of how to implement different highly optimized memory management schemes for when garbage collection is inadequate in a program.[19]

In functions, `struct` instances are by default allocated on the stack, while `class` instances by default allocated on the heap (with only reference to the class instance being on the stack). However this can be changed for classes, for example using standard library template `std.typecons.scoped`, or by using `new` for structs and assigning to a pointer instead of a value-based variable.[20]

In functions, static arrays (of known size) are allocated on the stack. For dynamic arrays, one can use the `core.stdc.stdlib.alloca` function (similar to `alloca` in C), to allocate memory on the stack. The returned pointer can be used (recast) into a (typed) dynamic array, by means of a slice (however resizing array, including appending must be avoided; and for obvious reasons they must not be returned from the function).[20]

A `scope` keyword can be used both to annotate parts of code, but also variables and classes/structs, to indicate they should be destroyed (destructor called) immediately on scope exit. Whatever the memory is deallocated also depends on implementation and class-vs-struct differences.[21]

`std.experimental.allocator` contains a modular and composable allocator templates, to create custom high performance allocators for special use cases.[22]

Source: https://en.wikipedia.org/wiki/D_(programming_language)

- **https://wiki.dlang.org/Memory_Management**

Garbage Collection

- https://dlang.org/spec/garbage.html

1. D is a systems programming language with support for garbage collection. Usually it is not necessary to free memory explicitly. Just allocate as needed, and the garbage collector will periodically return all unused memory to the pool of available memory.

2. D also provides the mechanisms to write code where the garbage collector is **not involved**. More information is provided below.

3. Programmers accustomed to explicitly managing memory allocation and deallocation will likely be skeptical of the benefits and efficacy of garbage collection. Experience both with new projects written with garbage collection in mind, and converting existing projects to garbage collection shows that:

- Garbage collected programs are often faster. This is counterintuitive, but the reasons are:
  - Reference counting is a common solution to solve explicit memory allocation problems. The code to implement the increment and decrement operations whenever assignments are made is one source of slowdown. Hiding it behind smart pointer classes doesn't help the speed. (Reference counting methods are not a general solution anyway, as circular references never get deleted.)
  - Destructors are used to deallocate resources acquired by an object. For most classes, this resource is allocated memory. With garbage collection, most destructors then become empty and can be discarded entirely.
  - All those destructors freeing memory can become significant when objects are allocated on the stack. For each one, some mechanism must be established so that if an exception happens, the destructors all get called in each frame to release any memory they hold. If the destructors become irrelevant, then there's no need to set up special stack frames to handle exceptions, and the code runs faster.
  - Garbage collection kicks in only when memory gets tight. When memory is not tight, the program runs at full speed and does not spend any time tracing and freeing memory.
  - Garbage collected programs do not suffer from gradual deterioration due to an accumulation of memory leaks.
- Garbage collectors reclaim unused memory, therefore they do not suffer from "memory leaks" which can cause long running applications to gradually consume more and more memory until they bring down the system. GC programs have longer term stability.
- Garbage collected programs have fewer hard-to-find pointer bugs. This is because there are no dangling references to freed memory. There is no code to explicitly manage memory, hence no bugs in such code.
- Garbage collected programs are faster to develop and debug, because there's no need for developing, debugging, testing, or maintaining the explicit deallocation code.

Tutorials

- https://en.wikibooks.org/wiki/D_Programming/Garbage_collector/Thoughts_about_better_GC_implementations
- https://news.ycombinator.com/item?id=14592457
- https://www.reddit.com/r/ProgrammingLanguages/comments/r4x2do/why_isnt_dstyle_hybrid_memory_management_with/
- https://users.rust-lang.org/t/dlang-adds-a-borrowchecker-called-the-ob-system-for-ownership-borrowing/42872
- https://news.ycombinator.com/item?id=13914308
- https://dev.to/jessekphillips/memory-free-programming-in-d-1mbo
- https://tour.dlang.org/tour/en/basics/memory
- …

2.3 Safety

- https://dlang.org/spec/function.html#function-safety
- …

2.3.1 Memory Safe

- https://dlang.org/spec/memory-safe-d.html

1. *Memory Safety* for a program is defined as it being impossible for the program to corrupt memory. Therefore, the safe subset of D consists only of programming language features that are guaranteed to never result in memory corruption. See this article for a rationale.

2. Memory-safe code cannot use certain language features, such as:

   - Casts that break the type system.
   - Modification of pointer values.
   - Taking the address of a local variable or function parameter.

- Usage

1. There are three categories of functions from the perspective of memory safety:

   - `@safe` functions
   - `@trusted` functions
   - `@system` functions

2. **`@system`** functions may perform any operation legal from the perspective of the language, including inherently memory unsafe operations such as pointer casts or pointer arithmetic. However, compile-time known memory corrupting operations, such as indexing a static array out of bounds or returning a pointer to an expired stack frame, can still raise an error. **`@system`** functions may not be called directly from **`@safe`** functions.

3. **`@trusted`** functions have all the capabilities of **`@system`** functions but may be called from **`@safe`** functions. For this reason they should be very limited in the scope of their use. Typical uses of **`@trusted`** functions include wrapping system calls that take buffer pointer and length arguments separately so that **`@safe`** functions may call them with arrays. **`@trusted`** functions must have a safe interface.

4. **`@safe`** functions have a number of restrictions on what they may do and are intended to disallow operations that may cause memory corruption. See `@safe` functions.

5. These attributes may be inferred when the compiler has the function body available, such as with templates.

6. Array bounds checks are necessary to enforce memory safety, so these are enabled (by default) for **`@safe`** code even in **`-release`** mode.

Scope and Return Parameters

1. The function parameter attributes **`return`** and **`scope`** are used to track what happens to low-level pointers passed to functions. Such pointers include: raw pointers, arrays, **`this`**, classes, **`ref`** parameters, delegate/lazy parameters, and aggregates containing a pointer.

2. **`scope`** ensures that no references to the pointed-to object are retained, in global variables or pointers passed to the function (and recursively to other functions called in the function), as a result of calling the function. Variables in the function body and parameter list that are **`scope`** may have their allocations elided as a result.

3. **`return`** indicates that either the return value of the function or the first parameter is a pointer derived from the **`return`** parameter or any other parameters also marked **`return`**. For constructors, **`return`** applies to the (implicitly returned) **`this`** reference. For void functions, **`return`** applies to the first parameter *iff* it is **`ref`**; this is to support UFCS, property setters and non-member functions (e.g. **`put`** used like **`put(dest, source)`**).

4. These attributes may appear after the formal parameter list, in which case they apply either to a method's **`this`** parameter, or to a free function's first parameter *iff* it is **`ref`**. **`return`** or **`scope`** is ignored when applied to a type that is not a low-level pointer.

5. **Note:** Checks for **`scope`** parameters are currently enabled only for **`@safe`** code compiled with the **`-dip1000`** command-line flag.

Limitations

1. Memory safety does not imply that code is portable, uses only sound programming practices, is free of byte order dependencies, or other bugs. It is focussed only on eliminating memory corruption possibilities.

...

2.3.2 SafeD

SafeD [edit]

SafeD[23] is the name given to the subset of D that can be guaranteed to be memory safe (no writes to memory that has not been allocated or that has been recycled). Functions marked `@safe` are checked at compile time to ensure that they do not use any features that could result in corruption of memory, such as pointer arithmetic and unchecked casts, and any other functions called must also be marked as `@safe` or `@trusted`. Functions can be marked `@trusted` for the cases where the compiler cannot distinguish between safe use of a feature that is disabled in SafeD and a potential case of memory corruption.[24]

Scope lifetime safety [edit]

Initially under the banners of DIP1000[25] and DIP25[26] (now part of the language specification[27]), D provides protections against certain ill-formed constructions involving the lifetimes of data.

The current mechanisms in place primarily deal with function parameters and stack memory however it is a stated ambition of the leadership of the programming language to provide a more thorough treatment of lifetimes within the D programming language[28] (influenced by ideas from Rust programming language).

Lifetime safety of assignments [edit]

Within @safe code, the lifetime of an assignment involving a reference type is checked to ensure that the lifetime of the assignee is longer than that of the assigned.

For example:

```
@safe void test()
{
    int tmp = 0; // #1
    int* rad;    // #2
    rad = &tmp;  // If the order of the declarations of #1 and #2 is reversed, this fails.
    {
        int bad = 45; // Lifetime of "bad" only extends to the scope in which it is defined.
        *rad = bad;   // This is kosher.
        rad = &bad;   // Lifetime of rad longer than bad, hence this is not kosher at all.
    }
}
```

Function parameter lifetime annotations within @safe code [edit]

When applied to function parameter which are either of pointer type or references, the keywords *return* and *scope* constrain the lifetime and use of that parameter.

The language standard dictates the following behaviour:[29]

Storage Class	Behaviour (and constraints to) of a parameter with the storage class
scope	References in the parameter cannot be escaped. Ignored for parameters with no references
return	Parameter may be returned or copied to the first parameter, but otherwise does not escape from the function. Such copies are required not to outlive the argument(s) they were derived from. Ignored for parameters with no references

An annotated example is given below.

```
@safe:

int* gp;
void thorin(scope int*);
void gloin(int*);
int* balin(return scope int* p, scope int* q, int* r)
{
     gp = p; // error, p escapes to global gp
     gp = q; // error, q escapes to global gp
     gp = r; // ok

     thorin(p); // ok, p does not escape thorin()
     thorin(q); // ok
     thorin(r); // ok

     gloin(p); // error, gloin() escapes p
     gloin(q); // error, gloin() escapes q
     gloin(r); // ok that gloin() escapes r

     return p; // ok
     return q; // error, cannot return 'scope' q
     return r; // ok
}
```

Source: https://en.wikipedia.org/wiki/D_(programming_language)

- https://dlang.org/articles/safed.html

2.4 Low-level programming
2.4.1 Inline ASM
2.4.1.1 X86/X86_64

- https://dlang.org/spec/iasm.html

1. D, being a systems programming language, provides an inline assembler. The inline assembler is standardized for D implementations across the same CPU family, for example, the Intel Pentium inline assembler for a Win32 D compiler will be syntax compatible with the inline assembler for Linux running on an Intel Pentium.

2. Implementations of D on different architectures, however, are free to innovate upon the memory model, function call/return conventions, argument passing conventions, etc.

3. This document describes the **x86** and **x86_64** implementations of the inline assembler. The inline assembler platform support that a compiler provides is indicated by the **D_InlineAsm_X86** and **D_InlineAsm_X86_64** version identifiers, respectively.

30.1 Asm statement

```
AsmStatement:
    asm FunctionAttributesopt { AsmInstructionListopt }

AsmInstructionList:
    AsmInstruction ;
    AsmInstruction ; AsmInstructionList
```

1. Assembler instructions must be located inside an **asm** block. Like functions, **asm** statements must be anotated with adequate function attributes to be compatible with the caller. Asm statements attributes must be explicitly defined, they are not infered.

```
void func1() pure nothrow @safe @nogc
{
    asm pure nothrow @trusted @nogc
    {}
}

void func2() @safe @nogc
{
    asm @nogc // Error: asm statement is assumed to be @system - mark it with '@trusted' if it is not
    {}
}
```

Labels

1. Assembler instructions can be labeled just like other statements. They can be the target of goto statements. For example:

```
void *pc;
asm
{
    call L1          ;
  L1:                ;
    pop  EBX         ;
    mov  pc[EBP],EBX ; // pc now points to code at L1
}
```

Operand Types

```
AsmTypePrefix:
    near ptr
    far ptr
    word ptr
    dword ptr
    qword ptr
    FundamentalType ptr
```

1. In cases where the operand size is ambiguous, as in:

```
add [EAX],3     ;
```

it can be disambiguated by using an *AsmTypePrefix*:

```
add  byte ptr [EAX],3 ;
add  int ptr [EAX],7  ;
```

2. **far ptr** is not relevant for flat model code.

Struct/Union/Class Member Offsets

1. To access members of an aggregate, given a pointer to the aggregate is in a register, use the **.offsetof** property of the qualified name of the member:

```
struct Foo { int a,b,c; }
int bar(Foo *f)
{
    asm
    {
        mov EBX,f                   ;
        mov EAX,Foo.b.offsetof[EBX] ;
    }
}
void main()
{
    Foo f = Foo(0, 2, 0);
    assert(bar(&f) == 2);
}
```

2. Alternatively, inside the scope of an aggregate, only the member name is needed:

```
struct Foo   // or class
{
    int a,b,c;
    int bar()
    {
        asm
        {
            mov EBX, this   ;
            mov EAX, b[EBX] ;
        }
    }
}
void main()
{
    Foo f = Foo(0, 2, 0);
    assert(f.bar() == 2);
}
```

Special Symbols

$

Represents the program counter of the start of the next instruction. So,

```
jmp  $  ;
```

branches to the instruction following the jmp instruction. The $ can only appear as the target of a jmp or call instruction.

__LOCAL_SIZE

This gets replaced by the number of local bytes in the local stack frame. It is most handy when the **naked** is invoked and a custom stack frame is programmed.

Stack Variables

1. Stack variables (variables local to a function and allocated on the stack) are accessed via the name of the variable indexed by EBP:

```
int foo(int x)
{
    asm
    {
        mov EAX,x[EBP] ; // loads value of parameter x into EAX
        mov EAX,x      ; // does the same thing
    }
}
```

2. If the [EBP] is omitted, it is assumed for local variables. If **naked** is used, this no longer holds.

...

2.4.1.2 LDC inline assembly expressions

- https://wiki.dlang.org/LDC_inline_assembly_expressions

LDC supports an LLVM-specific variant of GCC's extended inline assembly expressions. They are useful on platforms where the D asm statement is not yet available (i.e. non-x86), or when the limitations or it being a statement are problematic. Being an expression, extended inline expressions are able to return values!

Additionally issues regarding inlining of function containing inline asm are mostly not relevant for extended inline assembly expressions. Effectively, extended inline assembly expression can be used to efficiently implement new intrinsics in the compiler.

Contents [hide]

Interface

To use them you must import the module containing the magic declarations:

```
import ldc.llvmasm;
```

Three different forms exist:

No return value:

```
void __asm (char[] asmcode, char[] constraints, [ Arguments... ] );
```

Single return value:

```
template __asm(T) {
  T __asm (char[] asmcode, char[] constraints, [ Arguments... ] );
}
```

Multiple return values:

```
struct __asmtuple_t(T...) {
  T v;
}
template __asmtuple(T...) {
  __asmtuple_t!(T) __asmtuple (char[] asmcode, char[] constraints, [ Arguments... ] );
}
```

...

An example for ARM

The following code adds two multibyte numbers and a carry and returns the final carry. It uses an interface similar to multibyteAddSub() in biguintnoasm.d.

```
uint multibyteAdd(uint[] dest, const(uint) [] src1,
    const (uint) [] src2, uint carry) pure @nogc nothrow
{
    assert(carry == 0 || carry == 1);
    assert(src1.length >= dest.length && src2.length >= dest.length);
    return __asm!uint(`   cmp    $2,#0                @ Check dest.length
                          beq    1f
                          mov    r5,#0                @ Initialize index
                        2:
                          ldr    r6,[${3:m},r5,LSL #2] @ Load *(src1.ptr + index)
                          ldr    r7,[${4:m},r5,LSL #2] @ Load *(src2.ptr + index)
                          lsrs   $0,$0,#1             @ Set carry
                          adcs   r6,r6,r7             @ Add with carry
                          str    r6,[${1:m},r5,LSL #2] @ Store *(dest.ptr + index)
                          adc    $0,$0,#0             @ Store carry
                          add    r5,r5,#1             @ Increment index
                          cmp    $2,r5
                          bhi    2b
                        1:`,
                      "=&r,=*m,r,*m,*m,0,~{r5},~{r6},~{r7},~{cpsr}",
                      dest.ptr, dest.length, src1.ptr, src2.ptr, carry);
}
```

The constraint string in this example is complex.

- A value is returned in a register which is marked as early clobber (`=&r`). This register is not used for input values.
- The pointers uses the memory constraint (`=*m` and `*m`). The pointer is passed in a register which is considered read-only.
- The carry parameter is tied to output parameter `0` and therefore uses the same register.
- The code clobers some registers and the current program status register.

Please note the following details:

- Even if you tie an input parameter to an output parameter you may need to mark the output register as early clobber. If LLVM can prove that input parameters always have the same value (e.g. all array length and the carry are 1) then the same register is used for these inputs. This happens even if the input parameter is tied to an output parameter! Marking the output as early clobber prevents this.
- Use can use only local labels (`1:`, `2:` and so on) and you have to specify the direction (`f` = forward, `b` = backward) if you refer to them.
- The parameter for a memory constraint is replaced with a memory access: `[` register `]`. E.g. the parameter `$1` from the example above may be expanded as `[r0]`. If you need only the register then you must use the modifier `m`: `${1:m}` is expanded as `r0`. (The register is chosen by LLVM!)
- `@` is used to mark a comment. Other architectures use other characters.

2.4.2 Pointer

- https://dlang.org/spec/type.html#pointers

1. A pointer to type **T** has a value which is a reference (address) to another object of type **T**. It is commonly called a *pointer to T* and its type is **T***. To access the object value, use the * dereference operator:

```
int* p;

assert(p == null);
p = new int(5);
assert(p != null);

assert(*p == 5);
(*p)++;
assert(*p == 6);
```

2. If a pointer contains a *null* value, it is not pointing to a valid object.
3. When a pointer to *T* is dereferenced, it must either contain a *null* value, or point to a valid object of type *T*.
4. **Implementation Defined:**
    1. The behavior when a *null* pointer is dereferenced. Typically the program will be aborted.
5. **Undefined Behavior:** dereferencing a pointer that is not *null* and does not point to a valid object of type *T*.
6. To set a pointer to point at an existing object, use the **&** *address of* operator:

```
int i = 2;
int* p = &i;

assert(p == &i);
assert(*p == 2);
*p = 4;
assert(i == 4);
```

- https://dlang.org/spec/expression.html#pointer_arithmetic

Tutorials

- https://www.tutorialspoint.com/d_programming/d_programming_pointers.htm
- https://en.wikibooks.org/wiki/D_(The_Programming_Language)/d2/Pointers,_Pass-By-Reference_and_Static_Arrays
- …

2.4.3 ABI

- https://dlang.org/spec/abi.html

1. A D implementation that conforms to the D ABI (Application Binary Interface) will be able to generate libraries, DLLs, etc., that can interoperate with D binaries built by other implementations.

38.1 C ABI

1. The C ABI referred to in this specification means the C Application Binary Interface of the target system. C and D code should be freely linkable together, in particular, D code shall have access to the entire C ABI runtime library.

38.2 Endianness

1. The endianness (byte order) of the layout of the data will conform to the endianness of the target machine. The Intel x86 CPUs are *little endian* meaning that the value 0x0A0B0C0D is stored in memory as: `0D 0C 0B 0A`.

38.3 Basic Types

bool	8 bit byte with the values 0 for false and 1 for true
byte	8 bit signed value
ubyte	8 bit unsigned value
short	16 bit signed value
ushort	16 bit unsigned value
int	32 bit signed value
uint	32 bit unsigned value
long	64 bit signed value
ulong	64 bit unsigned value
cent	128 bit signed value
ucent	128 bit unsigned value
float	32 bit IEEE 754 floating point value
double	64 bit IEEE 754 floating point value
real	implementation defined floating point value, for x86 it is 80 bit IEEE 754 extended real
char	8 bit unsigned value
wchar	16 bit unsigned value
dchar	32 bit unsigned value

- ...

2.5 BetterC

The D programming language has an official subset known as "Better C".[34] This subset forbids access to D features requiring use of runtime libraries other than that of C.

Enabled via the compiler flags "-betterC" on DMD and LDC, and "-fno-druntime" on GDC, Better C may only call into D code compiled under the same flag (and linked code other than D) but code compiled without the Better C option may call into code compiled with it: this will, however, lead to slightly different behaviours due to differences in how C and D handle asserts.

Features included in Better C [edit]

- Unrestricted use of compile-time features (for example, D's dynamic allocation features can be used at compile time to pre-allocate D data)
- Full metaprogramming facilities
- Nested functions, nested structs, delegates and lambdas
- Member functions, constructors, destructors, operating overloading, etc.
- The full module system
- Array slicing, and array bounds checking
- RAII
- **scope(exit)**
- Memory safety protections
- Interfacing with C++
- COM classes and C++ classes
- **assert** failures are directed to the C runtime library
- **switch** with strings
- **final switch**
- **unittest** blocks
- **printf** format validation

Features excluded from Better C [edit]

- Garbage collection
- TypeInfo and ModuleInfo
- Built-in threading (e.g. `core.thread`)
- Dynamic arrays (though slices of static arrays work) and associative arrays
- Exceptions
- *synchronized* and `core.sync`
- Static module constructors or destructors

Source: https://en.wikipedia.org/wiki/D_(programming_language)

2.6 History and Future

CHINA OpenInfra Days

Walter Bright started working on a new language in 1999. D was first released in December 2001[1] and reached version 1.0 in January 2007.[35] The first version of the language (D1) concentrated on the imperative, object oriented and metaprogramming paradigms,[36] similar to C++.

Some members of the D community dissatisfied with Phobos, D's official runtime and standard library, created an alternative runtime and standard library named Tango. The first public Tango announcement came within days of D 1.0's release.[37] Tango adopted a different programming style, embracing OOP and high modularity. Being a community-led project, Tango was more open to contributions, which allowed it to progress faster than the official standard library. At that time, Tango and Phobos were incompatible due to different runtime support APIs (the garbage collector, threading support, etc.). This made it impossible to use both libraries in the same project. The existence of two libraries, both widely in use, has led to significant dispute due to some packages using Phobos and others using Tango.[38]

In June 2007, the first version of D2 was released.[39] The beginning of D2's development signaled D1's stabilization. The first version of the language has been placed in maintenance, only receiving corrections and implementation bugfixes. D2 introduced breaking changes to the language, beginning with its first experimental const system. D2 later added numerous other language features, such as closures, purity, and support for the functional and concurrent programming paradigms. D2 also solved standard library problems by separating the runtime from the standard library. The completion of a D2 Tango port was announced in February 2012.[40]

The release of Andrei Alexandrescu's book *The D Programming Language* on 12 June 2010, marked the stabilization of D2, which today is commonly referred to as just "D".

In January 2011, D development moved from a bugtracker / patch-submission basis to GitHub. This has led to a significant increase in contributions to the compiler, runtime and standard library.[41]

In December 2011, Andrei Alexandrescu announced that D1, the first version of the language, would be discontinued on 31 December 2012.[42] The final D1 release, D v1.076, was on 31 December 2012.[43]

Code for the official D compiler, the *Digital Mars D compiler* by Walter Bright, was originally released under a custom license, qualifying as source available but not conforming to the open source definition.[44] In 2014, the compiler front-end was re-licensed as open source under the Boost Software License.[3] This re-licensed code excluded the back-end, which had been partially developed at Symantec. On 7 April 2017, the whole compiler was made available under the Boost license after Symantec gave permission to re-license the back-end, too.[4][45][46][47] On 21 June 2017, the D Language was accepted for inclusion in GCC.[48]

Source: https://en.wikipedia.org/wiki/D_(programming_language)

- **Origins of the D Programming Language**

https://dl.acm.org/doi/abs/10.1145/3386323

Origins of the D programming language

Authors: Walter Bright, Andrei Alexandrescu, Michael Parker Authors Info & Claims

Proceedings of the ACM on Programming Languages, Volume 4, Issue HOPL • June 2020 • Article No.: 73, pp 1–38 • https://doi.org/10.1145/3386323

https://zhuanlan.zhihu.com/p/129027674

- **https://wiki.dlang.org/Language_History_and_Future**

III. LDC

Compilers for 

- https://wiki.dlang.org/Compilers

DMD »

Digital Mars D compiler

The official reference D compiler.

DMD Portal »

GDC »

GCC D compiler

The DMD compiler front end coupled with the GCC compiler back end. Fast and open source.

GDC Portal »

LDC »

LLVM D compiler

The DMD compiler front end coupled with the LLVM compiler back end. Fast and open source.

LDC Portal »

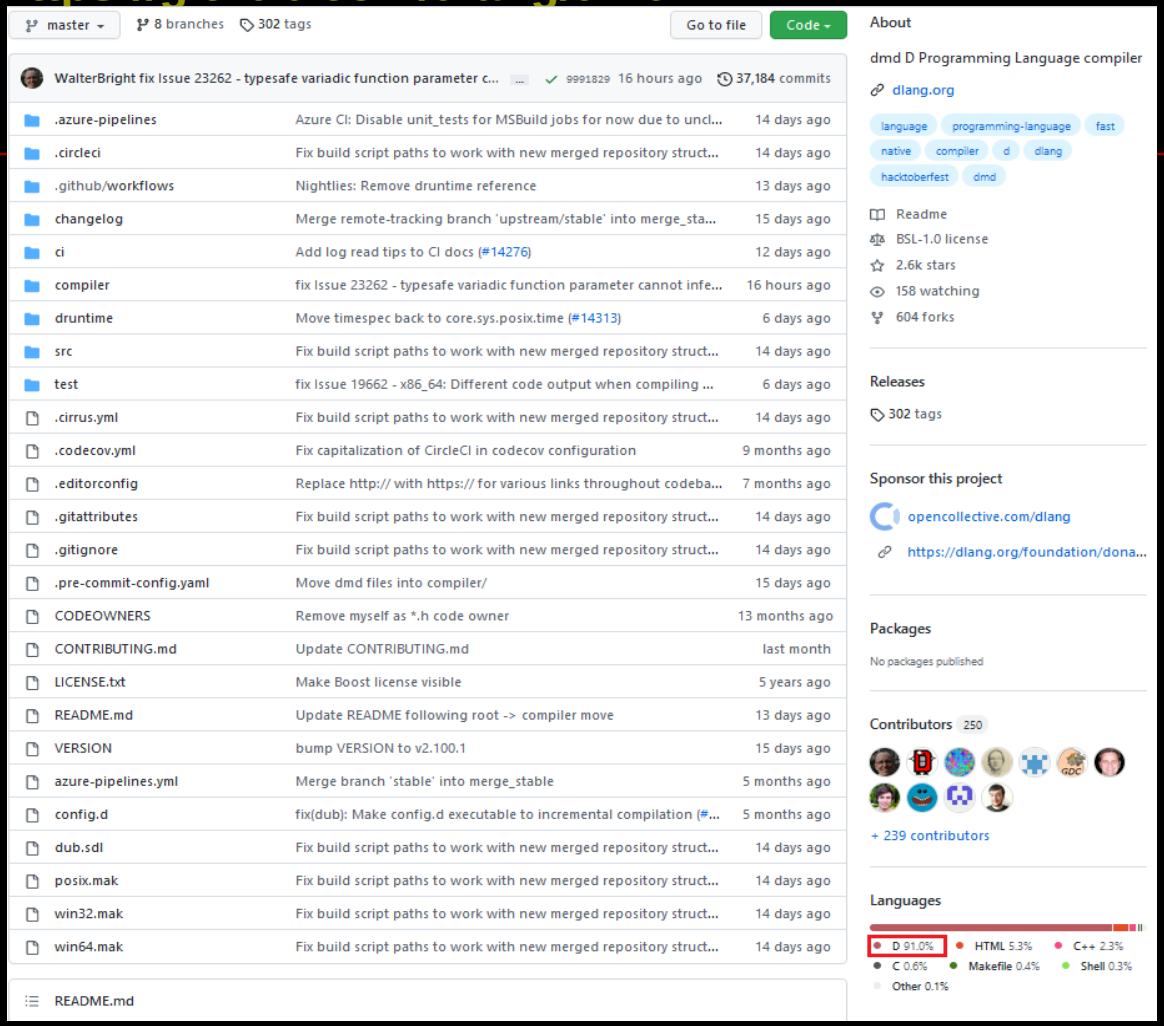
...

- https://wiki.dlang.org/DMD
  https://github.com/dlang/dmd
- https://wiki.dlang.org/GDC
  https://gcc.gnu.org/gcc-9/changes.html
  Support for  has been added to GCC 9.1!
- ...

CHINA OpenInfra Days

DMD

- https://github.com/dlang/dmd



master · 8 branches · 302 tags · Go to file · Code

WalterBright fix Issue 23262 - typesafe variadic function parameter c... 9991829 16 hours ago 37,184 commits

.azure-pipelines	Azure CI: Disable unit_tests for MSBuild jobs for now due to uncl...	14 days ago
.circleci	Fix build script paths to work with new merged repository struct...	14 days ago
.github/workflows	Nightlies: Remove druntime reference	13 days ago
changelog	Merge remote-tracking branch 'upstream/stable' into merge_sta...	15 days ago
ci	Add log read tips to CI docs (#14276)	12 days ago
compiler	fix Issue 23262 - typesafe variadic function parameter cannot infe...	16 hours ago
druntime	Move timespec back to core.sys.posix.time (#14313)	6 days ago
src	Fix build script paths to work with new merged repository struct...	14 days ago
test	fix Issue 19662 - x86_64: Different code output when compiling ...	6 days ago
.cirrus.yml	Fix build script paths to work with new merged repository struct...	14 days ago
.codecov.yml	Fix capitalization of CircleCI in codecov configuration	9 months ago
.editorconfig	Replace http:// with https:// for various links throughout codeba...	7 months ago
.gitattributes	Fix build script paths to work with new merged repository struct...	14 days ago
.gitignore	Fix build script paths to work with new merged repository struct...	14 days ago
.pre-commit-config.yaml	Move dmd files into compiler/	15 days ago
CODEOWNERS	Remove myself as *.h code owner	13 months ago
CONTRIBUTING.md	Update CONTRIBUTING.md	last month
LICENSE.txt	Make Boost license visible	5 years ago
README.md	Update README following root -> compiler move	13 days ago
VERSION	bump VERSION to v2.100.1	15 days ago
azure-pipelines.yml	Merge branch 'stable' into merge_stable	5 months ago
config.d	fix(dub): Make config.d executable to incremental compilation (#...	5 months ago
dub.sdl	Fix build script paths to work with new merged repository struct...	14 days ago
posix.mak	Fix build script paths to work with new merged repository struct...	14 days ago
win32.mak	Fix build script paths to work with new merged repository struct...	14 days ago
win64.mak	Fix build script paths to work with new merged repository struct...	14 days ago

README.md

About

dmd D Programming Language compiler

dlang.org

language · programming-language · fast · native · compiler · d · dlang · hacktoberfest · dmd

Readme
BSL-1.0 license
2.6k stars
158 watching
604 forks

Releases

302 tags

Sponsor this project

opencollective.com/dlang
https://dlang.org/foundation/dona...

Packages

No packages published

Contributors 250

+ 239 contributors

Languages

D 91.0% · HTML 5.3% · C++ 2.3% · C 0.6% · Makefile 0.4% · Shell 0.3% · Other 0.1%

- https://dconf.org/2015/talks/murphy.html

"DDMD and Automated Conversion from C++ to D"

1) Overview

- https://wiki.dlang.org/LDC

The LDC project aims to provide a portable D programming language compiler with modern optimization and code generation capabilities. The compiler uses the official DMD frontend to support the latest D2 version and relies on the LLVM Core libraries for code generation.

LDC is fully Open Source; the parts of the code not taken/adapted from other projects are BSD-licensed (see the LICENSE file for details).

D1 is no longer supported in the current development tree; the last version supporting it can be found at the `d1` Git branch.

Architectures

x86 / x86_64	Fine, CI-tested.
64-bit ARM	Usable, a few testsuite failures remain (at least for non-Apple OS). See tracker issue and latest Linux CI logs for current status.
32-bit ARM	Should be okay-ish, not really CI-tested though (Android ARMv7-A somewhat is). E.g., see tracker issue and Minimal semihosted ARM Cortex-M "Hello World".
WebAssembly	Should work fine; currently limited to `-betterC` (lacking druntime/Phobos support). See Generating WebAssembly with LDC for a quick how-to.
PowerPC	Incomplete; has been worked on (32-bit, 64-bit big & little endian) but not touched/tested in years. E.g., see tracker issue.
MIPS	Incomplete; has been worked on but not really touched/tested in years. E.g., see tracker issue.
Others	Basic code generation should work as long as there's an LLVM backend, and simple `-betterC` code might just work. druntime/Phobos support is most likely lacking (and probably requires a new `version` predefined by the compiler). Proper C(++) interop most likely requires a new `TargetABI` implementation in the compiler.

Operating Systems

Linux	glibc environments are working fine & CI-tested (on i686/x86_64 and AArch64). musl/uClibc should be usable but aren't tested.
macOS	Fine, CI-tested on x86_64. Requires macOS v10.7+ for native TLS. 32-bit support was dropped with LDC v1.15.
Windows	Fine for 'native' MSVC (Microsoft Visual C++) environments, CI-tested on Win32 & Win64. **NOTE**: LDC uses 64-bit double precision for `real`, unlike DMD, for MSVC `long double` compatibility. 32-bit MinGW support was almost complete at one stage but got neglected; a manageable number of adaptations (mainly in druntime) would probably suffice to support it again.
Android	Should be usable for ARMv7-A / AArch64 / i686 / x86_64, only rudimentarily CI-tested though (compilability). **NOTE**: Requires an LDC linked against our LLVM fork due to a custom TLS emulation scheme. Check out the Build D for Android article for how to get started.
iOS / tvOS / watchOS	Should be usable with 64-bit ARM (all druntime/Phobos unittests pass on iOS with LDC v1.21), only rudimentarily CI-tested though. The official prebuilt macOS package supports cross-compilation out-of-the-box, just use `-mtriple=arm64-apple-ios12.0`.
FreeBSD	Fine, CI-tested on x86_64. Requires LLD 9+ to pull in `libexecinfo` automatically when linking static druntime.
Other BSDs	Should be usable, not CI-tested though.
Solaris	Ditto.

...

- **Release**

https://github.com/ldc-developers/ldc/releases/tag/v1.30.0

▾ Assets 17

ldc-1.30.0-src.tar.gz	7.91 MB
ldc-1.30.0-src.zip	10.3 MB
ldc2-1.30.0-android-aarch64.tar.xz	30.1 MB
ldc2-1.30.0-android-armv7a.tar.xz	27.6 MB
ldc2-1.30.0-freebsd-x86_64.tar.xz	17.3 MB
ldc2-1.30.0-linux-aarch64.tar.xz	51.3 MB
ldc2-1.30.0-linux-x86_64.tar.xz	62.3 MB
ldc2-1.30.0-osx-arm64.tar.xz	65.5 MB
ldc2-1.30.0-osx-universal.tar.xz	136 MB
ldc2-1.30.0-osx-x86_64.tar.xz	72.8 MB
ldc2-1.30.0-windows-multilib.7z	50.1 MB
ldc2-1.30.0-windows-multilib.exe	53.1 MB
ldc2-1.30.0-windows-x64.7z	36 MB
ldc2-1.30.0-windows-x86.7z	35.1 MB
ldc2-1.30.0.sha256sums.txt	1.31 KB
Source code (zip)	
Source code (tar.gz)	

- **Src**
  https://github.com/ldc-developers/ldc/

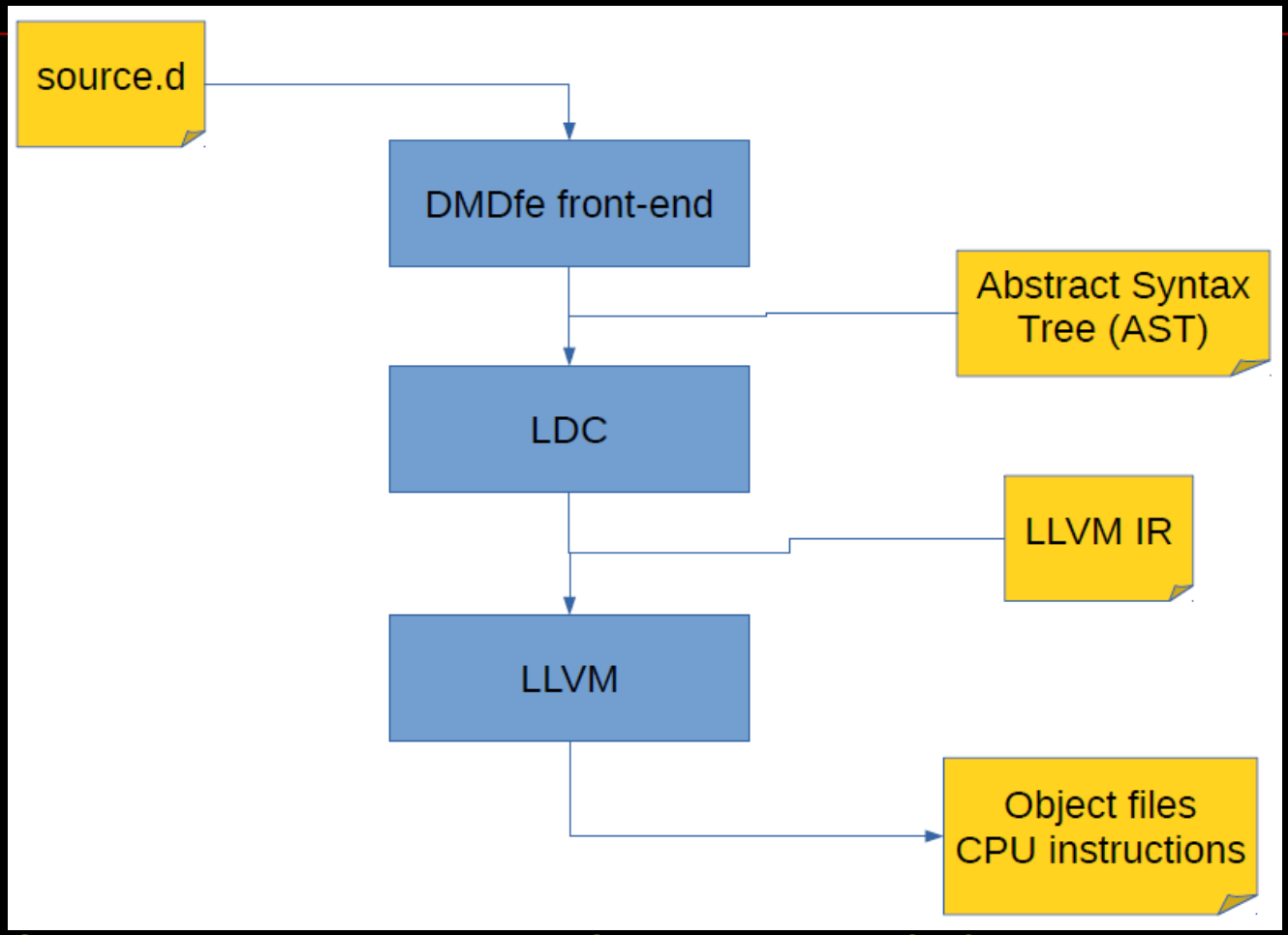
```
[mydev@fedora ldc-master]$ git branch
* master
[mydev@fedora ldc-master]$ git log -1
commit 8edd8de439ed007622b408de5540cf2b13d9c02f (HEAD → master, origin/master, origin/HEAD)
Author: Johan Engelen <jbc.engelen@gmail.com>
Date:   Sat Jul 23 12:06:44 2022 +0200

    Add --fno-discard-value-names : Do not discard value names in LLVM IR (#4012)
[mydev@fedora ldc-master]$
[mydev@fedora ldc-master]$ tokei
===============================================================================
 Language            Files        Lines         Code     Comments       Blanks
===============================================================================
 Assembly                1          114           56           41           17
 GNU Style Assembly      2          761          437          268           56
 Autoconf                9         1027          722          110          195
 BASH                    1           29           11           13            5
 Batch                   1           13           13            0            0
 C                     166        17804        12219         3378         2207
 C Header              165        34806        25773         5465         3568
 CMake                  19         3188         2330          540          318
 C++                   158        65996        48859         7914         9223
 D                    5261      1072367       771816       179533       121018
 JSON                    4         1318         1318            0            0
 Makefile               42         4771         3280          477         1014
 Module-Definition       9         1772         1765            6            1
 MSBuild                 1           34           34            0            0
 Objective-C             8          193          155            3           35
 Python                  1           17           12            2            3
 R                       2           79           57           11           11
 Shell                  61         1340          781          211          348
 Plain Text             33        24358            0        18199         6159
 YAML                    4          465          405           20           40
-------------------------------------------------------------------------------
 HTML                   87        16799        14278          984         1537
 ├─ CSS                 83        40863        33737            2         7124
 ├─ JavaScript           1            1            1            0            0
 (Total)                            57663        48016          986         8661
-------------------------------------------------------------------------------
 Markdown               25         2579            0         2036          543
 ├─ BASH                 1            2            2            0            0
 ├─ D                    1           25           14            9            2
 (Total)                            2606           16         2045          545
===============================================================================
 Total                6060      1290721       918075       219222       153424
===============================================================================
[mydev@fedora ldc-master]$
```

```
[mydev@fedora ldc-master]$ tree -L 2 -d .
.
├── cmake
│   └── Modules
├── dmd
│   ├── common
│   ├── res
│   └── root
├── docs
├── driver
├── gen
│   ├── dcompute
│   └── passes
├── ir
├── packaging
│   └── bash_completion.d
├── runtime
│   ├── druntime
│   ├── jit-rt
│   ├── phobos
│   └── TestRunnerTemplate
├── tests
│   ├── baremetal
│   ├── codegen
│   ├── compilable
│   ├── d2
│   ├── debuginfo
│   ├── driver
│   ├── dynamiccompile
│   ├── fail_compilation
│   ├── instrument
│   ├── linking
│   ├── PGO
│   ├── plugins
│   ├── sanitizers
│   ├── semantic
│   └── tools
├── tools
│   └── ldc-profdata
└── utils

38 directories
```

2) Internals
2.1 Overall
Workflow



Source: "LLVM-backed goodies in LDC", Johan Engelen, DConf 2018.

2.2 Front-end

- https://github.com/ldc-developers/ldc/tree/master/dmd

Directory structure

Folder	Purpose
dmd/	The dmd driver and front-end
dmd/backend/	Code generation for x86 or x86-64. Shared by the Digital Mars C compiler, but not LDC or GDC.
dmd/common/	Code shared by the front-end and back-end
dmd/root/	Meant as a portable utility library, but "it wasn't very good and the only project left using it is dmd".

Driver

File	Purpose
mars.d	The entry point. Contains `main`.
cli.d	Define the command line interface
dmdparams.d	DMD-specific parameters
globals.d	Define a structure storing command line options
dinifile.d	Parse settings from .ini file (`sc.ini` / `dmd.conf`)
vsoptions.d	Detect the Microsoft Visual Studio toolchain for linking
frontend.d	An interface for using DMD as a library
errors.d	Error reporting functionality
target.d	Manage target-specific parameters for cross-compiling (for LDC/GDC)
compiler.d	Describe a back-end compiler and implements compiler-specific actions

Lexing / parsing

File	Purpose
lexer.d	Convert source code into tokens for the D and ImportC parsers
entity.d	Define "\&Entity;" escape sequence for strings / character literals
tokens.d	Define lexical tokens.
parse.d	D parser, converting tokens into an Abstract Syntax Tree (AST)
cparse.d	ImportC parser, converting tokens into an Abstract Syntax Tree (AST)

Semantic analysis

Symbols and declarations

File	Purpose
dsymbol.d	Base class for a D symbol, e.g. a variable, function, module, enum etc.
identifier.d	Represents the name of a `Dsymbol`
id.d	Define strings for pre-defined identifiers (e.g. `sizeof`, `string`)
dscope.d	Define a 'scope' on which symbol lookup can be performed
dtemplate.d	A template declaration or instance
dmodule.d	Define a package and module
mtype.d	Define expression types such as `int`, `char[]`, `void function()`
arraytypes.d	For certain Declaration nodes of type `T`, provides aliases for `Array!T`
declaration.d	Misc. declarations of `alias`, variables, type tuples, `ClassInfo` etc.
denum.d	Defines `enum` declarations and enum members
attrib.d	Declarations of 'attributes' such as `private`, `pragma()`, `immutable`, `@UDA`, `align`, `extern(C++)` and more
func.d	Define a function declaration (includes function literals, `invariant`, `unittest`)
dversion.d	Defines a version symbol, e.g. `version = ident`, `debug = ident`

AST nodes

File	Purpose
ast_node.d	Define an abstract AST node class
astbase.d	Namespace of AST nodes that can be produced by the parser
astcodegen.d	Namespace of AST nodes of a AST ready for code generation
astenums.d	Enums common to DMD and AST
expression.d	Define expression AST nodes
statement.d	Define statement AST nodes
staticassert.d	Define a `static assert` AST node
aggregate.d	Define an aggregate (`struct` , `union` or `class`) AST node
dclass.d	Define a `class` AST node
dstruct.d	Define a `struct` or `union` AST node
init.d	Define variable initializers

AST visitors

File	Purpose
parsetimevisitor.d	General visitor for AST nodes
permissivevisitor.d	Subclass of ParseTimeVisitor that does not `assert(0)` on unimplemented nodes
strictvisitor.d	Visitor that forces derived classes to implement `visit` for every possible node
visitor.d	A visitor implementing `visit` for all nodes present in the compiler
transitivevisitor.d	Provide a mixin template with visit methods for the parse time AST
apply.d	Depth-first expression visitor
sapply.d	Depth-first statement visitor
statement_rewrite_walker.d	Statement visitor that allows replacing the currently visited node

...

2.3 Back-end

- https://github.com/ldc-developers/ldc/tree/master/gen

dcompute	[dcompute] add support for OpenCL image I/O	10 months ago
passes	Factor out legacy implementation details from Custom passes (#3977)	3 months ago
aa.cpp	Adapt to new TOK and EXP enum classes	5 months ago
aa.h	Adapt to new TOK and EXP enum classes	5 months ago
abi-aarch64.cpp	Simplify dmd/ldcbindings.{d,h}	10 months ago
abi-aarch64.h	Fix comments in abi-aarch-64.h (#3707)	15 months ago
abi-arm.cpp	Simplify dmd/ldcbindings.{d,h}	10 months ago
abi-arm.h	Replace old-school header guards by `#pragma once`	4 years ago
abi-generic.h	Fix insufficient alignment of BaseBitcastABIRewrite allocas (#3698)	15 months ago
abi-mips64.cpp	Adapt to TY enum class	11 months ago
abi-mips64.h	Replace old-school header guards by `#pragma once`	4 years ago
abi-nvptx.cpp	Merge pull request #3873 from kinke/no_reverse	5 months ago
abi-nvptx.h	Replace old-school header guards by `#pragma once`	4 years ago
abi-ppc.cpp	Adapt to TY enum class	11 months ago
abi-ppc.h	Replace old-school header guards by `#pragma once`	4 years ago
abi-ppc64le.cpp	Adapt to TY enum class	11 months ago
abi-ppc64le.h	Replace old-school header guards by `#pragma once`	4 years ago
abi-spirv.cpp	Merge pull request #3873 from kinke/no_reverse	5 months ago
abi-spirv.h	Replace old-school header guards by `#pragma once`	4 years ago
abi-win64.cpp	ABI: Handle extern(C++) delegates for 32-bit MSVC with __thiscall con...	5 months ago
abi-win64.h	Replace old-school header guards by `#pragma once`	4 years ago
abi-x86-64.cpp	Merge pull request #3873 from kinke/no_reverse	5 months ago
abi-x86-64.h	Replace old-school header guards by `#pragma once`	4 years ago
abi-x86.cpp	Merge pull request #3873 from kinke/no_reverse	5 months ago
abi-x86.h	Replace old-school header guards by `#pragma once`	4 years ago
abi.cpp	Merge pull request #3873 from kinke/no_reverse	5 months ago
abi.h	Merge pull request #3873 from kinke/no_reverse	5 months ago
arrays.cpp	codegen: Support {Not,Logical,Identity,Cmp}Exp with non-bool type (fo...	3 months ago
arrays.h	Pass isCfile parameter for all defaultInit() calls, for some extra im...	3 months ago
asm-gcc.cpp	Simplify dmd/ldcbindings.{d,h}	10 months ago

2.3.1 DCompute

- https://github.com/libmir/dcompute

DCompute: Native execution of D on GPUs and other Accelerators.

This project is a set of libraries designed to work with LDC to enable native execution of D on GPUs (and other more exotic targets of OpenCL such as FPGAs DSPs, hereafter just 'GPUs') on the OpenCL and CUDA runtimes. As DCompute depends on developments in LDC for the code generation, a relatively recent LDC is required, use 1.8.0 or newer.

There are four main parts:

- std: A library containing standard functionality for targetting GPUs and abstractions over the intrinsics of OpenCL and CUDA.
- driver: For handling all the compute API interactions and provide a friendly, easy-to-use, consistent interface. Of course you can always get down to a lower level of interaction if you need to. You can also use this to execute non-D kernels (e.g. OpenCL or CUDA).
- kernels: A set of standard kernels and primitives to cover a large number of use cases and serve as documentation on how (and how not) to use this library.
- tests: A framework for testing kernels. The suite is runnable with `dub test` (see `dub.json` for the configuration used).

Example

Kernel:

```
@kernel void saxpy(GlobalPointer!(float) res,
                   float alpha,
                   GlobalPointer!(float) x,
                   GlobalPointer!(float) y,
                   size_t N)
{
    auto i = GlobalIndex.x;
    if (i >= N) return;
    res[i] = alpha*x[i] + y[i];
}
```

Invoke with (CUDA):

```
q.enqueue!(saxpy)
    ([N,1,1],[1,1,1]) // Grid & block & optional shared memory
    (b_res,alpha,b_x,b_y, N); // kernel arguments
```

equivalent to the CUDA code

```
saxpy<<<1,N,0,q>>>(b_res,alpha,b_x,b_y, N);
```

For more examples and the full code see `source/dcompute/tests`.

2.3.2 CTFE

- https://en.wikipedia.org/wiki/Compile-time_function_execution

Compile-time function execution (or **compile time function evaluation**, or **general constant expressions**) is the ability of a compiler, that would normally compile a function to machine code and execute it at run time, to execute the function at compile time. This is possible if the arguments to the function are known at compile time, and the function does not make any reference to or attempt to modify any global state (is a pure function).

If the value of only some of the arguments are known, the compiler may still be able to perform some level of compile-time function execution (partial evaluation), possibly producing more optimized code than if no arguments were known.

Here's an example of compile time function evaluation in the D programming language:[3]

```
int factorial(int n) {
    if (n == 0)
       return 1;
    return n * factorial(n - 1);
}

// computed at compile time
enum y = factorial(0); // == 1
enum x = factorial(4); // == 24
```

This example specifies a valid D function called "factorial" which would typically be evaluated at run time. The use of `enum` tells the compiler that the initializer for the variables must be computed at compile time. Note that the arguments to the function must be able to be resolved at compile time as well.[4]

CTFE can be used to populate data structures at compile-time in a simple way (D version 2):

```
int[] genFactorials(int n) {
    auto result = new int[n];
    result[0] = 1;
    foreach (i; 1 .. n)
        result[i] = result[i - 1] * i;
    return result;
}

enum factorials = genFactorials(13);

void main() {}

// 'factorials' contains at compile-time:
// [1, 1, 2, 6, 24, 120, 720, 5_040, 40_320, 362_880, 3_628_800,
//  39_916_800, 479_001_600]
```

CTFE can be used to generate strings which are then parsed and compiled as D code in D.

- https://dlang.org/blog/2017/04/10/the-new-ctfe-engine/
- https://wiki.dlang.org/Compile-time_vs._compile-time

IV. D for Linux Kernel

1) Latest Trends

Moving to modern C

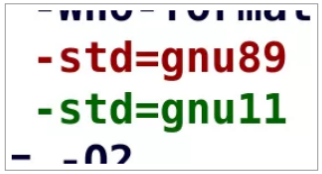
- Linux Kernel Moving Ahead With Going From C89 To C11 Code

Written by Michael Larabel in Linux Kernel on 28 February 2022 at 07:06 AM EST. 69 Comments

It looks like for the Linux 5.18 kernel cycle coming up it could begin allowing modern C11 code to be accepted rather than the current Linux kernel codebase being limited to the C89 standard.

Following mailing list discussions, Linus Torvalds entertained the idea of bumping the C version target from C89 up to C99. But it turns out with the current minimum version compiler requirements of the kernel and the condition of the current code, they can actually begin building the kernel with C11 in mind.

Thanks to Linux 5.15 raising the compiler requirement to GCC 5.1 and other recent improvements to the code-base, they can now begin safely building the Linux kernel using C11/GNU11 for its accepted C version.

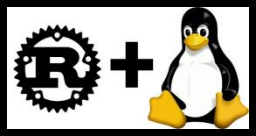


Allowing modern C code into the Linux kernel!

This morning Arnd Bergmann sent out the new patch allowing the Linux kernel to default to "*-std=gnu11*" in specifying the GNU dialect of C11. Thus moving forward the kernel will allow usage of nice C99/C11 features rather than being limited to C89. As this change already has the blessing of Linus Torvalds, it will likely go forward in the next kernel merge window assuming no fundamental issues are uncovered.

- https://lwn.net/Articles/885941/
- https://en.wikipedia.org/wiki/C11_(C_standard_revision)
- ...

Rust heads into Linux Kernel



CHINA OpenInfra Days

- **What's happening!**

  https://lwn.net/Articles/889924/ //Rustaceans at the border
  https://lwn.net/Articles/853423/ //Rust heads into the kernel?
  https://lwn.net/Articles/852704/ //Rust in the Linux kernel
  https://lwn.net/Articles/849849/ //Rust support hits linux-next
  https://lwn.net/Articles/829858/ //Supporting Linux kernel development in Rust
  …

- **https://github.com/Rust-for-Linux**

> The goal of this project is to add support for the Rust language to the Linux kernel. This repository contains the work that will be eventually submitted for review to the LKML.
>
> Feel free to contribute! To start, take a look at `Documentation/rust`.

- **Rust for Linux patch 9**

  https://www.phoronix.com/news/Rust-For-Linux-v9-Patches

Earlier this week saw the Rust for Linux v8 patches posted that introduced a number of new abstractions and expanding the Rust programming language integration to more areas of the kernel. Those patches amounted to 43.6k lines of new code while "Rust for Linux v9" was posted today and comes in at just 12.5k lines of new code.

Rust for Linux v9 is significantly smaller than the prior patches due to removing a lot of extra features and integration. The hope is now to take a more initial minimal route with the Rust for Linux integration until that initial mainlining and then from there can build things up with the enhanced integration and allowing more involved review/feedback of the various abstractions and subsystem-specific patches.

  https://lore.kernel.org/lkml/20220805154231.31257-1-ojeda@kernel.org/

```
89 files changed, 12548 insertions(+), 51 deletions(-)
```

- **https://www.phoronix.com/news/Rust-For-Linux-5.20-Possible (Linux 6.0)**

- **Memory Safety for the World's Largest Software Project**

  https://lwn.net/Articles/899164/

  Miguel Ojeda has posted an update on the Rust-for-Linux project.

  > This second year since the RFC we are looking forward to several milestones which hopefully we will achieve:
  >
  > - More users or use cases inside the kernel, including example drivers – this is pretty important to get merged into the kernel.
  > - Splitting the kernel crate and managing dependencies to allow better development.
  > - Extending the current integration of the kernel documentation, testing and other tools.
  > - Getting more subsystem maintainers, companies and researchers involved.
  > - Seeing most of the remaining Rust features stabilized.
  > - Possibly being able to start compiling the Rust code in the kernel with GCC.
  > - And, of course, getting merged into the mainline kernel, which should make everything else easier!

- **https://www.memorysafety.org/blog/memory-safety-in-linux-kernel/**
- **r-linux**

  https://github.com/bus1/r-linux

  Capability-based Linux Runtime

  The r-linux project provides direct access to the application programming interfaces of the linux kernel. This includes direct unprotected accessors to the kernel API, as well as rustified traits and functions to access the kernel API in a safe, capability-based way.

- **…**

2) D for a @safer Linux Kernel

- https://dconf.org/2019/talks/militaru.html

Abstract:

D is advertised as a safe and fast systems programming language. “Fast code, fast." is the mantra of the D community. But how fast is it really and how suitable for systems programming? We thought to investigate this by porting a Linux kernel driver to D, documenting the process, and assessing the results. The talk will delve into the details of integrating a D kernel module with the C-written kernel, discuss the difficulties encountered, and present the performance and safety benchmarks obtained. If you ever wondered if writing a Linux driver in D would be a good idea and how to do that, then this talk could be a good starting point for you.

Description

The talk aims to present the steps we took to port a Linux device driver to D, namely virtio_net. We will comment on the main difficulties we encountered and how we overcame them, the safety improvements we achieved using features implemented in the D language, and the performance and safety benchmarks obtained. There will be five big topics discussed:

- linking a D object file with the other Linux kernel objects
- porting C code to D: issues, design decisions and shortcomings
- a better C: the features we used to improve the code’s safety and how we did that
- assessing the D-written driver’s behaviour: benchmarks and results
- conclusions

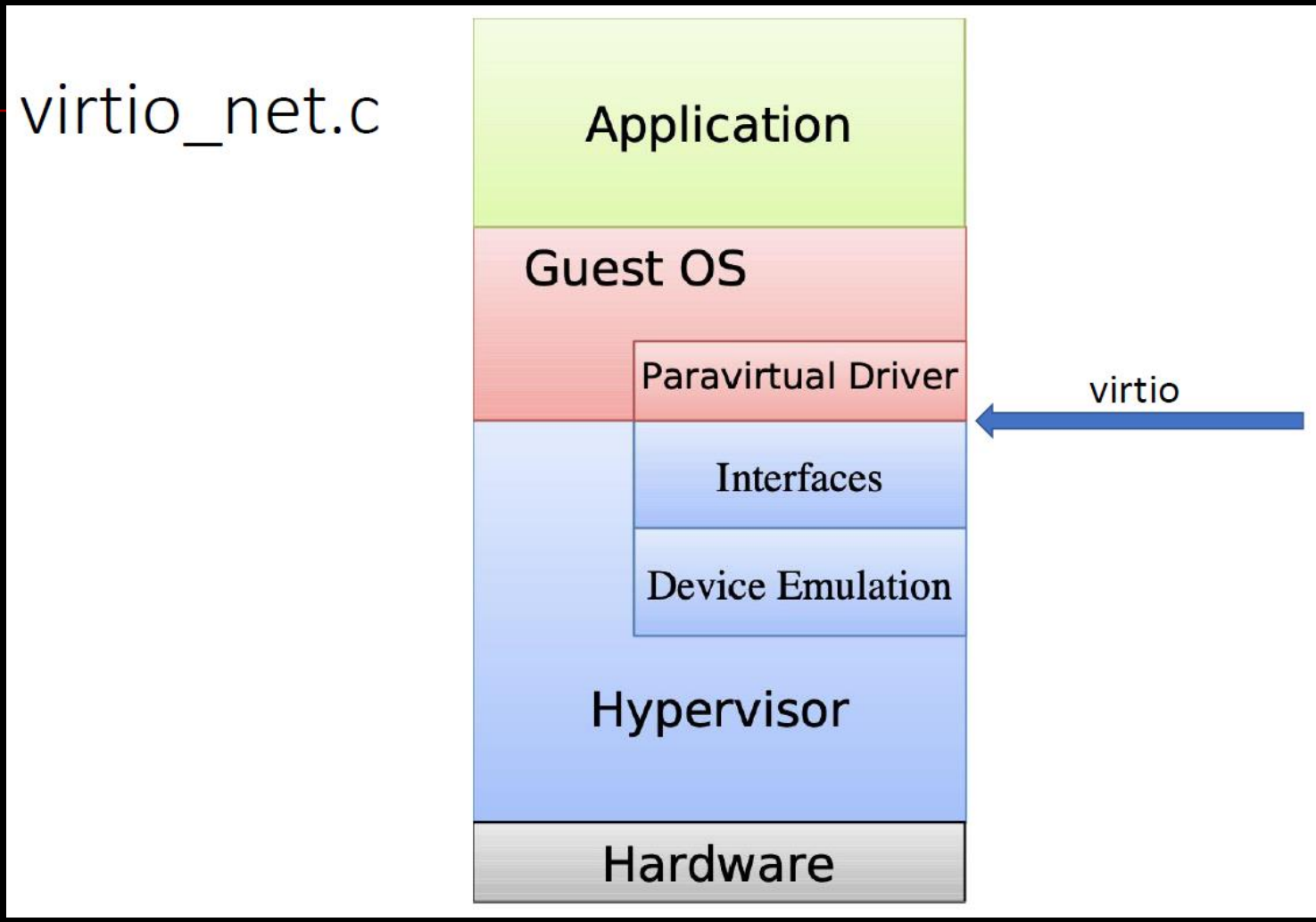
The talk offers a starting point and proposes a methodology for those who want to port or implement a Linux driver in D and, in the end, reflects on how suitable D is for kernel programming, how fast it is in such an environment, and what improvements it brings.

- Conclusions

- D can keep up with C in the Linux kernel
- better write a driver from the ground in D
- a compiler switch for the issues encountered

Example: Virtio Driver

- https://wiki.osdev.org/Virtio
-



Source: "D for a @safer Linux Kernel", Alexandru Militaru, DConf 2019.

- **Src**

https://github.com/alexandrumc/d-virtio/tree/master/drivers/net/dfiles

alexandrumc solved bounds checking fault — 209a397 on 30 Apr 2019 — History

..		
Makefile	added bounds checking support	3 years ago
bpf_prog_h.d	ported virtnet_info and control_buf structs	3 years ago
cache_h.d	continued the porting process	3 years ago
control_buf_h.d	ported virtnet_info and control_buf structs	3 years ago
cpu_state_h.d	continued the porting process	3 years ago
device_h.d	ported struct net_device	3 years ago
dvirtio_net.d	solved bounds checking fault	3 years ago
gfp_h.d	continued the porting process	3 years ago
kobject_h.d	ported struct device	3 years ago
link_state_h.d	continued the porting process	3 years ago
list_head_h.d	ported struct device	3 years ago
lockdep_map_h.d	ported struct device	3 years ago
mod_devicetable_h.d	automated build process	3 years ago
mutex_h.d	ported struct net_device	3 years ago
napi_struct_h.d	ported napi_struct	3 years ago
net_device_h.d	nearly finished porting the driver	3 years ago
old_cfunc.h	code ported in D	3 years ago
page_h.d	ported struct page	3 years ago
receive_queue_h.d	template virtnet_info, ported some heavily used functions	3 years ago
send_queue_h.d	template virtnet_info, ported some heavily used functions	3 years ago
sk_buff_h.d	small refactoring	3 years ago
sock_h.d	continued the porting process	3 years ago
spinlock_types_h.d	ported struct net_device	3 years ago
test.sh	automated testing process	3 years ago
uapi_h.d	nearly finished porting the driver	3 years ago
virtio_h.d	template virtnet_info, ported some heavily used functions	3 years ago
virtnet_info_h.d	added bounds checking support	3 years ago

https://github.com/alexandrumc/d-virtio/blob/master/drivers/net/dfiles/Makefile

alexandrumc added bounds checking support — Latest commit 9d25e18 on 24 Apr 2019 — History

1 contributor

27 lines (20 sloc) | 993 Bytes — Raw | Blame

```
1  CC=ldc2
2  #CFLAGS=-betterC -c -release -boundscheck=on -O1 -enable-inlining -disable-red-zone
3  CFLAGS=-betterC -c -release -boundscheck=on -enable-inlining -aarch64-enable-cond-br-tune -aarch64-enable-condopt
4  #CFLAGS=-betterC -c
5  ext=.o_shipped
6  CC_PATH=~/dlang/ldc-1.14.0/activate
7  SHELL := /bin/bash
8  SRC=$(wildcard *.d)
9  TAR=$(SRC:.d=.o_shipped)
10 D_FLAGS=-d-version=CONFIG_LOCKDEP,CONFIG_DEBUG_LOCK_ALLOC,CONFIG_DEBUG_SPINLOCK,CONFIG_NETPOLL,CONFIG_64BIT,CONFIG_MUTEX_SPIN_ON_OWNER,CONFIG_DEBUG_MUTEXES,CONFIG_SMP,CONFIG_N
11
12 all: build
13
14 build: $(TAR) copy
15
16 %.o_shipped: %.d
17         source $(CC_PATH) && $(CC) $(CFLAGS) $(D_FLAGS) $< -of=$@
18
19 copy:
20         mv $(TAR) ..
21
22 print:
23         echo $(TAR)
24
25 clean:
26         cd .. ; rm -f $(TAR); rm -f virtio_net_tmp* virtio_net.o
27
```

Second benchmark

- C Host to Guest: 398 Mbits/sec
- D-safe Host to Guest: 390 Mbits/sec
  - 2.25% decrease
- C Guest to Host: 182 Mbits/sec
- D-safe Guest to Host: 175 Mbits/sec
  - 4% decrease
- C Guest to Guest: 249 Mbits/sec
- D-safe Guest to Guest: 245 Mbits/sec
  - 1.6% decrease

Source: "D for a @safer Linux Kernel", Alexandru Militaru, DConf 2019.

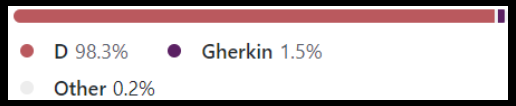
3) DPP

- https://github.com/atilaneves/dpp

Directly include C headers in D source code.

To directly `#include` C and C++ headers in D files and have the same semantics and ease-of-use as if the file had been `#included` from C or C++ themselves. Warts and all, meaning that C `enum` declarations will pollute the global namespace, just as it does "back home".

- **Languages**



C++ support

C++ support is currently limited. Including any header from the C++ standard library is unlikely to work. Simpler headers might, the probability rising with how similar the C++ dialect used is to C. Despite that, dpp currently does try to translate classes, templates and operator overloading. It's unlikely to work on production headers without judicious use of the `--ignore-cursor` and `--ignore-namespace` command-line options. When using these, the user can then define their own versions of problematic declarations such as `std::vector`.

Limitations

- Only known to work on Linux with libclang versions 6 and up. It might work in different conditions.
- When used on multiple files, there might be problems with duplicate definitions depending on imports. It is recommended to put all `#include`s in one `.dpp` file and import the resulting D module.
- Not currently able to translate Linux kernel headers.

- ...

Example

```
// c.h
#ifndef C_H
#define C_H

#define FOO_ID(x) (x*3)

int twice(int i);

#endif
```

```
// c.c
int twice(int i) { return i * 2; }
```

```
// foo.dpp
#include "c.h"
void main() {
    import std.stdio;
    writeln(twice(FOO_ID(5)));  // yes, it's using a C macro here!
}
```

At the shell:

```
$ gcc -c c.c
$ d++ foo.dpp c.o
$ ./foo
$ 30
```

- https://github.com/atilaneves/dpp/tree/master/examples

Automatic Integration of D Code With the Linux Kernel

CHINA OpenInfra Days

- https://ieeexplore.ieee.org/document/9638307

Abstract

Document Sections

I. Introduction

II. Background

III. DPP Architecture

IV. DPP Improvements

V. Evaluation

VI. Conclusion

Abstract:

The Linux kernel is implemented in C, an unsafe programming language, which puts the burden of memory management, type and bounds checking, and error handling in the hands of the developer. Hundreds of buffer overflow bugs have compromised Linux systems over the years, leading to endless layers of mitigations applied on top of C. In contrast, the D programming language offers automated memory safety checks and modern features such as OOP, templates and functional style constructs. In addition, interoper-ability with C is supported out of the box. However, to integrate a D module with the Linux kernel it is required that the needed C header files are translated to D header files. This is a tedious, time consuming, manual task. Although a tool to automate this process exists, called DPP, it does not work with the complicated, sometimes convoluted, kernel code. In this paper, we improve DPP with the ability to translate any Linux kernel C header to D. Our work enables the development and integration of D code inside the Linux kernel, thus facilitating a method of making the kernel memory safe.

- ...

V. DHDL

1) Overview

- https://github.com/luismarques/dhdl

The D Hardware Design Language.

- Languages

- Model

- Built-in data types and their semantics
- Expressing combinational circuits using D operators
- Maintaining state with registers
- The semantics of assignment and concurrent execution
- Grouping logic into hardware blocks

Source: https://dconf.org/2017/talks/marques.html

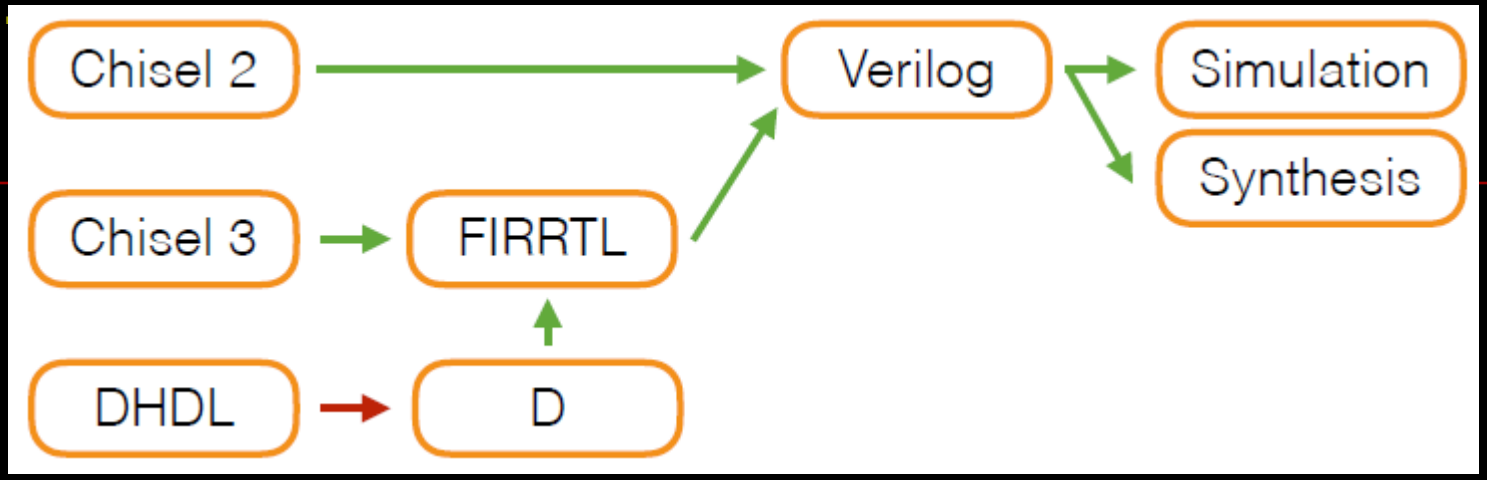
Design

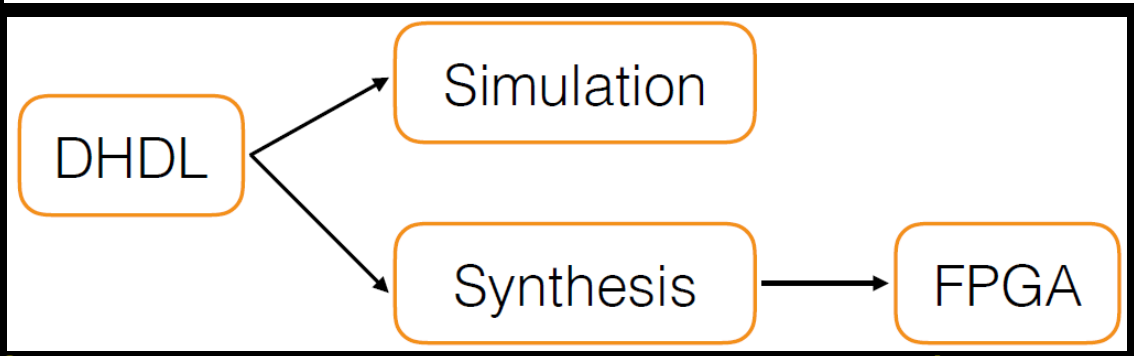
Creating Hardware

- Approach 1: High-level synthesis
  Approach 2: Event-based
  Approach 3: RTL builder
- VHDL & Verilog: Follow mostly approach 2
- DHDL (and Chisel, SpinalHDL, etc.): Follows mostly approach 3

Source: "DHDL: The D Hardware Description Language", Luís Marques, DConf 2017.

How it works





Source: "DHDL: The D Hardware Description Language", Luís Marques, DConf 2017.

Hello World

helloworld.dhdl

```
circuit HelloWorld
{
    port in bool a;
    port out bool b;

    b := a;
}
```


```
unittest
{
    auto hello = peekPokeTester!HelloWorld;

    hello.a = false;
    hello.eval();
    assert(hello.b == false);

    hello.a = true;
    hello.eval();
    assert(hello.b == true);
}
```

HelloWorld.fir

```
circuit HelloWorld :
    module HelloWorld :
        input reset : UInt<1>
        input clock : Clock
        input a : UInt<1>
        output b : UInt<1>

        b is invalid

        b <= a
```

HelloWorld.v

```
module HelloWorld(
  input   reset,
  input   clock,
  input   a,
  output  b
);
  assign b = a;
endmodule
```

Source: “DHDL: The D Hardware Description Language”, Luís Marques, DConf 2017.

VI. D-based DSLs

1) Overview

- …

1) Vox

- https://github.com/MrSmith33/vox

  Vox language compiler. AOT/JIT/Linker. Zero dependencies.
  A multiparadigm programming language inspired by D(60%), Jai(30%), and Zig(10%).

- Languages

- Main features

  - Fast compilation
  - Strong metaprogramming
  - Can be used for scripting and standalone programs (both JIT and AOT compilation)
  - No dependencies (except D compiler)

- Platforms

Supported:

- windows-x64 - host and target
- linux-x64 - host and target
- macos-x64 - only jit-mode

Planned:

- linux-arm64
- wasm
- windows-arm64?
- spirv (Vulkan/OpenCL/OpenGL shaders)

Differences from 

Similarities to the D language

- Same syntax for most portions of the language (struct, function, enum, for, while, if, UFCS, slices, arrays)
- Conditional compilation
- Templates, Variadic templates, and functions
- C interoperability
- Modules / Packages

Differences from the D language

- No GC, minimal runtime, no classes (only structs), no exceptions
- More compile-time features, faster CTFE
- Using templates for heavy calculations is discouraged, instead, CTFE can be used for introspection, and code generation.
- Macros (WIP)
- No C++ interoperability

Design

Project goals

- Strong focus on application extensions
- Maximize user productivity
- Maximize application performance
- AOT and JIT, plugin support, runtime compilation, embedded compiler, tiered compilation
- Static typing
- Great error messages
- Fast / Incremental compilation
- Minimize effort needed for installation, setup, integration
  - Minimal dependencies/encapsulate dependency into module or package
  - Runtime as a library/minimal runtime/no runtime
  - Embedding/extern C
  - Code driven compilation, extending compiler from inside of the program being compiled
- Processor intrinsics
- Conditional compilation
- CTFE, Templates, Introspection, Code generation

Examples

```
i32 fib(i32 number) {
    if (number < 1) return 0;
    if (number < 3) return 1;
    return fib(number-1) + fib(number-2);
}

struct Point {
    i32 x;
    i32 y;
}

T min[T](T a, T b) {
    if (a < b) return a;
    return b;
}
```

- Cross-platform hello world https://gist.github.com/MrSmith33/34a7557ad5ac23ebe6cf27bef15a39a6
- Fibonacci https://gist.github.com/MrSmith33/9645d9552b567fdbdc1a4d8822b4f1f7
- Fannkuch https://gist.github.com/MrSmith33/ac14e66a83b9d047793adede464ca1ef
- Roguelike tutorial using SDL2 - repo

- Voxel engine that uses Vox as a scripting language: Voxelman 2

https://github.com/MrSmith33/voxelman2

- Example of JIT compilation for amd64 from D code:

▼ code

```
// Source code
string source = q{
    void test(i32* array, i32 index, i32 value) {
        array[index] = value;
    }
};

// Error handling is omitted
Driver driver;
driver.initialize(jitPasses);
scope(exit) driver.releaseMemory;
driver.beginCompilation();
driver.addModule(SourceFileInfo("test", source));
driver.compile();
driver.markCodeAsExecutable();

// Get function pointer
auto testFun = driver.context.getFunctionPtr!(void, int*, int, int)("test");

// Use compiled function
int[2] val = [42, 56];
testFun(val.ptr, 1, 10);
assert(val[1] == 10);
```

VII. Pros & Cons

1) Overview

Pros

Development Mode	mainly community-driven https://github.com/dlang/DIPs/ https://dlang.org/foundation/
Productivity	a combination of C++/C/Java/Scala/Python…
Flexibility	e.g., hybird(auto/manually) memory management
Binary-compatible with C	essentially, this means that it should be possible to compile a C source file with a C compiler and combine the output into a program that is written in D and compiled with a D compiler, or vice versa
Low level programming	pointer, inline assembler…
Interop	easily interface with legacy code written in C/C++/Lua/Python…
Multi-paradigm	includes but not limited to imperative, functional, object-oriented, metaprogramming, and concurrent
Good support for HPC	e.g., DCompute in LDC, project Mir…
Built-in unit test support	QA friendly
…	…

- *Cons*

  Lack of popular open source projects;
  Lack of good application frameworks/libraries;
  Not as mature as commercial products, e.g. GC algorithms etc;
  It seems that some of the D-based open-source projects may not be well-tested outside X86;
  The ecosystem is still weak when compared with that of C++, Go, Rust…

should pay more attention on ARM & RISC-V.

2) Performance

Tests from Kostya

- https://github.com/kostya/benchmarks

Env

CPU: Intel(R) Xeon(R) E-2324G

Base Docker image: Debian GNU/Linux bookworm/sid

Language	Version
.NET Core	6.0.203
C#/.NET Core	4.1.0-5.22128.4 (5d10d428)
C#/Mono	6.8.0.105
Chez Scheme	9.5.4
Clojure	"1.11.1"
Crystal	1.4.1
D/dmd	v2.100.0
D/gdc	12.1.0
D/ldc2	1.29.0
Elixir	1.12.2
F#/.NET Core	12.0.1.0 for F# 6.0
Go	go1.18.2
Go/gccgo	12.1.0
Haskell	9.2.3
Java	18.0.1.1
Julia	v"1.7.3"
Kotlin	1.6.21
Lua	5.4.4
Lua/luajit	2.1.0-beta3
MLton	20210117
Nim	1.6.6
Node.js	v18.2.0
OCaml	4.14.0
PHP	8.1.5
Perl	v5.34.0
Python	3.10.4
Python/pypy	7.3.9-final0 for Python 3.9.12
Racket	"8.5"
Ruby	3.1.2p20
Ruby/jruby	9.3.4.0
Ruby/truffleruby	22.1.0
Rust	1.61.0
Scala	3.1.2
Swift	5.6.1
Tcl	8.6
V	0.2.4 a3c0a9b
Vala	0.56.1
Zig	0.9.1
clang/clang++	13.0.1
gcc/g++	12.1.0

Measurements

The measured values are:

- time spent for the benchmark execution (loading required data and code self-testing are not measured);
- memory consumption of the benchmark process, reported as `base` + `increase`, where `base` is the RSS before the benchmark and `increase` is the peak increase of the RSS during the benchmark;
- energy consumption of the CPU package during the benchmark: PP0 (cores) + PP1 (uncores like GPU) + DRAM. Currently, only Intel CPU are supported via the powercap interface.

All values are presented as: `median` ± `median absolute deviation`.

UPDATE: 2022-05-30

Some results

Base64

Testing base64 encoding/decoding of the large blob into the newly allocated buffers.

Base64

Language	Time, s	Memory, MiB	Energy, J
C/clang (aklomp)	$0.100_{\pm 0.001}$	$2.04_{\pm 00.04} + 0.00_{\pm 00.00}$	$4.73_{\pm 00.07}$
C/gcc (aklomp)	$0.101_{\pm 0.000}$	$2.09_{\pm 00.02} + 0.00_{\pm 00.00}$	$4.80_{\pm 00.05}$
Rust	$0.969_{\pm 0.000}$	$2.58_{\pm 00.03} + 0.01_{\pm 00.00}$	$39.56_{\pm 00.22}$
V/clang	$0.995_{\pm 0.001}$	$1.91_{\pm 00.02} + 0.00_{\pm 00.00}$	$37.66_{\pm 00.14}$
C/clang	$0.997_{\pm 0.000}$	$2.01_{\pm 00.04} + 0.00_{\pm 00.00}$	$37.14_{\pm 00.36}$
C/gcc	$1.012_{\pm 0.000}$	$2.05_{\pm 00.04} + 0.00_{\pm 00.00}$	$37.95_{\pm 00.09}$
Nim/clang	$1.024_{\pm 0.001}$	$2.82_{\pm 00.03} + 4.44_{\pm 00.00}$	$41.08_{\pm 00.18}$
D/ldc2	$1.074_{\pm 0.003}$	$3.58_{\pm 00.03} + 3.58_{\pm 00.00}$	$45.02_{\pm 00.50}$
Nim/gcc	$1.081_{\pm 0.002}$	$2.31_{\pm 00.03} + 4.44_{\pm 00.06}$	$42.48_{\pm 00.56}$
V/gcc	$1.092_{\pm 0.000}$	$1.63_{\pm 00.01} + 0.44_{\pm 00.04}$	$42.03_{\pm 00.17}$

Json

Testing parsing and simple calculating of values from a big JSON file.

Few notes:

- gason mutates input strings;
- simdjson requires input strings with batch of trailing zeros: a special zero padding for SIMD instructions;
- DAW JSON Link "NoCheck" skips some JSON structure correctness checks;
- DAW JSON Link, gason, default (not "Precise") RapidJSON, and D implementations except Mir-based have some inaccuracies in number parsing:
  - DAW JSON Link's number parsing issue
  - gason's number parsing issue
  - D stdlib number parsing issue

Json

Language	Time, s	Memory, MiB	Energy, J
C++/g++ (simdjson On-Demand)	$0.067_{\pm 0.000}$	$113.45_{\pm 00.09} + 59.81_{\pm 00.00}$	$2.78_{\pm 00.03}$
C++/clang++ (simdjson On-Demand)	$0.068_{\pm 0.000}$	$112.55_{\pm 00.07} + 60.36_{\pm 00.06}$	$2.83_{\pm 00.01}$
C++/clang++ (DAW JSON Link NoCheck)	$0.077_{\pm 0.000}$	$112.44_{\pm 00.03} + 0.00_{\pm 00.00}$	$3.14_{\pm 00.01}$
C++/g++ (DAW JSON Link NoCheck)	$0.079_{\pm 0.000}$	$113.19_{\pm 00.03} + 0.00_{\pm 00.00}$	$3.16_{\pm 00.02}$
C++/g++ (DAW JSON Link)	$0.087_{\pm 0.000}$	$113.22_{\pm 00.07} + 0.00_{\pm 00.00}$	$3.58_{\pm 00.03}$
C++/clang++ (DAW JSON Link)	$0.089_{\pm 0.000}$	$112.44_{\pm 00.02} + 0.00_{\pm 00.00}$	$3.67_{\pm 00.04}$
Rust (Serde Custom)	$0.107_{\pm 0.000}$	$111.99_{\pm 00.05} + 0.00_{\pm 00.00}$	$4.56_{\pm 00.02}$
Rust (Serde Typed)	$0.112_{\pm 0.000}$	$111.97_{\pm 00.07} + 12.03_{\pm 00.00}$	$4.72_{\pm 00.01}$
C++/g++ (gason)	$0.133_{\pm 0.000}$	$113.14_{\pm 00.05} + 96.80_{\pm 00.06}$	$5.28_{\pm 00.03}$
C++/clang++ (simdjson DOM)	$0.136_{\pm 0.001}$	$112.48_{\pm 00.03} + 177.15_{\pm 00.06}$	$5.79_{\pm 00.03}$
C++/g++ (simdjson DOM)	$0.138_{\pm 0.000}$	$113.01_{\pm 00.09} + 176.39_{\pm 00.19}$	$5.88_{\pm 00.01}$
D/ldc2 (Mir Asdf DOM)	$0.146_{\pm 0.000}$	$112.73_{\pm 00.04} + 61.27_{\pm 00.03}$	$6.09_{\pm 00.02}$

...

Web Framework Benchmarks

CHINA OpenInfra Days

- https://www.techempower.com/benchmarks/

Env

Environment details

This project measures performance in two common deployment scenarios: cloud instances and physical hardware. To-date, each round has used a single representative environment for each of these scenarios. The particular specifications of the environments have evolved over time as shown below.

Cloud environments

Azure (rounds 13 onward)

Microsoft Azure D3v2 instances; switched gigabit Ethernet.

AWS (rounds 1 through 12)

Amazon EC2 c3.large instances (2 vCPU each); switched gigabit Ethernet (m1.large was used through Round 9).

Physical hardware environments

Citrine (rounds 16 onward)

Three homogeneous Dell R440 servers each equipped with an Intel Xeon Gold 5120 CPU, 32 GB of memory, and an enterprise SSD. Dedicated Cisco 10-gigabit Ethernet switch. Provided by Microsoft.

ServerCentral (rounds 13 through 15)

Dell R910 (4x 10-Core Intel Xeon E7-4850 CPUs) application server; Dell R710 (2x 4-Core Intel Xeon E5520 CPUs) database server; switched 10-gigabit Ethernet. Provided by ServerCentral.

Peak (rounds 9 through 12)

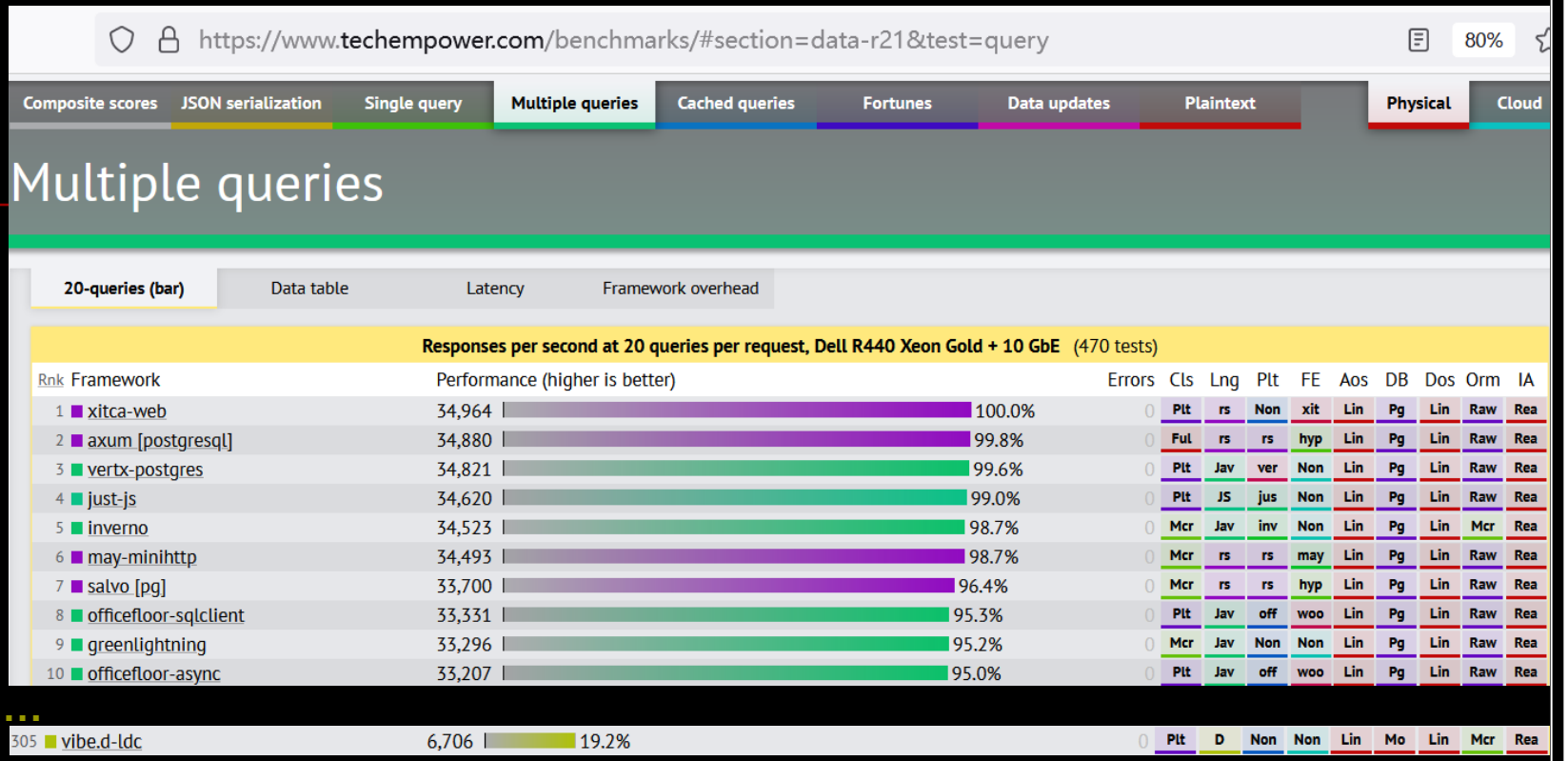
Dell R720xd dual Intel Xeon E5-2660 v2 (40 HT cores) with 32 GB memory; database servers equipped with SSDs in RAID; switched 10-gigabit Ethernet. Provided by Peak Hosting.

i7 (rounds 1 through 8)

In-house Intel Sandy Bridge Core i7-2600K workstations with 8 GB memory (early 2011 vintage); database server equipped with Samsung 840 Pro SSD; switched gigabit Ethernet.

https://github.com/TechEmpower/FrameworkBenchmarks/wiki/Project-Information-Framework-Tests-Overview

Some results



Composite Framework Scores

Each framework's peak performance in each test type (shown in the colored columns below) is multiplied by the weights shown above. The results are then summed to yield a weighted score. Only frameworks that implement all test types are included. 142 total frameworks ranked, 139 visible, 3 hidden by filters. See filter panel above.

Rnk	Framework	JSON	1-query	20-query	Fortunes	Updates	Plaintext	Weighted score	
1	just	1,526,714	673,201	34,620	538,414	24,454	6,982,125	**8,453**	100.0%
2	may-minihttp	1,546,221	642,348	34,493	520,976	24,192	7,023,484	**8,334**	98.6%
3	xitca-web	1,207,053	638,244	34,964	587,955	24,488	6,996,736	**8,287**	98.0%
4	drogon	1,086,998	622,274	29,935	616,607	21,877	5,969,800	**7,801**	92.3%
5	actix	1,498,561	512,830	29,198	512,422	20,635	7,017,232	**7,667**	90.7%
6	officefloor	1,374,439	558,932	33,331	432,309	23,691	6,383,827	**7,492**	88.6%
7	asp.net core	1,306,635	483,762	26,350	458,677	19,644	7,023,107	**7,077**	83.7%
8	salvo	1,082,630	631,785	33,700	542,547	23,733	1,928,951	**7,061**	83.5%
9	axum	847,891	612,714	34,880	498,541	24,324	3,780,458	**6,982**	82.6%
10	wizzardo-http	1,479,464	630,207	31,770	307,614	17,393	7,013,230	**6,851**	81.0%
...									
92	vibed	335,478	97,997	6,706	52,043	3,344	586,029	**1,187**	14.1%
93	fastapi	167,592	65,123	12,611	50,877	5,803	156,541	**1,184**	14.0%
...									

compiler-benchmark

- https://github.com/nordlow/compiler-benchmark

Sample run output

The output on my AMD Ryzen Threadripper 3960X 24-Core Processor running Ubuntu 20.04 for the sample call

```
./benchmark --function-count=200 --function-depth=200 --run-count=1
```

results in the following table (copied from the output at the end).

Lang-uage	Temp-lated	Check Time [us/fn]	Compile Time [us/fn]	Build Time [us/fn]	Run Time [us/fn]	Check RSS [kB/fn]	Build RSS [kB/fn]	Exec Version	Exec Path
Vox	No	1.5 (best)	N/A	5.2 (3.3x)	42 (1.2x)	1.1 (2.8x)	3.6 (8.1x)	master	vox
Vox	Yes	2.0 (1.4x)	N/A	6.1 (3.9x)	65 (1.8x)	2.0 (5.1x)	4.4 (9.9x)	master	vox
D	No	6.3 (4.2x)	13.4 (7.4x)	17.9 (11.4x)	72 (2.0x)	4.6 (11.5x)	12.2 (27.2x)	v2.097.0-275-g357bc9d7a	dmd
D	No	7.4 (5.0x)	90.8 (49.9x)	99.6 (63.5x)	219 (6.2x)	5.7 (14.3x)	19.7 (43.8x)	1.26.0	ldmd2
D	No	6.4 (4.3x)	240.5 (132.3x)	237.5 (151.5x)	40 (1.1x)	4.5 (11.2x)	19.2 (42.6x)	10.3.0	gdc

D	Yes	12.7 (8.5x)	21.9 (12.0x)	25.9 (16.5x)	64 (1.8x)	13.0 (32.5x)	21.4 (47.6x)	v2.097.0-275-g357bc9d7a	dmd
D	Yes	14.2 (9.6x)	102.0 (56.1x)	110.6 (70.6x)	302 (8.6x)	14.9 (37.4x)	29.3 (65.3x)	1.26.0	ldmd2
D	Yes	12.4 (8.3x)	287.4 (158.0x)	286.8 (182.9x)	56 (1.6x)	13.2 (33.1x)	28.3 (63.0x)	10.3.0	gdc
C	No	1.8 (1.2x)	1.8 (best)	1.6 (best)	44 (1.3x)	0.4 (best)	0.4 (best)	0.9.27	tcc
C	No	5.3 (3.5x)	N/A	N/A	N/A	1.7 (4.2x)	N/A	unknown	cproc
C	No	8.2 (5.5x)	274.2 (150.8x)	282.0 (179.8x)	55 (1.6x)	2.9 (7.4x)	14.3 (31.9x)	9.3.0	gcc
C	No	8.2 (5.5x)	273.6 (150.4x)	278.8 (177.8x)	54 (1.5x)	3.0 (7.5x)	14.3 (31.8x)	9.3.0	gcc-9
C	No	6.0 (4.0x)	220.4 (121.2x)	224.9 (143.4x)	55 (1.6x)	2.8 (7.1x)	14.3 (31.9x)	10.3.0	gcc-10
C	No	14.7 (9.9x)	121.9 (67.0x)	125.9 (80.3x)	1045 (29.8x)	1.8 (4.4x)	9.4 (20.9x)	10.0.0-4	clang-10
C	No	15.6 (10.5x)	121.7 (66.9x)	124.7 (79.5x)	376 (10.7x)	1.9 (4.7x)	9.4 (21.0x)	11.0.0-2	clang-11

...

3) Ecosystem

- https://dlang.org/orgs-using-d.html

- https://github.com/dlang-community/awesome-d
- http://wiki.dlang.org/Libraries_and_Frameworks
- https://wiki.dlang.org/IDEs
- http://code.dlang.org/
- https://wiki.dlang.org/Open_Source_Projects
- https://github.com/trending/d
- https://github.com/dlang-community
- ...

3.1 weka.io

- https://www.weka.io
- https://www.weka.io/why-weka/

CHINA OpenInfra Days

About the Weka.io product

- "Software only" storage product
- Low latency, high performance
- Written in D
- About 280,000 LoC
  - Not including 114,663 lines in a single auto-generated file.
- Compiled using waf

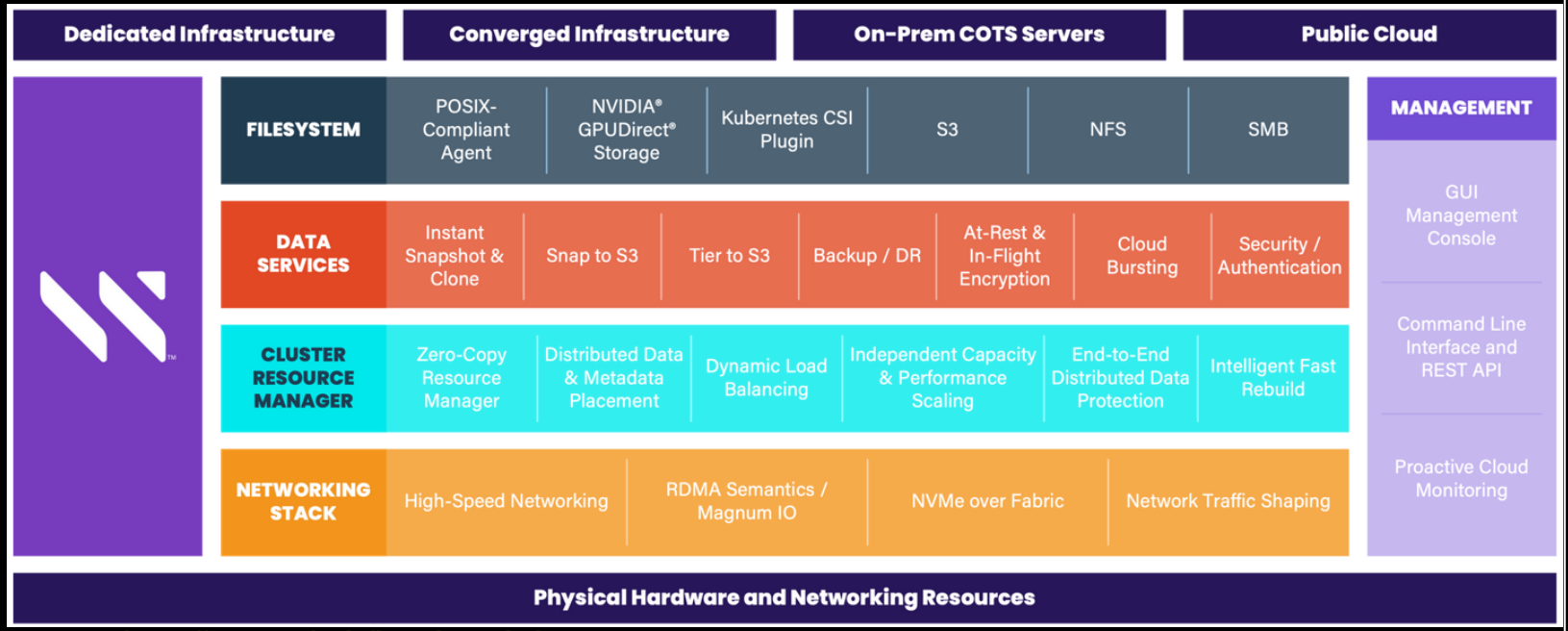
More About the Code

- Internally called "wekapp"
- Extremely latency sensitive
  - As little GC as possible
  - As few system calls as possible
- Performance sensitive
  - As little copying of data as possible
- Micro-threading (Fibers) based

What do we care about?

- Safety
- Performance
- Brevity
- Ability to manage complexity

Source: "Announcing Mecca", Shachar Shemesh, DConf 2018.



Source: https://www.weka.io/how-it-works/

- https://github.com/dlang-community/mecca

3.2 D in China

- https://www.d-programming-language-china.org/ (inactive)
- http://ddili.org/ders/d.en/index.html

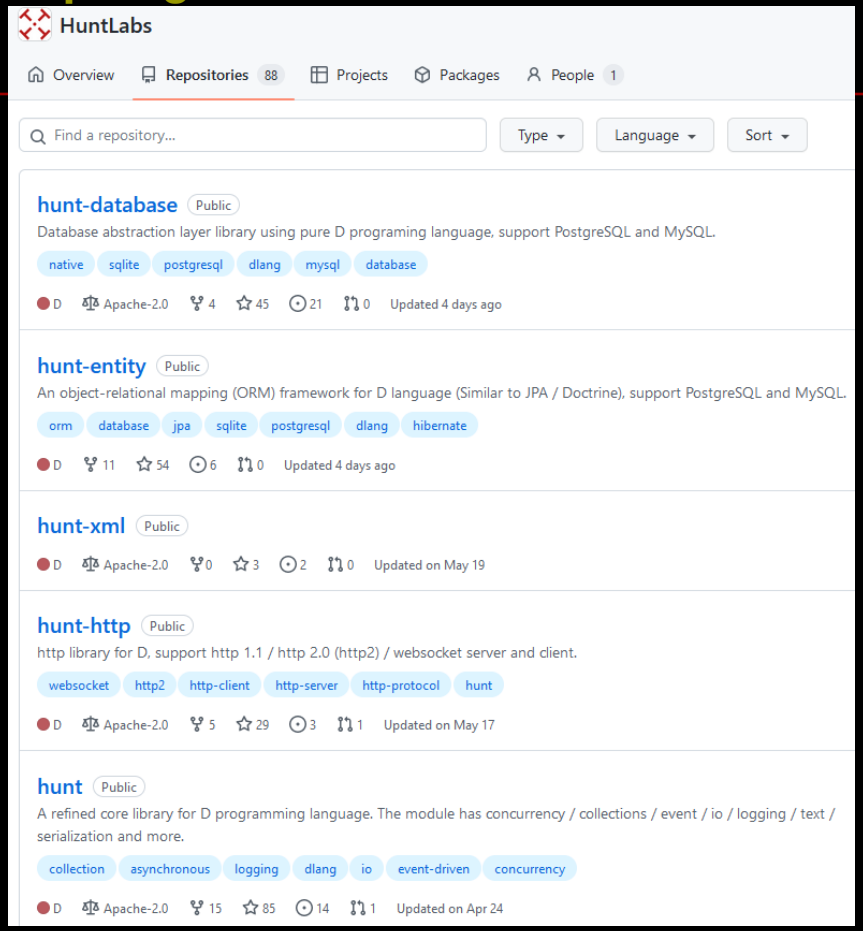
Random stats of the day:

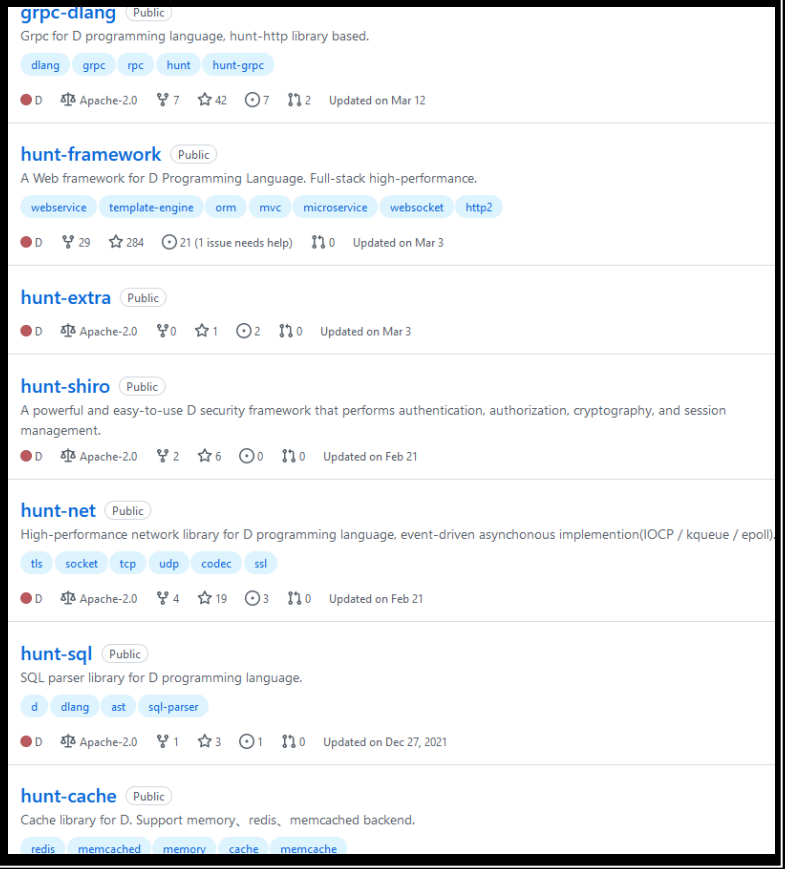
Location	Pages	Hits	Bandwidth
United States	34,237	42,608	1.34 GB
China	28,616	29,040	543.10 MB
Turkey	16,121	46,814	929.62 MB
Russian Federation	10,205	12,616	525.24 MB
Netherlands	8,559	8,747	148.16 MB
Norway	7,247	7,324	79.20 MB
Thailand	7,045	7,052	78.29 MB
Germany	6,172	7,734	495.69 MB
Brazil	5,272	5,604	128.59 MB

[...]

Putao

- https://www.huntlabs.net
- https://github.com/huntlabs





...

4) D 3.0

- https://github.com/dlang/vision-document
- https://forum.dlang.org/post/bpwczoipsprmfmlhjgwf@forum.dlang.org
- …

VIII. My ideas

A desired system language from our perspective

- Comparable performance to C/C++/Rust/Go.
- Guarantee memory-safety and thread-safety as Rust.
- Low learning curve and high productivity, especially when compared to C++/Rust, while close to the level of Python/Java is mostly preferred.
- Support Multi-paradigm programming, especially for Functional and Metaprogramming, as well as Concurrent.
- Good interoperability for most of the popular programming languages.
- Built-in support for concurrency that closer to Go.
- A self-hosting compilers is mostly preferred.
- The ability of bare-metal programming.
- Can be used for Linux Kernel development, and more.
- Natively support for Heterogeneous Parallel Computing.
- Easier to implement new DSLs base on it.
- With high-level abstraction ability for HW/SW Co-design, Co-development, Co-simulation, and Co-debugging etc.
- Come with a certain foundation of ecosystem.

Through a comprehensive evaluation, we do think  has tremendous potential to meet all of our needs to a great extent.

And let's start to build more progressive enhancements to  since 2022!

1) Ideas

1. Leverage  in a HW-SW co-designed Edge Computing Infrastructure

- Please look forward to an upcoming talk -- "The 3rd round discussion on eBPF-centric new approach for Hyper-Converged Infrastructure & Edge Computing".

  And go back in history, you may refer to my previous talks:

  1. "Rethinking Hyper-Converged Infrastructure for Edge Computing“ at OpenInfra Days China 2019(Shanghai).
  2. "Revisiting the eBPF-centric new approach for Hyper-Converged Infrastructure & Edge Computing” at K+ Summit 2021(Shanghai).

2. Re-implement LLVM with the selected platforms, features, and targets in 

3. An enhanced LDC version with better support for ARM, RISC-V, Wasm, and eBPF

4. More lightweight  runtimes like LWDR(https://github.com/hmmdyl/LWDR) for resource-limited computing devices

5. Add support for  in GraalVM (https://www.graalvm.org/)

6. Extend Vox to support more platforms like ARM, RISC-V, Wasm, eBPF, and SPIR-V

---

7. Add a  implementation of the Jupyter kernel protocol (https://github.com/jupyter/jupyter/wiki/Jupyter-kernels)

…

IX. Wrap-up

- A New Golden Age for Computer Architecture!
  A New Golden Age for Compiler Design!
  A New Golden Age for System Programming Language!

- We do think  is one of the best candidate for next generation system programming, especially in the HW-SW co-designed Edge Computing Infrastructure which has more freedom to innovate.

-  has great potential, but still has a long way to go…

- You may look forward to our follow-ups "Revisiting D as a better system programming language", "Rethinking D for HW-SW co-designed system", and "The first exploration of Smart Runtime" with more supplementary materials and detailed explanation in the near future.

CHINA
OpenInfra Days

Q & A

2022 | OpenInfra Days China
中国开源基础设施活动日

致谢！

李枫

hkli2013@126.com

CHINA
OpenInfra Days

Reference

Slides/materials from many and varied sources:

- http://en.wikipedia.org/wiki/
- http://www.slideshare.net/
- https://en.wikipedia.org/wiki/Memory_safety
- https://dlang.org/blog/2022/01/05/new-year-dlang-news-hello-2022/
- https://github.com/readme/featured/functional-programming
- https://jonathan2251.github.io/lbd/llvmstructure.html
- https://github.com/rui314/mold
- https://wiki.dlang.org/Commonly-Used_Acronyms
- https://dlang.org/blog/2019/10/15/my-vision-of-ds-future/
- https://dlang.org/blog/2022/06/21/dip1000-memory-safety-in-a-modern-system-programming-language-pt-1/
- https://insights.dice.com/2021/08/05/4-exotic-programming-languages-popular-with-malware-developers/
- https://p0nce.github.io/d-idioms/
- https://en.wikibooks.org/wiki/D_Programming/Garbage_collector
- https://ziglang.org/learn/why_zig_rust_d_cpp/
- https://lwn.net/Articles/686602/
- https://www.memorymanagement.org/mmref/lang.html
- https://aradaelli.com/blog/why-i-like-d/

- https://caiorss.github.io/C-Cpp-Notes/cpp-alternatives.html
- https://mcturra2000.wordpress.com/2021/12/31/my-goal-is-to-use-dlang-in-2022/
- …