VAST: MLIR Compiler for C/C++

Henrich Lauko, Lukáš Korenčik, and Robert Konicar

Fork me on GitHub
github.com/trailofbits/vast

VAST is a Program Analysis–Focused Compiler

Fine-Grained Steps

Various abstraction levels are useful for different program analyses. VAST enables viewing source code at various stages of translation from AST to LLVM IR. Each step of the LLVM code generation process is modeled as a distinct pass or a dialect.

Provenance

To link results of low-level analysis to the source code or to incorporate high-level structural insights into low-level analysis, it is essential for VAST dialects to maintain bidirectional provenance information across representations.

Program Abstractions

Not all information is necessary for specific analyses. A different source view can yield more precise results and simplify analysis design. For that, VAST supports user-defined program abstractions (dialects) compatible with the rest.

Compilation

VAST is designed as a Clang driver wrapper. It introduces a new AST consumer for MLIR generation, guided by two key principles:

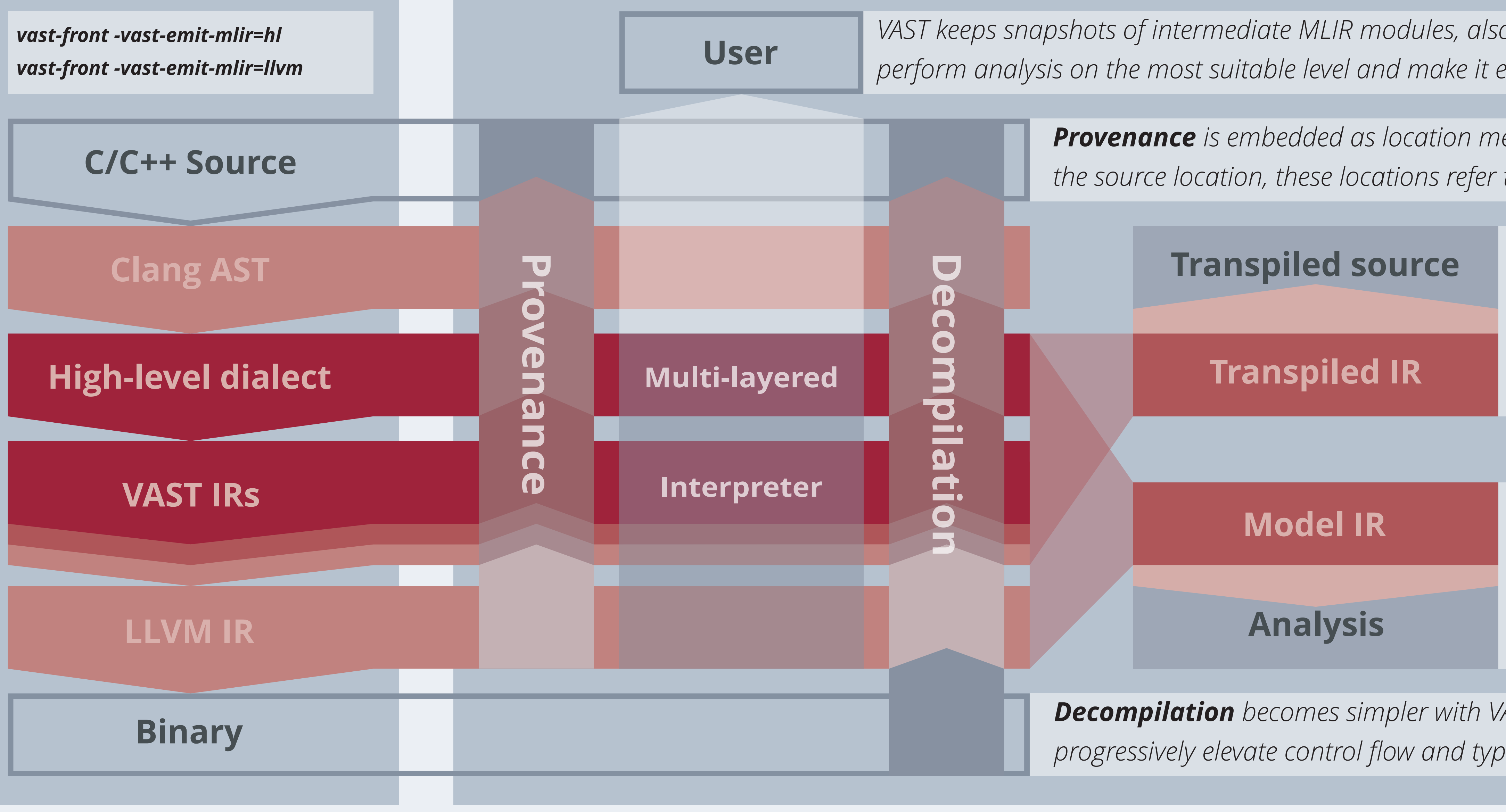
1. Fine-grained code generation: VAST differentiates itself by breaking down the LLVM codegen process into distinct MLIR passes and dialects, such as type desugaring or ABI translation. This approach not only facilitates easier tracing of code provenance but also allows stopping at the most suitable representation for program analysis.

2. Commutativity of independent steps: This allows independent codegen steps to be interchangeable, fostering the creation of interleaved high-level and low-level code representations.

VAST leverages both specialized VAST dialects and standard MLIR dialects, culminating in the generation of LLVM IR. Future developments aim to incorporate ClangIR as a target dialect.

vast-front -vast-emit-mlir=hl
vast-front -vast-emit-mlir=llvm

Applications



VAST keeps snapshots of intermediate MLIR modules, also called the Tower of IRs. These allow us to perform analysis on the most suitable level and make it easier to link analysis results back to the user.

***Provenance** is embedded as location metadata in operations. Rather than directing to the source location, these locations refer to the prior snapshots of generated MLIR.*

MLIR provides tooling to create IRs for transpilation to languages like Rust or newer C/C++ versions. VAST can serve as a generator for the most suitable IR for this task.

***Modeling IRs** streamline the program analysis by eliminating unnecessary details, focusing on specific aspects like aliasing or function calls for devirtualization.*

***Decompilation** becomes simpler with VAST, as it offers IRs for granular steps. We can progressively elevate control flow and type information to more expressive dialects.*

MLIR Dialects

***High-level dialect** resembles a Clang AST–like dialect and serves as the starting point for VAST, retaining as much information as possible for later stages of code generation.*

***Builtin dialect** specifies Clang's C/C++ built-in operations and types.*

***ABI dialect** describes the mapping between high-level types and the LLVM ABI.*

***Low-level dialect** is akin to an LLVM dialect, designed to be compatible with high-level structured control flow and high-level types.*

***Core dialect** defines generic interfaces and types used in VAST, like symbols, functions, or scopes.*

***Meta dialect** enables operations to be tagged with user-defined locations (IDs) and connects operations across the layers of the Tower of IRs.*

***Analyses dialects** represent programs in simplified representation for specific scenarios like points-to analysis, devirtualization, invariants, etc.*

***Future dialects** are planned to incorporate more specific information, such as concurrency, libc, library-specific abstractions, lifetimes, and more.*

Tower of IRs

```
#include <stdint.h>
uint32_t gcd(uint32_t a, uint32_t b) {
  while (b ≠ 0) {
    uint32_t temp = b;
    b = a % b;
    a = temp;
  }
  return a;
}
```

Each operation in the Tower of IRs employs MLIR locations to reference the preceding layer, depicted here through arrows for illustrative purposes only, with just a few examples shown.

Codegen through Clang AST

vast-front -vast-emit-mlir=hl

```
hl.typedef "uint32_t" : !hl.int<unsigned>
hl.func @gcd(
  %arg0: !hl.lvalue<!hl.typedef<"uint32_t">>,
  %arg1: !hl.lvalue<!hl.typedef<"uint32_t">>
) → !hl.elaborated<!hl.typedef<"uint32_t">> {
  hl.while {
    %2 = hl.ref %arg1 : !hl.lvalue<!hl.typedef<"uint32_t">>
    %3 = hl.implicit_cast %2 LValueToRValue
    %4 = hl.const #core.integer<0> : !hl.int
    %5 = hl.implicit_cast %4 IntegralCast
    %6 = hl.cmp ne %3, %5
    hl.cond.yield %6 : !hl.int
  } do {
    %2 = hl.var "temp" = { /* init */ }
    %3 = hl.ref %arg0 : !hl.lvalue<!hl.typedef<"uint32_t">>
    %4 = hl.implicit_cast %3 LValueToRValue
    %5 = hl.ref %arg1 : !hl.lvalue<!hl.typedef<"uint32_t">>
    %6 = hl.implicit_cast %5 LValueToRValue
    %7 = hl.urem %4, %6
    %8 = hl.assign %7 to %3
     /* ... */
  }
  %0 = hl.ref %arg0 : !hl.lvalue<!hl.typedef<"uint32_t">>
  hl.return %0
}
```

Simplify

vast-front -vast-emit-mlir=hl -vast-simplify

```
module attributes {dlti.dl_spec = #dlti.dl_spec<
  #dlti.dl_entry<!hl.int<unsigned >, vast.abi_align.key = 32, vast.dl.bw = 32>
>, vast.core.lang = C, vast.core.target_triple = "x86_64-pc-linux-gnu"} {
  hl.func @gcd(
    %arg0: !hl.lvalue<!hl.int<unsigned>>, %arg1: !hl.lvalue<!hl.int<unsigned>>
  ) → !hl.int< unsigned > {
    /* high-level dialect with desugared types and resolved typedefs */
    %2 = hl.var "temp" : !hl.lvalue<!hl.int<unsigned>>
    /* the rest of the function */
    hl.return %0 : !hl.int<unsigned>
  }
```

Emit ABI and use standard types

vast-opt --vast-hl-lower-types --vast-emit-abi

```
abi.func @vast.abi.gcd(%arg0: !hl.lvalue<ui32>, %arg1: !hl.lvalue<ui32>) {
  %0:2 = abi.prologue {
    %4 = abi.direct %arg0 : !hl.lvalue<ui32>
    %5 = abi.direct %arg1 : !hl.lvalue<ui32>
    abi.yield %4, %5 : !hl.lvalue<ui32>, !hl.lvalue<ui32>
  } : !hl.lvalue<ui32>, !hl.lvalue<ui32>
  %4 = hl.var "temp" : !hl.lvalue<ui32> = {
     %5 = hl.ref %0#1 : (!hl.lvalue<ui32>) → !hl.lvalue<ui32>
  /* the rest of function with ABI transformed values */
  %3 = abi.epilogue { /* ... */ } : ui32
  hl.return %3 : ui32
```

Lower high-level dialect

vast-opt --vast-hl-to-ll-vars --vast-hl-to-ll-cf ...

• • •

Emit LLVM dialect or LLVM IR

vast-front -vast-emit-mlir=llvm or vast-front -vast-emit-llvm

***Disclaimer:** This example simplifies the representation and omits types for readability. For the full example, follow the QR code to Compiler Explorer.*

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA).

The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

Explore on Compiler Explorer