Learn Llvm 12 PDF (Limited Copy)

Kai Nacke

Learn
LLVM 12

A beginner's guide to learning LLVM compiler tools and core libraries with C++

Kai Nacke

BooKey

More Free Book

Scan to Download

Learn Llvm 12 Summary

Mastering Compiler Construction with LLVM and C++

Written by Books OneHub

More Free Book

Scan to Download

About the book

Dive into the fascinating world of compiler construction and code optimization with "Learn LLVM 12" by Kai Nacke, a comprehensive guide designed for both beginners and experienced programmers alike. This book unravels the intricacies of the LLVM (Low Level Virtual Machine) project, offering a detailed exploration of its architecture and capabilities that empower developers to create high-performance compilers and innovative programming tools. Through practical examples and clear explanations, readers will gain hands-on experience with LLVM's modular structure, intermediate representation, and optimization techniques, equipping them with the skills to leverage LLVM's powerful features in real-world scenarios. Whether you aim to enhance your software development expertise or embark on a new project that demands cutting-edge performance, this enlightening resource paves the way to mastering LLVM and transforming how you approach code generation and optimization.

More Free Book

Scan to Download

About the author

Kai Nacke is a prominent figure in the realm of compiler development and programming languages, known for his deep expertise in LLVM and related technologies. With a solid academic background in computer science, he has contributed significantly to the open-source community, particularly through his work on LLVM, a powerful compiler infrastructure. Throughout his career, Kai has not only engaged in developing advanced tools and optimizations but has also dedicated himself to educating others about the intricacies of compiler design and implementation. His insights and practical guidance make him a respected author and educator, capable of bridging complex theoretical concepts with hands-on applications, as exemplified in his book "Learn LLVM 12." Whether through his writings or lectures, Kai Nacke has played a key role in advancing the understanding of LLVM in both academic circles and the software development industry.

More Free Book

Scan to Download

Ad

Try Bookey App to read 1000+ summary of world best books

Unlock 1000+ Titles, 80+ Topics

New titles added every week

Brand | Leadership & Collaboration | Time Management | Relationship & Communication

ness Strategy | Creativity | Public | Money & Investing | Know Yourself | Positive P

Entrepreneurship | World History | Parent-Child Communication | Self-care | Mind & Spi

Insights of world best books

Free Trial with Bookey

Summary Content List

Chapter 1: Installing LLVM

Chapter 2: Touring the LLVM Source

Chapter 3: The Structure of a Compiler

Chapter 4: Turning the Source File into an Abstract Syntax Tree

Chapter 5: Basics of IR Code Generation

Chapter 6: IR Generation for High-Level Language Constructs

Chapter 7: Advanced IR Generation

Chapter 8: Optimizing IR

Chapter 9: Instruction Selection

Chapter 10: JIT Compilation

Chapter 11: Debugging Using LLVM Tools

Chapter 12: Create Your Own Backend

More Free Book

Scan to Download

Chapter 1 Summary: Installing LLVM

In Chapter 1 of "Learn LLVM 12," titled "Installing LLVM," the author, Kai Nacke, guides readers through the essential steps of setting up their development environment for LLVM, a powerful collection of tools for compiling and building software. Starting from scratch, Nacke emphasizes the importance of compiling LLVM from its source to gain a deeper understanding of its functioning.

The chapter begins by detailing the prerequisites needed for a successful installation, which includes having a compatible operating system like Linux, FreeBSD, macOS, or Windows, and sufficient disk space—ideally around 30 GB. For those building LLVM on less powerful machines, such as a Raspberry Pi, patience is required due to longer compilation times. Recommendations for optimal hardware are given, such as a quad-core CPU and an SSD, based on the author's own experience using a high-performance setup.

Next, Nacke lists the necessary software, highlighting tools like Git for version control, CMake for generating build files, and Ninja as the preferred build system due to its efficiency. The chapter provides a comprehensive overview of various compatible compilers and their required versions to ensure LLVM will compile correctly. Python is also mentioned, as it plays a critical role in generating build files and running tests.

More Free Book

Scan to Download

The text then transitions into how to install these prerequisites using the package managers native to different operating systems, providing straightforward commands for popular ones like Ubuntu, Fedora, FreeBSD, macOS, and Windows. Following this, readers are instructed on configuring Git for version control, an essential part of committing changes and tracking progress in software development.

Building LLVM entails cloning the repository from GitHub, which is explained in detail, including steps for managing line-ending issues specific to Windows users. Nacke elaborates on the necessity of creating a separate build directory which contrasts with many projects that allow inline builds. Furthermore, he walks through generating the build system files using CMake, emphasizing the importance of proper command syntax depending on the operating system.

After generating build files, readers learn how to compile LLVM and Clang, using Ninja, along with best practices like running a test suite to verify the installation. There's advice on managing CPU resources during the compilation process to allow for multitasking if necessary.

In summary, this chapter serves as a detailed, step-by-step introduction tailored for developers eager to dive into LLVM. It lays down the groundwork necessary to build and customize LLVM installations,

More Free Book

Scan to Download

underscoring the flexibility and control it offers for various programming needs. Looking ahead, Nacke prepares readers for the next chapter's exploration of the LLVM mono repository and the various projects within it, promising an enriching journey into the intricacies of LLVM’s architecture.

More Free Book

Scan to Download

Chapter 2 Summary: Touring the LLVM Source

In Chapter 2 of "Learn LLVM 12," the author, Kai Nacke, takes readers on an exploratory journey through the LLVM (Low-Level Virtual Machine) mono repository, unpacking its structure and the various projects contained within. This chapter emphasizes the importance of understanding LLVM's layout to maximize its use in personal projects.

Starting with an overview, Nacke highlights that the LLVM mono repository encompasses several crucial components, which can be grouped into three primary types: core libraries, compilers and tools, and runtime libraries. The core libraries, located in the `llvm` directory, include essential libraries for code optimization and generation, alongside tools for inspecting object files and debugging. Projects like Polly and MLIR introduce advanced optimizations and multi-level representations, respectively.

Moving to compilers and tools, Nacke introduces Clang, the featured C/C++ compiler that leverages LLVM's capabilities. Clang not only compiles but also supports formatting and static analysis, with additional tools like `clang-tidy` and the `lld` linker completing the suite. The chapter also mentions LLDB, the debugger, which shares a structure similar to GDB, facilitating debugging across supported languages.

The runtime libraries provide necessary support for the languages LLVM

More Free Book

Scan to Download

compiles. Projects such as `compiler-rt` for runtime support functions, `libunwind` for exception handling, and `libcxx` for the C++ standard library are examined. Each project maintains a structured approach that enhances reusability, contrasting with the more monolithic design of GCC.

Nacke then delves into the layout common to all LLVM projects, detailing how each project typically contains directories for libraries, utilities, documentation, and more, all managed with CMake for build file generation. This layout not only promotes organization but also allows developers to easily leverage existing libraries.

As a practical exercise, readers are guided in creating their own project, dubbed `tinylang`, which exemplifies a simple language built using LLVM libraries. Nacke outlines the directory structure and necessary CMake setup for compiling a basic "Hello, world!" application, alongside the process for linking it to LLVM libraries.

The chapter also introduces the concept of cross-compiling, essential for targeting different CPU architectures, and explains the use of compiler triplets in describing the host and target systems. Nacke underscores the importance of portable code and provides a step-by-step guide to setting up a cross-compilation environment, emphasizing the tools and libraries required.

By the chapter's conclusion, readers have a solid foundation in LLVM's

More Free Book

Scan to Download

structure and functionality, paving the way for further exploration of compiler construction in subsequent chapters. This blend of technical insight and practical guidance makes the content both informative and engaging.

Scan to Download

Chapter 3: The Structure of a Compiler

In Chapter 3 of "Learn LLVM 12," the author, Kai Nacke, delves into the intricate structure of a compiler and its essential components, making the complex subject accessible and engaging. The chapter introduces us to the dual framework of a compiler, consisting of the frontend and backend. The frontend caters to language-specific features, parsing the source code to create a semantic representation, usually an annotated abstract syntax tree (AST). In contrast, the backend executes the task of converting this semantically analyzed representation into optimized machine code, highlighting the importance of having a well-defined interface for reusability.

Using an arithmetic expression language as a case study, Nacke guides us through the construction of a simple compiler that can understand and process expressions. Key components such as lexical analysis, syntactical analysis, and semantic analysis are laid out clearly. The lexer is responsible for breaking down the input into tokens, while the parser organizes these tokens into a logical structure—the AST. This AST serves not only as a

Install Bookey App to Unlock Full Text and Audio

Free Trial with Bookey

Ad

Why Bookey is must have App for Book Lovers

30min Content

The deeper and clearer interpretation we provide, the better grasp of each title you have.

Text and Audio format

Absorb knowledge even in fragmented time.

Quiz

Check whether you have mastered what you just learned.

And more

Multiple Voices & fonts, Mind Map, Quotes, IdeaClips...

Free Trial with Bookey

Chapter 4 Summary: Turning the Source File into an Abstract Syntax Tree

In Chapter 4 of "Learn LLVM 12," we embark on the intriguing journey of building the frontend of a compiler for a simple programming language called tinylang, which is a subset inspired by Modula-2. This chapter breaks down the complex world of compilers and guides us through essential steps to translate source code into an abstract syntax tree (AST), setting us up for future code generation.

The chapter kicks off by introducing the foundational grammar of tinylang. The grammar is defined in a way that allows for the declaration of modules, constants, variables, and procedures. Each of these declarations has specific rules, providing structure and meaning to the language. For example, declaring a procedure involves specifying its name and parameters, along with its body, which could contain further statements or declarations. The structure is a bit more complex than earlier examples, laying solid groundwork for later programming tasks.

Next, we create the project layout. This involves organizing code into different components like the lexer, parser, and semantic analyzer. By maintaining clear dependencies among these components, efficient code reuse becomes possible. The chapter emphasizes good project management practices, illustrating how neatly structured code can significantly simplify

More Free Book

Scan to Download

tasks like debugging and expanding the compiler's capabilities.

An essential part of any compiler is error management, and this chapter details how to handle source files and user messages effectively. We learn about the llvm::SourceMgr class that manages multiple input files and produces user-friendly error messages when something goes wrong in the code. This focus on user experience ensures that developers can easily correct mistakes and understand their code's behavior.

The chapter progresses into building the lexer, which is responsible for tokenizing the input source code. We learn to separate the lexer into modular components and create a Token class that categorizes various types of tokens: keywords, punctuators, and identifiers. The intricacies of handling keywords, particularly given their overlap with identifiers, lead us into more advanced techniques like keyword filtering and the use of hash tables for efficient lookups.

Next, we delve into constructing a recursive descent parser, where we apply grammar rules to create methods in C++ that mirror the grammar's structure. This is a fascinating part of the chapter, as it highlights the connection between theory and practical implementation. Here, we also tackle challenges like left-recursive rules and ambiguity in grammar, promoting best practices in grammar design that facilitate effective parsing.

More Free Book

Scan to Download

For those who prefer less manual coding, the chapter introduces bison and flex—tools that automatically generate parsers and lexers from specifications. This section emphasizes the advantages of these tools, particularly when prototyping new languages, and shows how to translate our grammar into formats suitable for bison and regular expressions for flex.

As we near the end of the chapter, we layer in the semantic analysis, intertwining it with the parser to evaluate the correctness of parsed constructs. The semantic analyzer checks names for proper declarations and types, enforcing the "declare-before-use" rule and ensuring that variables and procedures are appropriately scoped.

With a focus on efficiency, we build our AST, populated with nodes holding vital information required for code generation. The chapter concludes on an enthusiastic note, as readers are encouraged to test their newly built frontend by parsing a simple GCD algorithm in tinylang.

In summary, Chapter 4 equips you with the key skills and frameworks necessary to create a functioning compiler frontend. By parsing a well-defined programming language and managing syntax and semantic errors with grace, you move closer to mastering compiler theory and practice. The groundwork laid in this chapter promises exciting possibilities in code generation and further compiler development in the subsequent chapters!

More Free Book

Scan to Download

Chapter 5 Summary: Basics of IR Code Generation

Chapter 5 of "Learn LLVM 12" takes you on an insightful journey into the essentials of generating LLVM Intermediate Representation (IR) code from an Abstract Syntax Tree (AST). After crafting a decorated AST for a programming language, the chapter emphasizes the need to transform this high-level structure into the more manageable LLVM IR, which resembles three-address code.

You dive into the various topics necessary for IR generation, including how to create the LLVM IR from the AST, the use of Static Single Assignment (SSA) form, and the techniques to emit assembler text and object code. As the chapter unfolds, it lays the groundwork for building a code generator tailored to your programming language, helping you integrate it seamlessly into your own compiler.

Starting with the transformation of the AST, you'll explore the mechanics behind the LLVM code generator, which processes a module defined in IR format, ultimately generating object code or assembly text. This involves creating three crucial classes: CodeGenerator, CGModule, and CGProcedure, each with distinct roles in managing the state required for generating IR code relevant to compilation units and individual functions.

The chapter introduces you to LLVM IR's building blocks—basic

More Free Book

Scan to Download

blocks—structured sequences of instructions that provide a clear entry and exit point. This allows you to understand how control flow structures, like WHILE and IF statements, translate into LLVM IR. These structures lead to the creation of basic blocks that define how the program executes through branching and conditional logic.

One of the standout techniques you'll learn is AST numbering, an approach for managing the values of local variables across these basic blocks. This process includes maintaining the current values of variables and establishing a system of phi instructions to facilitate the SSA form, which streamlines operations and optimizations in IR code generation.

Additionally, the chapter explores the load-and-store model for local variables in the generation process, illustrating how parameters and variables are accessed and manipulated within the IR. It covers essential aspects like sealing blocks, updating phi instructions, and ensuring that basic blocks maintain their integrity and functionality throughout code generation.

As you delve deeper, you also touch on the nuances of designing a module in LLVM that regroups all functions and global variables. This includes handling names, managing visibility through linkage styles, and utilizing name mangling to create unique identifiers in the context of modules.

Finally, the chapter wraps up with the procedures for emitting assembler text

More Free Book

Scan to Download

and object code, outlining the necessary LLVM pipeline to take your IR through various optimization passes before outputting the final product. The excitement culminates as you learn how to execute commands to generate an object file from a high-level language, ushering you into the realm of practical compiler design.

By the end of this chapter, you emerge equipped with the knowledge to implement a code generator that not only outputs LLVM IR but also produces assembler code, putting you one step closer to mastering your compiler design.

Topic	Description
Overview	Chapter 5 focuses on generating LLVM Intermediate Representation (IR) code from an Abstract Syntax Tree (AST).
AST Transformation	Transforming a decorated AST into LLVM IR, which resembles three-address code.
Code Generator	Building a tailored code generator for integration into a compiler.
Key Classes	Creation of CodeGenerator, CGModule, and CGProcedure to manage IR code generation.
Basic Blocks	Introduction to basic blocks and their role in control flow structures like WHILE and IF statements.
AST Numbering	A technique to manage local variable values across basic blocks, involving phi instructions for SSA form.
Load-and-Store	Managing local variables and parameters in the IR generation

More Free Book

Scan to Download

Topic	Description
Model	process.
Module Design	Designing a module that contains all functions and global variables with managing names and visibility.
Output Generation	Procedures for emitting assembler text and object code with LLVM optimization passes.
Outcome	Equipped to implement a code generator that outputs LLVM IR and assembler code, advancing in compiler design.

More Free Book

Scan to Download

Chapter 6: IR Generation for High-Level Language Constructs

In Chapter 6 of "Learn LLVM 12" by Kai Nacke, the focus is on transforming high-level language features into LLVM Intermediate Representation (IR), particularly around aggregate data types and object-oriented programming (OOP) constructs. The chapter kicks off by discussing the challenges posed by complex data types and how the LLVM IR supports these through constructs such as arrays and structures. It emphasizes the need to understand how different platforms handle the passing of aggregate types to ensure compatibility and efficiency in function calls.

The author dives into how to represent arrays and structs in LLVM IR, detailing their memory organization. Arrays are defined with a fixed size, while structures can consist of various data types. Important instructions like `extractvalue` and `insertvalue` are introduced, which allow manipulation of elements within these aggregates. The chapter also emphasizes the significance of pointers in LLVM, illustrating how global variables and

Install Bookey App to Unlock Full Text and Audio

Free Trial with Bookey

Ad

22k 5 star review

Positive feedback

Sara Scholz

tes after each book summary
erstanding but also make the
and engaging. Bookey has
ding for me.

Fantastic!!!

Masood El Toure

I'm amazed by the variety of books and languages Bookey supports. It's not just an app, it's a gateway to global knowledge. Plus, earning points for charity is a big plus!

Fi

Ab
bo
to
my

José Botín

ding habit
's design
ual growth

Love it!

Wonnie Tappkx

Bookey offers me time to go through the important parts of a book. It also gives me enough idea whether or not I should purchase the whole book version or not! It is easy to use!

Time saver!

Bookey is my go-to app for
summaries are concise, ins
curated. It's like having acc
right at my fingertips!

Awesome app!

Rahul Malviya

I love audiobooks but don't always have time to listen to the entire book! bookey allows me to get a summary of the highlights of the book I'm interested in!!! What a great concept !!!highly recommended!

Beautiful App

Alex Walk

This app is a lifesaver for book lovers with
busy schedules. The summaries are spot
on, and the mind maps help reinforce wh
I've learned. Highly recommend!

Free Trial with Bookey

Chapter 7 Summary: Advanced IR Generation

In Chapter 7 of "Learn LLVM 12," the focus broadens to advanced IR generation, delving into the intricacies of implementing crucial compiler functionalities such as exception handling and metadata management. This chapter builds on earlier concepts, introducing readers to the techniques necessary for addressing real-world compiler challenges, particularly in modern programming languages that utilize exceptions.

The chapter opens with a detailed exploration of exception handling in LLVM IR, highlighting its importance and connection to runtime libraries, specifically libunwind. It presents practical examples using C++, where functions like `bar()` can throw exceptions of different types, and `foo()` demonstrates how to catch these exceptions. The mechanics of throwing an exception involve two runtime calls—one to allocate memory for the exception and the other to raise it, employing LLVM's IR syntax.

The introduction of the `invoke` instruction marks a significant shift from the standard `call` instruction, allowing different control paths depending on whether exceptions occur. This leads to the creation of landing pads, which capture exception information and provide the necessary structure for handling such exceptions correctly. The chapter meticulously details the various components of exception handling, including type metadata and cleanup codes, making it clear how LLVM relies on this structured approach

More Free Book

Scan to Download

to manage exceptions elegantly.

Next, we see how to enhance optimization through type-based alias analysis (TBAA). By adding metadata to LLVM instructions, the optimizer can gain insights into pointer relationships, allowing for more aggressive optimizations. The chapter explains how different types and structures are related within the LLVM hierarchy, emphasizing the interplay between type definitions and their metadata, which aids the optimizer in being more efficient.

As the narrative progresses, the author transitions into adding debug information to LLVM IR, essential for enabling source-level debugging. This process involves crafting detailed metadata using the `DIBuilder` class to describe everything from files to functions to variable types. Such metadata assists debuggers in linking machine code back to its source context, fostering a smooth debugging experience.

The chapter culminates with practical implementations of the concepts discussed. The author outlines how to integrate exception handling and metadata generation into a simple calculator compiler, effectively demonstrating how these advanced features come together in real-world scenarios. As readers digest these technical details, they are equipped with the knowledge to extend their compilers with robust functionalities.

More Free Book

Scan to Download

In summary, Chapter 7 of "Learn LLVM 12" enriches the reader's knowledge base, combining theoretical concepts with practical implementation in LLVM IR, addressing essential compiler features while exposing the intricate workings behind exception handling, metadata management, and debugging support. As the chapter closes, readers are set to explore optimization techniques in the following chapter, paving the way to further enhance their compiler design skills.

More Free Book

Scan to Download

Chapter 8 Summary: Optimizing IR

In Chapter 8 of "Learn LLVM 12," we dive deep into the intricacies of optimizing the intermediate representation (IR) of a compiler using LLVM's Passes and the Pass manager. The chapter begins with an introduction to the LLVM Pass manager, a vital component that orchestrates a series of optimizations on the IR. Each optimization step, known as a Pass, either transforms the IR or analyzes it for dependencies and information. The Pass manager ensures that these Passes are executed in the proper sequence, adjusting to the varying needs of developers who may prioritize faster compilation speeds during development or more sophisticated optimizations for production applications.

As we progress, we learn how to implement a new Pass, termed `countir`, which counts the number of IR instructions and basic blocks. This involves adding specific files and defining the Pass in the LLVM source tree, showcasing both the technical aspects and organization of working with LLVM's libraries. The chapter guides us through the steps to register this new Pass with LLVM, allowing it to integrate seamlessly with the standard optimization tool, `opt`.

Furthermore, the story of development takes an interesting turn as the text explains adapting the same Pass for LLVM’s legacy Pass manager, showcasing the evolutionary leap from old to new architectures within

More Free Book

Scan to Download

LLVM. The old Pass manager had a more cumbersome inheritance requirement, whereas the new Pass manager introduces a streamlined, concept-based approach, improving performance and efficiency.

As we fold in the creation of a new Pass plugin, the chapter emphasizes how to preserve functionality while ensuring compatibility across different versions of the Pass manager. This development not only aids in personal projects but can also benefit the larger LLVM community by contributing to its expansive toolkit.

The latter half pivots towards practical application, detailing how to add an optimization pipeline to a compiler, particularly the `tinylang` compiler discussed throughout the book. By utilizing the PassBuilder class, we can define and execute an optimization pipeline directly in the compiler, thus transforming `tinylang` into a more robust and optimizing compiler, akin to clang.

Finally, the chapter wraps up by discussing extension points within the Pass pipeline, allowing users to customize their compilation process further. The addition of various user-defined and predefined Passes at key moments underscores LLVM's flexibility and power.

In summary, this chapter demystifies the technical workflows behind LLVM optimizations, from implementing and adapting Passes to integrating them

More Free Book

Scan to Download

into a functional compiler. The knowledge gained here equips readers with the skills to harness the full potential of LLVM in their development efforts.

Section	Summary
Introduction to LLVM Pass Manager	Overview of the Pass manager that orchestrates optimization passes on the IR, allowing adjustments based on development needs.
Implementing a New Pass	Guide on creating the `countir` Pass to count IR instructions and blocks, including file addition and registration with LLVM.
Transitioning from Legacy to New Pass Manager	Explains adaptation of Pass for the legacy Pass manager, highlighting improvements in the new manager's approach and efficiency.
Create Pass Plugin	Focuses on developing a Pass plugin that maintains backward compatibility while enhancing functionality for personal and community projects.
Adding Optimization Pipeline	Details on incorporating an optimization pipeline into the `tinylang` compiler using the PassBuilder class to improve its robustness.
Extension Points in Pass Pipeline	Discussion on customizing the compilation process with user-defined and predefined Passes, showcasing LLVM's flexibility.
Conclusion	Summarizes the workflows for LLVM optimizations, equipping readers with skills for effective development in LLVM.

More Free Book

Scan to Download

Critical Thinking

Key Point: The importance of optimizing processes for efficiency

Critical Interpretation: Imagine applying the principles of LLVM's Pass manager to your daily life. Just as the Pass manager orchestrates optimizations in the compiler, you can streamline your own tasks, finding the right order and methods to tackle your responsibilities. By analyzing what works best for you and prioritizing speed or quality as needed, you can enhance your productivity, adapt to challenges, and ultimately lead a more efficient and satisfying life.

More Free Book

Scan to Download

Chapter 9: Instruction Selection

In Chapter 9 of "Learn LLVM 12," the focus is on instruction selection, a crucial phase in the LLVM backend. After optimizing LLVM Intermediate Representation (IR), this process translates it into machine instructions, which can be done via three distinct methods: Selection DAG, Fast Instruction Selection (FastISel), and Global Instruction Selection (GlobalISel).

The chapter begins by outlining the structure of the LLVM target backend, detailing the essential tasks it performs, such as constructing a directed acyclic graph (DAG) from LLVM IR, selecting machine instructions, optimizing their order, and generating target-specific assembly code. As the IR transitions into machine code, each instruction morphs into various representations until it reaches a final form suitable for coding.

To assist in testing and debugging, Machine IR (MIR) is introduced, allowing developers to inspect the state of machine instructions. Through practical exercises, readers learn to use tools like llc to generate relevant

Install Bookey App to Unlock Full Text and Audio

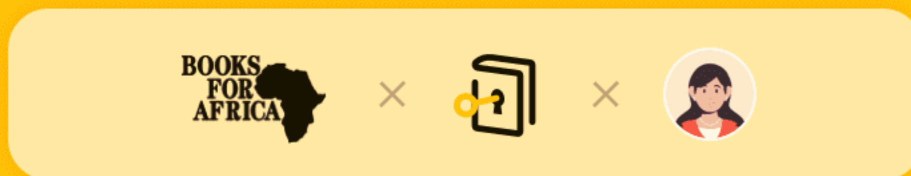
Free Trial with Bookey

Ad

Read, Share, Empower

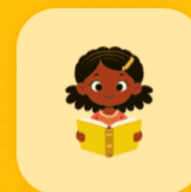
Finish Your Reading Challenge, Donate Books to African Children.

The Concept



This book donation activity is rolling out together with Books For Africa. We release this project because we share the same belief as BFA: For many children in Africa, the gift of books truly is a gift of hope.

The Rule



Earn 100 points **Redeem a book** **Donate to Africa**

Your learning not only brings knowledge but also allows you to earn points for charitable causes! For every 100 points you earn, a book will be donated to Africa.

Free Trial with Bookey

Chapter 10 Summary: JIT Compilation

In Chapter 10, titled "JIT Compilation," of "Learn LLVM 12," you dive into the fascinating world of Just-In-Time (JIT) compilation using LLVM. The chapter begins by explaining the essence of JIT compilation, highlighting how it differs from Ahead-Of-Time (AOT) compilers by allowing code to be compiled at runtime, thus enabling immediate execution without storing object code on disk. You get a comprehensive overview of the LLVM ExecutionEngine component, which is crucial for creating JIT compilers that can execute Intermediate Representation (IR) code directly from memory.

Throughout the chapter, the author draws attention to various practical applications of JIT compilers, such as virtual machines (like those used in Java and C#), expression evaluation in spreadsheet applications, and database query execution. By understanding how functions in languages can be optimized during runtime and how IR code can be compiled to machine code on-the-fly, you grasp the power and flexibility that JIT compilers offer.

As you explore the chapter, you learn how to implement your own JIT compiler, starting with an overview of LLVM's JIT architecture, including its layered design. This design features compile and link layers, as well as transformation layers that can customize JIT behavior. The chapter introduces existing LLVM tools, particularly the `lli` tool, which serves as both an interpreter and dynamic compiler for running LLVM IR.

More Free Book

Scan to Download

The text provides a hands-on approach, encouraging readers to engage with practical examples. You learn how to set up and compile a simple "Hello World" application in LLVM IR, gradually building toward more complex implementations. Detailed instructions guide you through creating a JIT compiler using the `LLJIT` utility class, with successful examples solidifying your understanding.

Once you have a grasp of the basic JIT compiler setup, you are encouraged to develop a more customizable JIT compiler class using the ORC API. This section outlines the construction and initialization of this more flexible JIT compiler, demonstrating how to manage various layers to enrich functionality.

The chapter concludes by exploring the implications of JIT compilation within static compilation contexts. You dive into how this can optimize language processing and evaluation, highlighting the conditions under which JIT execution is beneficial, especially for "pure" functions. However, it also raises thoughtful considerations about language semantics, security, and implementation complexities.

By the end of the chapter, you're equipped with a thorough understanding of both the theoretical and practical aspects of JIT compilation. You've gained skills to implement your own JIT compiler and consider its integration into

More Free Book

Scan to Download

broader compiler designs, setting the stage for a deeper exploration of CPU architecture in subsequent chapters.

More Free Book

Scan to Download

Chapter 11 Summary: Debugging Using LLVM Tools

Chapter 11 of "Learn LLVM 12" by Kai Nacke is all about debugging applications using LLVM tools, primarily focusing on the LLVM and Clang libraries designed to catch a variety of bugs and performance issues. It introduces readers to the powerful tools provided by LLVM, such as sanitizers for identifying specific bugs, performance profilers, static analyzers, and methods for creating custom tools based on Clang.

The chapter starts by detailing how to instrument applications with sanitizers, introducing three main types: the Address Sanitizer, Memory Sanitizer, and Thread Sanitizer. The Address Sanitizer is particularly useful for detecting common memory errors, like accessing memory after it's been freed or overrunning allocated blocks. It replaces standard memory management functions with custom ones that validate memory access, providing detailed error reports that include where the error occurred in the code.

Next, the chapter explains the Memory Sanitizer, which finds bugs related to the use of uninitialized memory, an issue common in C and C++. The Thread Sanitizer follows, which helps to spot data races in multi-threaded applications, a critical concern as modern software increasingly utilizes threading for performance.

More Free Book

Scan to Download

The discussion then shifts to fuzz testing, enabled by libFuzzer, a powerful tool for exposing hidden bugs in programs by applying random or mutated inputs. It highlights the utility of fuzz testing, illustrating how it locates vulnerabilities that typical unit tests might miss.

Performance profiling is another key focus, where the XRay tool is introduced. XRay allows developers to visualize where time is being spent in their code and identify performance bottlenecks effectively. Users can generate flame graphs for intuitive presentations of execution performance, facilitating quick identification of heavy functions.

The chapter also delves into the Clang Static Analyzer, a tool that performs deeper analyses of source code than standard compiler checks, helping to catch potential runtime errors at compile-time. It provides examples of how the static analyzer can identify tricky coding errors, such as division by zero, through symbolic interpretation of the code's flow.

For those interested in extending Clang's functionality, the chapter provides a guide on how to create custom checkers for the static analyzer. It explains how to develop a new checker that monitors function naming conventions, reinforcing best practices in code.

Lastly, readers are introduced to Clang plugins, which allow the integration of custom behavior into the compilation process, ranging from simple

More Free Book

Scan to Download

warnings about naming conventions to more complex tasks like modifying the Abstract Syntax Tree (AST) during compilation.

By the end of Chapter 11, readers gain a comprehensive understanding of various debugging and profiling techniques available through LLVM, empowering them to build robust applications while efficiently identifying and correcting potential issues throughout their development lifecycle. This enables software developers to not only fix existing bugs, but also to proactively avoid new ones, thereby enhancing overall code quality.

More Free Book

Scan to Download

Critical Thinking

Key Point: The importance of proactive debugging and performance profiling.

Critical Interpretation: Imagine incorporating the practice of proactive debugging in your daily life, not just in coding but in all your endeavors. Just like LLVM tools empower developers to catch and fix issues before they escalate, you too can adopt a mindset of anticipation and foresight. By regularly evaluating your personal and professional actions, identifying areas for improvement, and addressing challenges before they arise, you create a path toward continuous growth and success. This approach encourages resilience and adaptability, allowing you to handle life's unpredictable moments with confidence, ensuring that you not only respond to problems but also prevent them from occurring in the first place.

More Free Book

Scan to Download

Chapter 12: Create Your Own Backend

In Chapter 12 of "Learn LLVM 12," the reader embarks on an exhilarating journey to create a new backend for the LLVM compiler infrastructure, specifically targeting the historically significant Motorola M88k CPU architecture. The chapter begins by setting the stage, highlighting that while LLVM's architecture is versatile and flexible, developing a new backend is a substantial task, often necessary for commercial applications or hobby projects.

First, readers learn about the various components that constitute the backend, starting by incorporating the M88k architecture into the LLVM's Triple class, allowing the compiler to recognize the new target architecture. This involves updating enumerations and methods within the existing LLVM codebase, thus laying the groundwork for all further development.

Next, the chapter delves into extending the Executable and Linkable Format (ELF) definition to accommodate M88k-specific relocations. This process ensures that the necessary binaries and object files can be generated and

Install Bookey App to Unlock Full Text and Audio

Free Trial with Bookey

World' best ideas unlock your potencial

Free Trial with Bookey

Scan to download