Zachary Waldowski

Professor Bob Waters

CS 2340 A

15 April 2012

M12 Individual Case Study

Nearly all modern computing is affected by Arthur C. Clarke's cliché, "Any sufficiently advanced technology is indistinguishable from magic." Compilers are no exception. Code is significantly more portable today than during C's heyday, with about 3 major architectures and a mere fistful of compilers; compare to the smattering of CISC and RISC, from Motorola 68K to POWER in the early 90s', each with their own assembly and compilers. A "standard" C was laughable. For those who weren't even alive then, it might be prudent to make a comparison to web browsers in the 2000's each having a special take on how to render a page. Like Mozilla's rise revolutionized the Internet, the rise of the GNU Compiler Collection (GCC) was slow, but no less revolutionary.

Many of GCC's philosophies come from this halcyon era of proprietary software. Corporate politics could've easily killed the compiler before it was even born. So GNU took an ironically dictatorial approach in writing it, by making the backend (i.e., the parts that generate assembly and mangle it into machine code) so incomprehensible that no single entity could claim a part of the code without GNU's blessing. So if Intel wanted to add a proprietary feature to speed up `for` loops on only Intel hardware, well, good luck. This choice made GCC surprisingly stable, though some might call it stagnant.

When struggling giant Apple acquired NeXT for the GUI of Mac OS X, their Hail Mary attempt at saving the company, the technical decision had already been made to base the OS on the XNU kernel and a BSD-like Unix stack. For this, the NeXT compiler wasn't going to cut it; they needed a true C compiler, so Objective-C support was grafted onto GCC. However, what started as a necessity became a technical debt: adding code generation features to the language, porting the platform to 64-bit (for a transition from PowerPC to Intel), and to ARM (for the eventual iPhone) would all require modifying the aforementioned GCC backend. Not an impossible feat, but impossible to maintain: an "official" Apple GCC never made it beyond 4.2.

In 2005, a five-year-old university research project called LLVM caught the company's eye. Originally meaning *Low-Level Virtual Machine* - in the Java sense of the phrase VM, and even then only etymologically-speaking - LLVM was no for-want-of-a-nail attempt at a GCC replacement, LLVM's goal was to be an infrastructure. It was intrinsically and unstoppably modular. Unlike, really, any compiler project (particularly open source) that come before it, LLVM was designed under the assumption that we're doing something wrong today and should therefore be able to rip it all out later and start over. Moreover, LLVM's modularity was what saved it from the bane of other alternative compiler projects. Developing a compiler is hard; it takes time, money, and rewrites to even get close to something like GCC. In the following sections, we'll take a short look

at LLVM's development practice, code generation, and library usability to see why this software is particularly unique.

The key to LLVM's power was that it became better in baby steps. Its start in the industry was inauspicious, using just the code generation backed in Mac OS X 10.4 "Tiger" to emulate higher-level GPU functionality on machines lacking the appropriate horsepower. This development style allowed the core LLVM backend to stabilize before focusing on standard library implementations, the Clang front-end for C and Objective-C, support for C++ and Objective-C++ in Clang, and eventually even LLDB, a debugger build using the LLVM backend parsers. All the while, a version of the GCC frontend integrated with the LLVM backend would allow developers to take advantage of its rapidly-maturing features without risking the stability of their toolchain.

Furthermore, LLVM benefits from the subtle changes in development practice inherent to a project started and refined "today". This is no fly-by-night, seat-of-the-pants setup; the software is rigorously and continuously tested, community-maintained with commercial backers, and flexibly organized. Entire software trees are checked into the testing system to be tested and re-tested for regression. Tools are maintained in-house to minimize the effort required to test components of the system, massively helped by the intermediate code generation that reduces the layers a bug might manifest itself in to just one.

One of LLVM's most powerful features comes as a mere side effect from its modularity. Every step in the compilation process is a module, and each of these modules must have appropriately compatible inputs and outputs. Frontends must output an intermediate form of code that is explicitly compatible with each of the other components in the toolset. Most other toolchains keep their parsed and optimized code in a black box, often in memory and other times exclusively compatible with that specific version of the compiler and nothing else. In LLVM, a piece of C code and a piece of Haskell code that do precisely the same things step by step order could theoretically produce identical intermediate representations. This design feature is indescribably useful and extends LLVM from the realm of compiler technology to the realm of meta-coding, in no small part thanks to the library-based structure of the project.

The final aspect of LLVM we will analyze is the project's library-based design. Components of the project with discrete functionality are factored as libraries. For bog-standard use as a compiler, most of these libraries include a reference binary, such as the assembler and optimization routines. Because of this approach, a user of LLVM could easily - almost trivially - integrate just one of the pieces of functionality provides with no effort. Xcode (Apple's IDE), and increasingly more open source projects use LLVM to provide code analysis even if the project's primary compiler isn't Clang. One project offers a replacement for the entire second half of LLVM's toolchain, compiling LLVM intermediate code - including C and C++ - to JavaScript that in some tests is only twice as slow as natively-built code. Certain tools build classically interpreted languages like Python and Ruby into lightning-fast native versions. All these and more implementations would simply not have happened or would have required thousands of man-hours to even get off the ground.

Evaluating from a more traditional FURPS perspective might be just as enlightening. The functionality of the projects are astounding, and though the still young derivatives like LLDB aren't yet at parity with GDB or Microsoft's debugger, the project

does not appear to be stopping its breakneck speed of improvement. Similarly, on the subject of reliability, the compiler toolchain itself tries its best to stay constantly reliable through thorough and automated testing; many errors with LLVM/Clang are often from poorly-written code or code that assumes nothing other than GCC would ever have an advanced feature set.

On the usability front, the project being young is to its great advantage; each tool is exhaustively documented and every attempt is made for them to be consistent with one another. However, raw C++ API of the libraries making up the project itself moves equally quickly with little respect towards backward-compatibility, which would dramatically slow performance. Though it may be inconvenient, these specific libraries that change are less often used than the higher-level C API it provides.

With respect to performance is where LLVM blows its competition away. Simplicity is the key with LLVM; optimizations aren't performed by loading every bit of code into memory and performing ad-hoc voodoo incantations on the assembly, but instead in compartmentalized and reusable pattern identifiers that apply to all intermediate code. The proof is in the pudding: a self-hosted, fully-functional LLVM toolchain compiles in about 15 minutes, whereas the typical GCC triplet takes several hours to build to the same point. Projects using Clang build almost impossibly fast.

Finally, LLVM started with and continues to be the epitome of supportable software. Eight years and 22 (soon-to-be 23) releases later, the project has transformed from a research compiler to a project umbrella encompassing 11 tools that are the backbone of the modern computer and advancing web. On this basis alone, I can say with a fair amount of confidence that the project's software design is validated, if not justified, almost in its entirety. Going forward, LLVM has the potential to make strict per-architecture compilation a thing of the past without the performance loss of managed environments. Google's Native Client, Mozilla's Rust, and the aforementioned to-JavaScript compiler might even enable us to say goodbye to installing software entirely. The future of software development is impressively bright, and it is one with LLVM standing proudly at the helm.