LLVM in the FreeBSD Toolchain

David Chisnall

1 Introduction

FreeBSD 10 shipped with Clang, based on LLVM [5], as the system compiler for x86 and ARMv6+ platforms. This was the first FreeBSD release not to include the GNU compiler since the project's beginning. Although the replacement of the C compiler is the most obvious user-visible change, the inclusion of LLVM provides opportunities for other improvements.

2 Rationale for migration

The most obvious incentive for the FreeBSD project to switch from GCC to Clang was the decision by the Free Software Foundation to switch the license of GCC to version 3 of the GPL. This license is unacceptable to a number of large FreeBSD consumers. Given this constraint, the project had a choice of either maintaining a fork of GCC 4.2.1 (the last GPLv2 release), staying with GCC 4.2.1 forever, or switching to another compiler. The first option might have been feasible if other GCC users had desired the same and the cost could have been shared. The second was an adequate stopgap, but the release of the C11 and C++11 specifications—both unsupported by GCC 4.2.1—made this an impossible approach for the longer term. The remaining alternative, to find a different compiler to replace GCC, was the only viable option.

The OpenBSD project had previously investigated PCC, which performed an adequate job with C code (although generating less optimised code than even our old GCC), but had no support for C++. The TENDRA compiler had also been considered, but development had largely stopped by 2007.

The remaining alternative was Clang, which was still a very young compiler in 2008, but had some significant commercial backing from companies including Apple and Google. In 2009, Roman Divacky and Pawel Worach begin trying to build FreeBSD with Clang and quickly got a working kernel, as long as optimisations were disabled. By May 2011, Clang was able to build the entire base system on both 32-bit and 64-bit x86 and so became a viable migration target. A large number of LLVM Clang bugs were found and fixed as a result of FreeBSD testing the compilation of a large body of code.

3 Rebuilding the C++ stack

The compiler itself was not the only thing that the FreeBSD project adopted from GCC. The entire C++ stack was developed as part of the GCC project and underwent the same license switch. This stack comprised the C++ compiler (g++), the C++ language runtime (`libsupc++`) and the C++ Standard Template Library (STL) implementation (`libstdc++`).

All of these components required upgrading to support the new C++11 standard. The runtime library, for example, required support for dependent exceptions, where an exception can be boxed and rethrown in another thread (or the same thread later).

The FreeBSD and NetBSD Foundations jointly paid PathScale to open source their C++ runtime library (libcxxrt), which was then integrated into the FreeBSD base system, replacing `libsupc++`. The LLVM project provided an STL implementation (`libc++`), with full C++11 and now C++14 support, which was duly integrated.

Using libcxxrt under `libstdc++` allowed C++ libraries that exposed C interfaces, or C++ interfaces that didn't use STL types, to be mixed in the same binary as those that used `libc++`. This includes throwing exceptions between such libraries.

Implementing this in a backwards-compatible way required some linker tricks. Traditionally, `libsupc++` had been statically linked into `libstdc++`, so from the perspective of all linked programs the `libsupc++` symbols appeared to come from `libstdc++`. In later versions in the 9.x series, and in the 9-COMPAT libraries shipped for 10, `libstdc++` became a filter library, dynamically linked to `libsupc++`. This allows symbol resolution to work correctly and allows `libsupc++` or `libcxxrt` to be used as the filtee, which actually provides the implementation of these symbols.

4 Problems with ports

The FreeBSD ports tree is a collection of infrastructure for building around 24,000 third-party programs and libraries. Most ports are very thin wrappers around the upstream distribution's build system, running autoconf or CMake configurations and then building the resulting make files or equivalent. For well-written programs, the switch to Clang was painless. Unfortunately, well-written programs make up the minority of the ports tree. To get the ports tree working with Clang required a number of bug fixes.

4.1 Give up, use GCC

The first stopgap measure was to add a flag to the ports tree allowing ports to select that they require GCC. At the coarsest granularity is the `USE_GCC` flag knob, which allows a port to specify that it requires either a specific version of GCC, or a specific minimum version.

This is a better-than-nothing approach to getting ports building again, but is not ideal. There is little advantage in switching to a new base system compiler if we are then going to use a different one for a large number of ports. We also encounter problems due to GCC's current inability to use `libc++`, meaning that it is hard to compile C++ ports with GCC if they depend on libraries that are built with Clang, and vice versa. Currently around 1% of the ports tree requires this. Quite a few more use the flags exposed in the `compiler` namespace for the port's `USES` flags. In particular, specifying `USES=compiler:openmp` will currently force a port to use GCC, as our Clang does not yet include OpenMP support.

This framework allows ports to specify the exact features of GCC that they require, allowing them to be switched to using Clang once the

4.2 The default dialect

One of the simplest, but most common, things to fix was the assumption by a lot of ports that they could invoke the `cc`, program and get a C89 compiler. POSIX97 deprecated the `cc` utility, because it accepts an unspecified dialect of C, which at the time might have been K&R or C89. Over a decade later, some code is still trying to use it. Today, it may require K&R C (very rare), C89 (very common), C99 (less common), or C11 (not yet common), and so should be explicitly specifying a dialect. This was a problem, because `gcc`, when invoked as `cc` defaults to C89, whereas `clang` defaulted to C99 and now to C11.

This is not usually an issue, as the new versions of the C standard are intended to be backwards compatible. Unfortunately, although valid C89 code is usually valid C99 or C11 code, very little code is actually written in C89. Most C ports are written in C plus GNU extensions. In particular, C99 introduced the **inline** keyword, with a different meaning to the **inline** keyword available as a GNU extension to C89. This change causes linker failures when C89 code with GNU-flavoured inline functions is compiled as C99. For most ports, this was fixed by adding `-fgnu89-inline` to the port's CFLAGS.

4.3 C++ templates

Another common issue in C++ code relates to two-phase lookup in C++ templates. This is a particularly tricky part of the C++ stack and both GCC and Microsoft's C++ compiler implemented it in different, mutually incompatible, wrong ways. Clang implements it correctly, as do new versions of other compilers. Unlike other compilers, Clang does not provide a fallback mode, accepting code with GNU or Microsoft-compatible errors.

The most common manifestation of this difference is template instantiations failing with an unknown identifier error. Often these can be fixed by simply specifying **this**−> in front of the variable named in the error message. In some more complex programs, working out exactly what was intended is a problem and so fixing it is impossible for the port maintainer.

This is currently the largest cause of programs requiring GCC. In particular, some big C++ projects such as the Sphinx speech recognition engine have not had new releases for over five years and so are unlikely to be fixed upstream. Several of these ports will only build with specific version of GCC as well and so are still built with GCC in the ports tree. Fortunately, many these (for example, some of the KDE libraries) are now tested upstream with Clang for Mac OS X compatibility and so simply updating the port to a newer version fixed incompatibilities.

4.4 Goodbye tr1

C++ Technical Report 1 (TR1) is a set of experimental additions to C++ that were standardised in between C++03 and C++11. It provided a number of extensions that were in headers in the `tr1/` directory and in the std :: tr1 namespace. In C++11, these were moved (with some small modifications) into the standard header directory and namespace.

The new C++ stack is a full C++11 implementation and does not provide the TR1 extensions to C++98. This means that code that references these will fail, complaining about a missing header. The simple fix for this is just to globally delete tr1 from the source files. Getting the code to also build with GCC is somewhat more problematic, but can be accomplished with a relatively small set of **#ifdef**s.

4.5 _Generic problems

In FreeBSD 10, we improved some of the generic macros in `math.h` to use the C11 _Generic expressions or GCC's type select extension if available. The old code dispatched arguments to the correct function by comparing **sizeof**(arg) against **sizeof**(**double**) and so on. Now, we are able to explicitly match on the type. Macros such as isnan() and isinf () will now raise compile-time errors if they are invoked with a type that is not one of the compatible ones.

This is something that we consider a feature. If you pass an **int** to isnan(), then you probably have a bug because there are no possible values of an **int** that are not numbers. Unfortunately, a surprising amount of code depends on the previous buggy behaviour. This is particularly prevalent in configure scripts. For example, Mono checks whether isnan(1) works, which checks whether there is a version of isnan() that accepts an integer argument. If it doesn't find one, then it provides an implementation of isnan() that accepts a **double** as the argument, which causes linker failures.

Fixing these was relatively easy, but time consuming. Most of the errors were in configure scripts, but we did find a small number of real bugs in code.

4.6 OpenMP

One of the current limitations of Clang as a C/C++ compiler is its lack of OpenMP support. OpenMP is a pragma-based standard for compiler-assisted parallelism and so is increasingly important in an era when even mobile devices have multiple cores. Intel has recently contributed an OpenMP implementation to Clang, but the code has not yet been integrated. This implementation also includes a permissively licensed OpenMP runtime, which would replace the GNU OpenMP library (`libgomp`).

Work is currently underway to finish importing the OpenMP support code into Clang. This is expected to be completed by LLVM 3.5, although some extra effort may be required to build the OpenMP support library on FreeBSD (Linux and Mac OS X are its two current supported configurations).

5 Looking forwards

Having a mature and easily extensible library-based compiler infrastructure in the base system provides a number of opportunities.

5.1 Install-time optimisation

A common misconception of LLVM, arising from the VM in its name, is that it would allow us to easily compile code once and run it on all architectures. LLVM uses an intermediate representation (IR) in the middle of the compiler pipeline. This is not intended as a distribution format or as a platform-neutral IR, in contrast to .NET or Java bytecode. This is an intrinsic problem for any target for C compilation: once the C preprocessor has run, the code is no longer target-neutral and much C code has different paths for things like byte order or pointer size.

Although LLVM IR is not architecture neutral, it is *microarchitecture* neutral. The same LLVM IR is generated for a Xeon and an Atom, however the optimal code for both is quite different. It would be possible for a significant number of ports to build the binary serialisation of LLVM IR ('bitcode') and ship this in packages. At install time, the `pkg` tool could then optimise the binaries for the current architecture.

To avoid long install times, packages could contain both a generic binary and the IR, allowing the IR to be stripped for people who are happy to run the generic code, or used for optimisation as a background task if desired. It's not clear how much overhead this would add to installation. Building large ports can be time consuming, however the slowest to build are typically C++ ports where the build time is dominated by template expansion. Generating a few megabytes of optimised object code from LLVM IR typically only takes a few seconds on a modern machine.

Microarchitectural optimisations are not the only applicable kind that could benefit from this approach. Link-time optimisation can give a significant speedup by doing interprocedural analysis over an entire program and using these results in optimisation. Typically, the boundary for this is a shared library, because you can not rely on code in a shared library not changing. If we are shipping both LLVM IR and binaries, however, it becomes possible to specialise shared libraries for specific executables, potentially generating much better code. The down side of this is that you end up without code shared between users of a library, increasing cache churn.

Fortunately, there is information available on a system about whether this is likely to be a good trade. The package tool is aware of how many programs link to a specific library and so can provide hints about whether reduction in code sharing is likely to be a problem. If you have a shared library that is only used by a single program, obviously you don't get any benefits from it. The kernel may also be able to profile how often two programs using the same library are running simultaneously (or after a short period) and so gaining any benefit from the sharing.

Of course, these are just heuristics and it may be that some library routines are very hot paths in all of their consumers and so would benefit from inlining anyway.

5.2 Code diversity

LLVM has been used by a number of other projects. One interesting example is the Multicompiler [3], which implements code diversity in LLVM with the goal of making return-oriented programming (ROP) more difficult. ROP turns the ability for an attacker to run a small amount of arbitrary code (e.g. control the target a single jump, such as a return instruction) into the ability to run large amounts of code. This works by stringing together short sequences of instructions ('gadgets') in a binary, connected by jumps. Gadgets are particularly common in code on x86, because the variable-length instruction encoding and byte alignment of instructions mean that a single instruction or instruction pair can have a number of different meanings depending on where you start interpreting it.

The Multicompiler combats this in two ways. First, it can insert nops into the binary, breaking apart particularly dangerous accidental sequences. Second, using a random seed, it performs various permutations of the code, meaning that different compiles can end up with code (including the surviving gadgets) in different places.

We are currently working to incorporate the multicompiler into the ports tree, so that users building site-local package sets can set a random seed and get deterministic builds that are nevertheless different in binary layout to those produced by everyone else. This makes generating an exploit that will work on all FreeBSD systems very difficult. We will also be able to incorporate this into the FreeBSD-provided binary packages, quickly running diversified builds when a vulnerability is found, requiring attackers to create new versions of their exploits. By rolling these out in a staggered fashion, we can make it hard to write an exploit that will work on all FreeBSD users, even within a single package version.

5.3 Sanitiser support

Clang, on Linux and Mac OS X, supports a number of 'sanitisers', dynamic checkers for various kinds of programming error. The compiler identifies particular idioms and inserts checks that are evaluated at run time and may potentially call routines in a supporting library. These include:

AddressSanitizer was the first of the family and is intended to provide similar functionality to Valgrind [6], with a much lower overhead. It detects out-of-bounds accesses, use-after-free and other related memory errors.

MemorySanitizer checks for reads of uninitialised memory. This catches subtle bugs where code can work fine on one system because memory layout happens to contain valid values, but fail on another.

ThreadSanitizer is intended to detect data races.

UndefinedBehaviorSanitizer performs run-time checks on code to detect various forms of undefined behaviour. This includes checking that **bool** variables only contain **true** or **false** values, that signed arithmetic does not overflow, and so on. This is very useful for checking portable code, as undefined behaviour can often be implemented in different ways on different platforms. For example, integer division by zero may trap on some architectures but may silently give a meaningless result on others.

DataFlowSanitizer allows variables to be labelled and their flow through the program to be tracked. This is an important building block for a category security auditing tools.

All of these require a small runtime library for supporting functionality, including intercepting some standard C library functions (e.g. malloc() and free()). These have not yet been ported to FreeBSD, but would provide significant benefits if they were. In particular, running the FreeBSD test suite with dynamic checks enabled on a regular basis would allow early detection of errors.

5.4 Custom static checks

The Clang static analyser provides generic functionality for understanding control and data flow inside compilation units. It also includes a number of checkers for correct usage of the relevant languages, for example checking that variables are not used uninitialised and NULL pointers are not dereferenced in all possible control flows. The more useful checks are those that incorporate some understanding of API behaviour.

By default, the analyser can check for correct usage of a number of POSIX APIs. Apple has also contributed a number of checkers for OS X kernel and userspace APIs. The framework is sufficiently generic that we can also provide plugins for FreeBSD APIs that are commonly misused.

Some of the checkers would be of more use if we provided more annotation in the FreeBSD code. For example, WITNESS allows dynamic lock order checking, but Clang can also perform some of these checks statically. It can also do some more subtle checks, for example ensuring that every access to a particular structure field has a specific lock acquired. Ideally, the static analyser would be combined with WITNESS, to elide run-time checks where static analysis can prove that they are not required.

5.5 Other analysis tools

The LLVM framework has been used to implement a number of other analysis tools. Of particular relevance to FreeBSD are SOAAP [4] and TESLA [1], which were both developed at the University of Cambridge with FreeBSD as the primary target.

TESLA is a framework for temporal assertions, allowing the programmer to specify things that must have happened (somewhere) before a line of code is reached, or which must happen subsequently. A typical example is that within a system call, by the time you get to the part doing I/O, some other code must have already performed a MAC check and then something else must later write an audit log event. These complex interactions are made harder to understand by the fact that the kernel can load shared libraries. TESLA uses Clang to parse temporal assertions and LLVM to instrument the generated code, allowing them to be checked at run time. A number of TESLA assertions were added to the FreeBSD kernel in a branch and used to validate certain parts of the system.

SOAAP is a tool to aid compartmentalising software. This is an important mitigation technique, limiting the scope of compromises. The Capsicum [9] infrastructure provides the operating system functionality required for running low-privilege sandboxes within an application but deciding where to place the partitions is still a significant engineering challenge. SOAAP is designed to make it easy to explore this design space, by writing compartmentalisation hypotheses, ensuring that all shared data really are shared, and simulating the performance degradation from the extra process creation and communication.

We anticipate a lot more tools along these lines being developed over the coming years and intend to take advantage of them.

5.6 The rest of the toolchain

We currently include code from either GNU binutils or the ELF Toolchain project. Most of this duplicates functionality already in LLVM. In particular, every LLVM back end can parse assembly and generate object code, yet we still have the GNU assembler. Various other tools, such as `objdump` have direct replacements available in LLVM and some others (e.g. `addr2line` would be simple wrappers around the LLVM libraries). The only complex tool is the linker.

There are two possible linkers available, both based on LLVM: MCLinker [2] and lld [8]. MCLinker is currently able to link the entire FreeBSD base system on i386, but lacks support for version scripts and so the resulting binaries lack symbol versions. It is a promising design, performing well in terms of memory usage and speed.

Lld is developed primarily by Sony and is part of the LLVM project. It is currently less mature, but is advancing quickly. Both use a scalable internal representation, with some subtle differences, inspired in part by Apple's 64-bit linker. MCLinker aims to be a fast ELF-only linker, whereas lld aims to link all of the object code formats supported by LLVM (ELF, Mach-O and PE/COFF). We are likely to import one of these in the near future.

We have already imported LLDB, the LLVM debugger, into the base system, although it was not quite ready in time for the 10.0 release. LLDB uses numerous parts of LLVM. When you type an expression into the GNU debugger command line, it uses its internal parser, which supports a subset of the target language. In LLDB, the expression is parsed with Clang. The parsing libraries in Clang provide hooks for supplying declarations and these are supplied by LLDB from DWARF debug information. Once it's parsed, the Clang libraries emit LLVM IR and the LLVM JIT produces binary code, which is copied into the target process's address space and executed.

5.7 Other compilers in FreeBSD

When people think of compilers in FreeBSD, the C and C++ compilers are the most obvious ones. There are a number of others, for domain-specific languages, in various places. For example, the Berkeley Packet Filter (BPF) contains a simple hand-written JIT compiler in the kernel. This produces code that is faster than the interpreter, but not actually very good in absolute terms.

Having a generic compiler infrastructure for writing compilers allows us to replace some of these with optimising compilers. In a simple proof of concept for an LLVM-based BPF JIT (a total of under 500 lines of code, implementing all of the BPF bytecode operations), we were able to generate significantly better code than the current in-kernel JIT. The LLVM-based JIT in its entirety (excluding LLVM library code) was smaller than the platform-dependent code in the in-kernel JIT and will work for any architecture that LLVM supports, whereas the current JIT only supports x86[-64].

It is not a simple drop-in replacement, however. LLVM is large and does not gracefully handle low-memory conditions, so putting it inside the kernel would be a terrible idea. There are two possible solutions to this. The first is to run the JIT in userspace, with the kernel streaming BPF bytecodes to a device that a userspace process reads, compiles, and then writes the generated machine code back into. The kernel can use the interpreter for as long as the userspace process takes to perform the compilation. The alternative is to use the NetMap [7] infrastructure to perform packet filtering entirely in userspace.

This is less attractive for BPF, where rule sets tend to be fairly simple and even the interpreter is often fast enough. It is more interesting for complex firewall rules, which change relatively infrequently (although the state tables are updated very often) and which can be a significant bottleneck.

6 Platform support

FreeBSD currently supports several architectures. We have enabled Clang/LLVM by default on x86, x86-64, and ARMv6 (including ARMv7). This leaves older ARM chips, SPARC, PowerPC, MIPS, and IA64 still using GCC. Support is progressing in LLVM for SPARC, PowerPC and MIPS.

We are able to compile working PowerPC64 kernels without optimisation, but there are still some optimisation bugs preventing Clang from becoming the default compiler on this architecture. On 32-bit PowerPC, LLVM still lacks full support for thread-local storage and position-independent code. SPARC support is progressing in LLVM, but it has not been recently tested.

We are currently compiling significant amounts of MIPS code (including FreeBSD libc) with LLVM and a large patch set. This includes significant improvements to the integrated assembler, but also support for MIPS IV. Currently, LLVM supports MIPS32, MIPS32r2, MIPS64 and MIPS64r2. The earlier 64-bit MIPS III and MIPS IV ISAs are still widespread. The changes required to support these in the back end are not very complex: simply disable the instructions that are not present in earlier ISA revisions. They should be upstreamed before LLVM 3.5 is released.

The (unfinished) IA64 back end in LLVM was removed due to lack of developer interest. It is unlikely that this architecture will ever be supported in LLVM, and it is doubtful that it has a long-term future in FreeBSD, as machines that use it are rare, expensive, and unlikely to be produced in the future.

7 Summary

Importing LLVM and Clang into the FreeBSD base system and switching Tier 1 platforms to use it was a significant amount of effort. So far, we have only just started to reap the benefits of this work. Over the next few years, LLVM is likely to be an important component of the FreeBSD base system.

This paper has outlined a few of the possible directions. It is likely that there are more that are not yet obvious and will emerge over time.

8 Acknowledgements

The task of importing LLVM into FreeBSD was begun by Roman Divacky and Pawel Worach in 2009 and without their initiative none of this work would have been possible. Since then, a large number of FreeBSD developers have worked to improve the state of the LLVM toolchain in FreeBSD, including Dimitry Andric, Ed Maste, Brooks Davis, Ed Schouten, and many others.

Portions of this work were sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-10-C-0237. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advan ced Research Projects Agency or the Department of Defense.

References

[1] Temporally enhanced security logic assertions (TESLA). `http://www.cl.cam.ac.uk/research/security/ctsrd/tesla/` (accessed 31/1/2014).

[2] Chinyen Chou. MCLinker BSD. In *BSDCan*, 2013.

[3] Michael Franz, Stefan Brunthaler, Per Larsen, Andrei Homescu, and Steven Neisius. Profile-guided automated software diversity. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–11, Washington, DC, USA, 2013. IEEE Computer Society.

[4] Khilan Gudka, Robert N. M. Watson, Steven Hand, Ben Laurie, and Anil Madhavapeddy. Exploring compartmentalisation hypotheses with soaap. In *Proceedings of the 2012*

IEEE Sixth International Conference on Self-Adaptive and Self-Organizing Systems Workshops, SASOW '12, pages 23–30, Washington, DC, USA, 2012. IEEE Computer Society.

[5] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[6] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.

[7] Luigi Rizzo and Matteo Landi. Netmap: Memory mapped access to network devices. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 422–423, New York, NY, USA, 2011. ACM.

[8] Michael Spencer. lld - the LLVM Linker. In *EuroLLVM*, 2012.

[9] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical capabilities for unix. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association.