Compiling C and C++ Programs for Dynamic White-Box Analysis*

Zuzana Baranová and Petr Ročkai

Faculty of Informatics, Masaryk University
Brno, Czech Republic
{xbaranov,xrockai}@fi.muni.cz

Abstract. Building software packages from source is a complex and highly technical process. For this reason, most software comes with build instructions which have both a human-readable and an executable component. The latter in turn requires substantial infrastructure, which helps software authors deal with two major sources of complexity: first, generation and management of various build artefacts and their dependencies, and second, the differences between platforms, compiler toolchains and build environments.
This poses a significant problem for white-box analysis tools, which often require that the source code of the program under test is compiled into an intermediate format, like the LLVM IR. In this paper, we present `divcc`, a drop-in replacement for C and C++ compilation tools which transparently fits into existing build tools and software deployment solutions. Additionally, `divcc` generates intermediate and native code in a single pass, ensuring that the final executable is built from the intermediate code that is being analysed.

1 Introduction

Automation is ubiquitous and essential, and this is no different in software engineering. Processes which are automated are cheaper, they reduce the chances of human error and are generally much more repeatable than processes which involve manual steps. Program compilation is one of the earliest software engineering tasks to have been automated. In addition to its intrinsic merits, build automation forms a key component in other process automation efforts within software engineering: automatic testing, continuous integration and continuous deployment, to name a few.

Another area of software engineering which can greatly benefit from automation is correctness and performance analysis of programs. Of course, this is a highly non-trivial problem and is the focus of intense research. A number of research tools exist and some of them aspire to eventually become production-ready. However, neither program source code nor the resulting machine code is

* This work has been partially supported by the Czech Science Foundation grant No. 18-02177S.

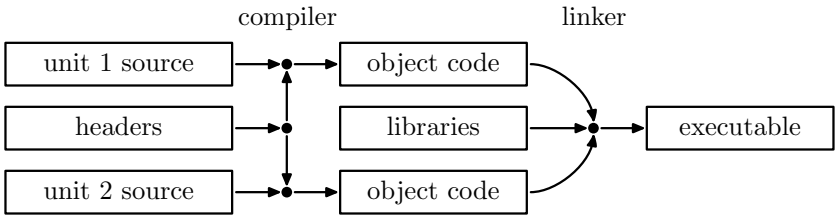

Fig. 1. The process of building an executable from 2 source files.

very convenient for automated analysis: instead, tools prefer to work with intermediate representations, such as LLVM.[1] Analysis tools fall into two coarse categories: static and dynamic. The latter are usually significantly more difficult to integrate into the workflows of large software projects – a deficiency which we aim to address.

Programming languages come in two basic flavours: interpreted and compiled. In the former case, the program is executed directly in its source form. However, interpretation is often deemed inefficient and many programs are written in languages which are first translated into *machine code*. Individual source files are *compiled* into *object code* (a form of machine code which is suitable for *linking* – the process of combining multiple compilation units to form a single program). The process that encompasses both compilation of the individual translation units, as well as the subsequent linking is then known as *building* (see Figure 1). A number of programs and/or libraries may result from a single build.

Since the build process is often complex, software implemented in compiled languages – especially C and C++ – usually ships with comprehensive *build instructions* which are automatically processed by a *build system*. Besides simply invoking the compiler and the linker, those build instructions often deal with building the software on different platforms and operating systems, locating build-time *dependencies*[2] and checking that a suitable version is correctly installed and so on.

1.1 Motivation

One of the first tools which is discovered in the configuration phase of a build is the system C compiler: it is common practice to use the compiler to perform subsequent platform checks. It is typically assumed that the compiler used in

[1] Of course, tools which work with machine code, known as black-box tools, do exist, but their use in software development is limited – they are mainly used in software forensics. In this paper, we focus on white-box methods, which work with source code or an intermediate representation thereof.

[2] A separate (often third-party) software package which needs to be installed in the system before the build can proceed – usually a library, sometimes a tool used in the build process.

the configuration phase of the build is the same as the compiler used to build the software package itself.[3]

Naturally, we would like to take advantage of existing build automation to obtain intermediate code (in our case LLVM IR) which can then be used for correctness and performance analysis. In ideal circumstances, such analysis would also be fully automated and incorporated into the continuous integration process. However, even when employed in mostly manual processes, it is extremely useful to always have the requisite and up-to-date intermediate form available. For this reason, we would like to seamlessly and automatically produce this intermediate form alongside standard libraries and executables.

There is a large number of tools which can benefit from improving the process of obtaining LLVM bitcode for entire programs. Tools that perform dynamic analysis can benefit the most.[4] Some of the tools in this category that use LLVM as their input representation are: the symbolic executor KLEE [3], the slicing-based bug hunting tool Symbiotic [4], the software model checker DIVINE [2] or the MCP model checker [13]. Likewise, stateless model checkers for weak memory models like Nidhugg [1] and RCMC [6] would be significantly easier to use on test cases that use external libraries. Similar benefits apply to bounded model checkers like LLBMC [12] or the LLVM-based IC3 tool VVT [5].

An important consideration is that very few of these tools offer comprehensive support for the standard C library, while support for the C++ standard library or for the widely used POSIX interfaces is even less frequent. Unfortunately, using a tool like `divcc` to build the system C library (`libc`) into a usable bitcode form is still a daunting task and it is not clear whether such bitcode could be sensibly used with any of the abovementioned tools.

Linking an analysis-friendly C library into the bitcode version of the program (providing bitcode definitions of functions that normally come from the system `libc`) effectively side-steps the problem. One of the goals of `divcc` is to make such substitution easy: in Section 3.5, we describe a variant of `divcc` which supplies the bitcode for standard C and C++ libraries provided by DiOS [11]. Out of the tools mentioned above, DiOS and the libraries it includes have been successfully ported to KLEE and DIVINE.

1.2 Related Work

A number of tools with related goals to `divcc` already exist. If we focus on LLVM-based tools, the most well-known tool which integrates white-box analysis into the standard build process is perhaps `scan-build` [7]. This tool shares the same fundamental technique of replacing the C/C++ compiler with a wrapper which, in this case, directly executes the `clang-analyzer` tool on each source file after compiling it using a standard compiler.

[3] Not doing so could lead to configuration mismatches between the two compilers causing build failures, or worse, miscompilation.

[4] This is true even in cases where such tools can work with partial programs – i.e. programs which use functions whose definitions are not available to the tool; however, this mode of operation negatively affects the precision of the analysis.

In the `scan-build` workflow, the compiler is only overridden temporarily, during the execution of `make` or another build tool – it is not in use during project configuration. This means that the compiler wrappers used by `scan-build` can be somewhat more lax about matching the behaviour of the underlying compiler perfectly. It is also simpler in the sense that it does not need to create persistent artefacts and bundle them with standard build products.

Another related tool, this time from the CBMC [8] toolkit is `goto-cc`, which is a `gcc`-compatible compiler which however does not produce executable binaries at all. For this reason, it rather heavily deviates from the behaviour of a standard compiler and as such can only work with comparatively simple build systems, which do not invoke external tools on their build products nor do they execute intermediate helper programs that were compiled as part of the build process.

An important source of inspiration in our effort was the *link time optimization* [10] subsystem of LLVM, which uses a special section in object files[5] to store the bitcode which resulted from compiling the corresponding source unit. In this case, the goal is not program analysis as such, but late-stage program optimization: interprocedural optimization passes can operate more efficiently if they see the entire program at once, instead of just a single unit at a time.

A tool perhaps most closely related to `divcc` is known as `wllvm` (where the 'w' stands for whole-program). Like many of the previously mentioned tools, `wllvm` provides a wrapper for the compiler which performs additional work – in this case, in addition to compiling the unit in a standard way, it runs the compiler again but instructs it to produce bitcode instead of machine code. Unlike the link time optimization system, this bitcode is not stored in the object file – instead, it creates a hidden file next to the original object and embeds the absolute path to the bitcode file in a section of the object file. Subsequently, a special tool called `extract-bc` needs to be used to extract those paths from a build product (a library or an executable) and link it into a single bitcode unit.

The approach taken by `wllvm` has a number of downsides: first, the creation of hidden files deviates from standard compiler behaviour, and sometimes interferes with the operation of build configuration tools. Second, even after installation into a target location, the build products refer to files present in the original build directory which therefore cannot be cleaned up as would be usual, making builds of systems which consist of multiple independent packages more difficult and error-prone. The `wllvm` tool offers workarounds for both these problems, though they are not free of their drawbacks. Even then, integration into automated build orchestration systems which build and package individual components, often in a distributed computing environment without shared file systems, would be very difficult, if at all feasible. On the other hand, the `wllvm` approach has one major upside: there is no need to perform any additional work during the linking stage of the compilation, since the `wllvm`-specific sections are correctly merged by a standard linker, somewhat simplifying the implementation.

[5] And subsequently also in static libraries, which on POSIX systems are simply archives of object files.

The whole problem is side-stepped by tools such as `valgrind` [9], which work directly with machine code and hence do not need any special tooling to integrate into build systems – at most, they re-use existing mechanisms to create *debug-enabled* builds. Finally, middle ground is occupied by the *clang sanitizers* family of dynamic analysis tools. Those tools are integrated in standard compilers (with support in both `clang` and `gcc`), but require special builds which differ from standard debug builds in the compiler flags passed to the compiler. They also require special versions of runtime libraries, which are usually shipped with the compiler in question.

1.3 Contribution

Our main contribution is an open-source tool, `divcc`,[6] which serves similar purpose as `wllvm` but mitigates its problems by taking the LTO-like approach of embedding the entire bitcode in object, library and executable files. Additionally, unlike `wllvm`, our tool first translates the input C or C++ code into bitcode and then compiles that bitcode into native code, saving effort and reducing the chance of discrepancies between the bitcode and the corresponding machine code.

Moreover, our approach allows analysis tools to provide their own header files and bitcode libraries, overriding the host system. This is crucial in scenarios where strict verification is desired, ensuring that only the functionality fully covered by the verification tool is made available to the program during build configuration. Finally, we have evaluated the usability and performance of `divcc` on a number of software packages. We report the results of this evaluation in Section 4.

2 Preliminaries

In this section, we will explain the terms and concepts that are in more-or-less common use and which are directly relevant to the remainder of this paper, most prominently Section 3.

2.1 Storing Machine Code

On UNIX systems, the standard format for storing machine code (i.e. the binary code understood by the CPU) is ELF, short for Executable and Linkable Format. It is the common format for representing files that figure in the process of compilation, such as object files, executables, or libraries.

During the process of building software, machine code exists in a number of related, but distinct forms. It is first generated by the compiler in the form of object code, which is usually stored in an object file. This form of the code is *relocatable*, meaning the routines and variables stored in the file have not been assigned their final addresses. A number of such object files can be bundled together, unaltered, to form a *static library* (also known as an *archive*),

[6] Source code & supplementary material at `https://divine.fi.muni.cz/2019/divcc`

using a special program – `ar`. Finally, object files and archives can be linked into executables or shared libraries. This final step is performed by another program, a linker (often known as `ld`) and consists of resolving cross-references and performing relocations.

The data and code in an ELF file is split into multiple *sections*. There are several well-known sections that have special roles in ELF files, the most important of which are:

- `.text` - contains instructions (machine code) of the program,
- `.data` and `.rodata` - constant-initialized data, e.g. string literals,
- `.bss` - zero-initialized data (the zeroes are not stored in the file).

However, there are no significant restrictions on the number or the names of individual sections. In particular, operating systems or compiler toolchains can create or recognize additional sections with the semantics of their choosing.

2.2 Compiler Architecture

Most C and C++ compilers follow a fairly standard architecture, which is depicted in Figure 2. The entire process is managed by a *driver*, which decides which stages and in what order need to be invoked. The responsibilities of the individual components are as follows:

1. the preprocessor reads the input source file and any header files it may refer to (via `#include` directives) and produces a single self-contained source file,
2. the frontend parses and analyses the source file produced by the preprocessor and generates an intermediate representation out of it,
3. the middle end performs transformations (mainly optimization) on the intermediate format, generating a new version thereof,
4. the backend, or code generator, translates the optimized intermediate representation into object code (i.e. relocatable machine code).

The linker is technically not a part of the compiler: in most cases, it is a separate program. However, it is usually the compiler driver that is responsible for executing the linker with the correct arguments – the linker then simply performs the tasks requested by the compiler. The selection and order of object files (including the system-specific components linked into every program, like `crt0.o`) and libraries (including system libraries like `libc`) to be linked is therefore the responsibility of the compiler driver.

Finally, an important consideration is the mechanics of archive linking: unlike shared libraries, which are indivisible and linked into each program in their entirety, or not at all, static libraries retain individual object files. By default, the linker will only include those object files from each archive that are required to provide symbols referenced by files already included. This optimization can influence program behaviour, because unlike shared libraries, global constructors which are defined in object files that are not referenced by the program (directly or indirectly) will not run. It is therefore important to replicate this behaviour in the bitcode linker component of `divcc`.

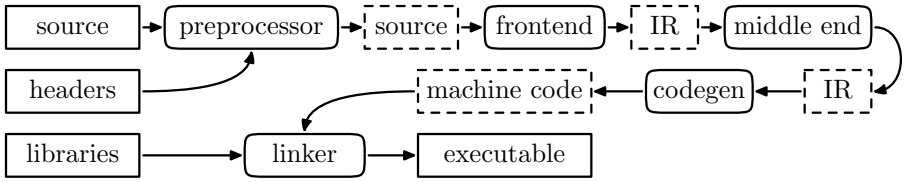

Fig. 2. The architecture of a typical compiler. The rounded boxes represent compiler components, the squares represent data. Dashed boxes only exist as internal objects within the compiler and will not be written into files unless requested by the user or by the build system. Out of the dashed boxes in the picture, typically only object code is written into a file.

2.3 Build Systems

Non-trivial software tends to be composed of numerous source files and header files, which are often organized into multiple libraries and executables. In addition to the source code shipped with the software itself, there are usually dependencies on external libraries and header files, which may be either part of the operating system, or provided by third parties as separate software packages.

Not only is it repetitive and error-prone for the programmer to carry out this process manually, it is also vital to automate it if the software is expected to be built by third parties, who are not sufficiently familiar with it. Many build automation systems have been proposed and implemented. In most cases, the software package is accompanied by build instructions which are read and performed by the build system or build tool in question. For instance, the `make` build system reads a file called `Makefile` which describes the steps for compiling and linking source code. The build process carried out by a typical build system is split into 2 phases:

1. Build configuration is mainly concerned with inspecting the build platform.
   i. The tool, taking into account the build instructions, examines the software installed in the system, to see what is available and whether it is possible to build the program at all.
   ii. To this end, it may attempt to compile and sometimes run *feature tests* – essentially tiny test programs; if the compilation fails, the tool concludes that the tested functionality is unavailable. Alternatively, it may contain a database of known systems and their properties.
   iii. At the end of the build configuration phase, the build tool will store the configuration information (like compiler flags and feature macro definitions) in a form which can be used during compilation.
2. The build proper, in which the software is compiled and linked.[7] The build system performs the steps specified in the build instructions to produce

[7] Most build systems also attempt to speed up repeated builds by avoiding recompiling files that are unchanged (and whose dependencies also remain up-to-date). This capability is important during development and testing, though of course it adds further complexity to the process.

libraries and executables which make up the package. The instructions are usually quite abstract and the particulars of tool orchestration are left to the build system.

3 Design & Implementation

In this section, we first summarize the functional requirements for a tool which would allow us to seamlessly integrate white-box dynamic analysis into existing build systems and workflows, then we spell out the specific design choices we made, describe the implementation, and discuss its limitations.

3.1 Functional Requirements

Our primary requirement when designing the tool was that it would serve as a drop-in replacement for a C (and C++) compiler. There are multiple issues that need to be considered, mainly to ensure compatibility with existing build systems. Our list of functional requirements for `divcc` is, therefore, as follows:

- compatible interface – avoiding the need to alter existing build instructions,
- compatible output – the build system expects that certain files are created, in a certain format so that it can work with them further,
- compilation – source code is compiled into intermediate and native code
- linking – both intermediate and native code is linked into executables and shared libraries,
- archive support – the handling of intermediate code in archives is semantically equivalent to the handling of object code therein,
- object bitcode – bitcode in object files needs to be stored in a format that can be linked to form shared libraries and executables,
- loadable bitcode – the final result must be in a format that the analysis tool can use as input, ideally with no changes to the tool
- single pass operation – no repeated front-end and middle end invocation, minimizing the overhead introduced by the tool into the build process.

Additionally, we have a few non-functional requirements:

- user-friendliness – this extends the functional requirement that the pre-existing build instructions do not need to be changed,
- re-use existing compiler code (CLang and LLVM),
- make it as easy as possible to keep up with changes in CLang and LLVM.

3.2 Intended Use

The expectation is that for the user, the only difference in building their programs is telling the build system to use `divcc` as the C compiler (and `divc++` as the C++ compiler if the software contains C++ source code), for example:

```
$ wget http://ftp.gnu.org/gnu/gzip/gzip-1.8.tar.gz
$ tar xzf gzip-1.8.tar.gz
$ cd gzip-1.8 && ./configure CC=divcc && make
$ echo hello world | ./gzip - > hello.gz
$ divine check --stdin hello.gz ./gzip -f -d -
$ divcc-extract gzip gzip.bc
$ klee -exit-on-error gzip.bc hello.gz
```

Fig. 3. An example use of `divcc` to build and analyse `gzip`.

- `./configure CC=divcc CXX=divc++` (with autotools-based builds)
- `cmake -DCMAKE_C_COMPILER=divcc -DCMAKE_CXX_COMPILER=divc++`
- `make CC=divcc CXX=divc++` (with plain `make`-based builds)

The remainder of the build process should be unaffected. If the analysis tool supports loading of bitcode from executables, it can be directly used. Otherwise, the `divcc-extract` helper script can extract a standalone bitcode (`.bc`) file corresponding to the given executable. The entire process is illustrated in Figure 3.

3.3 Design

To achieve our goals, we need to modify the flow of data through the compiler in a few places (the original data flow is illustrated in Figure 2, the modifications are highlighted in Figure 4). First, we need to obtain the intermediate representation after the middle end, so that we can store it alongside machine code in the object file. This also means that we need to alter the path on which the object file is written by the compiler, so that we can actually include the bitcode section[8] in it. The implementation of these alterations in the data flow is explained in Section 3.4. The other component which needs to be modified is the part of the compiler driver which supervises the invocation of the linker.

1. Like with the changes in the compiler proper, we need to alter the data flow – this time, to extract the bitcode from constituent object files and libraries, and to include linked bitcode in the output of the linker.
2. We need to include a new component: a bitcode linker, which combines the input bitcode files into a single bitcode module which can be inserted into the output of the linker.

As noted in Section 2.2, the bitcode linker needs to follow the semantics of the native linker, specifically when dealing with archives. While a bitcode linker is part of LLVM, this linker can only combine individual modules and does not directly support linking bitcode archives, much less archives which consist of object files with embedded bitcode. There are essentially three options:

[8] We use a section named `.llvmbc`, which is the same as the LTO subsystem. This section is recognized by some LLVM tools and is the closest there is to a 'standard' way to embed bitcode in object files.

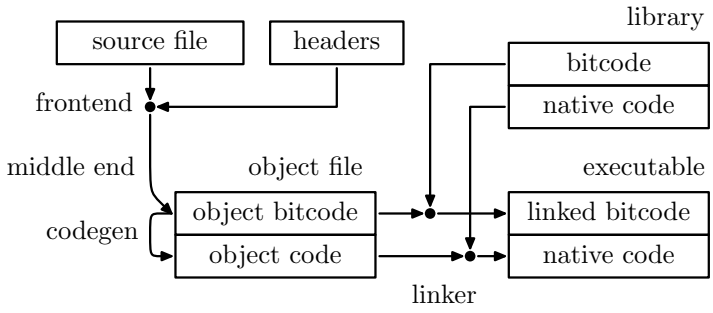

Fig. 4. The flow of the compiled code within `divcc`. The source code along with included header files is processed by the frontend and then middle end to generate LLVM IR. This IR is used by the code generator to produce object code, stored within the same object file. The linker then separately combines bitcode and machine code from object files and libraries to produce an executable which again contains both executable machine code and analysable bitcode.

1. Re-use the LTO infrastructure, which uses linker plugins to perform bitcode linking of modules selected by the native linker. This approach has significant portability issues, since it requires the ability to extend the native linker.
2. Use an auxiliary section in a fashion similar to `wllvm` to learn which objects were included by the native linker and perform the link based on those.
3. Extend the existing module-based bitcode linker to handle archive linking semantics.

Even though not the simplest, we have taken option 3, since it has an important advantage of also working with archives which only contain bitcode which is not accompanied by any native code.[9]

3.4 Implementation

The implementation was done in C++ for the following reasons:

- to gain direct access to individual CLang components and utility functions,
- to allow distribution of `divcc` as a self-contained, statically linked binary,
- to avoid the overhead associated with `fork`-based wrappers.

Like upstream CLang, `divcc` will by default use `fork` and `exec` to invoke the system linker for the actual linking of object files. However, it also includes experimental support for using the `lld` linker as a library, avoiding the need to interface with external programs altogether. The construction of the correct linker command is delegated to the upstream CLang driver. Likewise, processing command-line switches is mainly done by existing CLang code (making interface

[9] This is important with e.g. libraries provided by DiOS, which are normally only compiled into bitcode and packaged into bitcode archives.

compatibility a fairly straightforward matter), as is, obviously, all the heavy lifting of the compilation process itself.

A relatively minor but notable issue is that C++ programs need to link to additional libraries (the C++ runtime support library and the C++ standard library, and any system libraries these two language-specific libraries depend on – usually at least `libpthread`). For this reason, C++ compilers usually provide two binaries, one for compiling and linking C programs and another for C++ programs, the main difference being precisely the libraries which are linked into the program by default. A common solution, which `divcc` adopts as well, is to provide a single binary, which decides whether to use C or C++ mode based on the name it was executed with, so that `divc++` can be made a link to `divcc`.

The final implementation issue is related to functions with variable arguments. LLVM provides a special instruction (`va_arg`) which implements access to arguments passed to a function through ellipsis. Unfortunately, current versions of CLang do not emit this instruction and instead produce an architecture-specific instruction sequence which directly reads the arguments from machine registers or the execution stack. In the context of program analysis, this is far from optimal – for this reason, we alter the behaviour of CLang so that `divcc` instead emits the `va_arg` LLVM instruction.

3.5 Library Substitutions

As mentioned in Section 1.1, it is sometimes desirable to provide alternate, bitcode-only versions of system libraries to make analysis of the resulting bitcode easier. We provide an alternate version of `divcc`, called `dioscc`, that links C programs to the DiOS `libc` and C++ programs also to DiOS versions of `libc++` and `libc++abi`. Likewise, DiOS versions of header files which belong to those libraries are used during compilation. It is straightforward to build additional variants of `divcc` with different substitutions.

The most important issue which relates to library substitutions is ABI compatibility – the property that both libraries use the same in-memory layouts for data structures, same numeric values for various named constants and so on. If ABI compatibility is broken, either the bitcode or the native executable will misbehave. DiOS takes special precautions to make its `libc` binary compatible with the one provided by the host system.[10]

Besides bitcode libraries, `dioscc` includes native versions of `libc++` and `libc++abi`, since different implementations of C++ libraries are usually not binary compatible with each other, and installing multiple versions of the C++ standard library is rather inconvenient. Finally, another native library bundled with `dioscc` is `libdios-host.a`, which contains native versions of functions which are present in the DiOS `libc` but may be missing from the system one.

Please note that unlike `dioscc`, `divcc` uses standard system headers, like any other compiler would, and does not supply bitcode definitions for functions

[10] Unfortunately, `libpthread`, which is also provided by DiOS, is not yet ABI compatible with the host version – see also Section 4.2.

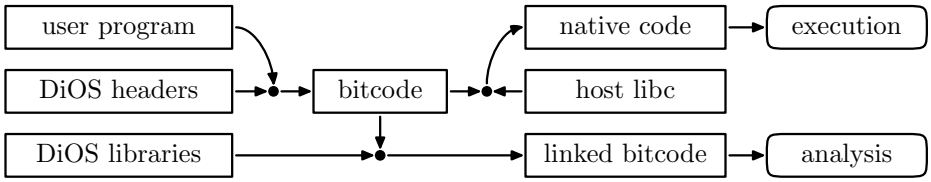

Fig. 5. The compilation process with library substitutions enabled.

from `libc`. It is up to the analysis tool in question to deal with the incomplete bitcode and the platform ABI defined in system headers.

3.6 Limitations

The main compromise in the current implementation is related to shared libraries. When a binary is linked to a shared library, the machine code version is linked in the usual way. However, we still link the bitcode statically, because no analysis tools can currently resolve dynamic dependencies and automatically load the bitcode from shared libraries.[11] The remaining limitations are mainly due to external causes:

1. Inline assembly is compiled to machine code as normal, but the LLVM IR will simply retain the architecture-specific assembly instructions, compromising its usefulness for analysis.[12]
2. When used with DiOS, builds may fail due to missing API coverage or may produce crippled binaries due to ABI compatibility issues. These problems need to be addressed in DiOS.

Finally, a limitation of the implementation (i.e. not inherent in the design) is that `divcc` currently only supports systems which use the ELF format for storing executable code.

4 Evaluation

In this section, we introduce the projects selected for evaluation, report on our findings and note issues we encountered with the build processes. We also provide measurements of build time for each of the packages (shown in Table 1).

[11] This decision may be reversed in a later version, if the situation with support for shared libraries in analysis tools improves.

[12] In some cases, it may be possible to reconstruct platform-neutral LLVM IR using the Remill decompilation library. This is especially pertinent to legacy software which may use inline assembly in applications which would be better served with compiler built-in functions. We will investigate using Remill in this capacity as an option in the future.

4.1 Summary

To evaluate our implementation, we have taken 7 existing C and C++ projects and built them from source using their respective build systems (which meant either CMake or configure, followed by make). Out of the tested projects, Eigen and zlib were built using CMake, the remaining projects used an autotools configure script which generates a `Makefile`.

Table 1. Total elapsed time for the configuration and compilation phases of different software packages. The `clang` and `gcc` columns are baseline compilers which only produce native executables. The `divcc` and `dioscc` columns are variants of the tool proposed in this paper (`divcc` uses native system libraries with no bitcode, `dioscc` uses DiOS replacements for `libc` and `libc++`). We included `wllvm`, which is an existing tool with similar goals to `divcc`, as another reference point. The `make` command was run with 4 jobs in parallel using `-j4`.

	`gcc`	`clang`	`divcc`	`dioscc`	`wllvm`
coreutils	1:33 + 0:22	2:28 + 0:31	3:59 + 0:42	3:46 + 0:45	7:01 + 1:04
db	0:18 + 0:20	0:33 + 0:26	0:43 + 0:45	0:51 + 0:47	1:11 + 0:53
eigen	0:14 + 0:00	0:21 + 0:00	0:27 + 0:00	0:30 + 0:00	0:34 + 0:00
gzip	0:21 + 0:02	0:45 + 0:04	1:04 + 0:05	1:13 + 0:05	2:08 + 0:09
libpng	0:05 + 0:09	0:10 + 0:10	0:16 + 0:11	0:17 + 0:12	0:28 + 0:18
sqlite	0:05 + 1:23	0:11 + 2:03	0:17 + 2:08	0:20 + 2:12	0:27 + 3:24
zlib	0:02 + 0:01	0:03 + 0:02	0:05 + 0:03	0:05 + 0:03	0:06 + 0:04

Each project was built in 5 configurations: with `divcc`, `dioscc`, CLang version 8, GCC version 8.3 and `wllvm`. All tools have built all the projects successfully, with some caveats described in Section 4.2.

- `coreutils` 8.31 is a set of over 100 GNU core utilities and various helper programs for file and text manipulation (such as `cat` or `ls`) and shell utilities (`env`, `pwd`, and others),
- `gzip` 1.10 – a data compression and decompression utility,
- Eigen 3.3.7 [C++] is a header-only template library that provides linear algebra structures, such as matrices and vectors and operations on them,
- SQLite 3.28.0 – a widely used SQL database engine for database management
- BerkleyDB 4.6.21 – another database management library, more closely coupled with the application
- `libpng` 1.6.37 – a library for reading and writing PNG image files
- `zlib` 1.2.11 – a compression library, included because it is required by `libpng`

The measurements (Table 1) show that the implementation is slightly slower than upstream CLang, in both configuration and building of the software, which is not surprising as bitcode manipulation incurs overhead. This is, however, not a significant cost when compared to `wllvm`, which compiles source code in two passes. The times for `wllvm` also exclude the additional time required

to link the bitcode when `extract-bc` is executed and the reason for the time advantage of `wllvm` when configuring Berkeley DB is that it had to be given the `WLLVM_CONFIGURE_ONLY=1` flag during configuration, as the bitcode files it otherwise produces were confusing the build system. Finally, GCC proved to be considerably faster than CLang and the remaining compilers (which are all based on CLang).

4.2 Package Details

Eigen This was the only project of the selection which uses CMake exclusively. Since it is also a header-only library, the build instructions mainly exist to build tests (with `make buildtests`) or build and run them (`make check`).[13] As some of the tools we used did not manage to build all test files, we did not include compilation of the tests in the time measurements.

Berkeley DB In this case, shared libraries have been disabled (using the `--disable-shared` configure flag), to include at least one statically-built library in the evaluation. In `dioscc`, several of the binaries result in a segmentation fault when run. This is due to the use of the `libpthread` library, as the system version is not ABI compatible with the DiOS `libpthread`.

In this case, it was also necessary to run the `configure` script specially for `wllvm`, passing `WLLVM_CONFIGURE_ONLY=1` in the environment.

SQLite This package was configured with `--disable-dynamic-extensions` because DiOS (and hence `dioscc`) does not currently support the `dlopen` family of functions. SQLite further exhibited the same problem as Berkeley DBD when built with `dioscc` due to ABI incompatibility of `libpthread`.

libpng This package was partly included in the evaluation since it has a dependency on a 3rd-party library, namely `zlib`. We built `zlib` version 1.2.11 using the same tool as `libpng` and provided the resulting `libz.so` or `libz.a` to `libpng` at configure time – in this case, we built both a static and a dynamic variant of `libpng` (along with a matching build of `zlib`).

5 Conclusions

We have designed and implemented a tool which makes integration of dynamic program analyses based on LLVM into the build and development processes significantly easier. Our design takes the best ideas from a number of related tools and combines them in a unique way to offer seamless integration into existing processes. Moreover, `divcc` optionally integrates with DiOS, making the resulting bitcode more analysis-friendly without compromising the guarantees stemming from the use of the same bitcode for native code generation and for analysis. Finally, we have evaluated `divcc` on a number of existing software packages, establishing its practicality and efficiency.

[13] This is the reason for zero build time of Eigen for all compilers.

Bibliography

[1] P. A. Abdulla, S. Aronis, M. F. Atig, B. Jonsson, C. Leonardsson, and K. Sagonas. Stateless model checking for TSO and PSO. *Acta Informatica*, 54(8):789–818, 2017. doi: 10.1007/s00236-016-0275-0.

[2] Z. Baranová, J. Barnat, K. Kejstová, T. Kučera, H. Lauko, J. Mrázek, P. Ročkai, and V. Štill. Model checking of C and C++ with DIVINE 4. 2017.

[3] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224. USENIX Association, 2008.

[4] M. Chalupa, M. Vitovská, M. Jonáš, J. Slabý, and J. Strejček. Symbiotic 4: Beyond reachability. In *TACAS*, volume 10206 of *LNCS*, pages 385–389. Springer, 2017.

[5] H. Günther, A. Laarman, and G. Weissenbacher. Vienna Verification Tool: IC3 for parallel software (competition contribution). In *TACAS*, pages 954–957, 2016. doi: 10.1007/978-3-662-49674-9_69.

[6] M. Kokologiannakis, O. Lahav, K. Sagonas, and V. Vafeiadis. Effective stateless model checking for C/C++ concurrency. *Proc. ACM Program. Lang.*, 2(POPL):17:1–17:32, 2017. doi: 10.1145/3158105.

[7] T. Kremenek et al. scan-build, 2009. URL `https://clang-analyzer.llvm.org/scan-build.html`.

[8] D. Kroening and M. Tautschnig. CBMC – C bounded model checker. In *TACAS*, pages 389–391. Springer, 2014. doi: 10.1007/978-3-642-54862-8_26.

[9] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[10] T. L. Project. LLVM Link Time Optimization, 2019. URL `https://www.llvm.org/docs/LinkTimeOptimization.html`.

[11] P. Ročkai, Z. Baranová, J. Mrázek, K. Kejstová, and J. Barnat. Reproducible execution of POSIX programs with DiOS. In *Software Engineering and Formal Methods*, LNCS. Springer, 2019. URL `https://divine.fi.muni.cz/2019/dios/`. To appear.

[12] C. Sinz, F. Merz, and S. Falke. LLBMC: A bounded model checker for LLVM's intermediate representation. In *TACAS*, volume 7214 of *LNCS*, pages 542–544. Springer, 2012. doi: 10.1007/978-3-642-28756-5_44.

[13] S. Thompson and G. Brat. Verification of C++ flight software with the MCP model checker. In *2008 IEEE Aerospace Conference*, pages 1–9, 2008. doi: 10.1109/AERO.2008.4526577.