Portable Executables

ISA Independent Executables for Linux

Linus Hagemann and Tom Wollnik

Hasso Plattner Institute for Digital Engineering
{firstname.lastname}@student.hpi.uni-potsdam.de

In this report we present an approch to build executables that can be ported between different instruction set architectures (ISAs). Our approach draws on LLVM Intermediate Representation (IR) (CITATION). We make use of a wrapper around `clang` (CITATION) the `LLVM` (CITATION) compiler for the `C` programming language. The IR allows for recompilation of the application if necessary. We provide a prototypical implementation that can build a portable version of `GNU sed` (CITATION). We end with some observations on the drawbacks (e.g. performance impact) of our approach and how they could be mitigated.

1 Context and Motivation

The ability to run programs on CPUs with different Instruction Set Architectures (ISA) is becoming increasingly important. The dominance of Intel x86 CPUs in modern mobile and desktop computers is being challenged by ARM. Apart from this shift in consumer devices, data centers are also becoming more heterogenous (CITATION). For example, one may find Intel x86, ARM, and PowerPC CPUs all in one data center. Also, there is a wide range of possible server configurations, e.g. with regard to the available accelerators. This strengthens the need for executables that can easily be ported between machines with different ISAs and different accelerator configurations.

A successful implementation of ISA independend executables could also support efforts for energy aware computing by making it easier to migrate programs and compute jobs between different machines. This aids the compute load balancing accross servers in a data center based on the energy supply. (CITATION)

2 Related Work

A multitude of approaches for portable binaries have been proposed throughout the years.
In this section we want to highlight some approaches that are strongly related to the one we chose for our project and also briefly comment on the respective differences.

2.1 Fat Binaries

The approach that arguably had the most impact for a wide range of users is known as Fat binaries, in which an executable contains code native to different ISAs. (CITATION) This results in filesizes larger than standard executables, thus their name. Often it is possible to engineer such formats in a way that some parts of the application (e.g. assets) can be used for both ISAs. For this reason Fat binaries can be smaller than the sizes of the respective standard executables combined.

For users the advantage of Fat binaries is that only a single file has to be provided for installation. Therefore, no special knowledge is required to identify which version of a program should be installed. Additionaly, this advantage does not come with a significant performance overhead, since native code is executed and the selection of the appropriate code is supported directly by the operating system. The only disadvantage is larger filesizes.
The following implementations supported only a predetermined combination of ISAs.

2.1.1 Apple's Universal Binaries

Apple's transitions from PowerPC to Intel and from Intel to ARM are well known cases of the usage of Fat binaries. In both instances, native code for both ISAs was contained in a single file. Additionaly, different word lengths could be supported. (CITATION) In June 2020, Apple announced the new version Universal 2 for the transition of x86 to ARM for their Macs and Macbooks. (CITATION)

2.1.2 FatELF

`FatELF` is a prototypical implementation of the Fat binary approach for `GNU/Linux` and its `ELF` (Executable and Linkable Format) (CITATION) format. Different ELF files are concatenated with some additional information in a header. The proof of concept implementation is available for Ubuntu 9.04. However, since currently there is no integration of this work into the mainline Linux Kernel and currently none is planned, involved changes to ones OS are neccesary in order to get support for `FatELF`. (CITATION)

2.2 Binary Translation

Another approach for portable executables is the translation of binary files before their execution. Therefore, the executable itself is not strictly portable but there exists tooling that translates code native to another ISA before a program is executed for the first time. (CITATIO) This process is transparent to the user, although it takes significant time. However, this translation has to happen only once. This makes it overall faster than emulation, if the program is used sufficiently often since in an emulated environment one has to deal with a constant overhead. (CITATION) Modern approaches use different strategies such as caching an memoization in order to translate as few lines of code as possible.

Again, Apple provides a prominent example for this approach with Rosetta and Rosetta 2. (CITATION)

2.3 Comparison to this project

We will see in section 4 that our implementation builds on both of the described ideas.
We recompile code for different ISAs from an architecture independent intermediate representation. Additionaly, we store the generated `ELF` files in order to only compile once per target configuration.
This leaves us with a worse performing solution (both in terms of space and performance) than the solutions above (see 6). Additionaly, we cannot bridge differences in processor word-length.
However, the on demand recompilation makes the provided prototype more flexible: The only limitation for an ISA to be supported should be that a `LLVM` backend exists. The set of supported ISAs does not have to be predetermined and no substantial changes to code, buildtools or operating system are neccesary in order to use our prototype.

3 Requirements and Goals

There are three main requirements for our implementation of portable executables.

The executable should be portable and extensible. Portable executables need to be able to run on machines with different ISAs. Additionally, they should be able to run on machines where different accelerators are available. It should not be neccesary to know the concrete configurations on these setups in advance.

It should be easy to create and run portable executables. Firstly, this means that users should not have to make significant changes to their existing workflows. We want to accomplish this by providing a compiler and a linker that can act as drop-in replacements for the `clang` compiler and linker in C projects.
Secondly, there should be only a minimal administration overhead. They should only use standard dependencies and no configuration should be necesseary to get started.
Thirdly, users who run the portable executable should not notice a difference to running a standard executable. Any startup logic needs to be hidden from the end user.

The first language to support is C. A first prototype should be able to create portable executables from `C` code.

4 Implemented Prototype

Our approach builds on the LLVM Intermediate Representation to generate portable executables. We use a container format. The general idea is that a PEX file contains a `tar` archive with the LLVM Intermediate Representation for the program and executables in the ELF format for different ISAs and compiler flag configurations.

4.1 LLVM Intermediate Representation

The LLVM Intermediate Representation is the foundation of our portable executable format. LLVM IR is an intermediate, assembly-like code format used by the LLVM compiler suite. It is designed to act as an intermediary between compiler frontends for different programming languages and backends for different ISAs. The IR is mostly machine-independent. There are some portability issues, as discussed in section 6.4.

We will now take a look at the IR for a simple Hello World program. Listing 1 shows the `C` code and listing 2 shows the corresponding LLVM IR. We will come back to this example when discussing the limitations of IR portability in section 6.4.

We make use of the LLVM IR's portability to build portable executables. Portable executables in our format (referred to as PEX files) contain the LLVM IR for the program. The specifics are discussed below in section 4.2. When a PEX file is executed on a machine with an ISA unknown to this PEX file, we can compile the program for the new ISA from the IR that is stored in the PEX file.

Listing 1: Simple Hello World program in C.

```c
#include <stdio.h>
int main() {
    printf("Hello World!");
    return 0;
}
```

Listing 2: LLVM IR for the simple Hello World program (shortended).

```
; ModuleID = 'helloworld.c'
source_filename = "helloworld.c"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

@.str = private unnamed_addr constant [13 x i8]
            c"Hello World!\00", align 1

; Function Attrs: noinline nounwind optnone uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  %2 = call i32 (i8*, ...) @printf(
      i8* getelementptr inbounds (
```

```
15                [13 x i8], [13 x i8]* @.str, i32 0, i32 0))
16    ret i32 0
17  }
18
19  declare i32 @printf(i8*, ...) #1
20
21  attributes #0 = {[...] "target-cpu"="x86-64"
22      "target-features"="+fxsr,+mmx,+sse,
23                              +sse2,+x87" [...] }
24  [...]
```

4.2 PEX Format

Having discussed the LLVM IR and how it can be used to achieve portability we will now take a closer look at the file format developed in this project. We use a container format as depicted in figure 1. PEX files consist of a loader Shell script and an integrated `tar` archive.

Tar archive. The `tar` archive in a PEX file contains the LLVM IR for all source files of its program, linker flags, and one or more bundles. A bundle is the compiled program for one ISA or one compiler flag configuration and its object files. Listing 3 shows the contents of the `tar` file for an example Hello World PEX file. The file `helloworld.ll` contains the IR for the single source file of the program. `LINKER_FLAGS` stores the flags that were used to link the different object files when the PEX was first created (i.e. the linker call in the Makefile). The example `tar` archive contains two bundles: `aarch64-unknown-linux-gnu` and `x86_64-pc-linux-gnu`. Each bundle holds the actual executable for one ISA (file `a.out`) and the object files compiled for this ISA. The names of the bundles tell us which ISA and OS the bundles were created for.

The loader script. The loader is a shell script that is run when a PEX file is executed. It unpacks the tar archive, recompiles the program from the IR if necessary, and executes the program. Note that the need for a recompilation is determined by checking if a bundle for the current machine exists (via `clang -dumpmachine`) as a default. However, this behaviour can be overriden. In that case the user specifies a bundle to create/execute. This allows for multiple configurations per ISA and operating system, e.g. due to different available accelerators.

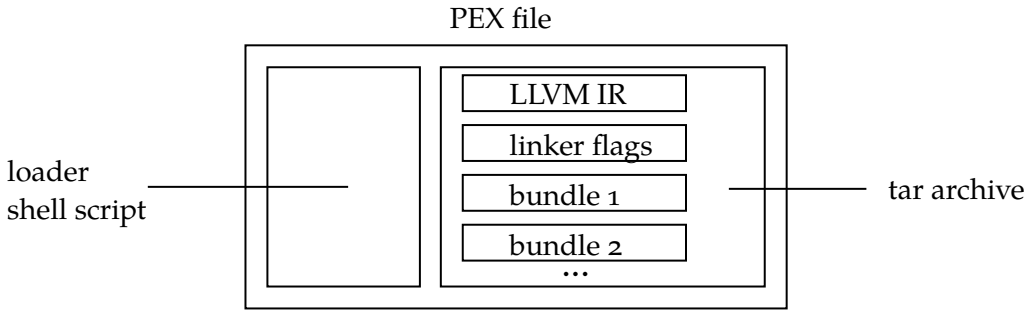

Figure 1: The proposed portable executable format PEX.

Listing 3: Contents of the tar archive for a simple hello world PEX file.

```
IR
   |--helloworld.ll
LINKER_FLAGS
aarch64-unknown-linux-gnu
   |-- a.out
   |-- src
       |-- helloworld.o
x86_64-pc-linux-gnu
   |-- a.out
   |-- src
       |-- helloworld.o
```

4.3 Workflow

In previous subsections we have looked at the PEX format. Now we want to describe how PEX files can be created for `C` projects. Let us take a look at an example Makefile that uses `clang` and compiles a simple `C` project with just two files (Listing 4). To generate a PEX file instead of a standard executable we only need to change the compiler and linker. This is done by setting the `CC` and the `LD` variables in the Makefile appropriately. This modified Makefile creates portable executables in the PEX format.

The `pex` compiler and linker can be used as a drop-in replacements for `clang` in most projects. This makes it simple to start generating PEX files. Note that we currently only support workflows with a single linker call that only links object files (see Section 6.3).

Listing 4: Makefile that compiles a small C project. Left: Original Makefile that uses `clang`. Right: Modified Makefile that uses the pex compiler and linker to create PEX files.

```
CC = clang
LD = clang

a.out: foo.o bar.o
        $(LD) -o $@ $^
%.o: foo.c bar.c
        $(CC) -c -o $@ $<
```

```
CC = pex
LD = pex

a.out: foo.o bar.o
        $(LD) -o $@ $^
%.o: foo.c bar.c
        $(CC) -c -o $@ $<
```

4.4 Managing PEX Files

To make working with PEX files more comfortable we provide a small tool that allows for the inspection and modification of PEX files. The manager program supports the following operations:

- `ls` and `tree`. These commands list the contents of the `tar` archive from a given PEX file using `ls` or `tree` [1].
- `rm`. The manager program can remove a bundle from a PEX file. This is especially useful to force recompilation for an ISA that the PEX file already knows.
- `extract`. Extract the content of the `tar` archive for a PEX file.
- `merge`. This command adds the possibility of merging two PEX files with disjoint bundle names into a new PEX file. Users can now compile a program multiple times using different compiler flags. The resulting PEX files can be merged and users can choose which program version should be used each time the resulting PEX file is executed. This allows for a great degree of flexibility and enables PEX files to use different accelerators depending on the compiler flags that were set for the selected bundle. This feature is directly related to the requirement that the executable should be portable accross accelerator configurations.

4.5 Summary

PEX files consist of a loader script and a `ttar` archive. The `tar` archive contains the IR, linker flags, and one or more bundles. The IR is used to recompile the program for previously unknown ISAs. We also provide a management program.

5 Proof of Concept

To demonstrate a more advanced usage of our prototype we built a PEX file for the text editor GNU `sed` on a x86 machine and an aarch machine. Both builds then could be executed on the other architecture.
We use `sed` here as an example since it is a complex, large, and well-tested `C` project. This makes it suitable as a proof of concept, since presumably all common features of the C language are used. By succesfully compiling `sed` as a PEX file we can therefore expect our tool to work for all `C` programs, within the limitations discussed later in section 6.
To compile `sed` to a PEX we set the `CC` and `LD` variables in `sed`'s Makefile to `pex`.
However, since this build process is more advanced than the processes we looked at before, another modification has to be made in order to receive a working PEX File. In the unmodified `sed` build process some object files are bundled as an `archive` (.a files). `Archives` are currently not supported by our implementation (see section 6 and 7). Therefore, we have to make a final call to `pex` with all object files generated during the unmodified build process. More details on this can be found in the project repository (CITATION).

[1] `https://linux.die.net/man/1/tree`

6 Limitations

In this section we want to discuss some limitations of our current implementation, as well as some limitations of the undelying approach.

6.1 Replicating `clang` Behaviour

A general problem arises from the fact that `pex` replaces `clang` in compilation workflows. Since the behaviour of `clang` differs depending on the given flags our implementation needs to support these behaviours as well. However, because of the variety of possible flags and their combinations we cannot guarantee correct behaviour in all cases. We provide POSIX compliant behaviour for all combinations of the `-o`, `-E`, and `-c` flags.

6.2 Performance Implications

Our approach of recompiling the entire program and bundling files in a `tar` archive has some serious performance implications.

First, a `PEX` file is significantly larger than the corresponding standard `ELF` file. For our `sed` example this means that a PEX generated on a x86 Ubuntu machine without any additional bundles is approximately 24MB large, while the on-system binary is drastically smaller than even 1MB.
The size of a `PEX` file grows with each new architecture it is executed on and with each configuration that is created (i.e. the number of contained bundles).

Second, the current implementation suffers from a siginificant slowdown at startup time. When we execute a `PEX`, the `loader` script is executed inside a Shell process and the `tar` archive is extracted. Depending on whether a configuration corresponding to the current ISA is found, a recompilation from IR is neccesary, as well as recreating the `tar` archive with the new content. For the final execution of the executable contained in the relevant bundle another Shell subprocess is started. For the example of sed we measured the time necessary to execute a simple sed call for a natively compiled sed and our PEX sed. While it took $\sim 0.004s$ for the executions of standard sed, the PEX calls took $\sim 0.02s$ on average. (CITATION REPO) With larger program runtimes the performance implications become less relevant.

6.3 Complex Build Processes

Another limitation lies with more advanced build processes (as seen in section 5) and library usage.
The problem here is that the knowledge about the build process lies completely outside of our tool, e.g. in a `Makefile`. Even in the simple case of bundling multiple object files inside a `.a` file this becomes a problem. The reason for this is that our tool currently assumes the existence of one source file with `IR` per object file in the project, as described in section 4. For the case of `archives` a possible solution to this

problem is discussed in section 7.
Similar problems could also arise when using shared libraries and dynamic linking. However this was not investigated as part of this project.

6.4 LLVM IR

When LLV IR is treated as architecture agnostic multiple problems arise.

6.4.1 `clang` Version Differences

Different versions of `clang` produce IR that is not neccesarily compatible.
For example, our testing machines ran `clang` version `6.0.0-1ubuntu2` (ARM) and `3.8.0-2ubuntu4` (x86). Initially, the IR created on the ARM machine could not be used to compile on x86.
We could track the problem to a single line that could be removed without altering the functionality of the `IR`. However, it cannot be assumed that there are no other cases in which different `clang` versions may create incompatible IRs or that no such cases will be introduced in the future.

6.4.2 Non-trivial ISA Differences and Dependent Code

Since it was possible to compile a working version of `sed` both from IR created on ARM and on x86, it is justified to assume that vast majority of C code can correctly be compiled from the corresponding IR. However, cases in which architecture dependencies exist in the C code remain problematic.[2]

Preprocessor Definitions The `C` preprocessor is run before the IR is created. Therefore architecture dependencies within Macros,... cannot be resolved by recompilation. For example, a definition with `#ifdef __x86_64__` will not be kept in the IR if first compiled on ARM, thus leading to unexpected behaviour if one later uses that IR to compile an executable on x86.

Data Type Sizes Differences in the bit-length of data types are hard coded into the IR, as seen in e.g. listing 2 line 3. Therefore, the current implementation does not allow for portability from e.g. a 16 Bit system to one with a 64 Bit architecture.

Inline Assembler If inline assembler code is present in the program (using `__asm__`,...) the resulting IR will not be portable due to the platform specicifity of the assembler code.

[2] Examples for these can be found in the `caveats` folder of `https://gitlab.hpi.de/osm/portable-binary`.

7 Future Work

As seen in section 6 there are multiple shortcomings of the approach described here and the implementation provided. In this section we want to highlight some areas where substantial improvements can be made.

7.1 Operating System Integration

As we saw earlier, the independence of our software of most parts of the underlying system comes with a significant cost in performance, i.e. startup time. The greatest part of the overhead we observed comes from the fact that we do not execute any ELF directly. Instead multiple other processes are started.
Similar to FatELF an integration via a Kernel patch could be used to solve this problem. But another, less invasive option could also be pursued: One could build an ELF file for the current architecure and then add the other parts of PEX (i.e IR,... see 4) as data in sections. Additionaly a custom handler via `binftm_misc` that triggers the recompilation if necessary should be provided.[3] This could lead to faster startup times with less necessary modifications of ones system. An approach like this also allows for PEX files to be correctly recognized as binary executables (e.g. with the `file` command).

7.2 Additional Language Support

Since we build upon the IR generated by an LLVM Compiler, it should be possible to transfer the presented approach and large parts of the presented implementation to other languages that have an LLVM frontend available.

7.3 Support for More Build Processes

7.3.1 Enhancement of the Presented Implementation

As we saw in section 5 our current implementation does not support linking libraries in the form of `.a` files. Since such files are just `archives` of object files it should be possible to adapt our implementation to extract the IR for each file contained in such an archive. We think this is possible without any major modifications to our format or the already existing logic.

7.3.2 Integration into `clang`

More complicated and advanced future work could also deal with the question if and how an approach for portable binaries could be supported directly by `clang` or other LLVM frontends. This would make adoption of portable binaries easier, since no additional tooling would be necessary. It would also allow for more certainty in regards to the possible usage of flags and allow arbitrarily complex build processes.

[3] `https://gitlab.hpi.de/osm/portable-binary`

7.3.3 How Portable is the LLVM IR?

A central question to the viability of our approach and also most of the proposed future work is: How portable is the IR generated by LLVM? Earlier, we identified some issues (e.g. data type length differences) we currently see as a hard limit for the portability of LLVM IR.
Future research could a) systematically investigate whether the problems named in this work are exhaustive and b) whether they can be mitigated, either due to manipulation one the generated IR or due to changes in the underlying language design.

8 Conclusion

We saw that it is possible to treat LLVM IR as portable between different instruction set architectures and use IR generated on one machine to compile a program on a machine with a different ISA. We presented a prototype that draws on this fact in order to build portable executables. Portability is achieved by recompiling the program if necessary. Integration in existing projects is easy in most cases, since our implementation is a wrapper around `clang`. We were able to demonstrate the potential of our approach by successfully applying it to GNU `sed`.
However, LLVM IR is not completely architecture agnostic and we also could observe implications on filesizes, performance and problems with complicated build processes.
Here we have identified several areas in which future research could improve the presented approach, e.g. regarding the question of how certain compatability problems in the LLVM IR could be mitigated.