Automatic Transformation of Bit-Level C Code to Support Multiple Equivalent Data Layouts

Marius Nita and Dan Grossman

Department of Computer Science & Engineering
University of Washington, Seattle WA 98195-2350, USA
{marius,djg}@cs.washington.edu

Abstract. Portable low-level C programs must often support multiple equivalent in-memory layouts of data, due to the byte or bit order of the compiler, architecture, or external data formats. Code that makes assumptions about data layout often consists of multiple highly similar pieces of code, each designed to handle a different layout. Writing and maintaining this code is difficult and bug-prone: Because the differences among data layouts are subtle, implicit, and inherently low-level, it is difficult to understand or change the highly similar pieces of code consistently.

We have developed a small extension for C that lets programmers write concise declarative descriptions of how different layouts of the same data relate to each other. Programmers then write code assuming only one layout and rely on our translation to generate code for the others. In this work, we describe our declarative language for specifying data layouts, how we perform the automatic translation of C code to equivalent code assuming a different layout, and our success in applying our approach to simplify the code base of some widely available software.

1 Introduction

C is sometimes referred to as a "portable assembly language" because it is implemented for almost every platform and allows direct bit-level access to raw memory. It remains the *de facto* standard for writing low-level code such as debuggers and run-time systems that manipulate low-level data representations or process external file formats. It is therefore quite ironic that C is *not* well-suited to writing *portable* bit-level code.

In practice, for such code to be portable it must support multiple equivalent data formats. A ubiquitous example is big-endian and little-endian byte order. The order of bit-fields in a struct can also vary with compilers. Additional idiosyncratic examples arise with each unusual file format, compiler, or architecture. Though such code is occasionally performance-critical (e.g., network-packet processing), it usually is not (e.g., file-header processing).

Writing portable code is time-consuming and error-prone. Simple web searches reveal hundreds of (known) endianness bugs. Even for bug-free code, data-layout portability often leads to large amounts of code duplication; using Google

L. Hendren (Ed.): CC 2008, LNCS 4959, pp. 85–99, 2008.

Code Search we found approximately one thousand open-source packages with near-identical code segments based on byte- or bit-order. Such code segments are often poorly documented and are difficult to maintain consistently, particularly because the code is inherently low-level.

1.1 Conventional Approaches

We believe the most common approach to supporting multiple equivalent data layouts is, indeed, code duplication and conditionals (typically with the preprocessor) to choose the correct variant. In our experience, a common approach to such duplication is that one version of the code is developed first (e.g., for a big-endian machine), then — perhaps much later when portability issues arise — the code is copied, one copy is edited, and an `#ifdef` chooses between copies. This process is error-prone and leads to maintenance problems as the copies must remain consistent.

A natural alternative is to abstract all data-layout assumptions into helper functions, confining code duplication to the smallest amount of code possible. We believe dismissing any code that does not follow this approach as "poorly written" is naive. First, if a nonportable code segment is already written, changing it to abstract out such assumptions could introduce bugs on mature platforms. Second, the resulting code can be much harder to read and understand: C *is* good at expressing bit-level operations directly so the code is often reasonably clear when specialized to a given data layout.

1.2 Our Approach

We have developed a tool that takes annotated C code and lets programmers (1) write one version of their code assuming one particular data layout and (2) declaratively specify multiple equivalent data layouts. We then perform a source-to-source transformation that automatically generates versions of the code for each data layout. In this way, we retain the coding effort and clarity of writing nonportable code while supporting portability. We simply use compiler technology to do what it does well: transform code to equivalent versions. Moreover, the declarative specifications are concise documentation of data-layout assumptions.

Our approach should fit well with typical software development. Programmers can still write nonportable code first and then add data-layout specifications and "port statements" (described later) to indicate where to perform our source-to-source transformation. Our tool can be used incrementally since only code executed in the lexical scope of a port statement needs transforming. By producing regular C code, the output of our tool can be distributed as open-source software, processed by conventional tools, or edited manually.

1.3 Outline

This paper describes our tool, how our transformation works, and our preliminary experience rewriting portions of the Gnu Debugger and Gnu Binary File

```
1   enum endian { BIG, LITTLE };
2   struct reloc { char idx[3]; char type; };
3   int idx, extern, pcrel, neg, length;
4   struct reloc *reloc;
    ...
5   if (bfd_header_endian(abfd) == BIG) {
6     reloc->idx[0] = idx >> 16;
7     reloc->idx[1] = idx >> 8;
8     reloc->idx[2] = idx;
9     reloc->type = ((extern ?  0x10 : 0) | (pcrel ?   0x80 : 0)
10                 | (neg ?     0x08 : 0) | (length << 5));
11  } else {
12    reloc->idx[2] = idx >> 16;
13    reloc->idx[1] = idx >> 8;
14    reloc->idx[0] = idx;
15    reloc->type = ((extern ?  0x08 : 0) | (pcrel ?   0x01 : 0)
16                 | (neg ?     0x10 : 0) | (length << 1));
17  }
```

Fig. 1. Example BFD code

Descriptor Library with our approach. Section 2 informally presents a real example to give a programmer's view of our extension. Section 3 then describes our language extensions completely. Section 4 describes our implementation, i.e., how we perform the source-to-source transformation. Section 5 describes some preliminary experience. Section 6 describes related work, and Section 7 concludes with several directions for future work.

2 Example

To give a flavor for how our tool works, we demonstrate its use on a code snippet from the Gnu Binary File Descriptor Library (BFD) [17], a library that facilitates working with binary formats such as a.out or ELF. Sections within some of these formats may be stored in either little- or big-endian order. Code that reads or writes these formats comes in two halves, each half handling one data layout.

Example 1 shows a snippet of such BFD code, rewritten slightly for conciseness. `reloc->idx` stores the three low-order bytes in `idx` either left-to-right (lines 6-8) or right-to-left (lines 12-14), depending on the byte order of the header section in the binary format being handled. The field `reloc->type` is an 8-bit piece of data holding six bit-flags and one 2-bit piece of data. Depending on the endianness of the header, the bit data is stored in either left-to-right (lines 9-10) or right-to-left (lines 15-16) order. Notice that the two representations of `reloc->type` are *not* related by the bitwise reverse function. The order of the two bits within `length` remains unchanged.

Changes to this code must be done simultaneously to both halves, taking into account the low-level details about how the representations differ. The bit

```
enum endian { BIG, LITTLE };
struct reloc {
  char idx[3] @ match endian,byte with BIG    -> 0:1:2
                                     | LITTLE -> 2:1:0;
  char type   @ match endian,bit  with BIG    -> 0:1:2:3:4:5:6:7
                                     | LITTLE -> 7:5:6:4:3:2:1:0;
};
int idx, extern, pcrel, neg, length;
struct reloc *reloc;
...
port (bfd_header_endian(abfd), BIG) {
  reloc->idx[0] = idx >> 16;
  reloc->idx[1] = idx >> 8;
  reloc->idx[2] = idx;
  reloc->type = ((extern ?  0x10 : 0) | (pcrel ?   0x80 : 0)
               | (neg ?     0x08 : 0) | (length << 5));
}
```

Fig. 2. Figure 1, rewritten for use with our tool

constants 0x10, 0x08, and 0x80 correspond to bitwise reverse analogues in the opposite half: 0x08, 0x10, and 0x01. The left-shift and bitwise-or on line 9 place the two bits in `length` in `reloc->type`'s bits 6 and 7. Since the representation is reversed on the other endianness, these bits will occupy positions 2 and 3, so a left-shift by 5 on big-endian formats must be accompanied by a left-shift by 1 on little-endian formats. The programmer must understand all these details when writing and changing this code, and ensure that changes to one half are propagated into the other half in a way that respects these low-level implicit relationships between data layouts.

Figure 2 shows the code in Figure 1, rewritten for use with our tool. Two points are worth emphasizing. First, we write only half the code, assuming one data representation. Second, the relationships between the two data representations for each `reloc->idx` and `reloc->type` are made explicit in the field declarations within `struct reloc`.

The extra declaration sections (to the right of `@`) on the two fields define how equivalent data layouts of the same data relate to each other. `match`, `byte`, `bit`, and `with` are built-in keywords, `endian` is a C enumeration type defined by the programmer and inhabited by the constants `BIG` and `LITTLE`, and the colon-delimited sequences specify how positions of the bytes or bits within data change from one data layout to another. The specifications case over the type `endian` and provide a data layout for each of its constants. The keywords `bit` and `byte` define the granularity of the specification — whether the numbers in the colon-delimited sequence denote bits or bytes.

The specification on field `idx` says that `idx` is laid out in two ways, each corresponding to an `endian` constant. The two layouts are the reverse of the

```
enum endian { BIG, LITTLE };
struct reloc { char idx[3]; char type; };
int idx, extern, pcrel, neg, length;
struct reloc *reloc;
...
int tmp = bfd_header_endian(abfd);
switch (tmp) {
case LITTLE: flip0(reloc->idx);
             flip1(& reloc->type);
             break;
case BIG:    break;
}
reloc->idx[0] = idx >> 16;
reloc->idx[1] = idx >> 8;
reloc->idx[2] = idx;
reloc->type = ((extern ?  0x10 : 0) | (pcrel ?   0x80 : 0)
             | (neg ?     0x08 : 0) | (length << 5));
switch (tmp) {
case LITTLE: unflip0(reloc->idx);
             unflip1(& reloc->type);
             break;
case BIG:    break;
}
```

Fig. 3. Our translation applied to the code in Figure 2

other (`0:1:2` vs `2:1:0`), at a byte-level granularity. The declaration on field `type` is at the granularity of bits, but the two representations are not quite the reverse of each other. Bits `5` and `6`, which represent the two `length` bits, remain in the same order.

The constants `BIG` and `LITTLE` associated with the layout declarations are used in the translation of the `port` statement. The `port` statement is written under the assumption that `bfd_header_endian(abfd)` evaluates to `BIG`. If it evaluates to `LITTLE`, our translation assumes that the bytes within `reloc->idx` should be represented in order `2:1:0`, i.e., the reverse of how they would be laid out when `bfd_header_endian(abfd)` evaluates to `BIG`. Therefore, when it evaluates to `LITTLE`, the bytes within `reloc->idx` are reversed prior to entering the body of the `port`. Likewise, the bits within `reloc->type` are shuffled according to the `7:5:6:4:3:2:1:0` specification. When `bfd_header_endian(abfd)` evaluates to `BIG`, the body is simply executed. The end result is that the code in Figure 2 executes exactly as the code in Figure 1, but is shorter, better-documented, and easier to write and maintain.

Figure 3 shows the code generated by our translation for the program in Figure 2. When `bfd_header_endian(abfd)` evaluates to `LITTLE`, the layouts of `reloc->idx` and `reloc->type` are flipped by functions $flip_0$ and $flip_1$ in accord with the specifications attached to the corresponding field declarations. When control enters what used to be the body of the `port` block, the two fields are laid

out as they would be when `bfd_header_endian(abfd)` evaluates to `BIG`, which matches the code's assumptions. When control exits this code, the layouts of the two fields are flipped back into their original forms by functions $unflip_0$ and $unflip_1$. The *flip* and *unflip* functions are automatically generated by our translation from the specifications on the two fields. The bodies of $flip_1$ and $unflip_1$ are shown in Figure 5 on page 94; the code for $flip_0$ and $unflip_0$ is straightforward. An analysis determines the data whose layouts should be flipped/unflipped, by inspecting all variable and field accesses and checking if their declarations contain a layout specification that cases over the type `endian`.

3 Description of the Extension

Having described our tool via an example, we now give a complete description of our annotations, their meaning, and how we perform our automatic translation to support multiple data layouts.

At the syntax level, our extension has two components. First, we extend C's declaration language to allow specifying multiple equivalent data layouts for variables and fields. Second, we introduce a new statement form `port(`e`,`c`){`e'`}` that allows programmers to write code assuming only one data representation. A translation takes code written with our extension and outputs code suitable for passing to a C compiler.

3.1 The Specification Language

A layout specification is written as

$$\texttt{match}\ \tau, g\ \texttt{with}\ c_1\ \texttt{->}\ s_1\ \texttt{|}\ c_2\ \texttt{->}\ s_2\ \texttt{|}\ \ldots\ \texttt{|}\ c_n\ \texttt{->}\ s_n$$

Symbols c_i are C enumeration constants belonging to enumeration type τ and s_i are colon-delimited sequences containing either natural numbers (starting at `0`) or the symbol '`_`'. g denotes the granularity of the specification. Our system supports granularities `bit`, `byte`, and `nibble`. Others are easily added. Layout specifications can appear on local and global variables and struct fields of integral type (`char`, `int`, etc.) and arrays thereof.

A sequence s assigns names to the underlying layout units in the data, according to the granularity g. For example, if the specification on a 4-byte piece of data has granularity `byte`, a sequence `0:1:2:3` assigns names `0` through `3` to the bytes within the data. In addition to numbers, sequences s may contain '`_`' symbols, meaning that the corresponding layout units do not contain useful data and need not be named, e.g., pad bytes. For example, a sequence `0:1:_:_` represents a 4-byte sequence containing two named data bytes and two pad bytes.

A sequence s is not useful in isolation. Two or more, however, can precisely describe how multiple equivalent layouts of the same data relate to each other. For example, the sequences `0:1:2` and `2:1:0` represent two layouts that are related by the reverse function. We say that they are *equivalent* because all the layout units (bits, bytes, nibbles) in one are present in the other.

Each sequence s is associated with a constant c. The set of constants in a specification is used by the `port` block (described in the next section) to specify its assumptions about the layouts of variables and fields used in its body.

Given two layout specification sequences s_1 and s_2, we can generate a *flip* function that takes a piece of data assumed to be laid out as described by s_1 and shuffles it such that the result is laid out according to s_2. In our tool, we must also generate an *unflip* function that undoes this effect, flipping data that is laid out according to s_2 back into its original layout, s_1. Functionally, *flip* is an isomorphism and *unflip* is its inverse. It would be unexpected, from the point of view of the programmer, for either function to "forget" bits or bytes within the data, with the exception of "don't care" (`_`) layout units.

In order to ensure that *flip* and *unflip* functions respect this behavior and to facilitate C code generation, we restrict the set of layout specifications that can be written to those that are *well-formed*. A specification (`match` τ`,`g `with` c_1 `->` s_1 `|` `...` `|` c_n `->` s_n) is well-formed if:

1. Constants do not overlap: $c_i \neq c_j$ when $i \neq j$.
2. All constants inhabiting type τ must be included in the specification.
3. All s_i's are equal in length. The lengths are multiples of 8 bits.
4. For all s_i, no number within s_i appears more than once.
5. Any number that occurs in an s_i must occur in all others.

The first two requirements ensure that specifications are complete and deterministic. Given a constant c, there is exactly one layout per variable or field associated with it. The rest of the requirements ensure that well-formed specifications do not contain any layout sequences that forget or add data. We also assume that the sequence lengths are multiples of 8 bits, as C types have sizes that are multiples of bytes. The assumption aids our code generation, which breaks layouts into bytes.

3.2 The `port` Statement

In addition to layout specifications on declarations, our extension provides a new `port` statement. The statement is provided as a means for programmers to delimit code that probes the in-memory layout of data with multiple possible layouts. Programmers write the body of `port` assuming one layout and the compiler generates code that will work as intended for the other layouts.

A `port` block is written as

$$\texttt{port}(e,\ c)\ \{\ e'\ \}$$

where c is a constant with enumeration type τ. The enumeration type and associated constants c_i in the layout specification language provide the connection between specifications and the `port` statement. For each piece of data used in e' whose declarations carry associated layout specifications at type τ (meaning the specifications case over τ), the constants inhabiting τ represent different ways to lay out the bytes, bits, and nibbles in that data. (Recall that each sequence s_k in

a layout specification corresponds to a constant c_k.) The programmer writes the body of the port block, e', under the assumption that variables and fields used in e' are laid out according to c. This is assumed to be the case when e evaluates to c. If e evaluates to a constant $c' \neq c$, the programmer's assumptions no longer hold. The compiler will then ensure that before e' is executed, the layouts of the variables and fields within e' are laid out according to c, and when control exits e', they are laid out as they were before control reached the `port` block.

Consider the following code, which prints the high-order byte within the layout of a 32-bit integer:

```
int x @ match endian,byte with BIG    -> 0:1:2:3
                              | LITTLE -> 3:2:1:0;
...
port(endianness(), BIG) { printf("%x", ((char*)&x)[0]); }
```

Here, the programmer assumes that `endianness()` evaluates to `BIG` and writes code that is correct for big-endian machines. If `endianness()` evaluates to `LITTLE`, the body of the `port` block is obviously incorrect — it does not print the high-order byte. In this case, the compiler uses the specification attached to `x` to ensure that its bytes are laid out according to the block's assumptions. In this case, the bytes within `x` are reversed.

More precisely, the semantics of `port(`e`,`c`){`e'`}` is as follows:

- If e evaluates to c, no further action is required and the body e' is executed.
- If e evaluates to $c' \neq c$, the layout of every variable and field used within e' with a specification at the type of c will be "flipped" to match the assumption that they are laid out according to c. The layouts are "unflipped" to their original states after e' is executed.

We allow nesting of `port` blocks as long as their associated constants c are of different types, to avoid re-flipping data that was already flipped by an outer `port` block. While we could allow arbitrary nesting and use a simple analysis to avoid re-flipping, in practice it makes little sense to nest blocks in this manner. (E.g., it is akin to nesting a block guarded by `#ifdef LITTLE` within one guarded by `#ifdef BIG`.)

3.3 Translation

Given a program written using our extensions, a translation produces plain C code that respects the semantics outlined in the previous section. The translation transforms the code in the following ways:

- It erases the data-representation specifications from variable and field declarations.
- If a variable or field with multiple layouts is accessed within a `port` block, the translation generates a *flip* function and an *unflip* function. The first flips the layout of the data to accord with the block's assumptions and the latter flips its layout back into its original form.

```
int tmp = e;
switch(tmp) {
  case c1: flip11(& x1); ... flip1m(& xm); break;        /* from c1 to c */
  ...
  case c: break;                                          /* already in c */
  ...
  case cn: flipn1(& x1); ... flipnm(& xm); break;        /* from cn to c */
}
e′;
switch(tmp) {
  case c1: unflip11(& x1); ... unflip1m(& xm); break;    /* from c to c1 */
  ...
  case c: break;                                          /* already in c */
  ...
  case cn: unflipn1(& x1); ... unflipnm(& xm); break;    /* from c to cn */
}
```

Fig. 4. Translation of `port(`e`,`c`){`e'`}`

– It rewrites `port(`e`,`c`){`e'`}` statements to plain C code that calls *flip* functions prior to entering e', executes e', and calls *unflip* functions upon exiting e'.

Given a statement `port(`e`,`c`){`e'`}` where c has enumeration type τ, the translation proceeds as follows. First, we gather all the variables and field accesses $x_1, x_2, \ldots, x_m$ that are used in e' and have associated layout specifications at type τ. Let the set of constants inhabiting τ be $c_1, c_2, \ldots, c_n$. Each of the variables and field accesses x_i will have an associated layout specification that assigns a layout to each constant c_i.

The translation scheme is shown in Figure 4. First, we generate code that evaluates e and saves the result in a fresh temporary *tmp*. Then we generate code that, depending on the result of e, flips the layouts of $x_1, \ldots, x_m$ so that they match the assumptions in e' — that the x_i are laid out according to the specifications corresponding to constant c. We then generate e' unchanged. After e', we generate code that flips the layouts of x_i back to their original forms. For each constant c_i, each variable and field access x_j will have c_i `->` s_i and c `->` s included in its associated specification. The function flip_{ij} changes the layout of x_i from s_i into s and and unflip_{ij} changes it back from s into s_i.

The translation calls flip/unflip functions for exactly the variables that (1) have multiple data layouts and (2) are accessed in the lexical scope of the port block. Notice any references to such variables passed to functions called in the port block will refer to flipped data, i.e., the flipping happens in place.[1] In theory, if two items that need flipping might alias, we need to check for aliasing dynamically to avoid double-flipping (and unflipping). In practice, we have not

[1] Conversely, we do not flip any data that is accessed in a callee but not mentioned directly in the port block. This can be an issue only with global variables or extremely convoluted code and this has not been a problem in practice.

```
void flip1(void * input) {
  char* t0 = (char*)input;
  char t1 = t0[0];
  t0[0] = 0;
  t0[0] |= ((t1 << 7) & 0x80);
  t0[0] |= ((t1 << 4) & 0x40);
  t0[0] |= ((t1 << 4) & 0x20);
  t0[0] |= ((t1 << 1) & 0x10);
  t0[0] |= ((t1 >> 1) & 0x08);
  t0[0] |= ((t1 >> 3) & 0x04);
  t0[0] |= ((t1 >> 5) & 0x02);
  t0[0] |= ((t1 >> 7) & 0x01);
}
```

```
void unflip1(void * input) {
  char* t0 = (char*)input;
  char t1 = t0[0];
  t0[0] = 0;
  t0[0] |= ((t1 << 7) & 0x80);
  t0[0] |= ((t1 << 5) & 0x40);
  t0[0] |= ((t1 << 3) & 0x20);
  t0[0] |= ((t1 << 1) & 0x10);
  t0[0] |= ((t1 >> 1) & 0x08);
  t0[0] |= ((t1 >> 4) & 0x04);
  t0[0] |= ((t1 >> 4) & 0x02);
  t0[0] |= ((t1 >> 7) & 0x01);
}
```

Fig. 5. Flip and unflip functions for `reloc->type` in Figure 2

encountered any code where such aliasing occurred, suggesting it may instead be reasonable and in the spirit of C to make such aliasing an unchecked error.

3.4 Generation of Flip Functions

Our tool generates *flip* and *unflip* functions that mutate the layouts of their input data in-place. The prototype of every flip function has the form `void` *flip*`(void*)`. Variables and fields whose layouts must be flipped are passed to their corresponding flip functions by address.

The body of a flip function breaks its input into bytes, via a cast to `char*`, and saves them in temporary variables. If the specified granularity is `byte`, flipping is a matter of assigning the temporaries into their new locations in the input. For smaller granularities, we generate bit-shifting and masking code to fetch the bits or nibbles from within the temporaries holding bytes and code to assign them to their new locations.

Figure 5 shows the *flip* and *unflip* functions generated by our translation from the specification on field `type` in Figure 2. For each bit in the input layout, we generate code that shifts it to the position specified by the output sequence and masks out the rest of the bits. The result is added to the output sequence by a bitwise-or. Code generation at `nibble` granularity is similar, except the only possible masks are `0xf0` and `0x0f`, and we shift by either 4 or 0 bits.

4 Implementation

Our prototype is implemented as a modification of the CIL frontend for C [14]. CIL inputs a C program, performs a series of transformations to simplify the code into a uniform subset of C, and outputs equivalent, human-readable C code. In addition, one can provide custom transformations that are applied to the intermediate representations before the output phase.

We modified CIL's parser, lexer, and abstract syntax to allow `port` statements and layout specifications in the input language. We then implemented a custom transformation that erases layout specifications and rewrites `port` blocks according to the translation scheme in Figure 4. In addition, *flip* and *unflip* functions are generated and inserted into the output program. We have also experimented with generating preprocessor macros instead of functions.

The tool outputs clean C code suitable for passing to a C compiler. As discussed more thoroughly in Section 5, a standard optimizing compiler is capable of entirely optimizing away byte-level flips when flips are generated as macros, and the overhead induced by bit- and nibble-level flips is manageable.

In our implementation, the annotation language is a syntactic extension to the C language. However, should it be desired, it is easy to encode annotations in stylized comments or empty preprocessor macros, such that an annotated program is still legal (but nonportable) C.

5 Experience

To assess the usefulness of the tool, we applied it to subsets of two pieces of software: the Gnu Debugger (GDB) [16] and the BFD library, which ship together as part of the GDB distribution. Preliminary experience suggests that our tool is a valuable addition to the developer's toolset. It improves readability, shrinks the code base, and aids in minimizing development and maintenance issues associated with code that is duplicated for the purpose of handling multiple data layouts. In the rest of this section, we describe how we simplified part of the GDB/BFD code base, present some quantitative results, discuss the limitations of our tool, and share our experiences modifying GDB/BFD code.

Simplifying the Code Base: To estimate the extent of the code-duplication problem in GDB/BFD, we manually examined 120 files in a source base of roughly 1700 C files and 1 million lines of code. In these files, we recorded 407 occurrences of snippets where multiple versions of the same code were specialized to particular data layouts. We counted roughly 3600 lines of duplicated code: code that could be potentially eliminated with our tool. While we focused on the part of the source base that we believe contains a lot of code doing low-level data processing, there is surely more such code in the part of the enormous source base that we have not inspected.

We applied our tool to 10 of these files, chosen in no particular fashion. Across the 10 files, we found 31 occurrences of highly similar code-pairs with each half specialized to a particular endianness. We used the `port` statement to eliminate half the code in each of these occurrences, totaling 188 lines (2,465 lexical tokens) of code. To ensure that the new code behaved the same as the old handwritten code, 11 data-layout annotations were required, each specifying two possible data layouts.

Two of the annotations (the ones shown in Figure 2) sufficed for 21 of the 31 `port` statements and contributed to eliminating 124 lines (1,894 lexical tokens) of code. The struct type with which they are associated is used by many files

in the BFD code base. Many of the other annotations were localized, on local variables within functions in which the `port` blocks were placed, and affected one or two occurrences of `port`. In one case, 3 annotations affected 2 blocks.

In addition to code being eliminated directly by `port` blocks, some related "scaffolding code" became superfluous. Developers tuck hard-to-understand bit-masks and flags into pairs of macro definitions, such as:

```
#define RELOC_STD_BITS_LENGTH_BIG       0x60
#define RELOC_STD_BITS_LENGTH_LITTLE    0x06
#define RELOC_STD_BITS_LENGTH_SH_BIG    5
#define RELOC_STD_BITS_LENGTH_SH_LITTLE 1
```

The former two are bit-masks used to identify the two `length` bits from Figure 1 in a byte. The latter two are amounts by which to shift left to place the `length` bits in a byte. In each case, big- and little-endian versions of the constants are provided. After applying our tool, half these constants were no longer needed.

Performance: None of our changes had an observable performance impact on GDB. First, none of the code blocks we found and changed were in inner loops or other performance-critical sections. Second, the overhead of our translation is small, as the underlying compiler optimizes our code efficiently.

To gain a preliminary understanding of the performance impact of the generated *flip* code currently generated by our system, we picked `port` statements from the ported BFD source and compared the quality of the generated code to the previous handwritten code. Since our generated code consists of flipping some layouts, executing handwritten code, then unflipping the layouts, the performance overhead consists entirely of executing flip and unflip functions. Of the 31 blocks we ported, the average number of required flips/unflips was 1.5 and the maximum was 3.

We noticed that if we generate preprocessor macros instead of functions, byte-level flips are entirely optimized away by `gcc -O3`. For example, `gcc` produces the same assembly code for `flip(x); y=x[0]; unflip(x);` as it does for `y=x[3];`. This is hardly surprising, as all that is needed for this optimization is copy propagation and dead-code elimination. In their current form, bit- and nibble-level flips are not optimized as efficiently and can add 50%–100% overhead in number of executed instructions. This is expected, as most of the time, the code executed between flips and unflips is roughly the size of a flip body.

Limitations: There were two low-level code-pairs that we could not port to our tool. Take, for example, the following snippet:

```
char valbuf[4];
...
if (TARGET_BYTE_ORDER == BFD_ENDIAN_BIG)
  memcpy (valbuf + (4 - len), val, len);
else
  memcpy (valbuf, (char *) val, len);
```

The code copies `val` into `valbuf` such that the bytes in `val` are right-justified on big-endian targets and left-justified on little-endian ones. Assuming the bytes within `val` are not already reversed on big-endian machines, we cannot write a static specification that handles this pattern, since `len` is a dynamic value. This makes clear the main limitation of our approach: it does not apply when the two equivalent data layouts cannot be specified statically.

Discussion: The process of determining relationships between equivalent data layouts by reading the code was difficult. First, bit-masks and shift constants are hidden under macros that are scattered across header files far away from the code that uses them. Second, code-pairs that touch the layout of data usually only inspect a subset of the underlying bits and bytes, so one must inspect several code-pairs before gaining a thorough understanding of the layout relationships. Third, identifying the code-pairs themselves is a problem, as there are many ways to express conditionals that run code depending on a particular layout. One may use `#ifdef`s or `if` statements, and in each case the predicate may be different (e.g., `bfd_header_endian()` vs. `TARGET_BYTE_ORDER == BFD_ENDIAN_BIG`).

We believe our rewritten code is much easier to understand than the original: Bit-level code is clearly delimited by port statements and the relationships between equivalent layouts are explicit. We do not necessarily advocate rewriting mature subtle bit-level code unless it is already being maintained or modified for other reasons; we did so to evaluate our research on real code used in practice that we did not write. We definitely do advocate using our approach for new code or when making code portable for the first time.

6 Related Work

We are unaware of any prior work that automatically translates bit-level C code to work for multiple data layouts. Our own recent prior work [15] was designed to find type-casts that rely on platform-specific assumptions to be memory-safe. That work, while useful for finding certain classes of bugs related to structure padding and word size, does not address the issues associated with multiple data layouts. More significantly, for bit-level differences such as endianness, our prior work will never find bugs, since assuming the wrong endianness does not violate memory safety — it just produces the wrong answer.

CCured [13], Deputy [3], SAFECode [4], and Cyclone [9] are projects that aim to make C safer and more expressive, in some cases enriching it with new programming abstractions, and compiling it in a way that prevents unchecked errors. These systems are similar to ours in that they perform source-to-source transformations and pass the output to a C compiler. However, they do not facilitate working with multiple data layouts. Programmers must resort to specializing code to each endianness as they would in plain C.

Some analyses over C programs (e.g. [12,19]) assume one bit-level layout for any piece of data, which is useful for precision, but not for writing portable code.

PADS [7], PacketTypes [10], and DataScript [1] are projects that facilitate working with data formats. PacketTypes lets programmers specify the layout of

network packets using a declarative language. Similarly, PADS uses a declarative language to allow specifying arbitrary ad-hoc data formats, both textual and binary, and automatically generates parsers that process the data. Like PADS, DataScript takes declarative specifications for binary formats and generates code that loads and processes binary files. It may be possible to modify software like BFD to use PADS or DataScript for processing binary formats. However, these projects do not handle discrepancies arising from how compilers/architectures lay out data in memory and leave it up to the programmer to handle multiple layouts of the same data.

Other work has focused on making it easier to work with bit-level data. Diatchki *et al.* [5] augment a Haskell-like language with *bitdata*: bit-level entities that can be manipulated in various high-level ways in a type-safe manner. Erlang bit patterns [8] allow pattern matching on binary data. Other projects (e.g. [6]) augment C and C++ with libraries that facilitate working with bit-level data. These projects do not facilitate working with multiple bit-level layouts.

Our *flip* and *unflip* functions are similar to relational lenses [2]. A lens is a pair of functions, *get* and *putback*. One extracts a representation (e.g., XML data) of an element in a concrete domain (e.g., a database entry) and the other puts the representation back into the concrete domain. Unlike *flip* and *unflip*, *get* and *putback* are not exact inverses of each other. That is, *get* is allowed to forget part of the data in the concrete domain.

Finally, some prior work has focused on making it easier to handle similar blocks of code (e.g., Simultaneous Editing [11] and Linked Editing [18]). These systems allow programmers to link together blocks of code that share a high-degree of syntactic similarity, such that modifications to certain regions of one block are automatically propagated to the others. However, they are unaware of semantic relationships: e.g., one cannot cause an index of "0" in a big-endian code block to be propagated as "3" to the corresponding little-endian block.

7 Conclusions and Future Work

We have designed, implemented, and evaluated a tool that provides direct support for writing code that is portable to multiple bit-level data representations. The key novelty is an approach where programmers write their algorithm in C with one representation in mind and declaratively specify what the equivalent representations are. A source-to-source transformation then produces C code with one version of the algorithm for each representation.

While we view our tool as successful, there are improvements that could make it more widely applicable. First, its current requirement that all data layouts be equivalent is too strong for scenarios where word size varies (e.g., 32-bit versus 64-bit machines). Second, for short, performance-critical code segments, our "flip on entry / unflip on exit" implementation strategy may be inferior to a more sophisticated transformation that modified the code segments. However, optimizing byte-endian code like `flip(x); y=x[3]; unflip(x);` into `y=x[0];` is within the capabilities of an optimizing C compiler, as discussed in Section 5.

We would also like to consider automating or semi-automating tasks we still leave with the programmer, such as identifying where port statements are necessary or editing legacy code to use our tool.

References

1. Back, G.: DataScript - A specification and scripting language for binary data. In: ACM Conference on Generative Programming and Component Engineering 2002 (2002)
2. Bohannon, A., Vaughan, J.A., Pierce, B.C.: Relational lenses: A language for updateable views. In: Principles of Database Systems 2006 (2006)
3. Condit, J., Harren, M., Anderson, Z., Gay, D., Necula, G.: Dependent types for low-level programming. In: European Symposium on Programming 2007 (2007)
4. Dhurjati, D., Kowshik, S., Adve, V.: SAFECode: Enforcing alias analysis for weakly typed languages. In: ACM Conference on Programming Language Design and Implementation (2006)
5. Diatchki, I.S., Jones, M.P., Leslie, R.: High-level views on low-level representations. In: ACM International Conference on Functional Programming (2005)
6. Dipperstein, M.: ANSI C and C++ bit manipulation libraries, `http://michael.dipperstein.com/bitlibs/`
7. Fisher, K., Mandelbaum, Y., Walker, D.: The next 700 data description languages. In: ACM Symposium on Principles of Programming Languages (2006)
8. Gustafsson, P., Sagonas, K.: Efficient manipulation of binary data using pattern matching. J. Funct. Program. 16(1) (2006)
9. Jim, T., Morrisett, G., Grossman, D., Hicks, M., Cheney, J., Wang, Y.: Cyclone: A safe dialect of C. In: USENIX Annual Technical Conference (2002)
10. McCann, P.J., Chandra, S.: Packet types: abstract specification of network protocol messages. In: Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (2000)
11. Miller, R.C., Myers, B.A.: Interactive simultaneous editing of multiple text regions. In: USENIX Annual Technical Conference (2002)
12. Miné, A.: Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In: ACM Conference on Language, Compilers, and Tool Support for Embedded Systems (2006)
13. Necula, G., Condit, J., Harren, M., McPeak, S., Weimer, W.: CCured: Type-safe retrofitting of legacy software. ACM Transactions on Programming Languages and Systems 27(3) (2005)
14. Necula, G., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: International Conference on Compiler Construction (2002)
15. Nita, M., Grossman, D., Chambers, C.: A theory of platform-dependent low-level software. In: ACM Symposium on Principles of Programming Languages (2008)
16. The GNU Project. GDB, The GNU Debugger, `http://sourceware.org/gdb/`
17. The GNU Project. GNU Binutils, `http://sources.redhat.com/binutils/`
18. Toomim, M., Begel, A., Graham, S.L.: Managing duplicated code with linked editing. In: IEEE Symposium on Visual Languages - Human Centric Computing (2004)
19. Wilson, R.P., Lam, M.S.: Efficient context-sensitive pointer analysis for C programs. In: ACM Conference on Programming Language Design and Implementation (1995)