

programming with multiple equivalent data layouts

marius nita

We have developed a **small extension** for C that allows programmers to **specify** how the different layouts of data relate to each other in a **declarative language** and write **only one version** of the code. A transformation automatically **generates** the **other versions** of the code.

```
enum endian { BIG, LITTLE };
int x @ match endian, byte with
    BIG -> 1:2:3:4
    | LITTLE -> 4:3:2:1;
port(endianness(), BIG)
{ printf("%x", ((char*)&x)[2]); }
```

The match expression specifies that the four bytes in `x` are **reversed** (4:3:2:1 vs. 1:2:3:4) on the **opposite endianness**. The `port` statement allows the programmer to delimit layout-specific code and **assume** only **one byte order** (BIG). The little-endian code is generated by our transformation.

Our **transformation** uses the specification on `x` to reverse its layout upon entering `port`:

```
enum endian { BIG, LITTLE };
int x;
int tmp = endianness();
switch(tmp) {
case BIG: break;
case LITTLE: flip(&x); break;
}
printf("%d\n", ((char*)&x)[2]);
switch(tmp) {
case BIG: break;
case LITTLE: unflip(&x); break;
}
```

The `flip` and `unflip` functions reverse the bytes in their input and are **generated** by our translation.

dan grossman

We **implemented a tool** that takes code written using our extension and yields plain C.

We **applied** our tool to parts of the Gnu Debugger (GDB) and Gnu Binary File Description Library (BFD).

We manually **inspected 120 files** in the source base of 1700 files. In these 120 files, we found **3600 lines of code** that could potentially be eliminated by using our tool.

We picked 10 files from the source base and modified them to work with our tool. We found **33 occurrences** of code with multiple versions, one for each data layout.

Our tool eliminated **188 lines** (2,465 lexical tokens) from the source base in exchange for **11 layout specifications**.

The **performance overhead** of our translation is **small** and byte-level flips and unflips are entirely **optimized away** by Gcc -O3. E.g., `flip(x); y=x[3]; unflip(x); is` rewritten to `y=x[0];`.

Often, low-level C programs must handle **multiple** in-memory **layouts of data**, e.g. little- versus big-endian order of bits, bytes, and bit-fields. Typically, programmers write **multiple versions** of the same code, each version specialized to a particular layout:

```
if (endianness() == BIG)
    printf("%d", ((char*)&x)[2]);
else
    printf("%d", ((char*)&x)[1]);
```

Writing and **maintaining** these versions is **bug-prone**. The versions must be maintained in-sync, the layout relationships are **subtle and implicit**, and the code is inherently low-level and hard to understand.

a WASP project
wasp.cs.washington.edu