

Program Execution Explorer lab

Useful pointers

- [Debugging with GDB](#) (2015)
- [Arithmetic operations](#) in the GNU Emacs manual (2015)

Background

The name of this assignment comes from the idea that a debugger like GDB is better thought of as a way of exploring program execution histories than merely as a debugger.

Although the GNU Emacs text editor is not intended for high performance numeric computation, its scripting language [Elisp](#) is reasonably widely used and Elisp applications need adequate (if not stellar) performance for numeric applications. One such application, [GNU Calc](#), is a desk calculator that does a lot of arithmetic internally. Your goal is to see how much overhead is imposed by Emacs when doing standard arithmetic operations, in particular multiplication with integer arguments, and to think about how to reduce the arithmetic overhead.

Keep a log

Keep a log in the file `pexexlab.txt` of what you do in the lab so that you can reproduce the results later. This should not merely be a transcript of what you typed: it should be more like a lab notebook, in which you briefly note down what you did and what happened. It should record not just what worked, but also what *didn't* work.

Gather instruction traces

You can multiply numbers with Emacs from the shell by running a command like this:

```
emacs -batch -eval '(print (* 6997 -4398042316799 179))'
```

Gather a trace for the key part of the above test case. This trace should include every instruction in the `Ftimes` function, which is the C implementation of the Elisp `*` function. It should also include every instruction in every function that `Ftimes` calls, either directly or indirectly.

For the purpose of this assignment, a trace is an ASCII text file. Each line corresponds to a single machine instruction executed in turn during the trace. Lines use the following format:

```
0x543502<arith_driver+2>data.c:2577      push %r14      M8[0x7fffffff508]=0x7fffffff520  rsp=0x7fffffff508
```

Columns should be separated by single tab characters. The first column gives the machine address of the instruction, both in hexadecimal and (in angle brackets) as an offset from the current function; this address is followed by the [basename](#) of the source file and line number that is most responsible for the instruction; the example source line is the { at the start of the `arith_driver` procedure, since the instruction is part of that function's [prolog](#). The second column gives the machine instruction in the same format used by GDB's `x/i` command, using a space to separate the instruction from operands. The third column gives the effect of the instruction on memory and general-purpose registers, again using hexadecimal values. The example above stores `0x7fffffff520` into the 8-byte word at address `0x7fffffff508`; the "8" in "M8" stands for an 8-byte memory access. The example also sets the `rsp` register to `0x7fffffff508`. List memory modifications in increasing address order, and register modifications in alphabetical order. Traces need not record modifications to status registers such as `rflags`.

To gather information for your trace (which you should put into the file `trace.tr`), use the executable `~eggert/bin64/bin/emacs-24.5` on the SEASnet Red Hat 7 GNU/Linux servers. The corresponding source code can be found in `~eggert/src/emacs-24.5/` (particularly its `src` subdirectory), and the executable was compiled for the x86-64.

The above example trace line corresponds to line 2577 of `~eggert/src/emacs-24.5/src/data.c`.

Examine integer overflow handling

Compile the following function:

```
#include <limits.h>
#include <stdbool.h>
long big = LONG_MAX;
bool
testovf (void)
{
    return big + LONG_MAX < big;
}
```

for the x86-64 in three ways: (1) with `-O2`, (2) with `-O2 -fsanitize=undefined`, (3) with `-O2 -fwrapv`. Compare the resulting assembly-language files, and describe and justify the differences that you see. Put your description into a plain ASCII text file `testovf.txt`.

A few more questions

Answer the following questions, in a plain text file `answers.txt`:

1. Explain why the instructions in the trace did not produce the correct mathematical result. Which instructions caused the problem, exactly?
2. Explain why the shell command `emacs -batch -eval '(print most-positive-fixnum)'` outputs 2305843009213693951. Where did the number 2305843009213693951 come from? Explain in terms of the Emacs source code.
3. Explain why the shell command `emacs -batch -eval '(print (* most-positive-fixnum most-positive-fixnum))'` outputs only 1.
4. The Emacs executable was compiled with GCC's `-O2` option. Suppose it had also been compiled with `-fsanitize=undefined`. Explain any problems the trace would run into, or if there would not be a problem explain why not.
5. Similarly, discuss whether and how `-fwrapv` would have caused problems.
6. Suppose we assume `-fwrapv` is used. Suggest changes to how Emacs does integer multiplication that should help improve its performance. Focus on integer multiplication; don't alter the machinery Emacs uses to decide which flavor of multiplication to do.
7. How significant are the efficiency differences discussed above, in the context of Emacs execution?

Submit

Submit a compressed tarball `pexex.tgz` containing the files mentioned above, namely `pexexlab.txt`, `trace.tr`, `testovf.txt`, and `answers.txt`.

© 2015 [Paul Eggert](#). See [copying rules](#).

\$Id: pexexlab.html,v 1.7 2015/10/21 22:56:37 eggert Exp \$