ESnet
ENERGY SCIENCES NETWORK

Run, Zeek, RUN!

How FAST can Zeek RUN?

Jim Mellander
Cybersecurity Engineer
ESNet

ZeekWeek 2019
Seattle WA
October 11, 2019

Goals for this presentation

- The Quest for Efficiency started Long Ago
- Can Zeek run faster without code changes?
  - ***Yes!***
- Trying a different compiler
- Rolling your own library
- Benchmarks
- Suggestions

ESnet

Optimization is not a new idea

Programmer's
Reference Manual
October 15, 1956

THE FORTRAN
AUTOMATIC CODING SYSTEM
FOR THE
IBM 704 EDPM®

This manual supersedes all earlier information about the FORTRAN system. It describes the system which will be made available during late 1956, and is intended to permit planning and FORTRAN coding in advance of that time. An Introductory Programmer's Manual and an Operator's Manual will also be issued.

ESnet

Optimization is not a new idea

FREQUENCY

GENERAL FORM	EXAMPLES
"FREQUENCY n(i, j, . . .), m(k, l, . . .), . . ." where n, m, . . . are statement numbers and i, j, k, l, . . . are unsigned fixed point constants.	FREQUENCY 30(1, 2, 1), 40(11), 50(1, 7, 1, 1)

The FREQUENCY statement permits the programmer to give his estimate, for each branch-point of control, of the frequencies with which the several branches will actually be executed in the object program. This information is used to optimise the use of index registers in the object program.

A FREQUENCY statement may be placed anywhere in the source program, and may be used to give the frequency information about any number of branch-points. For each branch-point the information consists of the statement number of the statement causing the branch, followed by parenthesis enclosing the estimated frequencies separated by commas.

Consider the example. This might be a FREQUENCY statement in a program in which statement 30 is an IF, 40 is a DO, and 50 is a computed GO TO. The programmer estimates that the argument of the IF is as likely to be zero as non-zero, and when it is non-zero it is as likely to be negative as positive. The DO statement at 40 is presumably one for which at least one of the indexing parameters (m's) is not a constant but a variable, so that the number of times the loop must be executed to make a normal exit is not known in advance; the programmer here estimates that 11 is a good average for that number. The computed GO TO at 50 is estimated to transfer to its four branches with frequencies 1, 7, 1, 1.

All frequency estimates, except those about DOs, are *relative;* thus they can be multiplied by any constant. The example statement, for instance, could equally well be given as FREQUENCY 30(2,4,2), 40(11), 50(3,21,3,3). A

ESnet

Modern Code Optimization

- Compiler has to make number of decisions
  - Is “**then**” more probable than “**else**”?
  - Is a function worth inlining here?
  - Should this loop be unrolled?
- Questions get down to branch probability assessment
  - Usually estimated by a number of heuristics
    - Loop exit condition usually estimated false, for instance

ESnet

Several ways to optimize Code Branches

- Manual
  - Then: Fortran's FREQUENCY statement providing hints for basic blocks.
  - Now: GCC's __builtin_expect() function, used by likely() and unlikely() macros in the Linux kernel.
  - However: *"(...) programmers are notoriously bad at predicting how their programs actually perform."* - GCC Manual
- Automated
  - Measure frequency of branches (not)taken during real workload execution.
  - Use gathered statistics to provide compiler hints.

ESnet

Switch Statement

```
switch(tcp_flag) {
    case SYN:
        do_syn();
        break;
    case FIN:
        do_fin();
        break;
    case ACK:
        do_ack();
        break;
    default:
        do_something_else();
    }
```

```
if (tcp_flag == SYN)
    do_syn();
else if (tcp_flag == FIN)
    do_fin();
else if (tcp_flag == ACK)
    do_ack();
else
    do_something_else();
```

ESnet

Most common TCP flag seen in traffic?

- *"(...) programmers are notoriously bad at predicting how their programs actually perform."*
  - But it's a good bet that **ACK** is the most common flag seen in actual traffic.
- So, to optimize the tests manually, we would want something like:

```
if (tcp_flag != ACK) goto NOTACK;
/* Process ACK Flag */
MAINLINE:
/* Continue with mainline of program */
..
NOTACK:
/* Test for 2nd most common flag */
if (tcp_flag != FIN) goto NOTFIN;
/* Process FIN Flag */
goto MAINLINE;
NOTFIN:
etc.
```

ESnet

Automated Optimization aka Profile Guided Optimization

- Compile code with hooks to gather statistics on branches taken/not taken.
- Run code against representative sample input, which gathers statistics.
- Recompile code using gathered statistics to optimize branches.

ESnet

Who uses Profile Guided Optimization?

- Firefox
  - Page rendering time: 13% faster.
- Chrome
  - Startup time: 16.8% faster.
  - Page load time: 5.9% faster.
  - New tab page load time: 14.8% faster.
- Python
  - Up to 20% faster.
- PHP
  - 7% faster.
- Zeek?

ESnet

Cliff Notes: Profile Guided Optimization

- Compile code with **`--coverage`** in {C|CXX|LD}FLAGS
- Run the binary
- Run your application/benchmark against that binary
- Recompile code with **`-fprofile-use`** (above steps will place lots of files in source tree, one per source code file actually executed)
- Code runs faster!

ESnet

Lets Compile Zeek

- `./configure; make; make install`
  - Builds with O2 optimization
- `CFLAGS='-O3' CXXFLAGS='-O3' ./configure; make; make install`
  - Still builds with O2 optimization ☹
- `./configure --build-type=Release; make; make install`
  - Builds with O3 optimization
- Can we do better?

ESnet

Lets Compile Zeek with PGO

- `CFLAGS='–coverage' CXXFLAGS='–coverage' ./configure --build-type=Release; make install`
- Run zeek against sample input, statistics dropped in source tree
- In source tree: `tar cvf gc.tar `find . –name '*.gc*'``
- `make distclean; CFLAGS='-fprofile-use -fprofile-correction -flto' CXXFLAGS='-fprofile-use -fprofile-correction -flto' ./configure --build-type=Release`
- `tar xvf gc.tar` (restore profiling information into build tree)
- `make; make install`

ESnet

How did we do?

- Against 150 GB pcap, compiled with Centos 7.5 default compiler: gcc 4.8.5 (average of 5 runs)
  - Before: 2231 seconds
  - After: 1965 seconds ~12% increase
- Can we do better than that?

ESnet

Maybe a Different Compiler?

- *gcc*
  - 9.2 release, 10 in development
- *clang*
- *Intel Parallel Studio*
  - 30 day free trial
- *AMD Optimizing C Compiler*
  - Free from AMD, based on clang
- *Open64 Compiler*
  - Free from AMD, based on SGI compiler
- *Portland Group PGI C/C++ Compiler*
  - Community Edition Free, popular on supercomputers, based on clang

ESnet

gcc 9.2

- Had trouble with other compilers, but did install gcc 9.2
  - –PGO runtime down to 1782 seconds
    ~20% faster!
  - –Can we do better than that?

ESnet

Compile for native architecture

- Default compile for any x86 processor
- Add `–march=native` to C|CXXFLAGS
- Now how are we doing?
  - Runtime down to 1744 seconds ~22% faster!
  - Can we do even better than that?

ESnet

Where's the Library?

- malloc dynamic memory library heavily used by zeek
- Are there additional efficiency gains by using an alternate malloc implementation?

ESnet

mallocs tested

- Centos 7.5 built in malloc – based on ptmalloc
- tcmalloc – aka gperftools
  - `--enable-perftools`
- jemalloc
  - `--enable-jemalloc`
- lockless malloc http://locklessinc.com/downloads/
- liblite-malloc https://github.com/Begun/lockfree-malloc
- mimalloc https://github.com/microsoft/mimalloc
- supermalloc https://github.com/kuszmaul/SuperMalloc
  - Supports Haswell transactional memory
- OpenBSD malloc https://github.com/andrewg-felinemenace/Linux-OpenBSD-malloc
  - Uses crypto for added security….

ESnet

Malloc implementations, The Good, The Bad, and The Ugly

- The Good
  - jemalloc 1541
  - tcmalloc 1470
  - llalloc 1409
  - mimalloc 1517
- The Bad
  - Standard malloc 1744
  - supermalloc 1885
  - liblite malloc 1767
- The Ugly
  - OpenBSD malloc 2852

ESnet

But wait, there's more

- For some reason, compiling Zeek with −march=native reduced performance in some cases
- The Good
  - jemalloc 1584
  - tcmalloc 1408
  - llalloc 1305
  - mimalloc 1373
- The Bad
  - Standard malloc 1782
  - supermalloc 1747
  - liblite malloc 1627
- The Ugly
  - OpenBSD malloc 2637

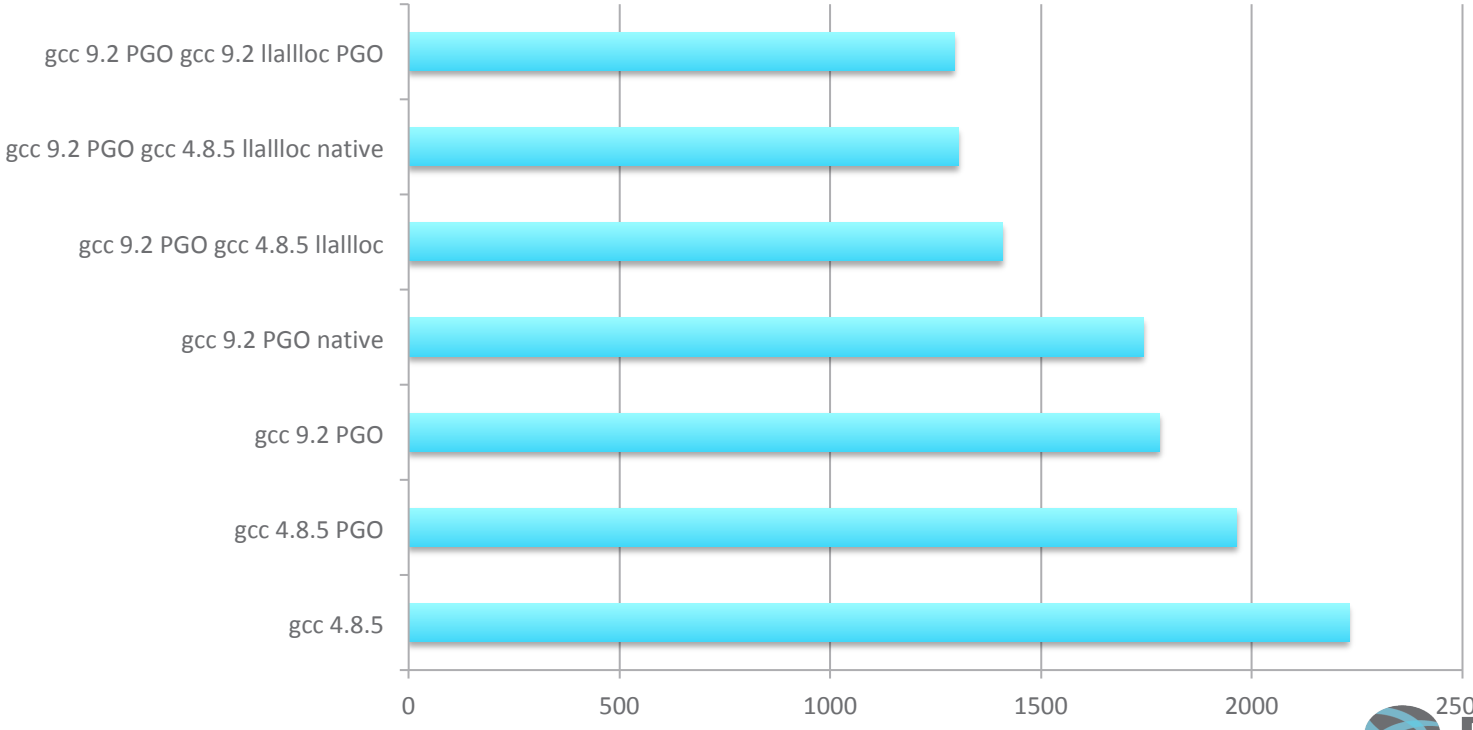
ESnet

What, even more?

- We can compile the malloc library with a more modern compiler (gcc 9.2) & use PGO, so that it is optimized for our use case.
- The Good:
  - jemalloc 1485
  - tcmalloc 1408
  - llalloc 1294 – THE WINNER!!!!! 42% speed increase over original compile
  - mimalloc 1305
- The Bad
  - Standard malloc 1782 (no recompile)
  - supermalloc 1622
  - liblite malloc 1566
- The Ugly
  - OpenBSD malloc 2445

ESnet

Chart


gcc 9.2 PGO gcc 9.2 llallloc PGO
gcc 9.2 PGO gcc 4.8.5 llallloc native
gcc 9.2 PGO gcc 4.8.5 llallloc
gcc 9.2 PGO native
gcc 9.2 PGO
gcc 4.8.5 PGO
gcc 4.8.5
0
500
1000
1500
2000
2500

ESnet

Next steps

- Other libraries may also benefit from Profile Guided Optimization
- Any other ideas?

ESnet

Recommendations

- Your mileage may vary, but….
  - Try Profile Guided Optimization against your traffic, both pcaps and network.
    - Also run against pcaps in Zeek distro to exercise little used code paths.
  - Check out alternatives to Standard Libraries.
  - Have fun!

THANK YOU!

Jim Mellander – jmellander@lbl.gov

ESnet