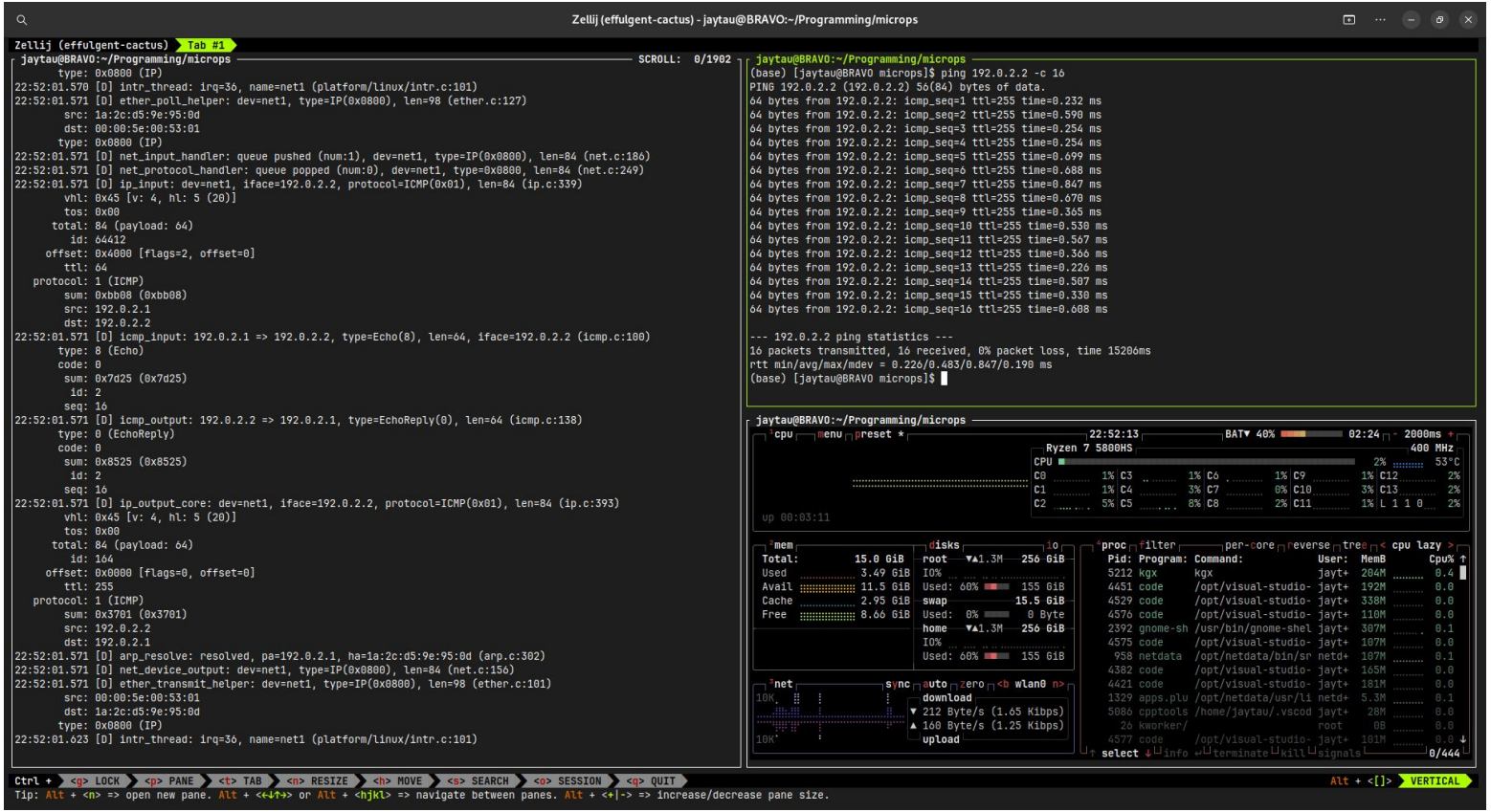
Milestone 1

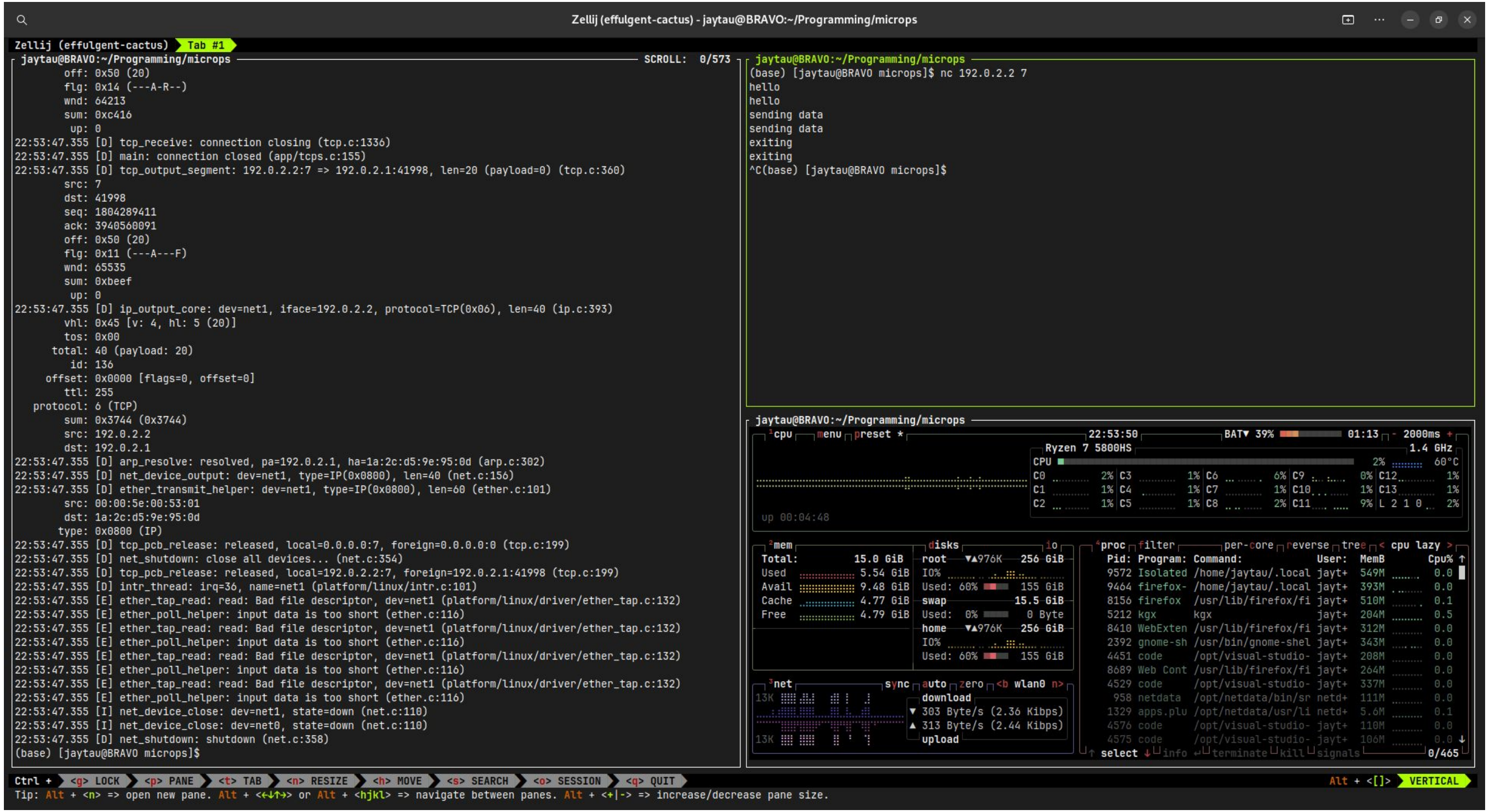
Ashman Mehra
Joel Tony
Arnav Gupta
Kumarakrishna Valeti

Project Description

We plan to port microps (a small C based TCP/IP protocol stack for learning) incrementally and study the challenges and performance impact of integrating Rust into existing C code. We also plan to study the possible memory safety guarantees on integration.

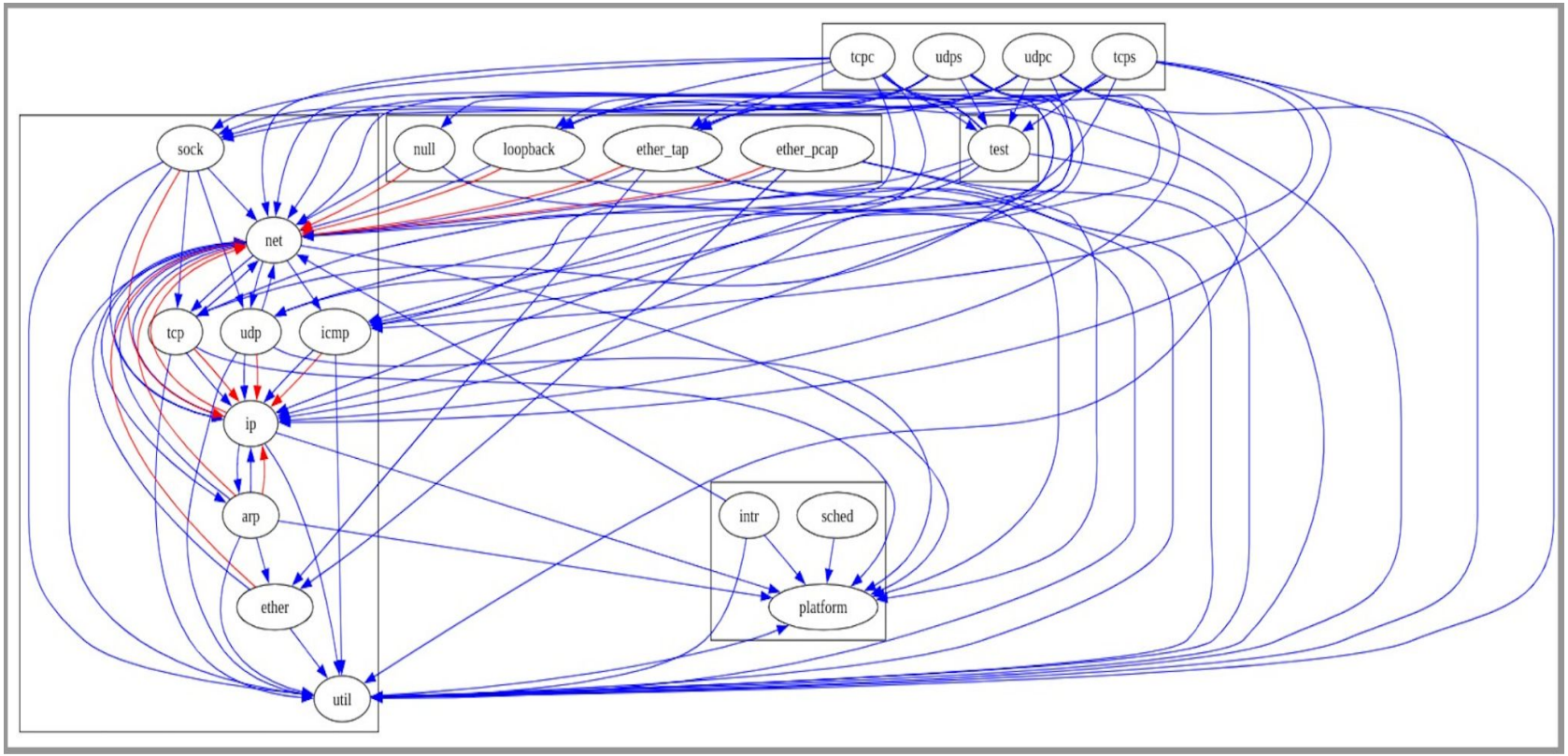


Microps Demonstration - ICMP Response



Microps Demonstration - TCP/IP Connection

Dependency Graph



First Port - Utils.c

Utils.c (approx 200 lines) is a collection of utility functions for network-related operations and data handling.

lprintf: This function is for logging and printing messages with a timestamp, log level, and source file information.

hexdump: This function is for printing hexadecimal and ASCII representations of binary data in a human-readable format

queue_init, queue_push, queue_pop, queue_peek, and queue_foreach: These functions implement a simple queue data structure for managing a list of elements.

endian, byteorder, byteswap16, byteswap32, hton16, ntoh16, hton32, and ntoh32: These functions handle byte order conversion for big-endian and little-endian systems

cksum16: This function calculates a 16-bit checksum for a given data block. It's commonly used in networking to verify data integrity.

lib.rs

We set up the Cargo library by creating a lib.rs file as the entry point for the Rust code.

One of the notable changes is the implementation of the queue data structure using a struct in rust with associated functions being implemented using ‘impl’.

We encountered challenges in finding Rust equivalents for the C libraries and functions used in the original code.

We also faced several errors with type compatibility.

Foreign Function Interface

FFI (Foreign Function Interface) is a way for functions written in one language to be called from another language. It allows Rust programs to interact with other programs and libraries, such as those written in C or C++.

Next chapter

```rust
gle]
pub extern "C" fn hello_from_rust() {
    println!("Hello from Rust!");
}
```

```c
extern void hello_from_rust();

int main(void) {
    hello_from_rust();
    return 0;
}
```

```
[lib]
crate-type = ["cdylib"]
```

```
gcc call_rust.c -o call_rust -lrust_from_c -L./target/debug
```

Integration - Procedure

We have divided the task into 2 parts:

1. Creating a shared library using FFI standards
    a. Creating a library module in rust
    b. Declaring the shared functions using pub extern “C” and exporting them as modules
    c. Building a cargo.toml file for dependency resolution

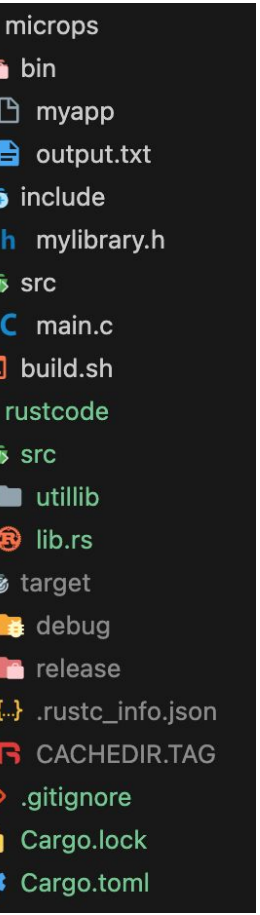
2. Importing the rust library in microps using header files
    a. Creating a build file for microps and including rust shared library in compilation
    b. Generating a header file to collect method definitions from rust
    c. Utilizing the rust functions in C for testing

Integration - Live code demonstration!

We will now demonstrate a working code for 2 functions from the util.c file:

1. dumphex - Memory Intensive
2. Std_err - I/O Intensive

```
~/Desktop/College/Projects/JAAK Project $ cd rustcode
Desktop/College/Projects/JAAK Project/rustcode $ cargo build --release
    Finished release [optimized] target(s) in 0.09s
Desktop/College/Projects/JAAK Project/rustcode $ cd ../microps
Desktop/College/Projects/JAAK Project/microps $ ./build.sh
Desktop/College/Projects/JAAK Project/microps $ ./bin/myapp
This is a message from C written by rust.
Write to stderr successful.
hexdump completed successfully.
```



Integration

Integrating with a C/C++ codebase has provided the following challenges. These concepts do not inherently exist in C/C++ which demands extra caution and vigilance

- Ownership model of Rust to leverage the borrow-checker to the full extent. Changing the structure of the code was often necessary for us to keep the ownership rules intact
- Lifetimes. Adding lifetimes required us going through the codebase thoroughly to make sure we didn't end the scope of a variable before it was supposed to
- Traits. Using traits provided a lot of freedom in terms of implementation of data types, which is significantly more robust and type-safe than in C/C++.

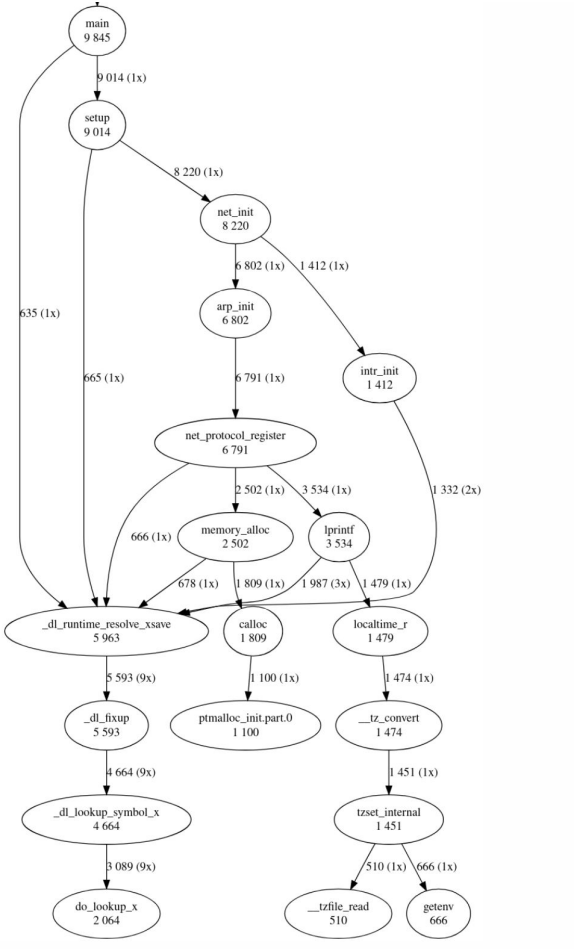
Preliminary Benchmarking Tools

Callgrind - Captures function call hierarchy, cache misses, and CPU cycles to profile program execution.

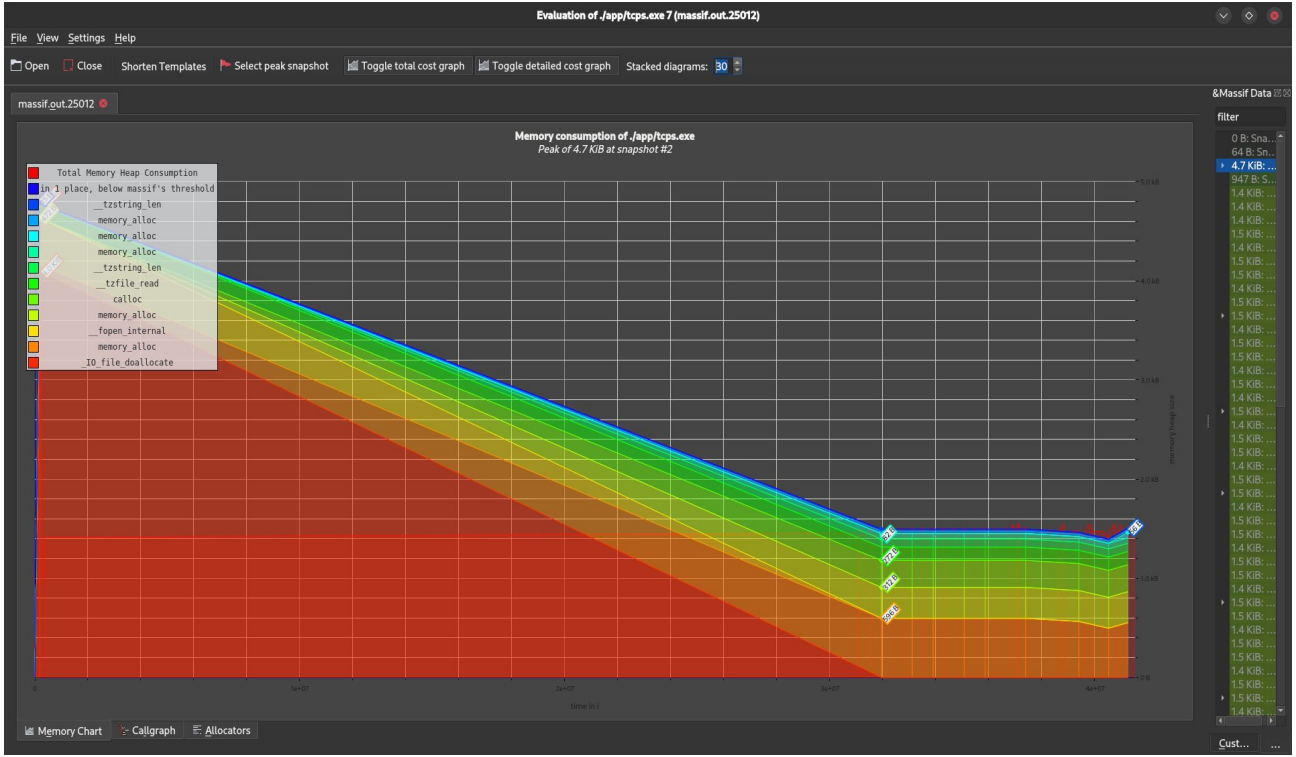
Massif is a heap profiler. It performs detailed heap profiling by taking regular snapshots of a program's heap. It produces a graph showing heap usage over time, including information about which parts of the program are responsible for the most memory allocations.

DHAT is a tool for examining how programs use their heap allocations. It tracks the allocated blocks, and inspects every memory access to find which block, if any, it is to.

Callgrind


main
9 845
9 014 (1x)
setup
9 014
8 220 (1x)
net_init
8 220
6 802 (1x)
1 412 (1x)
arp_init
6 802
635 (1x)
665 (1x)
6 791 (1x)
intr_init
1 412
net_protocol_register
6 791
2 502 (1x)
3 534 (1x)
1 332 (2x)
666 (1x)
memory_alloc
2 502
lprintf
3 534
678 (1x)
1 809 (1x)
1 987 (3x)
1 479 (1x)
_dl_runtime_resolve_xsave
5 963
calloc
1 809
localtime_r
1 479
5 593 (9x)
1 100 (1x)
1 474 (1x)
_dl_fixup
5 593
ptmalloc_init.part.0
1 100
__tz_convert
1 474
4 664 (9x)
1 451 (1x)
_dl_lookup_symbol_x
4 664
tzset_internal
1 451
3 089 (9x)
510 (1x)
666 (1x)
do_lookup_x
2 064
__tzfile_read
510
getenv
666

Massif


Evaluation of ./app/tcps.exe 7 (massif.out.25012)
File View Settings Help
Open Close Shorten Templates Select peak snapshot Toggle total cost graph Toggle detailed cost graph Stacked diagrams: 30
massif.out.25012
Memory consumption of ./app/tcps.exe
Peak of 4.7 KiB at snapshot #2
Total Memory Heap Consumption
In 1 place, below massif's threshold
__tzstring_len
memory_alloc
memory_alloc
memory_alloc
__tzstring_len
__tzfile_read
calloc
memory_alloc
__fopen_internal
memory_alloc
_IO_file_doallocate
&Massif Data
filter
Memory Chart
Callgraph
Allocators
Cust...

DHAT

```
Invocation {
  Mode:    heap
  Command: ./app/tcps.exe 7
  PID:     22341
}

Times {
  t-gmax: 1,98,530 instrs (1.37% of program duration)
  t-end:  1,45,16,490 instrs
}

▼ PP 1/1 (9 children) {
   Total:     9,063 bytes (100%, 624.32/Minstr) in 93 blocks (100%, 6.41/Minstr), avg size 97.45 bytes, avg lifetime 40,59,478.35 instrs (27.96% of program duration)
   At t-gmax: 4,781 bytes (100%) in 5 blocks (100%), avg size 956.2 bytes
   At t-end:  1,160 bytes (100%) in 19 blocks (100%), avg size 61.05 bytes
   Reads:     30,23,958 bytes (100%, 2,08,311.93/Minstr), 333.66/byte
   Writes:    13,057 bytes (100%, 899.46/Minstr), 1.44/byte
   Allocated at {
     #0: [root]
   }
  }
  ├── PP 1.1/9 {
  │    Total:     4,096 bytes (45.19%, 282.16/Minstr) in 1 blocks (1.08%, 0.07/Minstr), avg size 4,096 bytes, avg lifetime 3,414 instrs (0.02% of program duration)
  │    Max:       4,096 bytes in 1 blocks, avg size 4,096 bytes
  │    At t-gmax: 4,096 bytes (85.67%) in 1 blocks (20%), avg size 4,096 bytes
  │    At t-end:  0 bytes (0%) in 0 blocks (0%), avg size 0 bytes
  │    Reads:     214 bytes (0.01%, 14.74/Minstr), 0.05/byte
  │    Writes:    454 bytes (3.48%, 31.27/Minstr), 0.11/byte
  │    Allocated at {
  │      #1: 0x48FE499: _IO_file_doallocate (filedoalloc.c:101)
  │      #2: 0x490D278: _IO_doallocbuf (genops.c:347)
  │      #3: 0x490D278: _IO_doallocbuf (genops.c:342)
  │      #4: 0x490C1BC: _IO_file_xsgetn (fileops.c:1288)
  │      #5: 0x4909DB3: fread_unlocked (iofread_u.c:40)
  │      #6: 0x4952C53: __tzfile_read (tzfile.c:187)
  │      #7: 0x49526B4: tzset_internal (tzset.c:405)
  │      #8: 0x49528B6: __tz_convert (tzset.c:577)
  │      #9: 0x10AE35: lprintf (util.c:27)
  │      #10: 0x10BFCD: net_protocol_register (net.c:218)
  │      #11: 0x10DF90: arp_init (arp.c:331)
  │      #12: 0x10C613: net_init (net.c:374)
  │    }
  │  }
```

Questions?