Dennis Chan & Shuyi Qi // Data Structure Visualizer

Section: Project Overview

System Design: We developed a visualization tool for dynamically-linked data structures in C programs. Our system can be broken down into three distinct parts. The tracer, the user, and the “vhelpers” header file. The former acts as a wrapper that encapsulates and invokes most of the operations the visualizer requires. The user includes the “vhelpers” header file and a call to start_tracing on the root of this structure that they want traced. The tracer program instruments comprehensive memory inspection on the user program at a regular, predefined but customizable interval that shows the user the progression of memory use during the user program’s runtime in a d3-based visualizer on firefox upon a successful instrumentation of the user program.

High level points of implementation: The tracer holds the data structure and mechanism that keeps track of the user. Once the tracer invokes the user program, it uses ptrace to single step through the user program and moderates and instruments reads at a regular interval on the user program to collect the data necessary for our visualization. The user program requires the inclusion of the vhelper header file that implements a start_tracing method that report the size information for each pointer the tracer program examines. It then follows naturally that the user program must call start_tracing with the head_addr to order to set up the hook. The tracer communicates with the user through a named pipe to get head address and pointer sizes from the user because the malloced size of a pointer is only available in its own process. The tracer instruments the user’s memory at a regular interval and produce in a log file a complete mapping of all pointer-like values that reside in nodes reachable from head until the user program exits. Each visit to a pointer-like address adds a line to the log file, which a converter takes as input and produces output files that the visualizer takes as input.

Evaluation Strategy and Results: We based our evaluation on three areas: correctness, ease of usage and performance penalties. Ease of Use: our program excels in this department, requiring only two lines of addition on the instrumented program, and running such with our wrapper function, who manages everything from the invocation of the user program to the opening firefox to render the data at the end. Correctness: our visualizer successfully reproduces a correct visualization for all four of our test programs, which create linked-lists and binary trees. Performance: One of the test programs that inserts 1000 random nodes into a binary search tree took an excruciating long amount of time to render, while the rest three programs return modest performance results. Needless to say, our implementation remains a prototype and has a large margin for improvement.

Section: Design & Implementation

Structure of System: The visualizer project can be broken down into three primary distinct, sequential phases. The first phase is the data collect phase, where the program performs memory inspect over a predefined interval the memory reachable from the user-provided root through pointer chasing. The memory inspection process outputs a log file for the second phase of the visualizer, where a python script converts the said output file into a format that the d3-based visualization takes. The last part is create--where firefox opens an HTML page that loads the visualization javascript and renders the data file into a meaningful and visualized memory mapping from root over snapshots the tracer program takes, over the regular intervals. The tracer program alone oversees all three phases of our visualization. Our program relies on the linux pthread and ptrace library for intel x64-86 systems.

Setup: During the setup phase, the tracer is the only program the user (the person) directly invokes. The tracer performs an execvp call on the user defined program that starts the said program. Immediately, tracer performs a pthread_create and starts a new *get_head_thread*, which opens a pipe with name predefined in the “vhelpers” header file and waits for the user program’s head address to be written to the other end of the pipe. Meanwhile, tracer intervenes and initiates a ptrace single step call which will step through the entirety of the user program’s execution. Despite the heavy intervention of single-stepping, ptrace only performs a memory inspection (described in the next section) per the number of instructions executed specified in the tracer program. The user program runs in a manner similar to its normal execution, except a successful trace requires the user program to report the root of the memory it wishes to be traced by a call to start_tracing (head_addr). Start_tracing() writes the head address of the tracing memory, on a newly created thread, into the pipe that the tracer “listens” on. This thread will stay active throughout the execution of the user program. As soon as tracer receives the head address over the pipe, *get_head_thread* copies the information to a global variable and exits.

Collect--Memory Inspect: When the tracer detects that head address has been received from the user, it triggers memory inspect and recursively inspects each reachable node from root in a depth first manner. Memory inspect creates a new pthread (*get_size_thread*), which writes the pointer address being examined into the pipe, and the user program continues to listen on its thread to report back malloc_usable_size when it receives a pointer. *Get_size_thread* exits after it receives the size of the pointer it just queried and writes the pointer size in a global mapping (unordered_map<void*, size_t>) that maps pointer addresses to their malloced sizes. Inspect memory then examines, eight bytes at a time using PEEKDATA, the content in the child’s memory until it reaches the end of that specific pointer’s allocation. PEEKDATA returns -1 when the address it peeks on is not a pointer and a valid long value otherwise. We took a conservative approach on memory inspection: For any pointer-like value discovered by this process, inspect memory logs the parent and the child of the pointer into a log file and recursively (and therefore depth-first) inspect memory of the newly discovered pointer-like value until no new pointers are found, thus concluding the memory inspection for a time slice. This occurs repeatedly until the end of the user program’s execution.

Convert: When the user program exits, the tracer program calls execvp on a python script that converts the log output into the input format required by the d3’s csv loading function.

Create: When the script exists, tracer invokes firefox that renders the files the script outputs.

Section: Evaluation

Evaluation Strategy: We based our evaluation on three areas: correctness, ease of usage and performance penalties. Because our program targets a wide spectrum of users, including those of little experience with c, the ease of user (i.e. how to invoke the program, what setup options are required for successful instrumentation, etc) remains a primary concern. We qualitatively evaluate this aspect by looking at how many lines of additions the user makes to their program for it to be visualizer compatible and terminal instructions to execute the visualizer/their program. The correctness concerns with whether the visualization tool faithfully reproduces and captures during program runtime all reachable nodes from the root and does not miss any nodes. We measure this aspect by checking the consistency between the logical placement of these relational memory objects and visually inspecting the placement in the visualization. Lastly, performance penalties measure the overhead sample applications with and without the use of our visualization tool. We developed programs that fall under four primary areas as part of our test suite: programs that generate 1) 10-100 linear nodes on a linked-list like data structure, 2) sub-100 nodes on a balanced binary search tree, 3) sub-100 nodes on a unbalanced search tree, 4) 1,000 random node insertions into a binary search tree. We ran all these programs on two mathlan machines in Noyce 3813. The two computers share identical hardware: Intel i5-6600 CPU @ 3.3Ghz, which is a quad-core CPU with a six megabyte cache size, with 8 gigabytes of RAM. The tracer program inspected user's memory every 100 steps. We time these programs over at least three trials using the *time* command on terminal for measurements on the collect and convert phase. The create phase takes a negligible amount of time and cannot be measured on terminal's time because the visualization renders dynamically.

Results: Ease of Use: our program excels in this department, requiring only two lines of addition on the instrumented program, and running such with our wrapper function, who manages everything from the invocation of the user program to the opening firefox to render the data at the end. Correctness: our visualizer successfully reproduces a correct visualization for the first three types of sample programs that we developed. Performance: The fourth program took an excruciating long amount of time to render, when the collection and conversion time alone exceeded 15 minutes (16 minutes 43 seconds for collect & convert, and the create phase never fully completed in the remainder of the time up to 30 minutes), a 300,000 times slow down alone in the first two phases compared to an uninstrumented run. Other runs took a much more modest penalty of 1,000-10,000 times slow down for the first three types of test programs. Needless to say, our implementation remains a prototype and has a large margin for improvement.