DrStackVis

Toshi Piazza & Austin Chin

May 6, 2016

1 DrStackVis

The runtime stack is generally visualized on paper; the use of *Box and Pointer* Diagrams are generally the method by which stacks are taught in *CSCI1200* Data Structures. However, an interactive demonstration of the runtime stack is in wont; current attempts are flimsy and generally require introducing a large volume of source code into the target project. Also, it is impossible to determine which values are garbage and which are pointer values; this is only possible by running this code through a system like DrMemory or Valgrind.

1.1 Audience

Data Structures students who go to lab regularly come across the so-called *stack lab* which makes an attempt at this flimsy demonstration of the runtime stack in order to gain deeper understanding of the low level concepts of call-by-reference and call-by-value. Data Structures students who use this particular software will be able to answer all of the same questions, but at the same time will be able to perform queries against the stack which would not be possible using the old method.

1.2 Research Questions

Data Structures students use this stack lab to gain a deeper understanding of C++ calling conventions and argument passing. On this note, for any arbitrary binary we should be able to gleam at least the following:

1. Given an arbitrary binary, can we tell the difference between an argument that has been passed by reference vs. passed by copy?

2. Given a binary that copies the contents of an array `a` over to `b`, can the user determine which of the following cases is occurring?
    (a) The length of array `a` is greater than the length of `b`.
    (b) The length of array `a` is equal to the length of `b`.
    (c) The length of array `a` is less than the length of `b`.

2 Related Work

The data structures stack lab [3] is the primary inspiration for this project; again the idea is to be able to replace the stack lab by distributing both our GUI and some sample json files and associated code. This will make the lab a lot easier to understand. The stack lab and our visualization also greatly build upon existing box and pointer diagrams which are written by hand generally, also referenced in the stack lab [3]. This is the canonical reference to which our visualization subscribes.

Our aim is generally to bring Interactive Visualization to the concept of the stack, and in so doing foster an understanding in the Data Structures students; although the current stack lab demonstrates hands on behavior, as we can allow students to directly modify and make use of the sample code, the students are unable to directly view the stack values with as high a fidelity as with our tool, and so an interactive tool is clearly an improvement to learning [4].

The data mining algorithm is implemented as a DynamoRIO plugin; as a result we are able to perform meaningful computation while a program is running while interfacing with DynamoRIOs clean plugin and callback system [1]. This way, we are able to track all the writes to an applications runtime stack. DynamoRIO ships with a *memtrace* example from which this plugin takes heavy inspiration. Specifically, we lift the general algorithm from this sample, which tracks writes *and reads* from a running program. As with most DynamoRIO plugins, we *fire and forget* into a file that we then postprocess, for ease.

Meanwhile, the general algorithm involves preinserting code that mines the relevant information we would like to display, as well as the code which outputs this information to a file. We work around some differences between our algorithm and the *memtrace* algorithm in the next section.

We briefly considered an *in situ* approach; our current focus is on speed, and as such we generally use the *fire and forget* approach. In other words, our current data mining algorithm does as little computation as possible, so that the program runs unhindered and as fast as possible. However, if we consider the *in situ* approach outlined in *An Image-based Approach to Extreme Scale In Situ Visualization and Analysis* [2], we would instead spawn a GUI at the same time, which DynamoRIO in fact supports, and which would also require all the computation would be done online. This would hurt running time, but it would also remove the need for a postprocessing stage entirely, and also removing the required disk space necessary to save our json, required to run our GUI.

3 Core Contributions

3.1 Technical Implementation of DynamoRIO Plugin

Although a good portion of the algorithm which we employ was taken from the canonical *memtrace* example shipped with DynamoRIO (licensed under the OpenBSD 3-clause license), we also construct a novel approach to reading the values of addresses on the stack; whereas the *memtrace* example inserts instrumentation before the instruction, we must insert some instrumentation after the current instruction to get the value of memory written as well! Using this approach, however, we must also account for call instructions, which return after a while and so break our assumption of linear control flow. To counter this, we preinsert on call instructions and simulate the value written by them (these write values are predictable!).

We also outline a novel technique of reading the state of output of a program; we tell DynamoRIO to notify us on syscalls, and on write syscalls in particular we base64 encode the output (so that we can insert it into json just fine), and output this to a file.

Finally, our plugin also allows for Stack Annotations, a la the stack lab. We expose the following api: `SET_BREAKPOINT()`, `CLEAR_ANNOTATIONS()`, and finally `ADD_ANNOTATION(addr, label)`. As long as the source code can be modified to use this api, we can improve the readability and usability of the stack diagram in this way. These functions mimic the current stack lab: we can easily snap to a point in time specified by the impromptu breakpoint, and we can view annotations based on user supplied labels.

3.2 Technical Implementation of GUI

The GUI frontend for these json data files was written using processing, and first required a creation of a data structure for the stack that both members worked on. Data is read from the JSON file and updates a Stack class data structure. From this data structure, a list of Strings is generated that is then used to update the current ticks visual output.

The Stack class consists of the current tick, and the ceiling and base of the stack. Additionally, it contains two Maps of stderr and stdout, which are used to store output and their relation to ticks in the chart. The other two major components of the Stack class is an internal list of an Addr class containing the value of the addresses and whether or not they have been used; and a list of Mem objects. A Mem object contains all of the JSON information: sptr, addr, wmem, type and size.

Upon stack creation, the call to create a string of memory addresses with their values is used by first calling the addresses and values of the stack into an Addr array via getStack(), which is then converted to the list of Strings with convertByteStack2Stack(). One index of the String array has the following syntax: `"[" + address + "]" + " " + value`

Once the data structure is created, the GUI is implemented using a variety of Processing elements. PShape objects (in particular, PShape rectangles), datatypes that are used to store created Processing shapes, are utilized for all text box and stack elements. Processings imported functions: update(), draw() and mousePressed(), among others, are used to provide more complex interaction. The drawing updates itself every frame to account for the position of the mouse - in particular this comes into play when checking to see if buttons are highlighted, which allows for simple procedures such as buttons to be pressed and their color to change. Text is created using a .vlw file to store typography options, and creating text with Processings built in text() function. Unfortunately, these functions do not include a scroll bar for overflowing text boxes.

Overall, the GUI is dependent on the current tick of the Stack data structure, updating the Mem objects, their statuses and the status of the stdout and stderr. Each time the user moves forward in time, the canvas is cleared and the simulation is redrawn in the correct tick.

4 Evolution of Visualization Design

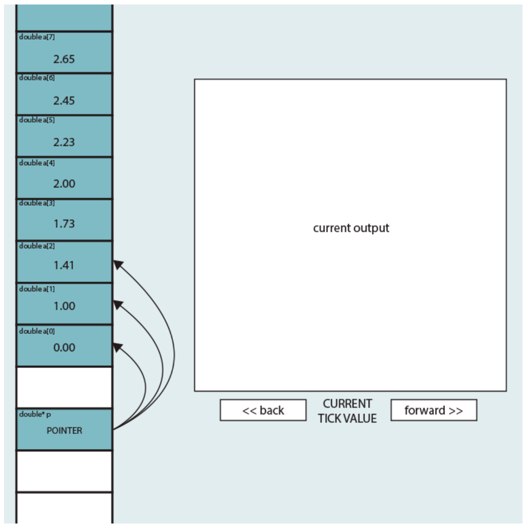
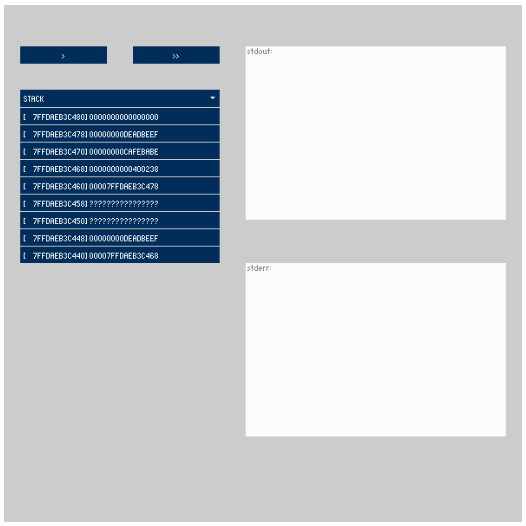


Figure 1: A very rough draft of our initial design. Note the key features, being the stack diagram, reminiscient of a box and pointer diagram of old, as well as the output box specifier.

The initial design of the visualization was created to resemble Box and Pointer diagrams that are typically used in Data Structures curriculum to teach students about the stack. This diagram is meant to be a scrollable list, so that you can examine each of the items on the stack and how they relate. Also on the canvas would be a standard output box that showed what was currently printed out at that tick value. The user could move forward and back a tick value with corresponding buttons.

Our visualization design evolved as we decided on where to build the visualization. Initially we were looking at a terminal based visualization using Go; ultimately we decided on creating the visualization in Processing, which allowed for a much more customizable and flexible user interface for our purposes. We also opted to add a JFileChooser at the beginning of the software so that the user would not have to input a path to the json file, instead being able to navigate their documents in an already intuitive form.

Our in-class user study featured a paper example that has been iterated upon from the initial sketch. Instead of outputting the entire stack, the stack outputs only the relevant information at that tick value. Additionally, a stderr box was also added. This design is the route that we continued to emulate, resulting in our current version.



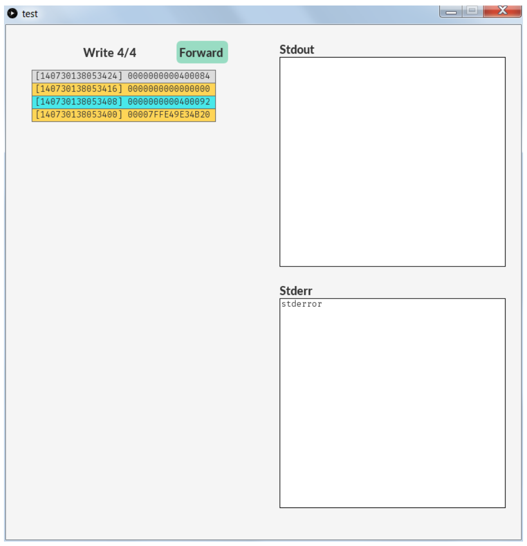
```
[  7FFDAEB3C480] 0000000000000000
[  7FFDAEB3C478] 00000000DEADBEEF
[  7FFDAEB3C470] 00000000CAFEBABE
[  7FFDAEB3C468] 0000000000400238
[  7FFDAEB3C460] 00007FFDAEB3C478
[  7FFDAEB3C458] ????????????????
[  7FFDAEB3C450] ????????????????
[  7FFDAEB3C448] 00000000DEADBEEF
[  7FFDAEB3C440] 00007FFDAEB3C468
```

Figure 2: A rough draft used for visual debugging purposes by Toshi. Note the horrible color choice, as well as the non-functioning output panes and awful font.

As we continued to work on the project, we experimented with the visualization of the stack, such as using plugin controlp5 to create a list of buttons for each memory address. Ultimately the available Processing plugins do not have the options that we need and we decided that it would be a better option to build these modules by scratch.

It became evident that due to time constraints, we would be unable to implement a back button - we have pushed the back button implementation to future iterations. We are still working on scrollbar implementation as well.

Visual-wise, we have elected to use monospaced type for the stack so that the user is better able to distinguish and identify the components of the stack. Color coding is based on the type of action made to the stack: push, calls, and moves are the most common types, and are specifically colored.



```
[140730138053424] 0000000000400084
[140730138053416] 0000000000000000
[140730138053408] 0000000000400092
[140730138053400] 00007FFE49E34B20
```

```
stderror
```

Figure 3: Our final design, as of the point of submission. Although not fully fleshed out feature- wise, clearly we can see the main features which we set out to achieve.

5 User Study Feedback

As previously stated, we used a paper demo for the in-class user study, and explained to users the purpose of the visualization,as well as the properties of the GUI.

The two questions that we had were

1. there is not a standardardized way to illustrate the stack: sometimes it is illustrated growing up, and other times it is illustrated growing down. Due to this, which way seems more intuitive to you?

2. confirmation that color choice is appropriate, legible and makes sense.

The majority of our users stated that they found that the stack growing down would be more intuitive. We did have significant support for enabling the user to decide this with a toggle. Most users found the colors to be discernible, although some did state that they may be a bit too bright. Following the study, we elected to first attempt to illustrate the stack growing down, and if time permitted would work on enabling a toggle system. Evidently we did not have time to implement a toggle, but the stack does grow down.

Looking ahead, we see this project as one that could be helpful for students to utilize and a source that can eventually be incorporated into classroom curriculum.

6 Future Considerations

Most of the future considerations hinge on usability concerns; stack annotations are supported by the backend, but not by the frontend. Care will be taken in the future to correctly display this Information. Also, in order for a user to know exactly where he/she is within the program, we should display a small box that visualizes the stack usage at the current time, and his/her place in the application. It also will snap-to user-defined breakpoints.

Also, for the future a better implementation of the DynamoRIO plugin is considered: we can easily optimize the plugin to output very infrequently. This allows us to guarantee that the native application will run at near-native speed. Also, to circumvent this hacky algorithm which preinserts instrumentation on call instructions and postinserts on everything else, we may decide to just expand call instructions into a push instruction, then a jmp. This could potentially fix our immediate problem, but could introduce a slew of other problems.

7 Work Breakdown

Generally, Toshi did all of the work concerning the DynamoRIO plugin, because of its niche usage and also because this is his area of study. Austin did all of the work implementing the Processing frontend, with the exception of a barebones library for simulating the stack, which is also generally a niche subject. All of the frontend was written without the help of a third party library, which is why Austin ended up doing a lot more work than was originally thought (it aint easy implementing scrollbars).

References

[1] Bruening, Derek L. Efficient, Transparent, and Comprehensive Runtime Code Manipulation. Thesis. MASSACHUSETTS INSTITUTE OF

TECHNOLOGY, 2004. Massachusetts: Massachusetts Institute of Technology, 2004. Efficient, Transparent, and Comprehensive Runtime Code Manipulation. Massachusetts Institute of Technology. Web. 3 May 2016. ¡http://www.burningcutlery.com/derek/docs/phd.pdf¿.

[2] Ahrens, James, John Patchett, Sebastien Jourdain, David H. Rogers, Patrick O'Leary, and Mark Petersen. *An Image-based Approach to Extreme Scale In Situ Visualization and Analysis. Tech. IEEE, n.d. Web. 3 May 2016.*

[3] Cutler, Barbara. "Pointers, Arrays, and the Stack." (n.d.): n. pag. *CSCI1200 Data Structures*. Web. 5 May 2016.

[4] Naps, Thomas L., Guido Robling, Vicki Almstrum, Wanda Dann, Rudolf Fleischer, Chris Hundhausen, Ari Korhonen, Lauri Malmi, Myles McNally, Susan Rodger, and ngel Velzquez-Iturbide. *Exploring the Role of Visualization and Engagement in Computer Science Education.* Diss. N.d. N.p.: n.p., n.d. Print.