Understanding the Linux Kernel, Visually

(Extended Abstract)

Hanzhi Liu
Nanjing University
Nanjing, China
hanzhiliu@smail.nju.edu.cn

Yanyan Jiang
Nanjing University
Nanjing, China
jyy@nju.edu.cn

Chang Xu
Nanjing University
Nanjing, China
changxu@nju.edu.cn

Abstract

Understanding the Linux kernel is challenging due to its large and complex program state. While existing kernel debugging tools provide full access to kernel states at arbitrary levels of detail, developers often spend a significant amount of time sifting through redundant information to find what is truly useful. This paper presents Visualinux, the first debugging framework that can simplify the program state of the Linux kernel to a level that can be visually understood with low programming complexity and effort. Evaluation results show that Visualinux can visualize various complex kernel components and efficiently assist developers in diagnosing sophisticated kernel bugs.

Keywords: Linux Kernel, Software Visualization, Debugging

1 Motivation

For years, developers have used GDB [5, 7, 11] and log/trace tools [8–10] to debug the Linux kernel. These tools provide mechanisms to examine kernel states at various levels of detail. However, considering that the human brain can only handle a limited amount of information at a time, developers are often overwhelmed by the abundance of information. Unfortunately, existing tools require significant human or programming effort to extract a specific subset of the information for a particular debugging objective.

As a typical example, the maple tree [3] introduced in the Linux kernel 6.1 is a scalable data structure for maintaining a set of ordered intervals, which replaces the two-decade-old red-black tree-based implementation of memory-mapped regions. We struggled to understand this new data structure: a textual interface naturally falls short of displaying high-dimensional information (e.g., an interconnected, complex data structure) in a readable way. We wrote scripts to unwrap the `union` type of nodes and parse the compressed pointers. Nevertheless, comprehending the list of tree nodes with indirect pointers remained mentally challenging.

The obstacle to understand the maple tree is rooted in the complexity of the Linux kernel: there are millions of live objects at runtime, with complex relations among them. Simply "printing" these objects as text produces an overwhelming amount of information for debugging purposes. Such complexity leads to the following question:

> *What kind of mechanism can help developers effectively and efficiently customize a view of the kernel object graph for the purpose of understanding kernel states?*

Existing kernel debugging techniques either do not focus on program state understanding [1, 5, 11, 13, 16, 19] or lack appropriate abstractions for high-level data structures [4, 6]. Moreover, most of existing tools are textual and are unable to produce readable results for high-dimensional information.

Past research has confirmed that program visualization can effectively help developers understand program states [17, 22, 26]. However, none of traditional visualized debuggers can handle the excessively large and complex program state of the Linux kernel [12, 20, 27]. The lack of suitable abstractions in existing tools forces developers to work with complete diagrams of entire massive objects and handle long and deep pointer chains.

2 Approach

We respond to the question in Section 1 by observing that the kernel state is essentially a graph, where kernel objects are pointer-connected nodes. A human debuggers' (implicit) goal is to *simplify* this graph to meet their debugging objectives. This paper argues that such simplified views can be created through two layers of simplification:

1. **ViewCL**, the View *Construction* Language, which allows for the creation of object graphs (plots) at customizable levels of abstraction. ViewCL employs three fundamental operators—*prune*, *flatten*, and *distill*—to programmatically reduce the kernel states.
2. **ViewQL**, the View *Query* Language, which enables customization of a simplified object graph in a developer-friendly manner. ViewQL provides the ability to exclude specific irrelevant object types or fields, eventually producing a human-readable plot.

Our approach is designed upon Daniel Jackson's *small scope hypothesis* [18], which suggests that a *small portion* (sufficiently small to be visually processed by the human brain) of the state suffices for understanding and debugging

EuroSys '25, March 30–April 3, 2025, Rotterdam, Netherlands
2025. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

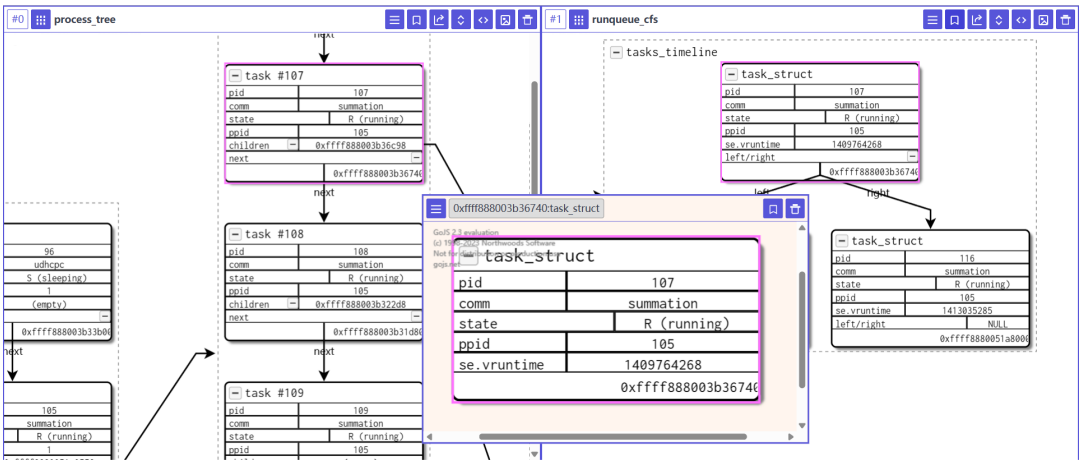


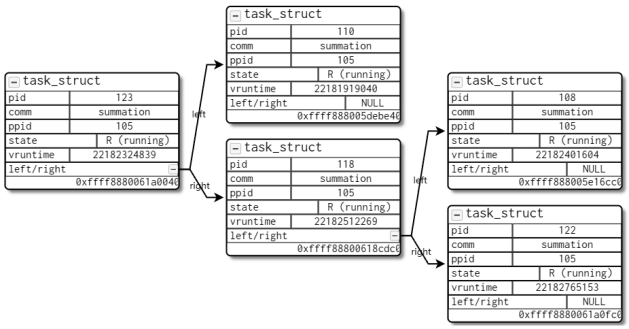
Figure 1. A screenshot of the Visualinux debugger.

a specific part of the system. Our two domain-specific languages work together to break down the large, complex kernel state into simpler state "views" that are visually tractable. With commonly used Linux kernel data structures and containers being predefined in ViewCL, most developers can work exclusively with ViewQL, often without even being aware of ViewCL's existence. The simplicity of ViewQL also enables large language models to customize plots based on user's textual descriptions, without requiring knowledge of either ViewCL or ViewQL.

Visualinux is the first debugging framework that enables developers to programmatically simplify the program state of the Linux kernel to a level that can be visually understood. This is beyond the capability of existing debugging abstractions [15, 21, 25]. Visualinux also offers developers a user-friendly, push-button interface to manage multiple plots and customize their displayed views, as Figure 1 illustrates.

Visualinux is capable of visualizing data structures at various levels of customizations. For instance, only a few lines of ViewCL code can yield a plot of the CFS Scheduler Run Queue, which is visualized as a binary search tree managing the Completely Fair Scheduler (CFS) [2] run queue:

```
1  // Declare a Box for a task_struct object
2  define Task as Box<task_struct> [
3      Text pid, comm
4      Text ppid: parent.pid
5      Text<string> state: ${task_state(@this)}
6                          // ${...} evaluates a C expression
7                          // @this refers to the Box itself
8      Text se.vruntime
9  ]
10 // cpu_rq(0) is the run queue of the first processor
11 root = ${cpu_rq(0)->cfs.tasks_timeline}
12 // RBTree is our predefined container data structure
13 // @root refers to the definition in Line 11
14 sched_tree = RBTree(@root).forEach |node| {
15     // For each node yield (create) a box of Task whose
16     // associated object is @node.se.run_node
17     yield Task<task_struct.se.run_node>(@node)
18 }
19 // Plot the object graph rooted at @sched_tree
20 plot @sched_tree
```



3 Implementation and Evaluation

We implement Visualinux prototype for interactive kernel debugging. It comprises two main components: the GDB extension, which is integrated into the GDB host to support ViewCL, and a visualizer front-end that supports ViewQL. The tool is publicly available at:

https://icsnju.github.io/visualinux

Our evaluation shows that Visualinux can handle various data structures and mechanisms in the Linux kernel. Specifically, it is capable of porting representative figures from the well-known (though now obsolete) textbook *Understanding the Linux Kernel* [14] to the latest Linux 6. All of the plots are available in the public site, including the maple tree mentioned in Section 1.

Visualinux can also assist in diagnosing real-world kernel bugs that require an understanding of the kernel state. We demonstrate its effectiveness through case studies involving the diagnosis of real-world Linux kernel CVEs [23, 24].

Finally, we evaluated Visualinux in two representative debugging scenarios and the results show that the performance overhead of Visualinux is generally acceptable.

References

[1] 2019. *A kernel debugger in Python: drgn.* https://lwn.net/Articles/789641/

[2] 2022. *CFS Scheduler - The Linux Kernel documentation.* https://docs.kernel.org/6.1/scheduler/sched-design-CFS.html

[3] 2022. *Introducing Maple Trees [LWN.net].* https://lwn.net/Articles/845507/

[4] 2024. *crash(8) — Linux manual page.* https://man7.org/linux/man-pages/man8/crash.8.html

[5] 2024. *Debugging kernel and modules via GDB.* https://docs.kernel.org/dev-tools/gdb-kernel-debugging.html

[6] 2024. *Documentation for Kdump - The Kexec-based Crash Dumping Solution.* https://www.kernel.org/doc/html/latest/admin-guide/kdump/kdump.html

[7] 2024. *GDB: The GNU Project Debugger.* https://sourceware.org/gdb/

[8] 2024. *Linux Tracing Technologies.* https://www.kernel.org/doc/html/latest/trace/index.html

[9] 2024. *Message logging with printk.* https://www.kernel.org/doc/html/latest/core-api/printk-basics.html

[10] 2024. *Perf Wiki.* https://perf.wiki.kernel.org/index.php/Main_Page

[11] 2024. *Using KGDB, KDB and the kernel debugger internals.* https://www.kernel.org/doc/html/latest/dev-tools/kgdb.html

[12] David B. Baskerville. 1985. *Graphic Presentation of Data Structures in the DBX Debugger.* Technical Report UCB/CSD-86-260. EECS Department, University of California, Berkeley.

[13] Tegawendé F. Bissyandé, Laurent Réveillère, Julia L. Lawall, and Gilles Muller. 2012. Diagnosys: automatic generation of a debugging interface to the Linux kernel. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (Essen, Germany) *(ASE '12)*. Association for Computing Machinery, New York, NY, USA, 60–69. https://doi.org/10.1145/2351676.2351686

[14] D. Bovet and M. Cesati. 2005. *Understanding the Linux Kernel, 3rd Edition.* O'Reilly Media.

[15] Úlfar Erlingsson, Marcus Peinado, Simon Peter, and Mihai Budiu. 2011. Fay: extensible distributed tracing from kernels to clusters. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 311–326. https://doi.org/10.1145/2043556.2043585

[16] Xinyang Ge, Ben Niu, and Weidong Cui. 2020. Reverse Debugging of Kernel Failures in Deployed Systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 281–292. https://www.usenix.org/conference/atc20/presentation/ge

[17] Jane Hoffswell, Arvind Satyanarayan, and Jeffrey Heer. 2018. Augmenting Code with In Situ Visualizations to Aid Program Understanding. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada) *(CHI '18)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3173574.3174106

[18] Daniel Jackson. 2012. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press.

[19] Dae R. Jeong, Minkyu Jung, Yoochan Lee, Byoungyoung Lee, Insik Shin, and Youngjin Kwon. 2023. Diagnosing Kernel Concurrency Failures with AITIA. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 94–110. https://doi.org/10.1145/3552326.3567486

[20] Tim Kräuter, Harald König, Adrian Rutle, and Yngve Lamo. 2022. The Visual Debugger Tool. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 494–498. https://doi.org/10.1109/ICSME55016.2022.00066

[21] Geoffrey Lefebvre, Brendan Cully, Christopher Head, Mark Spear, Norm Hutchinson, Mike Feeley, and Andrew Warfield. 2012. Execution mining. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments* (London, England, UK) *(VEE '12)*. Association for Computing Machinery, New York, NY, USA, 145–158. https://doi.org/10.1145/2151024.2151044

[22] J.I. Maletic, A. Marcus, and M.L. Collard. 2002. A task oriented view of software visualization. In *Proceedings First International Workshop on Visualizing Software for Understanding and Analysis*. 32–40. https://doi.org/10.1109/VISSOF.2002.1019792

[23] MITRE. 2022. *CVE-2022-0847.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-0847

[24] MITRE. 2023. *CVE-2023-3269.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-3269

[25] Andrew Quinn, Jason Flinn, Michael Cafarella, and Baris Kasikci. 2022. Debugging the OmniTable Way. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 357–373. https://www.usenix.org/conference/osdi22/presentation/quinn

[26] Gruia-Catalin Roman and Kenneth C. Cox. 1992. Program Visualization: The Art of Mapping Programs to Pictures. In *Proceedings of the 14th International Conference on Software Engineering* (Melbourne, Australia) *(ICSE '92)*. Association for Computing Machinery, New York, NY, USA, 412–420. https://doi.org/10.1145/143062.143157

[27] Andreas Zeller and Dorothea Lütkehaus. 1996. DDD—a Free Graphical Front-End for UNIX Debuggers. *SIGPLAN Not.* 31, 1 (jan 1996), 22–27. https://doi.org/10.1145/249094.249108