Discussion Notes on Arrakis : The Operating System is the Control Plane

Ying-Jheng Chen
yingjheng.chen@gmail.com

1. Main Idea

Arrakis achieved high performance through the isolation between control and data plane. This paper moved data plane from its own kernel to user mode because of virtualization supported by recent hardware devices.

Keywords Arrakis, Virtualization, Data Plane, Control Plane, Transparency, Networking, Multi-Core Processors, Barrelfish, Operating Systems

2. Contribution

Provided an OS prototype to achieve the separation of chores between the I/O devices (Networking/Disk), kernel, user applications. This prototype can then scale its capability to multi-core processors based on the Barrelfish[1]. Finally, authors evaluated their prototype through the quantified examination of Memcached[2], Redis[3], and HTTP load balancer.

Design Goals[4]

1. Minimize kernel involvement for data-plane operations.
2. Transparency to the application programmer.
3. Appropriate OS/hardware abstractions.

3. Questions

In this section, I will organize closed questions first, followed by open questions. Closed questions might have brief answers in class; however, these questions still need to be organized as well as verified. Open questions rarely have convincing conclusions during class discussions. Therefore, I will research, clarify, and present satisfying answers.

3.1 Closed Questions

Q1. ***Does it sacrifice obvious performance to achieve protection of user space I/O virtualization?***

A1. No, This paper claimed that Arrakis can achieve protection without compromising high performance.

Q2. ***How does it access control between users?***

A2. Through I/O Virtualization, Arrakis kernel authorizes the right to user applications remapping I/O memory. User applications can customize their information sharing through those memory region. Arrakis only initializes the communication between processes through the support of the global naming system and privilege management.

Q3. ***Is Arrakis applicable to data centers?***

A3. Although large data centers might apply this model for their software improvement, they should upgrade their networking devices for the support of virtualization. To conclude, the Arrakis might be costly to apply.

Q4. ***In earlier research, minimizing kernel aims for security, robustness, and customization while sacrificing performance[5][6][7][8][9][10]. Now, it surprisingly achieved high performance through the minimization of kernel. Why do you think minimizing kernel can achieve high performance?***

A4. Minimizing kernel to achieve high performance is not a new concept. However, because of recent development of advanced hardware, this concept becomes applicable. Authors sped up performance through kernel minimization and I/O virtualization supported by advanced hardware.

3.2 Open Questions

Q5. ***Can Arrakis be Applied to Virtualization?***

A5. Because this paper had discussed the capability of virtualization without any evaluation, I had to trace the related source code for the Arrakis, one of the subsystems of Barrelfish. It really provided the Struct Guest to support virtualization of hardware of Memory, I/O devices, and CPU. Through the examination of Struct Guest in Figure 1 defined in [11], we can see how Arrakis organized the virtualization of Memory and I/O devices for virtual machine guests. Those data members are related to memory devices such as apic, power management; pci, bus transmission; iopm, I/O memory map. Moreover, in the Struct Guest, we notice the ctrl with the type of struct guest_control in Figure 2 defined in [12]. The ctrl of the type, struct guest_control, can support virtualization of CPU through host/guest register set save/restore. Finally, Libarrakis is a new library for user applications to directly use CPU features without kernel intervention.

Q6. ***How does Arrakis Separate Control and Data plane?***

A6. In traditional OS design such as UNIX[13][14], kernel should be responsible for control and data plane; however, Arrakis already removed most part of the data plane. Now, I will outline how Arrakis made changes in data plane including networking devices and memory below.

- For I/O networking devices, Arrakis supports the virtual function driver so that user application can execute direct operations on e1000 and e10k networking devices.
- For operations in memory, Arrakis uses the IOMMU driver to provide the sharing space between users. Therefore, user applications can directly read/write the sharing spaces without intervention of kernel.

Q7. ***Did they prove that protection has no contradiction to high performance in Arrakis ?***

A7. Although there is no direct evidence that supports protection without contradiction to high performance, the authors sped up performance significantly. It seems that the

design of Arrakis can achieve high performance without compromising protection cost.

Q8. ***How does this paper demonstrate that Arrakis can achieve locality and load balance?***

A8. Arrakis used two kinds of evaluation to show the capability of locality and load balance.

- For locality, Arrakis achieve space locality on CPU through the bindings between cpu and user application. Arrakis would try its best to make each user application run on the previous processors.
- Authors used hundreds of thousands web connections per second at most to verify the capability of load balance. They achieved load balance through scalability to utilize multi-processors.

Q9. ***How did Arrakis support multi-core processors?***

A9. Arrakis still used most of original features already supported by Barrelfish. Arrakis provides the transparency between CPU and user applications through I/O virtualization.

Q10. ***How much data did this paper use for its evaluation?***

A10. This paper demonstrated limited amount of data in each I/O operation. As for evaluating Arrakis' performance to connect web services, such as HTTP, this paper used a simple static web page in 1,024 bytes. For Memcached Key-Value Store , the size of one key is only 64 bytes and the size of one value is merely 1KB. In sum, this paper did not perform any I/O transaction in big data size.

Q11. ***What is Memcahced?***

A11. This paper use the Memcahed as a benchmark for in-memory I/O operation. The following descriptions are quoted from https://memcached.org/

- Free and open source, high-performance, distributed memory object caching system, generic in nature, but intended for use in speeding up dynamic web applications by alleviating database load.
- Memcached is an in-memory key-value store for small chunks of arbitrary data (strings, objects) from results of database calls, API calls, or page rendering.
- Memcached is simple yet powerful. Its simple design promotes quick deployment, ease of development, and solves many problems facing large data caches. Its API is available for most popular languages.

4. Conclusion

I really appreciate this paper demonstrated the possibility that minimizing kernel can improve performance based one the modern hardware. I can not wait to see more microkernel OSes which can be sped up; therefore future microkernel OSes can have robustness as well as high performance. Finally, if this notes have any mistakes or ambiguous points, welcome to correct me through Email.

References

[1] Barrelfish. `http://www.barrelfish.org/`. Accessed: 2016-09-26.

[2] Memcached description. `https://memcached.org/`. Accessed: 2016-09-26.

[3] Redis. `http://redis.io/`. Accessed: 2016-09-26.

[4] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 1–16, Berkeley, CA, USA, 2014. USENIX Association.

[5] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Minix 3: A highly reliable, self-repairing operating system. *SIGOPS Oper. Syst. Rev.*, 40(3):80–89, July 2006.

[6] Andrew S. Tanenbaum. Lessons learned from 30 years of minix. *Commun. ACM*, 59(3):70–78, February 2016.

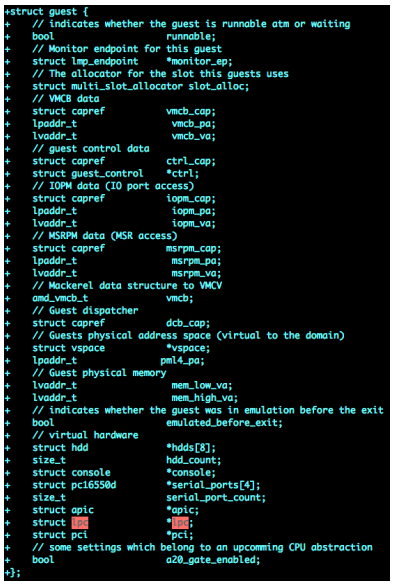
[7] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM.

[8] Gernot Heiser and Kevin Elphinstone. L4 microkernels: The lessons from 20 years of research and deployment. *ACM Trans. Comput. Syst.*, 34(1):1:1–1:29, April 2016.

[9] J. Liedtke. On micro-kernel construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 237–250, New York, NY, USA, 1995. ACM.

[10] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[11] guest data structure. `https://github.com/UWNetworksLab/`



```
+struct guest {
+    // indicates whether the guest is runnable atm or waiting
+    bool                runnable;
+    // Monitor endpoint for this guest
+    struct lmp_endpoint *monitor_ep;
+    // The allocator for the slot this guests uses
+    struct multi_slot_allocator slot_alloc;
+    // VMCB data
+    struct capref       vmcb_cap;
+    lpaddr_t             vmcb_pa;
+    lvaddr_t             vmcb_va;
+    // guest control data
+    struct capref       ctrl_cap;
+    struct guest_control *ctrl;
+    // IOPM data (IO port access)
+    struct capref       iopm_cap;
+    lpaddr_t             iopm_pa;
+    lvaddr_t             iopm_va;
+    // MSRPM data (MSR access)
+    struct capref       msrpm_cap;
+    lpaddr_t             msrpm_pa;
+    lvaddr_t             msrpm_va;
+    // Mackerel data structure to VMCV
+    amd_vmcb_t          vmcb;
+    // Guest dispatcher
+    struct capref       dcb_cap;
+    // Guests physical address space (virtual to the domain)
+    struct vspace       *vspace;
+    lpaddr_t            pml4_pa;
+    // Guest physical memory
+    lvaddr_t             mem_low_va;
+    lvaddr_t             mem_high_va;
+    // indicates whether the guest was in emulation before the exit
+    bool                emulated_before_exit;
+    // virtual hardware
+    struct hdd          *hdds[8];
+    size_t              hdd_count;
+    struct console      *console;
+    struct pc16550d     *serial_ports[4];
+    size_t              serial_port_count;
+    struct apic         *apic;
+    struct [illegible]  *[illegible];
+    struct pci          *pci;
+    // some settings which belong to an upcomming CPU abstraction
+    bool                a20_gate_enabled;
+};
```

Figure 1. [**?**] Struct guest

```
/**
 * \brief A VMKit guest control and state structure.
 *
 * Defines some control and state values shared beween VMKit kernel and a VM
 * monitor.
 */
struct guest_control {
    /// Space to store all regs not captured in the VMCB
    struct registers_x86_64 regs;
    struct registers_x86_64 host_regs;
    uint64_t guest_cr2;
    uint64_t host_cr2;
    uint64_t        num_vm_exits_with_monitor_invocation;
    uint64_t        num_vm_exits_without_monitor_invocation;
};
```

Figure 2. Struct guest_control

arrakis/blob/master/usr/arrakismon/guest.h#L21. Accessed: 2016-09-26.

[12] guest_control data structure. https://github.com/BarrelfishOS/barrelfish/blob/master/include/barrelfish_kpi/vmkit.h#L30. Accessed: 2016-09-26.

[13] Dennis M. Ritchie and Ken Thompson. The unix time-sharing system. *Commun. ACM*, 17(7):365–375, July 1974.

[14] Dennis Ritchie. The evolution of the unix time-sharing system. In *Proceedings of a Symposium on Language Design and Programming Methodology*, pages 25–36, London, UK, UK, 1980. Springer-Verlag.