Offloading Operating System Functions to the Cloud

Zhiyao Ma, Samantha Detor, and Lin Zhong
Yale University
{zhiyao.ma, sam.detor, lin.zhong}@yale.edu

ABSTRACT

This paper questions a fundamental assumption by a modern operating system (OS): it must run in the same computer it manages. We show that for many desirable OS functions, embedded systems often do not have the necessary resources. By carefully offloading some OS functions to another more resourceful computer, e.g., the cloud, one not only immediately overcomes the local resource limits but also opens the door for interesting optimizations because the remote computer becomes an advantageous point of aggregation and coordination. We discuss the challenges to offloading OS functions and their potential solutions. We also share some preliminary results of offloading system initialization logic and dynamic memory management from a microcontroller-based embedded system.

1 INTRODUCTION

An OS manages the resources of a computer. It is assumed that the OS runs on the *same* computer as it manages. This paper questions that assumption and makes a case for offloading (at least some) OS functions to another computer, e.g., the cloud.

We reached this position through our experience of porting an experimental OS called Theseus to microcontroller-based embedded systems (§2). Theseus [5] can recover from transient faults and update itself without rebooting, which makes it ideal for mission-critical embedded systems. To achieve this, Theseus consists of a large number of modules (called *cells*) that are loaded and linked at runtime. Cells, implemented as Rust crates, interact with each other via clearly defined, runtime-persistent boundaries. As a result, Theseus must maintain information about its numerous cells. The bookkeeping data, while negligible on 64-bit machines, becomes prohibitive for microcontroller-based embedded systems.

Our epiphany toward overcoming this problem was that the bookkeeping data is updated and used infrequently, i.e., when a cell is first used or reloaded for fault tolerance or live evolution. As a result, such data, along with the logic that manages it, does not have to reside in the embedded system. With reliable network connectivity, a remote, more capable computer can perfectly bookkeep cells and run the logic for fault recovery and live evolution. Extrapolating from this experience, we posit that many other OS functions could also run in a remote computer and as a result, overcome the resource limitation of the managed computer and create new opportunities of optimization (§2.1).

HOTMOBILE '24, February 28–29, 2024, San Diego, CA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0497-0/24/02.
https://doi.org/10.1145/3638550.3641127

This paper elaborates this position by discussing the challenges toward offloading OS functions (§3) as well as design solutions for them (§4). We present preliminary results from prototype embedded systems that offload system initialization logic and dynamic memory management as two example OS functions (§5). We discuss prior work that provides either inspirations or solutions for offloading OS functions in §6.

2 WHY OFFLOADING OS FUNCTIONS

Theseus is an OS written in Rust aiming for resilience against kernel panics and hardware transient faults, and it also supports live evolution of all kernel subsystems without requiring a reboot, all using the same set of mechanisms. These properties make Theseus particularly attractive for mission-critical embedded systems where robustness, resilience, and ease of update are highly valued. Because ARM Cortex M-based microcontrollers are popular with such embedded systems, we started porting Theseus from x86_64 to such microcontrollers but quickly encountered an insurmountable obstacle as the system ran out of memory (SRAM) and storage (FLASH).

Runtime Linking. For fault recovery and live evolution, Theseus maintains boundaries between cells, by linking the cells at runtime. However, the binary must contain substantial additional data (>50%) to facilitate linking at runtime, which is only consulted during linking operations. Specifically, all object files in Theseus must keep additional sections for runtime relocation, including the symbol and relocation table, i.e., the `.symtab`, `.strtab`, and `.rela.*` sections. As shown in Table 1, these sections account for more than 50% of the binary size, yet are consulted only while linking. Theseus does not employ position-independent code (PIC) but performs absolute relocation at runtime; as a result, it eschews dynamic symbol sections `.dyn*` and dynamic relocation section `.rel.dyn` for conventional dynamic linking which is based on PIC. We note that microcontroller-based embedded systems usually do not support runtime linking at all so they suffer from neither overhead.

Moreover, bookkeeping data for runtime linking also occupies significant memory, because Theseus tracks the addresses of loaded object sections, the inter-object dependencies, and their exported global symbols. 58.4% of the `.strtab` section stores global symbol names, which Theseus maintains in memory. Considering that the memory on microcontrollers is about 10 times smaller than the storage, even though the code and read-only data can be kept in flash, global symbols can occupy the whole memory by itself. Even worse, Theseus employs nested namespaces, providing different visibility to linked objects in each namespace to support live evolution. To ensure efficient retrieval of global symbols, Theseus incorporates a hash table into each namespace, leading to multiple copies of public symbol names thus significant memory consumption.

Table 1: Relative sizes of sections of Theseus. We note those of the Linux kernel are very different because the Linux kernel is statically linked and does not support fault recovery and live evolution as Theseus does.

Section name	Size percentage
`.text`	34.76%
`.data`	0.12%
`.rodata`	6.93%
`.symtab`	5.32%
`.strtab`	17.76%
`.rela.*`	26.96%
`.eh_frame`	7.15%
`.gcc_except_table`	1.01%

Error Handling. Theseus, like most OSes, includes extensive error handling paths in the code for robustness, which are rarely executed. Unlike most OSes, Theseus handles kernel panics through unwinding the stack, which requires extra sections in the object files called `.eh_frame` and `.gcc_except_table`. The former is the unwinding table used to recover callee-saved registers while back-tracing each function. The latter contains pointers to landing pads that are the code to invoke object destructors during unwinding. These sections are only consulted when there is a panic, but account for more than 8% of the binary size. The rarely executed portion of `.text` plus the two sections for unwinding can account for 35% of the binary size.

Dynamic Memory Management. Theseus employs a single heap for system-wide dynamic memory allocation. Memory allocated from the heap almost always incurs a bookkeeping overhead. Most heap implementations include a header in each memory chunk, which records the chunk size and possibly other data to facilitate fast free chunk lookup or merging. Because the memory allocation typically requires 4-byte alignment, the header will be at least 4-byte each. For a typical allocation size of around 36 bytes on microcontrollers, the 4-byte overhead can lead to 10% memory waste.

2.1 Benefits of Offloading OS functions

Our experience described above compels us to reach the idea of offloading because both the resource constraints of embedded systems and the resource usage by Theseus' fault recovery and live evolution mechanisms appear to be fundamental. Offloading appears to be particularly suitable for solving the challenges we encountered because much of the bookkeeping data and error handling logic is only needed occasionally, e.g., when a new task is created, a fault is detected, or a new kernel update is applied.

Extrapolating from the case of porting Theseus, we posit that many other OS functions could also run on a remote computer. Once we have reached the idea of offloading OS functions, we realize there are other benefits, *beyond* overcoming the resource constraints of the managed computer.

Central Aggregation & Coordination. When multiple computers offload OS functions to the same remote computer, the remote computer conveniently becomes a central point for aggregation and coordination, which enables otherwise impossible optimizations. For example, when Wi-Fi access points are in a dense deployment, locally selected channels are usually sub-optimal [8]. If all access points offload their channel selection function, usually part of the radio driver, to the same remote computer (or cloud), the latter can easily find the globally optimal channel assignment.

Seamless OS Function Evolution. Updating an OS function is known to be tricky and often requires rebooting. This, unfortunately, creates a tension between updating the OS, to make new features available or to fix a newly discovered vulnerability, and disrupting the operation (and usability). In practice, this means many OS updates are not applied timely. Offloading would make it possible to deploy an OS update in a way transparent to the users of the managed computer.

Better Security. Embedded systems, such as IoT devices, are notorious for their lack of security, partly because each of them has to defend against all the vulnerabilities associated with its OS and there are a large number of them out there. By offloading OS functions to the cloud, one reduces the OS code inside an embedded system, reducing its attack surface and shifting the responsibility of defense to the cloud. Furthermore, there has been a growing interest [6, 14, 30, 31] in mediating how the OS accesses user data, not trusting the OS at all. Recent work [20] has demonstrated that even the Linux kernel can properly function without unmediated access to local resources. By relocating the OS to an external machine, local software can oversee and regulate all operations from the remote OS, effectively preserving user privacy and even rendering side-channel attacks infeasible.

3 CHALLENGES

Offloading OS functions splits local software into a distributed system and as a result, subjects the OS itself to the challenges known to distributed systems, including existing application logic offloading work (See §6). We next elaborate on these challenges in the context of OS functions.

Latency. The obvious first challenge is the latency introduced by offloading. While a local OS function is available via a function call (or syscall), an offloaded OS function will add the latency from the network stack (on both ends of the network) and the network itself. That is, it suffers from the latency of a remote procedure call (RPC), i.e., 1s to 100s of milliseconds, instead of that of a function/system call, i.e., 10s to 1000s of nanoseconds.

Existing techniques to hide network latency may not be directly applicable in the case of OS function offloading. Batching or pipelining would be infeasible if operations depend on each other, like when a driver manipulates hardware registers. Executing OS functions speculatively is also not realistic since identifying application patterns and supporting OS operation roll-back can greatly increase system complexity. Caching, however, remains an applicable technique, which we discuss further in §4.

We argue that the challenge of network latency also may not be as bad as it initially appears. First of all, with the wide deployment of 5G mobile networks and their much improved wireless link latency (a few milliseconds) and potential deployment of edge data centers, the network latency to access an OS function in an edge data center will be within 10 milliseconds. Second, performance loss due to offloading may be acceptable, especially for OS functions that are not invoked frequently or in the critical path, such as fault recovery and live evolution logic. We also note that for many resource-constrained embedded systems, absolute performance is usually less important than predictability. Finally, we can tap the rich literature that deals with the overhead of syscalls, e.g., [29], and that of RPCs, e.g., [16, 27].

Disconnection. Worse than network latency, offloading also subjects OS functions to potential disconnection. This is particularly true for mobile and embedded systems that often rely on wireless networks. Disconnection may result in a system-wide stall in the case of OS function offloading, in contrast to affecting a single application when it relies on offloaded logic. Again, we note that 5G mobile networks promise much wider and more reliable coverage. Ironically, because many mobile and IoT services rely on their cloud end, they would become useless when disconnected, regardless of whether an OS function is available or not. On the other hand, drawing inspiration from *disconnected operation* from the Coda file system [17], one can *cache* OS function results locally to cope with intermittent disconnections, an idea we will explore further in §4.2.

Development. An OS function does not work in isolation: it interacts with other OS functions via function interfaces and global data structures. Offloading it invites the same set of considerations faced by offloading application logic. Ideally, the boundary of the offloaded function should be narrow and its interface with the rest of the OS well-defined. Related, the remote computer is likely to use a different CPU architecture and a different OS. One cannot simply take the OS code and expect it to run on the remote computer. In §4.1, we will discuss how virtualization technologies can be used to ease this challenge.

Trust. Offloading OS functions demands a secure and authenticated connection between the local and remote computers. An insecure connection invites the attacker to gain privileged access to the local computer. Similarly, sensitive information may be leaked if the attacker pretends to be either party. Mutual transport layer security (mTLS), championed by recent research [24, 28], stands out as a promising solution, ensuring bidirectional authentication and secure connection. Additionally, a low-cost hardware root of trust [10] is necessary for preventing credential forgery.

4 DESIGN CONSIDERATIONS

We next visit major design issues concerned with OS function offloading, especially in the context of the challenges identified above.

4.1 Enabling Technologies & Mechanisms

We visualize the possibilities of OS function offloading as a continuum depicted in Figure 1. At the left extreme is the conventional OS design in which all functions remain local. At the right extreme is a completely disembodied OS, wherein most of the OS functions run on the remote computer. Between these two extremes, design points exist where different subsets of OS functions may be offloaded, depending on the application and deployment context. As more OS functions are offloaded, the dependency on the network intensifies, and system complexity increases. We next discuss technologies to possibly address these challenges.

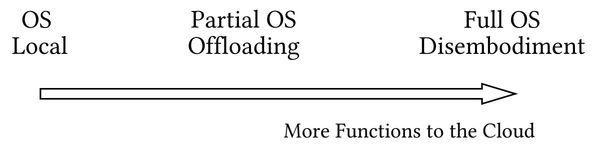

Figure 1: A continuum of OS function offloading between two extremes: (left) completely local OS and (right) completely remote OS. When an OS function is offloaded, we say it is disembodied.

Persistent Network Connectivity. The managed computer requires constant network access to a remote computer for OS functions. This can be achieved by incorporating network support into the managed computer's boot/reset handling logic, typically located in non-volatile memory, e.g., Flash in microcontrollers. In Theseus [5], this includes the *nano core*, a small module that is loaded upon reset/boot and is responsible for loading other modules.

Network-based Privileged Access. OS functions often require privileged access to the managed computer. As a result, the remote computer running an offloaded OS function must have such privileged access over the network. This access can be supported with privileged local logic, or with hardware support to potentially bypass the local CPU for efficiency. For example, in ARM microcontrollers, the debug port interface provides direct access to SRAM and memory-mapped peripherals without engaging the local CPU, which is actually used by ARM semihosting [3]. For more powerful computers, remote direct memory access (RDMA) and compute express link (CXL) similarly allow remote memory access without engaging the local CPU. We note such network-based privileged access is more fine-grained than what is afforded by technologies such as over-the-air (OTA) updates and network boot, which allow the entire OS image to be replaced.

Virtualization. Virtualization technologies can address the development challenge discussed in §3. First of all, the remote computer can employ a virtual machine to provide an identical system environment as the managed computer. One step further, the remote computer can maintain a *digital twin* [22] of the managed computer so that the transplanted OS function can run as if it were inside the managed computer. Moreover, research in OS-level virtualization [18] has discovered opportune boundaries inside mainstream OSes that can be exploited to decouple OS functions. For example, the device file boundary has been shown to be effective in decoupling device drivers from the rest of the OS [1, 2].

4.2 Useful Principles

We next consider several well-known design principles for determining what OS functions to offload and for optimizing their performance once offloaded.

Separation of Data vs. Control Planes. The data plane is where frequent, performance-sensitive actions take place, often in a distributed, uncoordinated manner, while the control plane is where infrequent operations happen, often in a logically central place. While this principle is widely practiced in software-defined networks, it has seen adoption by the software systems community. For example, in Arrakis [25], the OS itself is considered part of the control plane. By applying this principle to an even finer granularity, one can identify control plane functions in the OS as candidates for offloading. For example, the virtual memory subsystem can be considered as part of the data plane because all memory accesses have to go through it. In contrast, heap management can be considered a part of the control plane because it is invoked when new memory needs to be allocated.

Separation of Knowledge and Logic. The rule of representation [26] suggests to "fold knowledge into data, so program logic can be stupid and robust." Our experience with Theseus indicates that an OS often contains a lot of knowledge, both static (about the system) and dynamic (about resource usage). Knowledge that is infrequently consulted (along with the logic manipulating it) could be an excellent candidate for offloading. And the performance impact due to offloading can be ameliorated by caching, as discussed below.

Caching. To cope with the network latency and potential disconnection, the managed computer can *cache* some of the offloaded knowledge, logic, or the output of such logic. For example, as we will show in §5, an embedded system can cache a small number of heap allocations locally to substantially improve the performance when the heap management is offloaded. This caching mechanism introduces an interesting tradeoff between performance and local resource usage and brings new opportunities for optimization. For example, the locally cached data can be updated *speculatively* by the remote computer without engaging the local CPU, using hardware support for remote direct memory access.

4.3 OS Functions to offload

We next examine several OS functions (subsystems) as candidates for offloading.

Device Drivers. Drivers are known to be a major source of vulnerability and incompatibility. There have been various attempts to ship drivers away from the kernel itself. For example, microkernel operating systems "offload" drivers into the user space as processes [15]. LeVasseur et al [19] "offload" a driver into a virtual machine. With these efforts in mind, offloading to a remote computer only appears to be a logical next step and allows the same benefits without the resource taxation on the local computer.

Error Handling. From our experience of porting Theseus to ARM Cortex-M microcontrollers, it is obvious that the bookkeeping data and logic for error handling are excellent candidates for offloading: they consume a lot of resources but are only used infrequently, e.g., when a fault is detected or when a kernel update must be applied.

Heap and Stack Management. Heap management can be offloaded as we will show in §5. With that, it is further profitable to offload stack management. The segmented stack provides an alternative by dynamically allocating stacklets from the heap, providing better memory efficiency than the conventional contiguous stack [21]. Once the heap management is offloaded, it is straightforward to further offload both the logic and bookkeeping data for stacklet allocation, allowing sophisticated solutions to run in the remote computer to achieve higher memory efficiency in the managed computer.

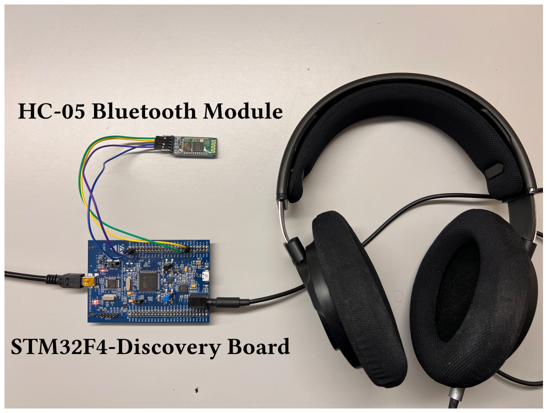

Figure 2: STM32F4-Discovery board connected with HC-05 Bluetooth module over UART, running a music player application and communicating with the server through HC-05.

Runtime Linking. The remote computer can manage the mapping between symbol names and their runtime addresses, where symbol and relocation sections in compiled objects are entirely offloaded. The remote computer performs the relocation and sends the linked binary to the local one. If a small modification to the relocation is to be made, the local computer receives directives from the remote one for the adjustment. Most of the time, applications do not require symbol information at runtime. However, in rare instances when it is necessary, the symbol query can be redirected to the remote computer.

5 PRELIMINARY IMPLEMENTATION

We experiment with a microcontroller-based embedded system to offload two OS functions. The first is system initialization, which is ideal because it is invoked only once per boot cycle and exhibits a higher latency tolerance. Our result highlights the potential of conserving storage through offloading. The second is dynamic memory management, which is substantially more ambitious because it is used frequently and often on the critical path of performance. It, however, allows us to expose challenges when moving rightwards along the offloading continuum depicted by Figure 1. Our findings indicate that caching can effectively counteract network latency.

5.1 System Initialization Code

We experiment with offloading system initialization code with a music player application developed for STM32F4-Discovery board written in C. We connect the board with a laptop PC acting as the server through Bluetooth. The system is shown in Figure 2. Upon boot-up, the board fetches the initialization code on-demand. Given

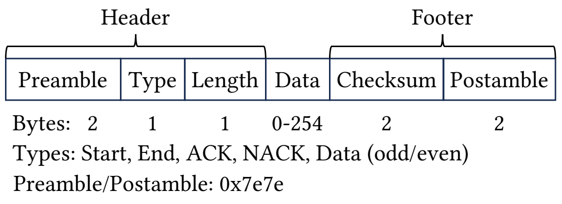

Figure 3: UART packet layout.

the transient nature of the initialization code, the board stores the code in the SRAM for execution and subsequently discards it. Other persistent code is stored in the flash and is executed in place.

The board requires preliminary initialization prior to fetching the code. Specifically, the board communicates with the HC-05 Bluetooth module via UART. The module forwards every byte between the Bluetooth radio and the UART lines. Since the HC-05 module retains its configuration across boot cycles, the board's boot-up procedure need only include the logic to enable the clock for the peripheral bus and to configure the GPIO pins for UART. The board also starts the system clock SysTick during boot-up to allow timing.

Since the UART connection is unreliable, we develop a simple protocol on top of UART to provide a reliable connection. UART transmits individual bytes at the bottom. We group bytes into packets, each bracketed by a header and a footer, as shown in Figure 3. Any byte in between the preamble and postamble that happens to be 0x7e will be escaped by prefixing the escape byte 0x7d. The sender starts or ends a session by sending a `start` or `end` packet, respectively. The sender re-transmits a packet upon timeout waiting for the `ACK` or if the receiver responds with `NACK` upon incorrect checksum. To prevent the receiver from getting duplicated data, the `data` packet type alternates between `odd` and `even`.

We allow convenient migration to offloading initialization code by compiling two binaries from a unified code base: the *local binary* to be stored on the board that contains the boot-up logic and the music player application code, and the *offloaded binary* to be stored on the PC containing the logic to initialize peripherals for the music player. Any function, independent of its final placement in either binary, can be defined in any source file. In practice, the local binary contains functions reachable from `main()` while the offloaded binary from `offloaded_main()`. To achieve this, we first compile each function into a separate ELF section using the `-ffunction-sections` compiler option. Subsequently, we enable the `-gc-section` linker option, keeping only the sections reachable from a given entry point into the linked binary. Kept code sections are merged back into a single `.text` section. The developer willing to offload initialization code needs only call initialization code from `offloaded_main()`, with other code remaining unchanged. Our framework provides a `run_offloaded()` function that downloads, runs, and finally discards the offloaded binary.

Our current implementation takes 1024 bytes on the board to store the UART initialization and the reliable transmission protocol logic. This overhead is constant and independent of the application running on the board. As for the music player application, the offloaded binary containing the peripheral initialization logic is 1080-byte large. We expect greater storage savings for more complicated applications. Fetching the offloaded binary takes on average below one second.

Our current implementation has two limitations. First, functions duplicated in the local and offloaded binaries waste SRAM. The offloaded binary of the music player application contains 40 bytes of duplicated functions. Second, similar duplication of static or global variables may introduce runtime error, because modification to a global variable in one binary will not be reflected in the other one. This can be solved by generating a linker script for the offloaded binary that contains symbol addresses of the functions and global variables from the local binary. The script will allow overriding the definitions in the offloaded binary and point them to the one from the local binary.

5.2 Memory Allocator

We next share some preliminary results from our attempt to offload dynamic memory management from STM32F4-Discovery board to a server, as part of our bigger effort to bring Theseus-like fault tolerance and live evolution to embedded systems. We chose to start with heap management because it has a very simple API (`malloc`/`free`) and its performance can be easily quantified.

At initialization, the microcontroller reports the memory region to be utilized as the heap to the server, which in turn creates the necessary data structures to manage allocated and free chunks. Software on the microcontroller uses the familiar `malloc` and `free` APIs to acquire and release heap memory. A library converts these function calls into remote procedure calls to the management process on the server.

We experiment with the idea of cache (§4.1) and are interested in how it may help cope with network latency and disconnection. For heap management, the cache saves recently freed heap allocations. When local software calls `malloc`, the cache will be consulted first. Only when no saved allocation can satisfy the request, it will be forwarded to the server. When local software calls `free`, the allocation will be returned to the allocation cache, which may evict entries by invoking the heap manager on the server. When disconnection happens, `malloc` can block or return failure if the request cannot be satisfied by the cache; `free` will be buffered until the connection is re-established.

Using artificial traces, we evaluate the efficacy of the cache in combating network latency, by varying the cache size and injecting network latency. We observe that a small cache can be highly effective in overcoming the performance impact of network latency. Specifically, we find that when the cache size is slightly larger than the average number of active allocations, the impact of network latency becomes negligible. Figure 4 shows the results from one of the settings we have tried. In this setting, the intervals between two consecutive `malloc` requests follow an exponential distribution with a mean of 100 µs, while the lifetime of allocated chunks follows a uniform distribution between 500 and 1500 µs. The allocation size follows a normal distribution with a mean of 36 bytes and a standard deviation of 8. In this setting, the average number of active allocations is about 10.

Given that the network latency is orders of magnitude greater than the allocation interval, a moderate cache miss rate causes a

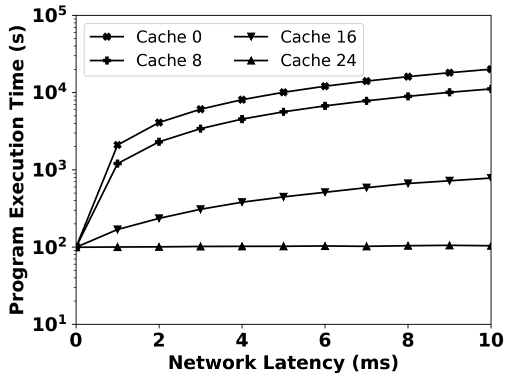

Figure 4: Impact on execution time of network latency and cache size. The cache effectively counters the impact of network latency. The effectiveness of cache is highly dependent on its size, while it can almost fully absorb the allocation request with moderate size.

substantial rise in execution time, as seen when the cache size is 8 in Figure 4. Increasing the cache size, however, can significantly decrease execution time. Furthermore, a moderate size of 24 cache entries renders the network latency's effect on execution time negligible. Note each cache entry consists of two integers, indicating the address and size of the allocation, respectively.

6 RELATED WORK

Microkernel and Library OS. Many have discussed where to place OS functions inside the computer. Microkernel OSes seek to move the OS functions out of the kernel and into the user space. Library OSes argue that OS functions should be implemented by the application itself as a library. While they do not discuss whether or how to move the OS functions outside the managed computer, they show that not all OS functions are equal and some can be placed farther away from others. Our proposal can be considered as a step further logically. More specifically, Chertiton and Duda [7] viewed the kernel execution of OS functions provided by an application as the kernel caches these functions, which, in spirit, is quite similar to the caching mechanism we discussed for offloaded OS functions.

Distributed OS. Offloading OS functions effectively creates a distributed system consisting of the local computer and the cloud where the cloud plays the role of central control. Therefore, it shares some of the challenges facing distributed systems. However, distributed systems usually focus on providing a single abstraction on top of distributed resources.

Computation Offloading and Thin Client. There is a rich literature about offloading computation from a resource-constrained computer to more powerful ones, including cyberforaging [11] and thin client [4, 23]. It focuses on offloading application logic and its programming [9] and OS [12, 13] support. Offloading OS functions can be considered as a special case of it, facing similar challenges of latency, disconnection, and coherence. On the other hand, OS functions often require privileged access and are more performance-sensitive. Because application logic may rely on OS functions, offloading OS functions may create even more opportunities for offloading application logic.

ACKNOWLEDGMENTS

This work is supported in part by NSF Award #2130257 and its REU supplement. The authors are grateful for the constructive and insightful feedback from the reviewers.

REFERENCES

[1] Ardalan Amiri Sani, Kevin Boos, Shaopu Qin, and Lin Zhong. 2014. I/O paravirtualization at the device file boundary. In *Proc. ACM ASPLOS*.

[2] Ardalan Amiri Sani, Kevin Boos, Min Hong Yun, and Lin Zhong. 2014. Rio: a system solution for sharing I/O between mobile systems. In *Proc. ACM MobiSys*.

[3] ARM. 2012. ARM Compiler toolchain version 5.01 update 1. https://developer.arm.com/documentation/dui0471.

[4] Ricardo A Baratto, Leonard N Kim, and Jason Nieh. 2005. Thinc: A virtual display architecture for thin-client computing. In *Proc. ACM SOSP*.

[5] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. 2020. Theseus: an experiment in operating system structure and state management. In *Proc. USENIX OSDI*.

[6] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. 2019. SANCTUARY: ARMing TrustZone with User-space Enclaves.. In *Proc. NDSS*.

[7] David R Cheriton and Kenneth J Duda. 1994. A caching model of operating system kernel functionality. In *Proc. USENIX OSDI*.

[8] Surachai Chieochan, Ekram Hossain, and Jeffrey Diamond. 2010. Channel assignment schemes for infrastructure-based 802.11 WLANs: A survey. *IEEE Communications Surveys & Tutorials* (2010).

[9] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. 2010. MAUI: Making Smartphones Last Longer with Code Offload. In *Proc. ACM MobiSys*.

[10] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. 2012. Smart: secure and minimal architecture for (establishing dynamic) root of trust. In *Proc. NDSS*.

[11] Jason Flinn. 2012. Cyber Foraging: Bridging Mobile and Cloud Computing. *Synthesis Lectures on Mobile and Pervasive Computing* (2012).

[12] Mark S Gordon, David Ke Hong, Peter M Chen, Jason Flinn, Scott Mahlke, and Zhuoqing Morley Mao. 2015. Accelerating mobile applications through flip-flop replication. In *Proc. ACM MobiSys*.

[13] Mark S Gordon, D Anoushe Jamshidi, Scott Mahlke, Z Morley Mao, and Xu Chen. 2012. Comet: Code Offload by Migrating Execution Transparently. In *Proc. USENIX OSDI*.

[14] Le Guan, Peng Liu, Xinyu Xing, Xinyang Ge, Shengzhi Zhang, Meng Yu, and Trent Jaeger. 2017. TrustShadow: Secure execution of unmodified applications with ARM TrustZone. In *Proc. ACM MobiSys*.

[15] Jorrit N Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S Tanenbaum. 2006. MINIX 3: A highly reliable, self-repairing operating system. *ACM SIGOPS Operating Systems Review* (2006).

[16] David B Johnson and Willy Zwaenepoel. 1993. The Peregrine high-performance RPC system. *Software: Practice and Experience* (1993).

[17] James J Kistler and Mahadev Satyanarayanan. 1992. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems (TOCS)* (1992).

[18] Oren Laadan and Jason Nieh. 2010. Operating system virtualization: practice and experience. In *Proc. Annual Haifa Experimental Systems Conference*.

[19] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. 2004. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Proc. USENIX OSDI*.

[20] Caihua Li, Seung-seob Lee, Min Hong Yun, and Lin Zhong. 2022. MProtect: Operating System Memory Management without Access. *arXiv preprint arXiv:2212.12671* (2022).

[21] Zhiyao Ma and Lin Zhong. 2023. Bring segmented stacks to embedded systems. In *Proc. ACM HotMobile*.

[22] Roberto Minerva, Gyu Myoung Lee, and Noel Crespi. 2020. Digital twin in the IoT context: A survey on technical features, scenarios, and architectural models. *Proc. IEEE* (2020).

[23] Jason Nieh, S Jae Yang, and Naomi Novik. 2000. *A comparison of thin-client computing architectures*. Technical Report CUCS-022-00. Columbia University.

[24] Sebastian Paul, Felix Schick, and Jan Seedorf. 2021. TPM-based post-quantum cryptography: a case study on quantum-resistant and mutually authenticated TLS for IoT environments. In *Proce. Int. Conf. Availability, Reliability and Security*.

[25] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2015. Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems (TOCS)* (2015).

[26] Eric S Raymond. 2003. *The art of Unix programming*. Addison-Wesley Professional.

[27] Michael D Schroeder and Michael Burrows. 1990. Performance of the Firefly RPC. *ACM Transactions on Computer Systems (TOCS)* (1990).

[28] Jaspreet Singh, Yahuza Bello, Ahmed Refaey Hussein, Aiman Erbad, and Amr Mohamed. 2020. Hierarchical security paradigm for iot multiaccess edge computing. *IEEE Internet of Things Journal* (2020).

[29] Livio Soares and Michael Stumm. 2010. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *Proc. USENIX OSDI*.

[30] Alexander Van't Hof and Jason Nieh. 2022. BlackBox: A Container Security Monitor for Protecting Containers on Untrusted Operating Systems. In *Proc. USENIX OSDI*.

[31] Minhong Yun and Lin Zhong. 2019. Ginseng: Keeping Secrets in Registers When You Distrust the Operating System. In *Proc. NDSS*.