Red Hat Research
Red Hat Funded this Research

Larry Woodman,
Ulrich Drepper,
Richard Jones,
Daniel Bristol de Olivera

Introducing the Chrono-kernel: Kernel Privilege for the People

Contact:
tommyu@bu.edu

Thomas Unger,
Arlo Albelli,
Ryan Sullivan,
Orran Krieger,
Jonathan Appavoo

BOSTON UNIVERSITY

1

Problem:
General purpose OSs like Linux meet the needs of many applications, yet some with extreme requirements (e.g. latency, throughput, security, etc.) slip through the cracks. Unfortunately, these apps are often the most valuable (e.g. high

2

Approach: Instead of resorting to kernel bypass or custom operating system construction, the goal of this project is to marry the convenience of application programming on Linux with the full power of the kernel's mode of execution.

A Chrono-kernel is a system that offers first-class support for application threads to access supervisor execution mode. We retrofit Linux with a new mechanism, kElevate, turning it into a Chrono-kernel. kElevate allows any set of application threads to toggle into the supervisor mode of hardware execution in ~10 nanoseconds. We call these "elevated" threads.

3

A Use Case: When running in kernel privilege, an elevated thread can exploit the low level internal interfaces of the kernel, in addition to the usual syscall API, utilizing the kernel as a library. We demonstrate using this to shortcut standard syscall paths as well as to perform more aggressive shortcutting deep in the kernel.

4

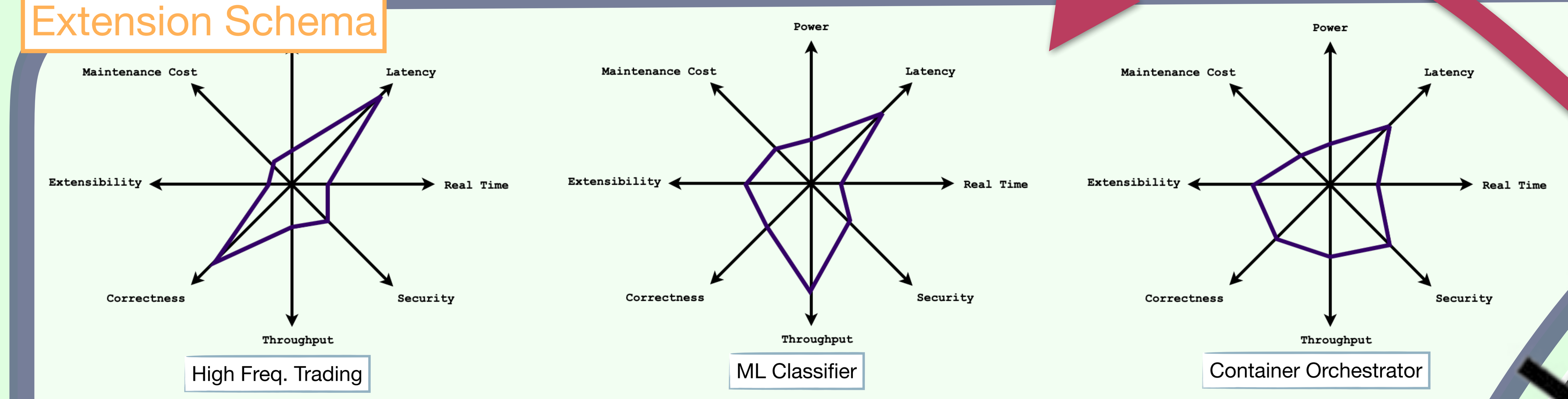
Context: Like Kernel Modules, elevated threads are unlimited in their power to change the system, so they are similarly protected behind superuser access. Unlike Kernel Modules, elevated threads are built to run from the application context.
Elevated applications use standard build processes. ABI compatibility may be maintained if desired. When running in kernel privilege, an elevated thread can make structural changes to Linux. It can interpose on system structures like the interrupt descriptor table, modifying or replacing core system components and code.

5

Deployment: kElevate has wide applicability, it runs baremetal, in containers, and in virtualization. It can be invoked from any language via the syscall interface.

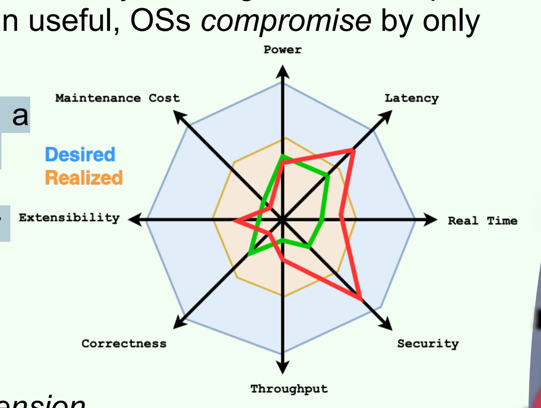
We have retrofitted Linux with the kElevate mechanism in an x86_64 prototype, and we have demonstrated the same technique is viable on an ARM64 proof of concept.

Extension Schema



Visualizing examples of applications' requirement spaces.

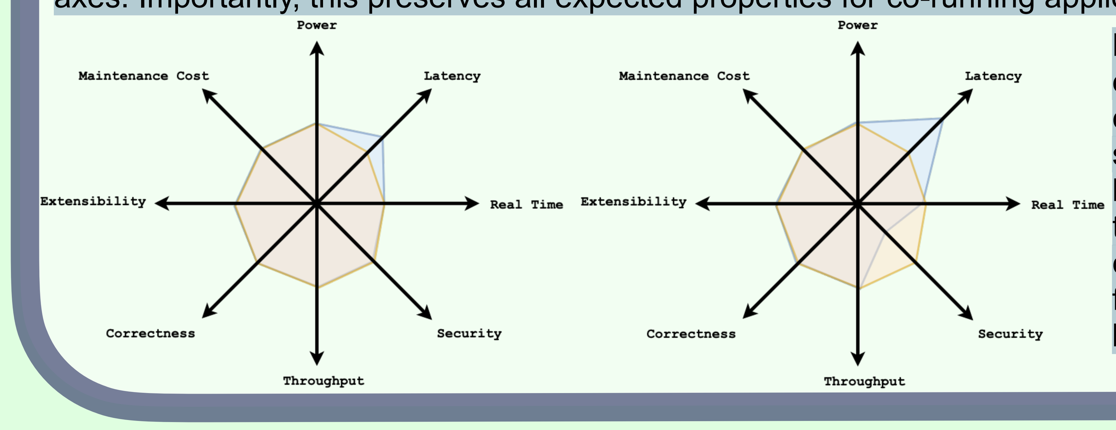
Applications have diverse requirements from the systems that run them.The requirement space of the union of *all* apps can be found by taking the maximum along each dimension: **it is large.** Because these requirements exist in a natural tension (e.g. security trades off with latency), simultaneously meeting all these requirements may not just be a hard problem, but a logical contradiction. To remain useful, OSs *compromise* by only delivering on a subspace of these requirements.



Right: In blue: the union of all application requirements, **in orange:** a visualization of the realized *OS compromise.* **In green:** an example application whose requirements are met by the OS. **In red:** an application app with more extreme requirements that are not met by the OS.

OS *extension* allows application or system library developers to renegotiate the OS compromise. This allows for modifying system interfaces or implementation with application specific changes. *Extension* is typically not a general improvement that applies to all applications. We differentiate between *conformant* and *transgressive* extensions (see below). Both optimize a chosen application along some axes. kElevate can be used to extend Linux at runtime following either model.

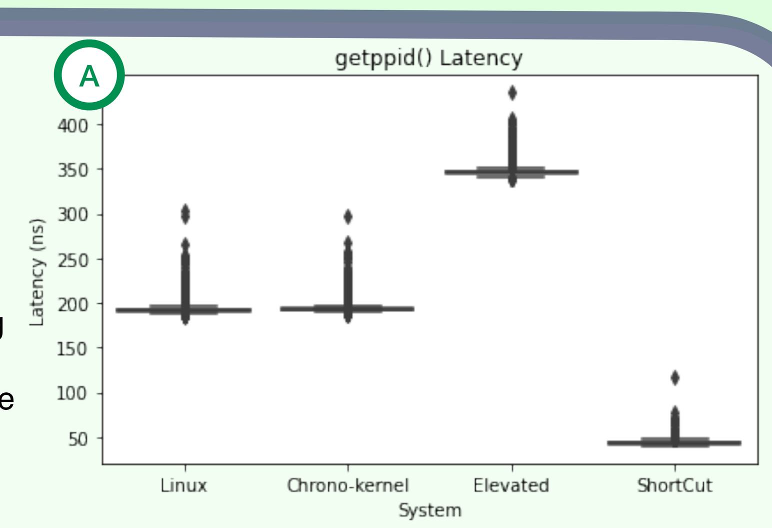
Left: Conformant extensions play along with the OS compromise by strictly improving along a subset of axes. Importantly, this preserves all expected properties for co-running applications.



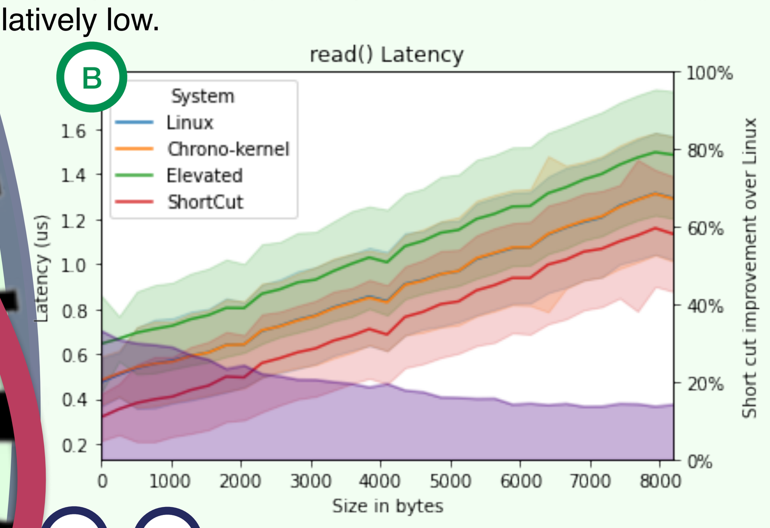
Right: Transgressive extension violates the OS compromise by regressing on a subset axes that are not required by the chosen app. This relaxes the constraints of the compromise, inserting *slack.* This frees the developer by opening a larger optimization space.

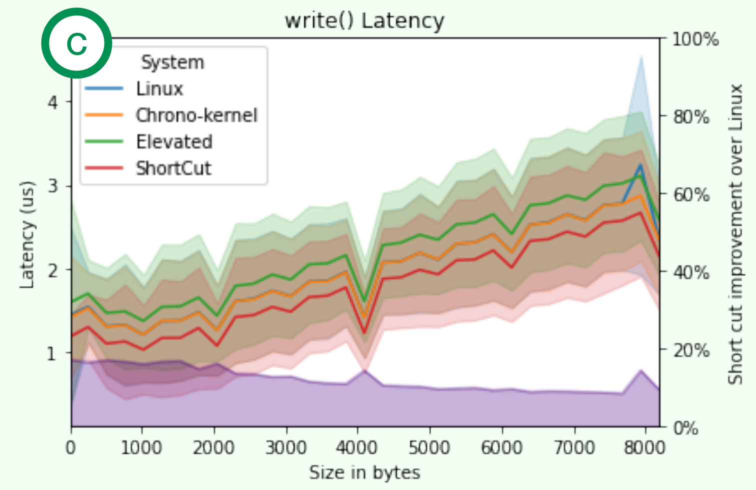
Microbenchmark Experiments

A Getting a Linux process's parent pid getppid() is one of the shortest syscalls. It approximates the total HW and SW costs of the syscall path. We call it in a loop and box plot the individual latencies.
"**Linux**" is a completely standard Fedora 35 environment.
"**Chrono-kernel**" is the same environment plus the kElevate mechanism simply showing a comparable result; the system call path has not been perturbed.
"**Elevated**" actually uses kElevate, showing the latency when running the application in supervisor mode.
"**ShortCut**" flattens the syscall handler away entirely because the elevated thread is able to make a function call to the getppid handler.



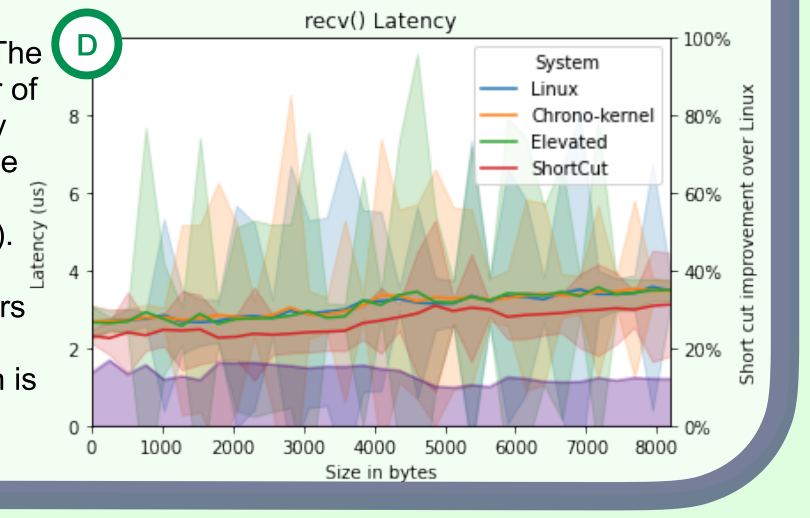
Discussion: syscalls take longer in the elevated case because a slow **Iret** is used instead of the faster **sysret** instruction. We use **Iret** out of convenience to preserve existing kernel exit code; when higher performance is desired, interposition can be used to change the **Iret** into a **ret** making elevated syscalls even faster than the Linux baseline. The time savings on the ShortCut case are mostly due to skipping system software on the ingress and egress paths, hardware overheads are relatively low.





B & **C** Small reads and writes can be accelerated by about 35% and 20% respectively. The relative savings decreases as a function of number of bytes because cpu bound work takes up a relatively larger fraction of execution time. ShortCut makes use of the access to the kernel internal interfaces ksys_read() and ksys_write() respectively (see below).

D The receive system call is a bit noisier than the others on most configurations. Interestingly not only does the ShortCut case improve on latency, the standard deviation is much smaller, which has positive implications on using kElevate in real-time contexts.



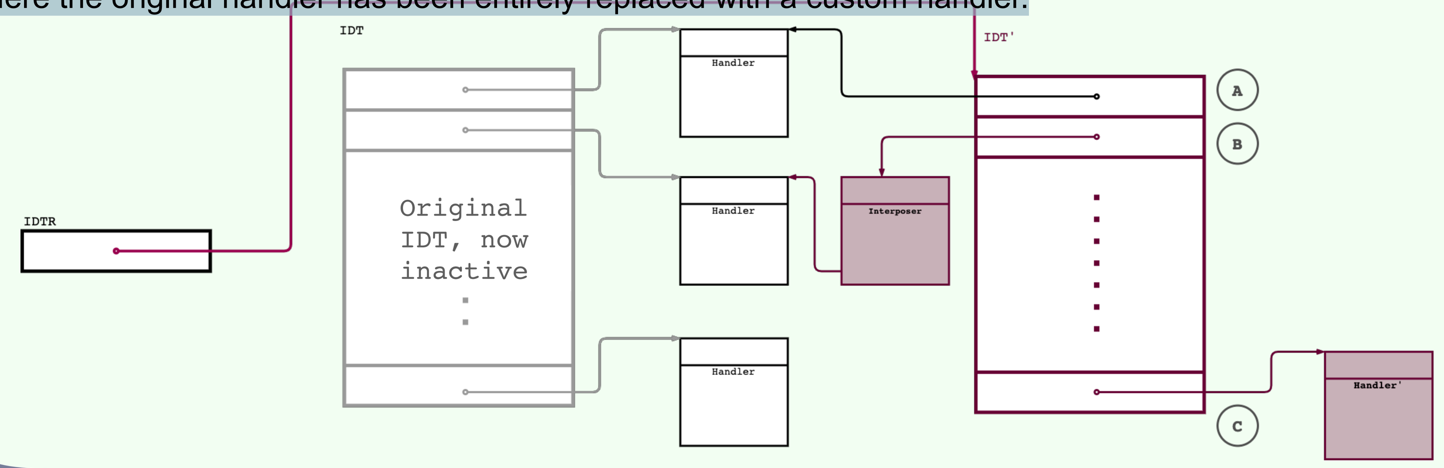
Modifying System Data Structures

Applications will not run in supervisor mode on Linux without system modification. Panics, kernel "Oops"s, and plain old processor resets will occur. Because Linux cleanly separates the application and kernel worlds using hardware mechanisms, we make use of *interposition* on the relevant data structures in order to modify system behavior. The system call handler and Interrupt Descriptor Table (IDT) are our main points of interest.

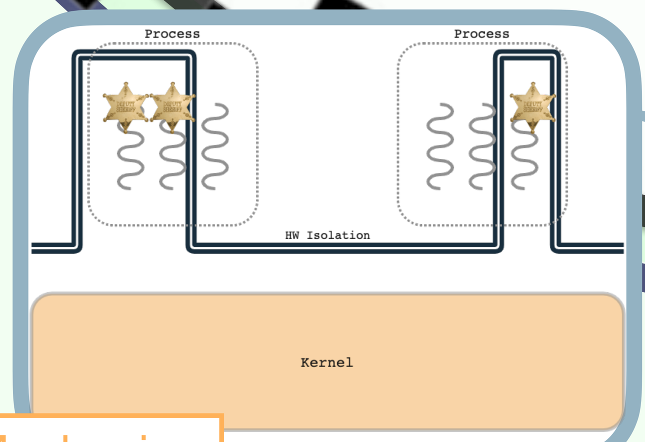
For a motivating example, consider the "stack starvation" problem. Kernel stacks do not fault because they are preallocated and pinned. Correspondingly, the Linux page fault handler does not manage stack faults in supervisor execution mode, instead, a panic occurs. Linux pages application stacks, so elevated threads will trigger this panic. A simple *mitigation* is to flip a single bit in the IDT, telling the hardware to use a special stack (interrupt stack tables). Of course this can be done with a memory write, but we use interposition to allow future context switching back to the unmodified Linux IDT. The program we use to enact this mitigation is a user process built using standard compilation with GCC and uses dynamic linking to our user level library. The only piece we add is our selective use of the kElevate syscall.

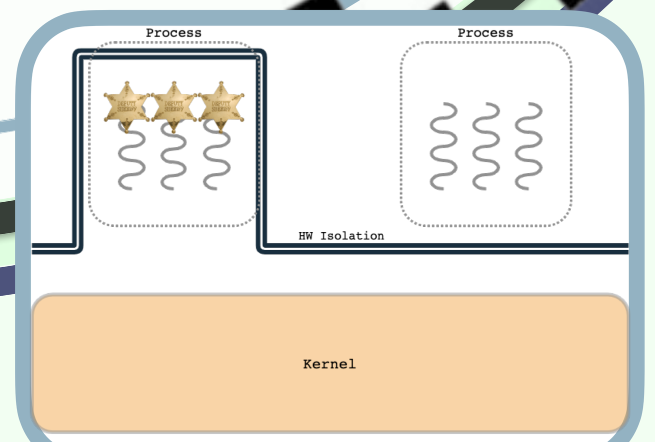
Implementation Detail: Mechanistically, our mitigator pre-allocates its own stack space in user mode and pins it (as any application could do on Linux). Then we make the kElevate syscall so we can (function) call the kernel's page allocator getting the address of a page. Still elevated, we read the protected system register, the IDTR which holds the location of the IDT. Then we use glibc's memcpy() function (demonstrating use of glibc) to copy the system IDT to our allocated page and flip the bit mentioned above. Finally we load the IDTR with the new location of our copied IDT and use the kElevate syscall to lower the thread back to user mode. Now any page faults on that core will use the interrupt stack table mechanism and we will never double fault on a stack page in the future.

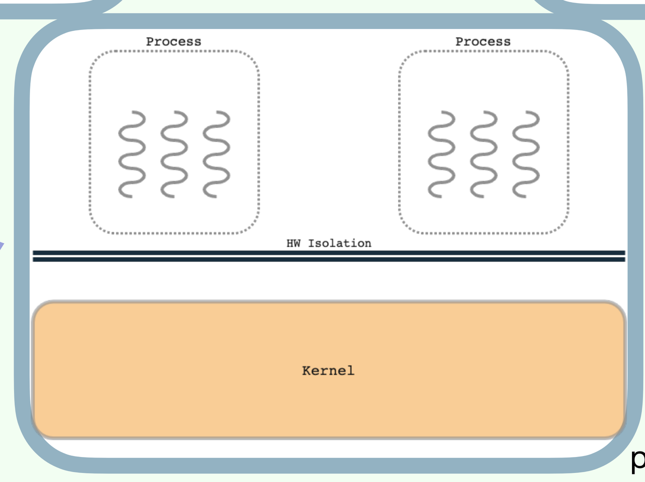
Three more approaches to interposition are illustrated below:
A: An unmodified entry points to the original handler.
B: A handler pointer is modified to point to an interposer which does work before continuing to the original handler.
C: Here the original handler has been entirely replaced with a custom handler.



Mechanism







Our Linux prototype uses the kElevate mechanism to turn Linux into a Chrono-kernel. At any point, an application can access the supervisor mode of the thread granularity. Any subset of threads in the system can be concurrently elevated. In these diagrams, elevated threads are marked with a badge.

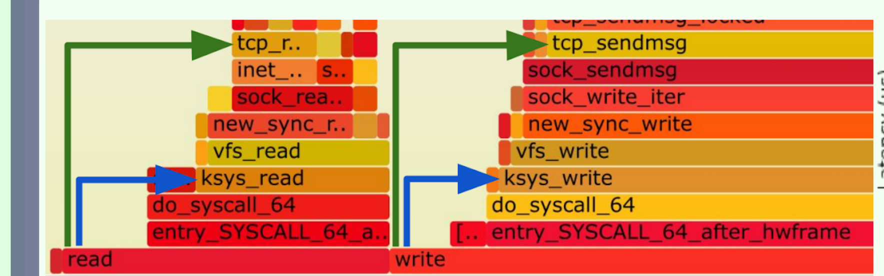
Our implementation introduces kElevate as a new system call. Interposition can be used to further optimize access to this primitive. For the X86_64 prototype, this involved a total of 357 LoC changes. We are able to keep changes to the kernel minimal because once the primitive of privileged execution is exposed, you can script with kernel power. Now, changes that used to go into the kernel can instead be executables living in the filesystem.

You and the Chrono-kernel!

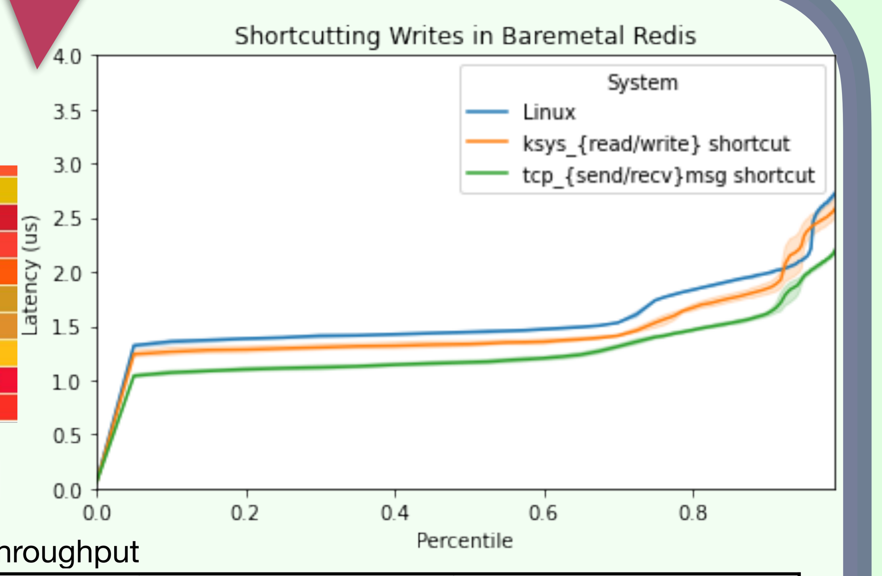
We have many more tasks than we have developers to implement them! We are looking for help from both types of developers: application and kernel. We are looking for applications with clear kernel bound latency/throughput/ real-time optimization objectives and kernel hackers who would be interested in helping us get through our to-do lists or bring their own perspectives to the work. If this sounds interesting to you please contact Tommy (tommyu@bu.edu).

Macrobenchmarks

Not Yet Peer Reviewed



Back traces from profiling Redis. In blue: skipping the syscall entry and exit code. In green: skipping the VFS and socket disambiguation.



Redis Throughput

	Average Throughput (MB/s)	Percent Throughput Improvement wrt Linux	Average Latency (msec)	Percent Latency Improvement wrt Linux
Linux	4.87	-	1.70	-
ksys_{read/write}	5.50	12.9%	1.51	11.2%
tcp_{send/recv}	6.45	32.4%	1.28	24.7%

We show baremetal latency plots demonstrating that a server using the kElevate mechanism improves significantly when using shortcutting directly into the ksys_read and ksys_write handlers. Further, we provide results for making shortcuts much deeper into the kernel, at tcp_sendmsg and tcp_recvmsg, cutting out the trip through the VFS layer to disambiguate the file descriptor.

Redis is an open source key value store, often used for in memory caching for microservices, it is one of the most popular applications by Docker's rankings. Redis makes read and write syscalls for its network traffic. The Redis server shortcuts significantly improve the throughput (by a third) and latency (a quarter) it achieves. A datacenter provider typically controls the software of both the Redis client and the server, so they have the opportunity to shortcut on both sides of the communication. We are currently quantifying the **power** savings as well.

Ongoing work: We are applying the same set of shortcuts to Memcached and Apache. This will demonstrate that our approach is general enough to be reused. We are interested in building tools that application developers can incorporate even if they are not kernel experts. These shortcuts may be used even when the application source code is not available (ABI compatible). Finally, we are also developing microkernel-like servers which can interface with unprivileged applications, securely accelerating their workloads.