Hijack$^{\text{COS}}_{\text{LINUX}}$: Toward Practical, Predictable, and Efficient OS Co-Location using Linux*

Qi Wang, Jiguo Song, Gabriel Parmer, John Wittrock, Yang Yang Wu, Tareque Hossain
The George Washington University
Washington, DC
{interwq,jiguos,gparmer, wittrock, rezie}@gwu.edu, tarequehossain@gmail.com

***Abstract*—Three of Linux's greatest assets, especially in embedded systems, are its extensive corpus of device drivers, its hardware compatibility layer that spans many architectures and platforms, and the broad spectrum of applications ported to it. This paper outlines an OS design that enables the co-location of both the Linux kernel, and other real-time operating system kernels on the same system. In executing other operating systems in this manner, they receive the advantage of Linux's assets, while also enabling the benefits of running other OSes with alternate structures and goals.**

This paper specifically focuses on the design and implementation of the compatibility layer with Linux that enables the co-located OS to selectively use specific Linux functionalities. The technique does not require modifications to the Linux kernel, and supports co-located kernels that provide user-level execution. An emphasis is on maintaining native performance and predictability of all co-located OSes. We detail the use of this technique to co-locate with Linux the COMPOSITE component-based OS – a real-time OS with a very different design. In doing so, we discuss the points of friction where COMPOSITE is adapted to ensure system correctness, or to avoid sacrifices in performance or predictability. We provide a preliminary evaluation of the system with specific emphasis on different means for the co-located OS to access the device drivers and applications of Linux. We believe this evaluation demonstrates the possibilities for such an approach, while introducing a new way to use the Linux code-base, and motivates further research. We present this to the Linux community to engender feedback, and gauge interest.

I. INTRODUCTION

Real-time and embedded systems are often very specialized around a specific set of tasks, and many operating systems exist to cater to these requirements. Linux is appealing for embedded systems due to its extensive development tools, its large application-base, its broad hardware portability, and extensive support for different devices.

This paper does a preliminary investigation of the possibility of using co-location of operating systems to achieve the benefits of both a specialized operating system with a different design, and the generality of Linux. We introduce HIJACK$^{\text{COS}}_{\text{LINUX}}$ that inserts itself as a layer between the hardware and each OS (thus hijacking the system) that multiplexes the hardware between them.

HIJACK$^{\text{COS}}_{\text{LINUX}}$ does not place both operating systems on equal footing, and instead requiring one, the *host OS*, to manage resources that cannot be shared such as device drivers. Though another OS could be used to provide co-location facilities, for the purposes of this work, we focus on Linux due its extensive device-driver support, strong hardware abstraction layer, and source availability. We refer to the OS that is co-located with the host as the *appendage operating system* or the AOS. In this paper we use the COMPOSITE[1] component-based OS as a case-study AOS for OS co-location with HIJACK$^{\text{COS}}_{\text{LINUX}}$. COMPOSITE is a research operating system focused on reliability, predictability, and configurability. It has a very different structure than Linux in that all system policies and most abstractions are implemented as user-level components that export functions through interfaces that other components invoke to access their functionality. This capability to run COMPOSITE (or COS) on Linux, thus the name HIJACK$^{\text{COS}}_{\text{LINUX}}$.

Figure 1 depicts both a normal Linux system and HIJACK$^{\text{COS}}_{\text{LINUX}}$ supporting an AOS. A HIJACK$^{\text{COS}}_{\text{LINUX}}$ Linux module interposes on hardware events, and routes them to the proper OS while enabling the AOS to communicate with and harness the host (Linux). To enable both OSes to share the system processor, the entire AOS is executed within a single host thread (with the exception of interrupt execution). When the host wishes to switch away from the AOS, it can do so by saving the thread's state as normal, and returning later. If the AOS wants predictable execution, it can be executed at the host's highest priority. As we will see, the AOS can harness host functions to manipulate the page tables associated with it's thread.

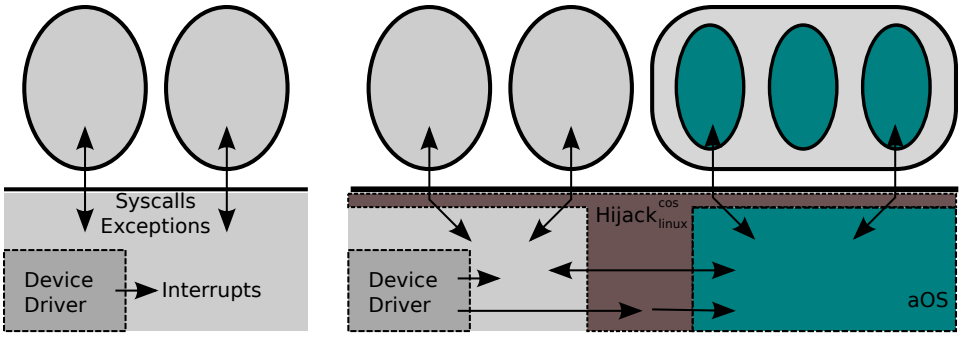
In providing facilities for OS co-location, we summarize the goals of HIJACK$^{\text{COS}}_{\text{LINUX}}$ here.

- **G1.** The HIJACK$^{\text{COS}}_{\text{LINUX}}$ mechanisms should be transparent to the user-level environments in both the host and the AOS.
- **G2.** The modifications to the AOS should be isolated from the core logic of the OS.
- **G3.** The impact on the host source to support HIJACK$^{\text{COS}}_{\text{LINUX}}$ should be minimal.
- **G4.** The communication between host and AOS should be predictable and not impose additional latency compared to traditional forms of IPC.
- **G5.** The host and AOS should be able to share I/O devices. They should both be able to issue relatively low-level commands to the devices.

Where we have not met these goals in the current prototype, we make specific note. In this paper, we wish to show the viability of the approach, and motivate further research.

In contrast to existing systems such as RTLinux [2] and

This material is based upon work supported by the National Science Foundation under Grants No. CNS 1137973, CNS 1149675, and CNS 1117243. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.



(a) Linux (b) HIJACK$^{\text{COS}}_{\text{LINUX}}$

Fig. 1. (a) A normal Linux system in which interrupts from devices, exceptions, and process system calls are directly processed by the kernel. (b) HIJACK$^{\text{COS}}_{\text{LINUX}}$ that interposes at kernel entry points, and multiplexes hardware events to either Linux or the AOS. HIJACK$^{\text{COS}}_{\text{LINUX}}$ provides abstractions for the co-located OSes to communicate, and for the AOS to access peripherals through the Linux drivers. The AOS hosts multiple user-level protection domains while Linux sees all AOS execution as a single Linux process.

RTAI [3], we do not attempt to reduce the response time or predictability properties of Linux. Instead the response time and execution bounds on operations is limited by both Linux and the AOS. Linux has made great strides in decreasing worst-case response time through the Linux-RT patches [4]. Instead, we focus on the orthogonal problem of providing the co-location of both OSes, while enabling them each to maintain the power to define possibly very divergent abstractions for user-level execution.

Contributions. The main contributions of this paper include 1) a discussion of the design and implementation of HIJACK$^{\text{COS}}_{\text{LINUX}}$, a system for co-locating OSes, 2) the application of this system to the COMPOSITE component-based OS, a system with a very different structure than the host, 3) the implementation of this support in Linux as a kernel module, and 4) and a preliminary evaluation of the basic mechanisms for sharing. Though we consider HIJACK$^{\text{COS}}_{\text{LINUX}}$ an existence proof demonstrating the feasibility of this approach, we believe that more research is warranted to determine the full promise of OS co-location.

This paper is organized as follows: Section II discusses how Linux provides the facilities for OS co-location, while Section III discusses our technique for dynamically overriding low-level hardware-triggered execution paths to enable co-located OS execution, and IV discusses our COMPOSITE implementation using this system. Section VI provides an evaluation of the system and Section VII discusses previous work that is related to HIJACK$^{\text{COS}}_{\text{LINUX}}$. Section VIII concludes and discusses future work.

II. THE HIJACK HOST ABSTRACTION LAYER: HAL REPLACEMENT FOR AOSES

The AOS must have access to hardware features required for kernel execution, but the AOS cannot directly access these features as the hardware must be multiplexed with Linux. To mediate hardware access, we present the the Hijack Hardware Abstraction Layer, or H^{2}AL, which is a replacement for the Hardware Abstraction Layer (HAL) of the AOS.

The H^{2}AL functionality is provided as a set of functions that are compiled with the AOS and inserted into the Linux kernel via the module interface. Consequently, HIJACK$^{\text{COS}}_{\text{LINUX}}$ does not require kernel modifications, thus satisfying G3. We discuss the most notable functionality in the H^{2}AL here.

- *Page-table management.* Functionality is included for creating, walking, allocating, and switching between page-tables. Switching between AOS page-tables not only includes rewriting the `cr3` register, but also changing the page-table pointer in the `mm_struct` for the Linux task devoted to the AOS.
- *Physical-memory management.* The AOS must receive physical memory to use for its own allocations. This memory is used to map into page-tables. Additionally, this includes the translation between physical and virtual addresses (backed by Linux's `__va` and `__pa`). If contiguous physical pages are not required, then the physical frames provided to the AOS can be allocated using `kmalloc` with the `GFP_KERNEL` flag. In this case, the H^{2}AL maintains a mapping between an AOS frame number, and actual physical page.
- *Kernel-memory allocation.* Though the AOS could use its own physical memory for its structures, in COMPOSITE these structures are so rare, that we provide functions within the H^{2}AL for kernel memory allocation and deallocation for convenience.
- *Saved register access.* Linux saves the registers of the current thread on the stack during interrupts and exceptions. The H^{2}AL provides functionality to access these registers so that the AOS can save and change them if a thread switch is required. Additionally, if an interrupt handled by the AOS occurs while in the context of a non-AOS thread, these functions will provide a reference to registers stored within the AOS's Linux thread. Thus, when that thread is dispatched to, the proper state of the AOS system is loaded. This is an essential step is decoupling the scheduling of the AOS and that of Linux.
- *Kernel stack.* The H^{2}AL provides facilities for tracking and

providing kernel stacks for the AOS. By default, the AOS always executes on the kernel stack of the Linux thread devoted to the AOS. More advanced functionality might enable the AOS to allocate its own kernel stacks, and the H^{2}AL would simply keep the current kernel stack for the AOS Linux thread consistent with the currently active AOS stack.

- *Timer interrupts.* The H^{2}AL provides facilities to retrieve information about the timer interrupts (e.g. frequency), and program the callback that is invoked each time a timer interrupt occurs. The H^{2}AL layer reprograms a Linux timer upon reception of a timer event so as to trigger the H^{2}AL every timer tick. This periodic reprogramming of the timer would have ill effects with `notick`, and there is certainly room for the H^{2}AL layer to improve.
- AOS *idle notification.* The H^{2}AL provides functionality that can be used by the AOS to notify Linux that there is no current activity. If the AOS's Linux thread is executed at the highest real-time priority, the idle notifications signify the only times that Linux processes are able to execute. AOS idling is implemented using typical Linux wait queues – when the AOS is idle it is blocked, and it is only woken up when it is receives an interrupt.
- *Hardware entry point abstraction.* Instead of directly programming the hardware to use AOS entry points for system calls, exceptions, and interrupts, the H^{2}AL provides functionality to activate the proper AOS handlers corresponding to specific hardware events. Section III contains the details about the requisite interposition and multiplexing between Linux and the AOS.

Additionally, we provide a couple of more specialized H^{2}AL features for accessing I/O of a specific type (e.g. network), and for communication with Linux processes (via the TRANSLATOR), discussed in Section V. These additions do not closely mimic hardware features and provide higher-level functionality, thus they likely require explicit integration into the AOS. These additions are closer to device drivers than they are a traditional hardware abstraction layer.

III. HARDWARE ENTRY POINT INTERPOSITION FOR OS CO-LOCATION

AOSes define their own system calls, exceptions, and interrupts. HIJACK$^{\text{COS}}_{\text{LINUX}}$ must multiplex the raw hardware events between Linux and the AOS. This function is performed in the HIJACK$^{\text{COS}}_{\text{LINUX}}$ module by reprogramming the hardware to activate HIJACK$^{\text{COS}}_{\text{LINUX}}$ logic that will jump appropriately to either Linux or the AOS in response to each event.

To provide OS co-location services via the interception of hardware entry points, we identify a number of implementation *requirements*.

R1. Code must be interposed on all target hardware entry points (including traps, interrupts, exceptions).

R2. Interposition code must determine the target OS to service the request, and forward execution to its handler.

R3. Establish a stack for kernel execution of the target OS.

R4. In the case of interrupts, complications must be resolved involving the execution of an AOS's interrupt handler while not in the context of the AOS thread. The H^{2}AL functions for saved register access are invaluable here.

The following subsections describe how these requirements are satisfied for a number of hardware entry points.

A. Case Study: System Calls

Our implementation focuses on x86-32, though many of the techniques are more broadly applicable (a x86-64 port is in progress). The primary mechanism for system calls is the `sysenter/sysexit` pair that tend to demonstrate significantly higher performance than using a system trap via `int 0x80` and `iret`. Hardware entry-point level interposition (R1) is done by simply writing the handler address to the appropriate model-specific register (`wrmsr(MSR_IA32_SYSENTER_EIP,...)`).

The system call handler must demultiplex the system call hardware event to a specific target OS (R2). Though this could be achieved easily by dispatching to the AOS system call handler when a specific flag value is set in the task structure of the current thread, or by checking if the current thread is amongst a set of threads managed by the AOS. For system calls, we decided against this approach as the overheads of retrieving the thread structure impacted a critical path for the AOS (IPC). Instead, we removed some transparency in the AOS (and weaken G1) and require the system call number (usually passed in `eax`) have a specific format. Specifically, when a user-level process makes a system call, the system call number of is offset by a bit count. Thus numerical values above this shift are AOS system calls to be dispatched to the AOS system call handler, and those lower than that are for normal Linux system calls. In both cases, a lazy check is made to ensure that AOS processes aren't trying to make Linux system calls, or that Linux processes aren't trying to make AOS system calls.

H^{2}AL uses the Linux-allocated kernel stack for the current Linux thread (for R3) for system call execution. We choose to use this stack even for the AOS as the Linux mechanisms for retrieving it (referencing the stack pointer field in the TSS on x86-32, and using segmentation on x86-64) are already low-overhead, convenient, and accessible from assembly.

After the system call is directed to the proper OS, the conventions of that OS for register and stack layout are enabled. Section II discusses how OSes additionally have control over the currently active address space through the H^{2}AL.The combination of control over the registers and active address space enable context switches between threads, and processes.

KMUX: System call demultiplexing between kernels. We have investigated a general mechanism for executing multiple pluggable "kernels" in a single "host" OS – Linux in this case. Each process' kernel invocations are handled by configurable kernels that provide sandboxing, and system call filtering. We call this system the kernel multiplexer, or KMUX[5]. We find that the overheads of such interposition for kernel compiles

is negligible [6]. KMUX provides an `ioctl` interface for controlling kernels, and mapping kernels to processes they control. In this paper, we focus on a less general mechanism for multiplexing system calls, and instead investigate the potential for effective, and predictable co-location of cooperating kernels.

B. Case Study: Page-Faults

Interposition for interrupts, exceptions, and other system traps is not significantly more difficult than for system calls. Interposition (R1) is provided by copying and modify the interrupt descriptor table, and using `sidt` to change the interrupt vector table to be the one referencing the interposition handlers. As this is a less performance-centric path that system calls, the interposition code explicitly checks if the current thread is the AOS thread, and correspondingly executes it's handlers (R2). The stack is assigned from the stack pointer in the TSS (via the semantics of x86-32), and is therefore the one that Linux has allocated for the task (R2).

However, providing $\text{HIJACK}^{\text{COS}}_{\text{LINUX}}$ interposition on the page-fault handler is significantly more difficult. These difficulties derive from a combination of practical factors in Linux: 1) Module code is paged into address spaces on demand (i.e. it is effectively `vmalloc`ed instead of `kmalloc`ed). 2) The Linux page fault handler is responsible for paging in these lazily-loaded kernel pages. If page-faults are interposed and multiplexed in the same way as system calls, the following string of events is likely: 1) the page fault handler is redefined to be the interposition code in the module, 2) an address space is switched to that does not have valid mappings in its page tables for the module, 3) a page fault occurs for any reason, 4) this activates the handler at an address in the module. This situation causes a double fault as the page fault handler is not present in the mappings of the current address space.

To enable module-defined page-fault handlers for $\text{HIJACK}^{\text{COS}}_{\text{LINUX}}$ interposition, we separate the page fault handler into two stages. The first is a trampoline that is accessible in any address space (*i.e.* in `kmalloc`ed memory), and dispatches to Linux if the fault is in the kernel. The second does the normal dispatch between co-located OS page-fault handlers and is only executed if the fault is outside of the kernel.

Page-fault trampoline. The page-fault trampoline is written in assembly, and crafted to only reference memory addresses using relative addressing within a page. This page contains pointers to both the default Linux page-fault handler, and the AOS handler. When the $\text{HIJACK}^{\text{COS}}_{\text{LINUX}}$ module is inserted, it finds the normal kernel address of the trampoline page (via translation from the page-tables). This address is accessible from within any address space, and the IDT entry is set point to it. This trampoline simply checks the x86 error code to determine if the fault occurred within kernel-level, in which case it invokes the Linux handler. Otherwise, the normal interposition code is invoked. This avoids the double fault problem described above, and enables module-defined page-fault handlers.

Page-fault multiplexing. The $\text{HIJACK}^{\text{COS}}_{\text{LINUX}}$ page-fault interposition code (post-trampoline) must decide if the fault should be vectored to the AOS, or to Linux. This is done by first checking if the current thread is the AOS thread. If so, it then checks if a virtual memory area exists for the Linux process for the fault address. If not, then the page fault is routed to the AOS. This choice has some implications: when initially creating the user-level environment, Linux memory facilities such as `mmap` can be harnessed, and virtual mappings can be shared between Linux and the AOS. However, it does imply that any such mappings should not share virtual addresses with other AOS mappings. Our AOS, COMPOSITE, is a single-address space OS [7], so this guarantee is provided by design. Other co-located systems could require explicit programming to provide this guarantee.

C. Case Study: I/O Events

We have not found a need to interpose at the hardware layer on I/O events. Linux provides an extensive corpus of device drivers and $\text{HIJACK}^{\text{COS}}_{\text{LINUX}}$ seeks to *share* these devices effectively, rather than partition the I/O devices across OSes. Specifically, we wish to enable Linux to manage the I/O devices with it's standard device drivers, and to harness the I/O devices via communication between the AOS and Linux. Here we outline how $\text{HIJACK}^{\text{COS}}_{\text{LINUX}}$ exposes I/O events to the AOS by focusing on one type of I/O: the timer. Network packet reception will be discussed in Section V.

Receiving timer interrupts in the AOS. Instead of manually reprogramming the local timer device, and interposing on its interrupts, $\text{HIJACK}^{\text{COS}}_{\text{LINUX}}$ simply uses Linux's timer abstraction layer. The timer event handler from the AOS is invoked from a timer event triggered by Linux that reprograms the timer on each timer tick to also trigger the next tick. This policy, though simple, is wasteful and disables the processor from going to sleep for long periods as would be the case with `hrtimers` and `notick`. We foresee closer integration with these features as a productive step forward. The current $\text{HIJACK}^{\text{COS}}_{\text{LINUX}}$ design does impose a latency on timer tick delivery to AOS proportional to the timer-delivery code in Linux. However, where this latency is acceptable, the AOS is able to harness the device abstractions of Linux.

It is often necessary to context switch between threads and address spaces when a timer interrupt is triggered. A high-level timer interrupt handler executes at user-level in COMPOSITE, requiring it to be dispatched when a timer occurs. Part of the H^2AL is a set of functions for saved register access for the AOS Linux task. If the current Linux thread preempted by the timer interrupt is the AOS thread, the registers are located on the stack as `pt_regs`. Otherwise, the context switch is finished by locating the registers within the AOS's task and modifying them. When the AOS thread returns to user-level, it will restore those registers, completing the AOS context switch.

Note that the current $\text{HIJACK}^{\text{COS}}_{\text{LINUX}}$ implementation assumes a single kernel-level stack for all AOS execution. This implies that AOSes are implemented in an event-driven style [8] where

preemptions are disabled. This enables execution for all user-level threads to be multiplexed onto a *single* kernel-level stack. This happens to be a good fit for HIJACK$^{\text{COS}}_{\text{LINUX}}$ where all execution for the AOS occurs in the context of a single Linux thread; all AOS user threads execute on the shared kernel stack provided by the Linux task. Though kernels implemented in this style do require non-preemptibility, it has been shown that response times can still be reasonable if the systems are simple [9], [10].

IV. COMPOSITE IN HIJACK$^{\text{COS}}_{\text{LINUX}}$

Though the focus of this paper is on effective OS co-location, here we give a background of COMPOSITE for context. The COMPOSITE component-based OS is a research operating system focused on reliability, predictability, and configurability. A functional system is composed of a set of user-level components; each component encapsulates both data and code, and provides some specific functionality, exports an interface through which that functionality can be harnessed by other components, and includes a set of dependencies on other interfaces required to provide the functionality. Each component executes in an isolated protection domain. The COMPOSITE kernel includes no policy, and instead relies on user-level components to define core system functions such as scheduling [11], memory management and mapping [12], [13], and synchronization [14]. Resource management can be arranged hierarchically [15] to provide fine-grained, expressive management policies, and heightened isolation that can provide finer-grained resource isolation properties than virtualization. Minimal systems with few components can focus on low memory usage and simplicity, while more complicated systems include web-servers with over 25 components that have performance competitive or better than conventional alternatives [16].

Optimized IPC. COMPOSITE requires an optimized IPC path for the invocation of functions in a component's interface. This operation involves two system calls and switching between two protection domains, and back. A challenge for HIJACK$^{\text{COS}}_{\text{LINUX}}$ is to maintain the efficiency and predictability [17] of this key path. The invocation path is accessed using a system call, thus our system call interposition discussed in Section III must be comparably efficient. The overheads imposed by HIJACK$^{\text{COS}}_{\text{LINUX}}$ on this key path are, first, the multiplexing at system call entry between Linux and COMPOSITE, and second, the overhead of updating the current Linux task's page-table pointer when the invocation results in a protection domain switch in COMPOSITE. The former cost is minimal as we lay out the assembly to optimize static branch prediction to choose the COMPOSITE path. The latter cost includes touching data-structures with associated costs for cache-line and TLB access. It should be noted that changing the Linux task's page table pointer is only necessary if the COMPOSITE thread is ever switched away from. Thus if COMPOSITE runs at the highest Linux priority, this cost can be avoided.

CPU Management. The COMPOSITE kernel does not include a scheduler, instead implementing policies for thread scheduling in user-level components. The kernel provides on simple means for dispatching between COMPOSITE threads. As COMPOSITE is an interrupt-driven kernel [8], the kernel does not maintain a stack or continuations [18], thus most thread state (excluding a small kernel structure including the thread's saved registers) is in user-level components, and unaffected by the HIJACK$^{\text{COS}}_{\text{LINUX}}$ co-location.

The most significant difficulty in scheduling thread execution in COMPOSITE is the scheduling of interrupt execution. As in the Linux-RT patch-set, COMPOSITE executes significant interrupt processing in dedicated, schedulable threads. Though the efficient and predictable user-level scheduling of interrupt threads is of interest [11], we focus here on the use of the HIJACK$^{\text{COS}}_{\text{LINUX}}$ support discussed in Section III-C. To switch COMPOSITE threads upon interrupt reception, the registers to be restored upon interrupt return are accessible to COMPOSITE through the H^{2}AL, and are used effect the thread switch. The support for switching page tables is also used if the interrupt thread is activated in a new protection domain.

Memory Management. Physical memory is retrieved from Linux upon insertion of the COMPOSITE module. The H^{2}AL functions are used to manipulate page-tables through a very simple COMPOSITE system call interface that enables a specific physical frame to be mapped at a specific (virtual address, page-table) pair. Specific components are given restricted access to physical frames, and provide higher-level memory mapping operations through their interface [15]. Page faults are routed to COMPOSITE by the hardware interposition layer, and COMPOSITE converts these faults into invocations to components equipped with logic to appropriately handle the faults.

A. *Booting* COMPOSITE

COMPOSITE components must be linked and loaded manually into their corresponding locations in the virtual address space. We have written a linker that will take all components and a ramdisk that we wish to load into COMPOSITE, and using the `bfd` library, it will link the components and generate their memory images accordingly. The thread and Linux protection domain for this linker process are used for the COMPOSITE system as a whole. Thus the linker adjusts its priority if we wish to execute COMPOSITE with the highest priority in the system, scheduled with `SCHED_RR`. The linker also uses the `rlimit` API to ensure that we can use 100% of the CPU. While executing at the highest priority, with access to all CPU bandwidth, the co-located OS effectively has control of the processor with the exception of interrupts (that can be interposed on).

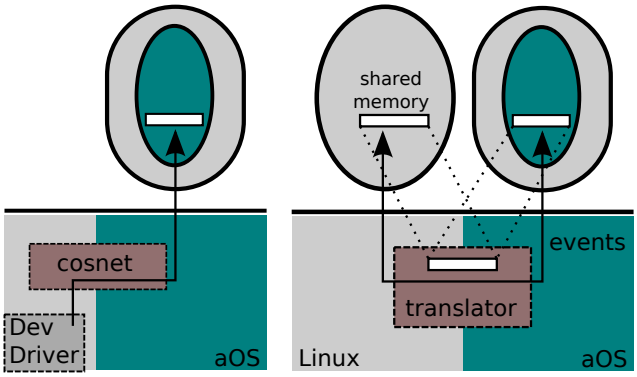
Once this is complete, a `loader` component will be loaded into Linux-allocated memory, and all component images are copied into its memory-space. This `loader` is executed immediately when COMPOSITE begins execution, and it creates the component protection domains (using COMPOSITE system calls), and loads the rest of the components.

Currently, this entire process requires `root` permissions. Though it is likely possible to avoid requiring `root` permis-

sions, the interposition module's API would require a thorough security audit.

V. Linux/Composite Coordination

Composite requires access to I/O to provide useful functionality. As Linux is used as the host system, it has access to all I/O devices. We wish to avoid a strict partitioning of the I/O devices between the two OSes, and instead provide multiple mechanisms for device access by both the aOS and Linux. These range from a specific technique that extends the H^2AL abstraction for a specific I/O device that gets native performance, to a generic mechanism for accessing I/O through specialized Linux processes that can be used to access any Linux I/O with the additional cost of IPC between this process and the aOS. Both of these mechanisms require that specialized components be provided in the aOS to interface with the I/O.



(a) (b)

Fig. 2. (a) Using the `cosnet` pseudo-device to access the network. (b) Communication between the aOS and Linux via the translator that provides shared memory, and event channels.

Figure 2 depicts the two styles of interaction between the aOS and Linux. In the first, a pseudo-device driver is inserted into the kernel (here `cosnet`) that interacts closely with the device driver to vector events to the aOS. In the second, a general module for communication between a Linux process and the aOS called the Translator is provided. The aOS accesses any Linux services through the proxy Linux process.

A. Linux Interface Extension: Access to Devices

Network packet reception in the aOS. Networking drivers can be quite complicated, and wireless drivers even more so. We investigate multiple approaches for packet reception. The option with the least overhead is to mimic the approach of the timer, and simply harness low-level Linux APIs to access the device drivers. We note that `tun` devices satisfy this requirement as they bypass all transport-layer processing, yet still permit differentiation via IP addresses between packets that arrive for Linux, and those that arrive for the aOS. We modify a `tun` device to provide `cosnet`. This pseudo-device interfaces with the Linux kernel identically to a `tun` device (with a different major number), but provides an interface to an aOS closer to that of a device driver. Namely, received packets are written into a ring buffer shared with a user-level virtual NIC, and the aOS network reception handler is called. As with the timer interrupt, this might cause task switches within the aOS that are controlled with direct access to the registers that will be activated upon return to user-level.

When sending packets, the aOS user-level virtual NIC again writes packets to transmit into a ring buffer, and makes a system call that uses the H^2AL to transmit the packet via `cosnet`.

Though this technique requires a Linux module per I/O type, and an addition to Composite to interface with it, this technique is able to achieve native delivery latencies, providing reasonable response-time bounds. We still rely on at least the low-level Linux processing of the device driver, thus any advances made in lowering bounds on response times in Linux-RT are necessary the aOS as well.

B. The Translator: Linux Process and Composite Communication

The previous technique for interfacing Composite with I/O relies on specialized code both in a Linux module for the I/O type, and in Composite. $\text{Hijack}^{\text{cos}}_{\text{linux}}$ provides a Translator framework for direct communication between a Linux process, and a Translator component in Composite. The translator process and component can send events to each other that designate that data has been added to a shared memory region shared by both. Figure 2 depicts the Translator and how it arbitrates communication between the co-located OSes.

Translator Linux Process. A $\text{Hijack}^{\text{cos}}_{\text{linux}}$ Linux module provides the `/dev/translator` device. A process can `read` and `write` to the device which will block waiting for an event, and send an event, respectively. The process `mmaps` the device to create a shared region of memory that is used to transfer data between the OSes. This is managed as a wait-free ring buffer. In many situations it is reasonable for the Linux process to have a high static priority (higher than the Composite thread) as this process' response time impacts that of any components in Composite attempting I/O.

Translator Composite component. Composite is extended to include a component that interfaces with custom system calls to both map the shared memory region previous created by the Linux process, and to send and receive events. A system call is provided to send events to wake a blocked Linux process. Events are received in Composite by triggering a virtual interrupt when the Linux process writes to the Translator file. This interrupt activates a Composite interrupt thread to handle the event. This thread can then read data out of the shared memory ring-buffer.

In this paper, we focus on using the Translatorto interface with I/O. However, it can be used for general communication between Linux processes and Composite to enable a system to be split between two operating systems, yet have them cooperate even at the application level.

VI. EVALUATION

All results in this section are run on an Intel i7-2600 CPU clocked at 3.4 Ghz, and statistics are reported over 1000 measurements. Unless otherwise specified, the threads running the tests execute at the highest real-time priority. We use a vanilla Linux version 2.6.33. Though we don't apply the real-time patches, we only compare to naive Linux latency. If this decreased, we'd expect a comparable decrease in HIJACK$^{\text{COS}}_{\text{LINUX}}$ overheads.

A. HIJACK$^{\text{COS}}_{\text{LINUX}}$ *Overhead*

Method	Latency
COMPOSITE: invocation	0.24 (0.00)
COMPOSITE: invocation w/o host pgtbl update	0.23 (0.00)
Hardware overhead: 2 syscall	0.08 (0.00)
Hardware overhead: 2 pgtbl switch	0.08 (0.00)
Hardware overhead: percent of invocation	69.85%
Linux: Pipe RPC	1.68 (0.68)
HIJACK$^{\text{COS}}_{\text{LINUX}}$: Kernel Compile	5m52.25s
Vanilla: Kernel Compile	5m51.96s

TABLE I
LATENCY (AVG(STDDEV) IN μSECS), UNLESS OTHERWISE NOTED.

HIJACK$^{\text{COS}}_{\text{LINUX}}$ imposes some overheads due to the multiplexing of hardware entry points, and due to the H^{2}AL page-table switching operations that update the Linux task's page-table pointer. The invocation (IPC) path in COMPOSITE is optimized and its performance and predictability is fundamental to the system. Table I shows the costs of an invocations (switching to another component, executing a null function at user-level, and returning to the previous component), the hardware costs of two system calls, and two page-table switches. The cost that software execution imposes on the invocation path in is 30% of the total execution time, thus most time is spent on hardware-limited operations. We also show the invocation cost without the H^{2}AL overhead of updating the Linux AOS thread's page-table pointer. This cost could be removed if the AOS always executed at highest Linux priority. This cost adds 26 cycles to an invocation which we believe is an acceptable overhead. For context, Table I shows the cost of an RPC between threads via pipes. A direct comparison should not be made as Linux is not designed around optimized IPC. Importantly, we note that the cost of the invocation is close to the raw hardware costs (including two system calls, and two page-table switches).

To evaluate the costs of HIJACK$^{\text{COS}}_{\text{LINUX}}$ on normal Linux execution, we do a kernel compile with and without the hardware interposition support, and find that the overhead is insignificant ($<$ 1 second).

B. Translator Performance

The TRANSLATOR facilitates communication between Linux and COMPOSITE. The event notification it provides on both ends bounds the response time for access to I/O and on any collaboration between OSes. Table II shows the latencies between when a message is sent from user-level in one OS, and received in user-level in the other. We compare versus the one-way latency for communication between two threads in the same process over a pipe.

Method	Latency
Pipe	0.90 (0.64)
Translator: cos$\rightarrow$ Process	0.51 (0.08)
Translator: Process $\rightarrow$ cos	0.73 (0.46)

TABLE II
IPC LATENCY (AVG(STDDEV) IN μSECS)

These results demonstrate that the TRANSLATOR does not impose significant overheads, and in fact that the TRANSLATOR module decreases latency in both cases. This demonstrates the possibility of having effective communication between AOS and host (thus satisfying G4). We believe this is due to the fact that there is a very simple interface for event delivery in COMPOSITE as opposed to the general VFS interface that the events for pipes use. We interpret these results as showing the promise of the TRANSLATOR for co-located OS communication.

C. COMPOSITE *I/O Response Time*

The response time of a system to an I/O event is an important factor in assessing if it can be used in a specific real-time environment. Here we assess the response time overheads of the different means for COMPOSITE to interface with I/O. We measure the latency between when an interrupt is triggered for an incoming UDP network packet, and when it is delivered for processing. We compare four different methods:

- A Linux process receiving the packets using sockets.
- A Linux process receiving packets via a TUN device.
- COMPOSITE using `cosnet` for direct reception.
- COMPOSITE receiving packets via the TRANSLATOR and a Linux process that in turn accesses the network via a TUN device.

To measure the latency between interrupt reception and packet delivery for processing, use the cycle-granularity time stamp counter (via `rdtsc`). All background tasks are run within the time-sharing priority band, and a medium priority, real-time task spins in a tight loop taking time stamp readings and writing them to a volatile variable shared with a higher priority thread that receives the packets. The high-priority thread blocks waiting for packets, and when awakened, it takes a time stamp reading, and compares that value to the medium priority thread's counter to retrieve the latency between when the interrupt preempted the medium priority thread, and delivered the packet to the high-priority thread. In the COMPOSITE systems, the high-priority thread is an interrupt thread. 16 byte UDP packets are sent at a low rate to the system under test, and we observe no nested interrupts. We report results both for normal sockets and for TUN in Linux. We report both as `cosnet` is more similar to TUN, while normal sockets are more widely used.

Table III shows the average and standard deviation of the response times for each method. These results show that the

Method	Latency
Linux: UDP	10.30 (0.50)
Linux: TUN	10.94 (5.13)
HIJACK$^{\text{COS}}_{\text{LINUX}}$: `cosnet`	9.69 (0.36)
HIJACK$^{\text{COS}}_{\text{LINUX}}$: translator	11.4 (0.59)

TABLE III
RESPONSE TIMES (AVG(STDDEV) IN μSECS)

latency for delivery to the AOS is within the same magnitude as Linux user-level access response times. Even using the TRANSLATOR, the kernel-level processing overheads of handling the interrupts, `softirq` execution, and other kernel operations overwhelms the translator overhead. We believe this shows that it is feasible to utilize the Linux support for a large number of device drivers, and architectures, without debilitating overheads or latencies, thus satisfying G5.

Note that oddly the TUN latency is larger than that for a process using sockets. This situation is reversed using the default Ubuntu 10.04 LTS, 2.6.32 kernel. We have not identified the reason, but note that they are both in the same overhead range as the HIJACK$^{\text{COS}}_{\text{LINUX}}$ methods.

VII. RELATED WORK

OS co-location for Real-Time. The most notable real-time OS co-location techniques are RTAI [3], and RTLinux [2]. Both attempt to provide an operating environment *below* Linux for hard real-time execution. In doing so, they require that devices be partition between the systems, and that only limited communication occur between the systems. We take a different approach by recognizing that Linux with the real-time patches [4] has acceptable response time for many application domains, and that we want to utilize the assets of Linux in a novel way.

OS virtualization. In contrast to virtualization and paravirtualization techniques such as Xen [19], L4Linux [20], Nova [21], and KVM [22], the entire guest operating system does not execute outside of the kernel in HIJACK$^{\text{COS}}_{\text{LINUX}}$. This simplifies the system, and enables native response times and efficiency, along with simplified collaboration with Linux. Virtualization has additionally been used to provide device-driver reuse[23], a goal shared with HIJACK$^{\text{COS}}_{\text{LINUX}}$, though the approach is quite different. The closest technique to this work provides a virtualization environment on commodity systems while also requiring only a module. I/O is conducted via helper processes [24]. This work does not focus on virtualization, and instead providing a kernel-level execution environment for AOSes.

VIII. CONCLUSIONS AND FUTURE WORK

We believe this is a significant potential to harness the quality of the Linux code base, its significant support for architectures and device drivers, and its advances in real-time performance by providing co-location facilities for the execution of specialized execution environments, including other OSes. The paper makes the argument that such techniques are technically feasible, and that more research is required in the area.

Future work involves considering the many factors we have simplified in this design. Certainly the co-management of system-wide CPU state is difficult. This includes power-state management and processor frequency. In the current prototype, we simply let Linux manage all power-related functionality for us. Whereas a normal OS would could go into an idle state when there are no tasks to execute, we switch out of the COMPOSITE process, and allow other Linux threads to run. Additionally, we are actively investigating how multiple cores can be used in HIJACK$^{\text{COS}}_{\text{LINUX}}$.

Links to the git repository, and documentation for the system can be found on the COMPOSITE webpage at composite.seas.gwu.edu.

REFERENCES

[1] "The COMPOSITE component-based system: http://composite.seas.gwu.edu."
[2] "Real-Time Linux: http://www.rtlinuxfree.com/."
[3] "The Real Time Application Interface: https://www.rtai.org/."
[4] "The real-time Linux patches: https://rt.wiki.kernel.org, retrieved 9/16/12."
[5] "KMUX: https://github.com/tarequeh/kmux, retrieved 9/21/12."
[6] T. Hossain, "Kmux: Kernel extension at the hardware interface," Master's thesis, The George Washington University, Washington, DC, USA, 2011.
[7] J. S. Chase, M. Baker-Harvey, H. M. Levy, and E. D. Lazowska, "Opal: A single address space system for 64-bit architectures," *Operating Systems Review*, vol. 26, no. 2, p. 9, 1992. [Online]. Available: citeseer.ist.psu.edu/58003.html
[8] B. Ford, M. Hibler, J. Lepreau, R. McGrath, and P. Tullmann, "Interface and execution models in the fluke kernel," in *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*. Berkeley, CA, USA: USENIX Association, 1999, pp. 101–115.
[9] B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, and G. Heiser, "Timing analysis of a protected operating system kernel," in *Proceedings of the 32nd IEEE Real-Time Systems Symposium*, Vienna, Austria, Nov 2011.
[10] B. Blackham, Y. Shi, and G. Heiser, "Improving interrupt response time in a verifiable protected microkernel," in *Proceedings of the 7th ACM european conference on Computer Systems*, ser. EuroSys '12, 2012.
[11] G. Parmer and R. West, "Predictable interrupt management and scheduling in the Composite component-based system," in *RTSS '08: Proceedings of the 29th IEEE International Real-Time Systems Symposium*. IEEE Computer Society, 2008.
[12] Q. Wang, J. Song, and G. Parmer, "Stack management for hard real-time computation in a component-based os," in *Proceedings of the 32nd IEEE Real-Time Systems Symposium (RTSS)*, ser. RTSS 11, Nov 2011.
[13] Q. Wang, J. Song, G. Parmer, G. Venkataramani, and A. Sweeney, "Increasing memory utilization with transient memory scheduling," in *Proceedings of the 33rd IEEE Real-Time Systems Symposium (RTSS)*, 2012.
[14] G. Parmer and J. Song, "Customizable and predictable synchronization in a component-based os," in *Proceedings of the International Conference on Embedded Systems and Applications (ESA)*, 2010.
[15] G. Parmer and R. West, "Hires: A system for predictable hierarchical resource management," in *Proceedings of the 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2011, pp. 180–190.
[16] ——, "Mutable protection domains: Adapting system fault isolation for reliability and efficiency," in *ACM Transactions on Software Engineering (TSE)*, July/August 2012.
[17] G. Parmer, "The case for thread migration: Predictable ipc in a customizable and reliable os," in *Proceedings of the Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT '10)*, 2010.

[18] R. P. Draves, B. N. Bershad, R. F. Rashid, and R. W. Dean, "Using continuations to implement thread management and communication in operating systems," in *Proceedings of the13th ACM Symposium on Operating Systems Principle*. Association for Computing Machinery SIGOPS, 1991, pp. 122–136. [Online]. Available: citeseer.ist.psu.edu/draves91using.html

[19] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer, "Xen and the art of virtualization," in *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2003. [Online]. Available: citeseer.ist.psu.edu/dragovic03xen.html

[20] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter, "The performance of μ-kernel-based systems," in *Proceedings of the Sixteenth Symposium on Operating Systems Principles*. ACM, October 1997.

[21] U. Steinberg and B. Kauer, "Nova: a microhypervisor-based secure virtualization architecture," in *Proceedings of the 5th European conference on Computer systems*, ser. EuroSys '10, 2010, pp. 209–222.

[22] "Linux KVM: http://www.linux-kvm.org, retrieved 9/16/12."

[23] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz, "Unmodified device driver reuse and improved system dependability via virtual machines," in *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*. Berkeley, CA, USA: USENIX Association, 2004, pp. 2–2.

[24] J. Sugerman, G. Venkitachalam, and B.-H. Lim, "Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor," in *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2001, pp. 1–14.