Hijack: Taking Control of COTS Systems to Enforce Predictable Service Guarantees

Richard West and Gabriel Parmer

Boston University
Boston, MA
{richwest,gabep1}@cs.bu.edu

Computer Science

Introduction

- Leverage commodity systems and generic hardware for QoS-constrained applications
  - Eliminate cost of proprietary systems & custom hardware
  - Use a common code base for diverse application requirements
    - e.g., use existing device drivers
  - ***BUT****…mismatch exists between QoS requirements of applications and the service provisions of commodity OSes*

Bridging the `Semantic Gap'

- There is a `semantic gap' between the needs of applications and services provided by the system
- Implementing functionality directly in application processes
  - **Pros:** service/resource isolation (e.g., memory protection)
  - **Cons:**
    - Does not guarantee necessary responsiveness
    - Must leverage system abstractions in complex ways
    - Heavyweight scheduling, context-switching and IPC overheads

Bridging the `Semantic Gap' Cont.

- Other approaches:
  - Special systems designed for extensibility
    - e.g., SPIN, VINO, Exo-/μ-kernels (Aegis / L4), Palladium
    - Do not leverage commodity OSes
    - Do not explicitly consider QoS requirements
      - e.g. bounded dispatch latencies and execution
  - Virtual machines
    - Have each VM provide system services for specific class of applications
    - BUT *hosted* VMs at mercy of unpredictable services of underlying host kernel
    - Here, we want to leverage underlying COTS system rather than replace it where possible!

Extending Commodity Systems

- Desktop systems now support QoS-constrained applications
  - e.g., Windows Media Player, RealNetworks Real Player
- Many such systems are monolithic and not easily extended or only support limited extensibility
  - e.g., kernel modules for device drivers in Linux
  - No support for extensions to override system-wide service policies

Objectives

- Aim to **extend** commodity systems to:
  - better meet the service needs of individual applications
  - provide first-class application-specific services
- Service extensions must be `QoS safe':
  - Need CPU-, memory- and I/O-space protection to ensure
    - Service isolation
    - Predictable and efficient service dispatching
    - Bounded execution of services

First-class Services

- Where possible, have same capabilities as kernel services but kernel can still revoke access rights
  - Grant access rights to subset of I/O-, memory-space etc
  - Dispatch latencies close to those of kernel-level interrupt handlers
  - Avoid potentially unbounded scheduling delays
    - Bypass kernel scheduling policies
    - Eliminate process context-switching
      - Eliminate expensive TLB flushes/reloads

First-class Services cont.

- Process, P_i, may register a service that runs even when P_i is not executing
  - Like a fast signal handling mechanism
  - Example usages:
    - Asynchronous I/O
    - Resource monitoring / management
      - e.g., P_i wishes to adjust its CPU usage even when not running perhaps because it wasn't getting enough CPU!

Contributions

- Comparison of kernel- and user-level extension technologies
  - "User-level sandboxing" (ULS) versus our prior SafeX work
  - Show how to achieve low service dispatch latency for app-specific services, while ensuring some degree of CPU-, I/O and memory protection
- Hijack
  - Next-generation ULS technique including interposition
  - Ability to intercept system calls and h/w interrupts for delivery to sandbox
    - Can predictably and completely control "guest" application execution

SafeX – Safe Kernel Extensions

- Extension architecture for general purpose systems
  - Allows applications to customize system behavior
  - Extensions run in context of a kernel "bottom half"
    - Enables low-latency execution in response to events & eliminates heavyweight process scheduling

SafeX Approach

- Supports compile- and run-time safety checks to:
  - Guarantee QoS
    - The QoS contract requirement
  - Enforce timely & bounded execution of extensions
    - The predictability requirement
  - Guarantee an extension does not improve QoS for one application at the cost of another
    - The isolation requirement
  - Guarantee internal state of the system is not jeopardized
    - The integrity requirement

SafeX Features

- Extensions written in Popcorn & compiled into Typed Assembly Language (TAL)
  - TAL adds typing annotations / rules to assembly code
- Memory protection:
  - Prevents forging (casting) pointers to arbitrary addresses
  - Prevents de-allocation of memory until safe
- CPU protection:
  - Requires resource reservation for extensions
  - Aborts extensions exceeding reservations
  - SafeX decrements a counter at each timer interrupt to enforce extension time limits

Synchronization

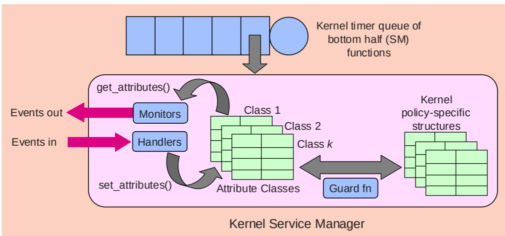
- Extensions cannot mask interrupts
  - Could violate CPU protection since expiration counter cannot decrement
- Problems aborting an extension holding locks
  - e.g., extension runs too long
  - May leave resources inaccessible or in wrong state

  - Extensions access shared resources via SafeX interfaces that ensure mutual exclusion

SafeX Kernel Service Managers

- Encapsulations of resource management subsystems
- Have policies for providing service of a specific type
  - e.g., a CPU service manager has policies for CPU scheduling and synchronization
- Run as bottom-half handlers (in Linux)
  - Invoked periodically or in response to events within system
- Invoke monitor and handler extensions
  - Can execute asynchronously to application processes
  - Apps may influence resource allocations even when not running

SafeX Kernel Service Managers (Cont.)



- Monitors & handlers operate on attribute classes
  - name-value pairs (e.g. process priority – value)
- Service extensions with valid access rights can modify attributes

Attribute Classes & Guards

- Attribute classes store name-value pairs for various app-specific service attributes
  - e.g., priority-value for CPU scheduling

  - Access to these classes is granted to the extensions of processes that acquire permission from the class creators

- Guard functions are generated by SafeX
  - Responsible for mapping values in attribute classes to kernel data structures
  - Can enforce range and QoS guarantee checks

SafeX Interfaces

- SafeX provides get_/set_attribute () interfaces
  - Extensions use these interfaces to update service attributes
  - Extensions are not allowed to directly access kernel data structures

- Interfaces can only be used by extensions having necessary capabilities
  - Capabilities are type-safe (unforgeable) pointers

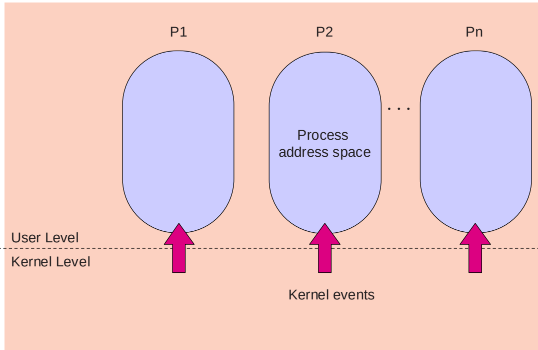
- Interfaces limit global affects of extensions
  - Balance application control over resources with system stability

User-Level Sandboxing (ULS)

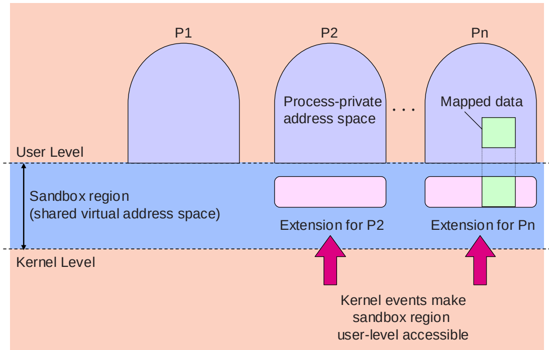
- Provide "safe" environment for service extensions
- Separate kernel from app-specific code
- Use only page-level hardware protection
  - Can use type-safe languages e.g., Cyclone for memory safety of extensions, SFI etc., or require authorization by trusted source

- Approach does not require (but may benefit from) special hardware protection features
  - Segmentation
  - Tagged TLBs

Traditional View of Processes



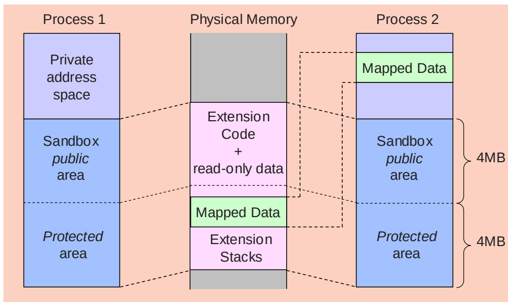
Sandbox Region Shared by Processes



ULS Implementation

- Modify address spaces of all processes to contain one or more shared pages of virtual addresses
  - Shared pages used for sandbox
    - Normally inaccessible at user-level
    - Kernel upcalls toggle sandbox page protection bits & perform TLB invalidate on corresponding page(s)
- Current x86 approach
  - 2x4MB superpages (one data, one code)
  - Modified libc to support mmap, brk, shmget etc
  - ELF loader to map code & data into sandbox
  - Supports sandboxed threads that can block on syscalls

Virtual-to-Physical Memory Mapping



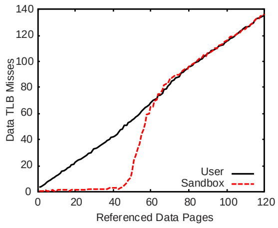
ULS Implementation (2)

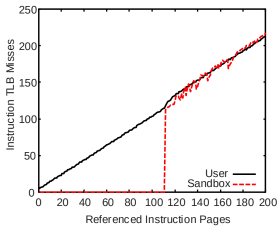
- Fast Upcalls
  - Leverage SYSEXIT/SYSENTER on x86
    - Support traditional IRET approach also
- Kernel Events
  - Generic interface supports delivery of events to specific extensions
  - Each extension has its own stack & thread struct
    - Extensions share credentials (including fds) with creator
  - Events can be queued ala POSIX.4 signals

Experimental Evaluation

- (a) Inter-Protection Domain Communication
  - Look at overheads of IPC between thread pairs
    - Exchange 4-byte messages
    - Vary the working set of one thread to assess costs
    - 1.4GHz P4, patched Linux 2.4.9 kernel
- (b) Adaptive CPU service management
  - Aim: to meet the needs of CPU-bound RT tasks under changing resource demands from a `disturbance' process
  - Compare ULS and SafeX to process-based approaches
  - 550 Mhz Pentium III, 256MB RAM, patched 2.4.20 Linux

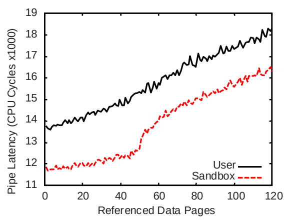
Data and Instruction TLB Misses

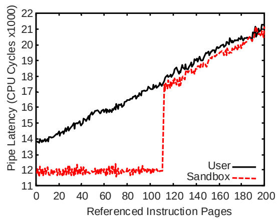




- Inter-protection domain communication costs
- Costs of 4-byte messages between two threads using pipes
- Vary working set of one process-private thread while other is in sandbox

Pipe Latency





- Pipe latency remains lower for RPC with sandboxed thread
  - Even when data TLB miss rates are similar
- NOTE: d-TLB sizes simulated by thread reading 4 bytes of data from addresses spaced 4160 bytes apart. i-TLB sizes simulated using relative jumps to instructions 4160 bytes apart.

System Service Extensions

- Can we implement system services in the sandbox?
- Here, we show performance of a CPU service manager (CPU SM)
  - Attempt to maintain CPU shares amongst real-time processes on target in presence of background disturbance
  - Use a MMPP disturbance w/ avg inter-burst times of 10s and avg burst lengths of 3 seconds

Kernel Service Management

- A service manager monitors CPU utilization and adapts process timeslices
  - Timeslices adjusted by PID function of target & actual CPU usage
  - Monitoring performed every 10mS
- Kernel monitoring functions invoked via timer queue

User-Level Management

- A periodic RT process acts as a CPU service manager
  - Reads /proc/pid/stat
  - Adapts service via kill() syscalls
    - Using SIGSTOP & SIGCONT signals

Experimental Setup

- 3 CPU-bound processes, P1, P2 & P3
  - P1 – target CPU = 40mS every period = 400mS
  - P2 – target CPU = 100mS every 500mS
  - P3 – target CPU = 60mS every 200mS
- An MMPP disturbance (CPU "hog")
  - 10 sec exponential inter-burst gap & 3 sec geometric burst lengths

Experimental Setup cont.

- Each app process has initial RT priority = 80 x (target / period)
  - target & period denote target CPU time in a given period
- User-level service manager & disturbance start at RT priority = 96
- Kernel daemons run at RT priority = 97
- Utilization points recorded over 1 sec intervals

Monitors and Handlers

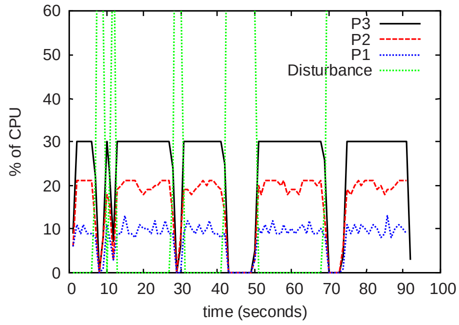
```
void monitor () {
  actual_cpu = get_attribute ("actual_cpu");
  target_cpu = get_attribute ("target_cpu");
  raise_event ("Error", target_cpu - actual_cpu);
}

void handler () {
  e[n] = ev.value; // nth sampled error

  /* Update timeslice adjustment by PID fn of error */
  u[n] = (Kp+Kd+Ki).e[n] - Kd.e[n-1] + u[n-1];

  set_attribute ("timeslice-adjustment", u[n]);
}
```

Guard Functions

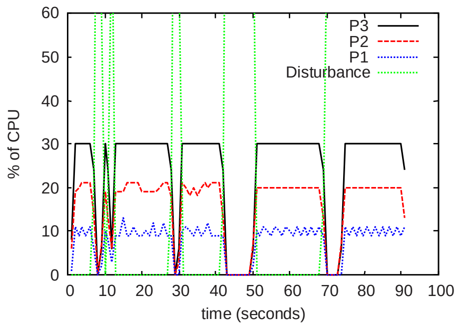
```
// Check the QoS safe updates to a process' timeslice

guard (attribute, value):
  if (attribute == "timeslice-adjustment")
    if (CPU utilization is QoS safe)
      timeslice = max (0, target_cpu + value);
    else block process;
```

• CPU utilization is deemed QoS safe if:
  Avg utilization over 2*period <= target utilization

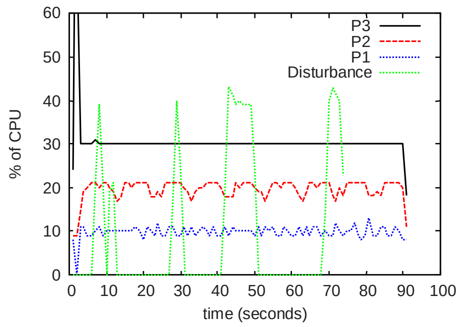
CPU SM: User-level Process



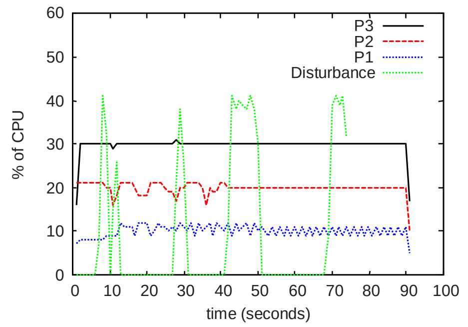
CPU SM: Sandbox Thread



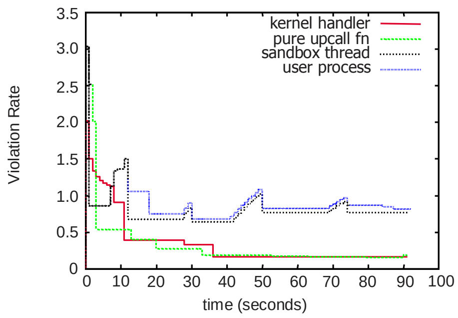
CPU SM: Pure Upcall



CPU SM: Kernel



Deadline Violation Rates



SafeX Benchmarks

- User-level:
  - Signal dispatch = 1.5µS
  - Context-switch between SM and app process = 2.99µS
  - Reading /proc/pid/stat = 53.87µS
  - Monitors and handlers (for 3 processes) = 190µS
- Kernel-level:
  - Executing monitors and handlers (for 3 processes) = 20µS

ULS Benchmarks

Operation	Cost in CPU Cycles
Upcall including TLB flush / reload	11000
TLB flush and reload *includes call to OpenSandbox()	8500
Raw upcall	2500
Signal delivery (current process)	6000
Signal delivery (different process)	46000

Hijack: Predictable Control of COTS Systems

- Provides mechanisms to redefine or hijack all COTS system policies concerning
  - Process execution
  - System service requests (system calls)
- Methodologies:
  - Create ULS-type memory region in address space of all hijacked processes
  - Interpose this layer on all hijacked process system calls
  - Allow the control of process execution (register state) and execution context (address space)

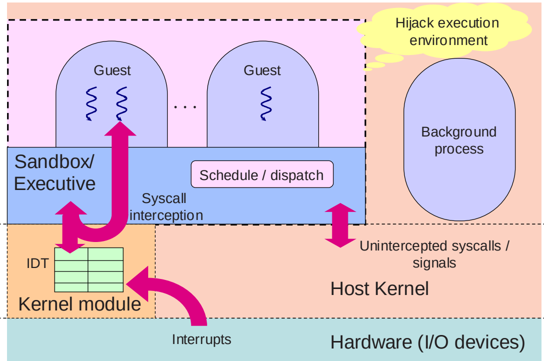
Hijack: Predictable Control of COTS Systems (continued)

- Like VMM, but interposes on the system call layer instead of the architectural
  - Can interpose on architectural level too!
  - Note: The Hijack approach was originally influenced by User-mode Linux (UML) that uses ptrace to interpose on syscalls
- Avoid changes to underlying host kernel
- Terminology:
  - ULS-type region defining hijack policies: Executive
  - Hijacked processes: Guests

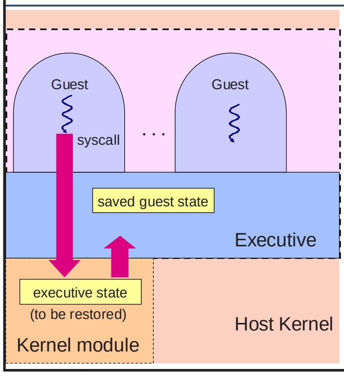
Hijack: Predictable Control of COTS Systems (continued)

- Use kernel loadable module to intercept syscalls & h/w interrupts
  - Intercepts trigger upcalls to executive (similar to ULS)
- Hijack is only a single kernel-thread to the host system with highest priority
  - Support multiple guest threads by multiplexing reg. state
- Can predictably & efficiently receive notification of host system events
  - e.g., SIGALRM signal generated by a timer interrupt in host kernel, for delivery to sandbox scheduler

Hijack System Architecture

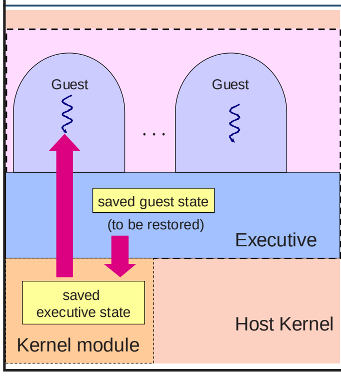


System Call Interposition



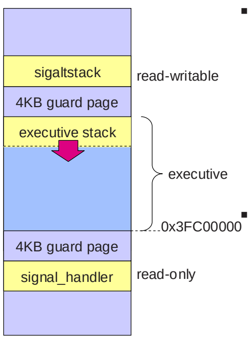
- Guest system calls are vectored to the executive
- Guest register state is stored at executive-defined location
  - Can alter register state
- Executive can make normal system calls in response to guest service request
- OR define its own policy for service

Control Flow from Executive to Guest



- Executive can resume guest execution
  - In any hijack address space
  - With executive-defined register state
- Executive controls
  - Scheduling
  - Entire execution environment of guest!
- Executive register saved in module when in guest

Hijack Virtual Address Space Layout

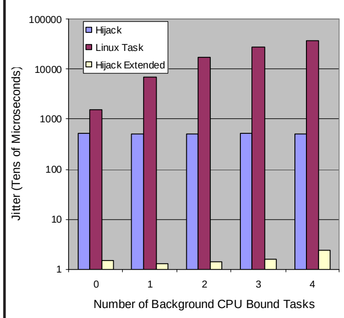


- Predictable host OS event notification (signals)
  - Must receive events promptly when executing in both guest and executive
  - Event handlers in executive BUT executive region inaccessible while in guest
- Define trampoline code (signal_handler) to receive signals
  - Open executive region if inaccessible
  - Execute executive handler

Hijack System Performance

Operation	Cost in CPU Cycles
System Call	430
RPC from Guest to Executive to Guest	4482
Interposition: RPC + System Call	5094
Interposition using POSIX ptrace	33613
IPC from Guest to Executive	1925
IPC from Executive to Guest	2563
RPC between two guests (separate page tables)	13476
RPC between two tasks using UNIX pipes	18661

Nanosleep Predictability

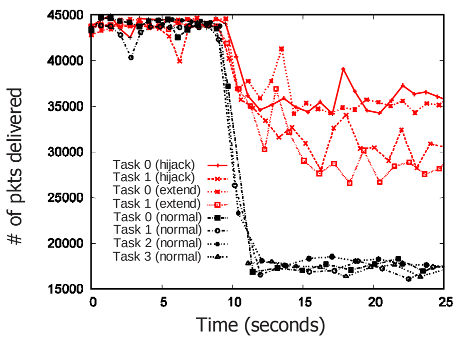


- nanosleep system call
  - Typically has minimum latency of system clock tick
  - Waking a sleeping process involves scheduling
- Unpredictable with multiple tasks in run-queue
- Perhaps appropriate for nanosleep provider to spin for sleep periods less than a clock-tick
  - Not a general solution

QoS Expts: Packet Delivery

- Demonstrate the definition of complex policies within executive
  - QoS for different tasks in terms of I/O capabilities
- Up to 4 streams of data sent to tasks
  - Small UDP packets
  - 44000 packets/second per stream
- Tasks “process” data by computing statistics on dropped packets and stream delivery jitter
  - Tasks output stats every 30000 packets processed
- Tasks with QoS requirements (pseudo-proportional share):
  - Task0: highest QoS
  - Task1: intermediate QoS
  - Task2/Task3: Best effort

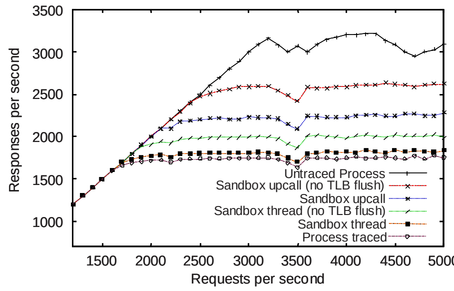
QoS Expts: Packet Delivery (cont)



Interposition Experiments

- Interposition
  - Simple syscall tracing extensions based on ptrace
  - Compare traditional ptrace implementation against:
    - Upcall handler implementation in sandbox
    - Kernel-scheduled thread in sandbox

Interposition Agents: ptrace of system calls



- Experiments on a 1.4GHz Pentium 4 w/ patched Linux 2.4.9
- Ptraced thttpd web server under range of HTTP request loads

Conclusions

- SafeX and ULS both capable of supporting app-specific service invocation without process scheduling / context-switching overheads
  - Avoid TLB flush/reload costs
  - Lower-latency, more predictable service dispatching
  - Both provide finer-grained service management than process-based approaches
    - No scheduling of processes for service management
    - Not dependent on scheduling policies and timeslice granularities
- Hijack is next step to full control of COTS system for predictable (QoS-based) services

Future Work

- Real-time upcall mechanism for deferrable services
- Better interrupt accounting and "bottom half" scheduling
- Support for complex virtual services
  - Use Hijack executive to control resource management amongst multiple hosted virtual machines
- In earlier work we showed how to use ULS to support user-space network protocol stacks, avoiding data-copying via host kernel
  - Could extend to multiple coordinated services across network of ULS/Hijack-controlled hosts
- Comparison with RTAI, RTLinux and similar approaches

Further Information

- www.cs.bu.edu/fac/richwest/sandboxing.html
- www.cs.bu.edu/fac/richwest/safex.html

- Richard West and Gabriel Parmer, *"Application-Specific Service Technologies for Commodity Operating Systems in Real-Time Environments,"* RTAS 2006
  - Extended version to appear in ACM Transactions on Embedded Computing Systems
- Richard West and Jason Gloudon, *"`QoS Safe' Kernel Extensions for Real-Time Resource Management,"* ECRTS 2002
- Xin Qi, Gabriel Parmer and Richard West, *"An Efficient End-host Architecture for Cluster Communication Services,"* Cluster Computing 2004
- Gabriel Parmer and Richard West, *"Hijack: Taking Control of COTS Systems for Real-Time User-Level Services,"* BU Technical Report (under review)
- Yuting Zhang and Richard West, *"Process-Aware Interrupt Scheduling and Accounting,"* BU Technical Report (under review)