Predictable Communication and Migration in the Quest-V Separation Kernel

Ye Li, Richard West, Zhuoqun Cheng and Eric Missimer

Computer Science Department
Boston University
Boston, MA 02215
Email: {liye,richwest,czq,missimer}@cs.bu.edu

***Abstract*—Quest-V is a separation kernel, which partitions a system into a collection of sandboxes. Each sandbox encapsulates one or more processing cores, a region of machine physical memory, and a subset of I/O devices. Quest-V behaves like a distributed system on a chip, using explicit communication channels to exchange data and migrate addresses spaces between sandboxes, which operate like traditional hosts. This design has benefits in safety-critical systems, which require continued availability in the presence of failures. Additionally, online faults can be recovered without rebooting an entire system. However, the programming model for such a system is more complicated. Each sandbox has its own local scheduler, and threads must communicate using message passing with those in remote sandboxes. Similarly, address spaces may need to be migrated between sandboxes, to ensure newly forked processes do not violate the feasibility of existing local task schedules. Migration may also be needed to move a thread closer to its required resources, such as I/O devices that are not directly available in the local sandbox. This paper describes how Quest-V performs real-time communication and migration without violating service guarantees for existing threads.**

***Keywords*—separation kernel; virtualization; real-time communication; process migration;**

I. Introduction

Multi- and many-core processors are becoming increasingly popular in real-time and embedded systems. As the number of cores per chip increases it becomes less desirable to use a global scheduling strategy to manage all the tasks in a system. Instead, a method that partitions, or distributes, tasks to separate cores where they are scheduled locally is preferable. This makes sense in situations where a single global scheduling queue would have a high probability of contention from threads running on separate cores. Taking this further, it would appear that partitioning a system into logically separate domains spanning separate cores, I/O devices and memory regions would help reduce resource contention, and increase both isolation and scalability.

Several systems already adopt the idea of partitioning resources into logically separate domains on many- and multi-core architectures, including Corey [1], FOS [2], and Barrelfish [3]. These systems focus on scalability rather than timeliness. However, separation of resources and system components into logically isolated domains has the potential to increase system predictability: tasks in one domain can avoid resource contention from tasks in another separate domain. The principle of separation has been around for a long time in the design of systems. Rushby [4] introduced the idea of a *separation kernel* as appearing indistinguishable from a physically distributed system, involving explicit communication channels between separate domains.

Separation kernels have gained popularity in recent years on multi- and many-core platforms, to support tasks of different criticality levels in automotive and avionics applications. PikeOS [5], for example, is a separation micro-kernel [6] that supports multiple guest VMs, and targets safety-critical domains such as Integrated Modular Avionics. Other systems such as XtratuM [7], the Wind River Hypervisor, and Mentor Graphics Embedded Hypervisor all use virtualization technologies to logically isolate and multiplex guest virtual machines on a shared set of physical resources. In our own work, we have been developing the Quest-V [8] separation kernel for use in real-time and safety-critical systems.

Where available, Quest-V leverages hardware virtualization features found on modern processors (e.g., Intel VT-x, AMD-V, and ARM Cortex A15) to partition the system into a collection of *sandboxes*. Each sandbox encapsulates a region of machine physical memory, one or more processing cores, a subset of I/O devices, and a collection of software components and tasks. In effect, each sandbox is like a compute node (or machine) in a traditional distributed system. Hardware virtualization support is not required in our system, but can be used to enforce safe and secure isolation of sandbox domains. This is valuable in safety-critical systems in which tasks of different criticality and trustworthiness can co-exist without jeopardizing overall functionality. For architectures lacking hardware virtualization support, a system using software techniques can be developed to isolate components in separate domains.

Unlike in a traditional distributed system, Quest-V

uses shared memory channels for communication. In all other fundamental respects, Quest-V is a distributed system, with each sandbox domain having its own local clock and scheduling queue. The lack of a global scheduler and the distributed nature of Quest-V does, however, impose some challenges of the design of applications, particularly those with real-time constraints. For example, a multi-threaded application might need to be assigned to multiple sandbox domains, because a single sandbox might otherwise be overloaded. Application threads in separate sandboxes might, in turn, need to communicate with one another. Consequently, the programming model for a distributed Quest-V application needs to support the assignment and migration of threads and address spaces to specific sandboxes and, hence, cores. It also needs to support a method for timely communication between threads in separate sandboxes.

In this paper, we describe how communication and migration are performed predictably in Quest-V. In particular, we show how threads and their address spaces can be dynamically created and migrated between separate sandbox domains without violating the timing guarantees of existing threads. Given that threads in separate sandboxes are scheduled independently, we analyze the worst-case communication delays between groups of sandboxes, and determine the conditions under which an address space can be migrated to a remote sandbox without affecting the schedulability of existing tasks.

In the next section, we briefly describe the Quest-V architecture. This is followed by a discussion of mechanisms and timing guarantees on inter-sandbox communication and migration. Experimental results are shown in Section V. A summary of related work is described in Section VI. Finally, conclusions and future work are discussed in Section VII.

II. Architecture Overview

Figure 1 shows a simplified view of the Quest-V system as it relates to the problem in this paper. Further details are available in related work [8].

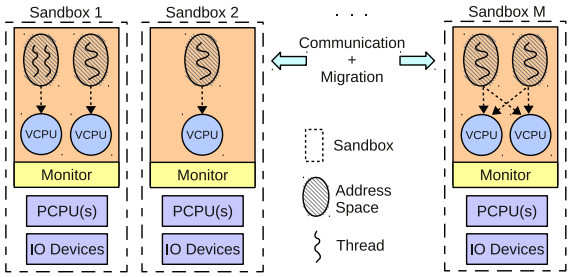

Fig. 1. Quest-V Architecture Overview

Quest-V is designed for safety-critical systems, comprising one or more sandboxes into which tasks and kernel services of different criticality levels can be mapped. A sandbox is a hardware partition that hosts a well-defined software environment. Each sandbox encapsulates one or more processor cores, a region of machine physical memory and a subset of I/O devices. The software environment for a sandbox consists of a monitor layer, a guest operating system (OS), and a group of application tasks. Collectively, this arrangement appears like a distributed chip-level system, or *separation kernel*. Each sandbox domain operates like a logically separate host, and communicates with other sandbox domains only through explicit channels. Failure of one sandbox should not compromise the functionality of another.

Quest-V uses machine virtualization techniques to partition hardware resources amongst separate sandboxes. Each sandbox has a trusted monitor that manages *extended page tables*, which assist in the mapping of guest-physical to machine-physical memory addresses. These page tables are similar, but in addition, to the traditional page table mappings in a conventional OS. In this case, a guest OS uses page tables to map guest-virtual to guest-physical memory addresses.

Quest-V does not use machine virtualization for any other purpose than to partition machine resources amongst sandboxes at boot time. Once booted, each sandbox monitor is only needed to establish communication channels with other sandboxes, and to assist in fault recovery. It is possible to detect a failed sandbox and recover its state without rebooting the entire system.

Failure of part or all of a sandbox functionality can be detected and recovered, as long as the monitor layer remains operational [1]. Although monitors are trusted entities in Quest-V, they have small memory footprints (typically less than a few kilobytes) and are not part of normal sandbox execution. Instead, each guest OS is responsible for managing the hardware resources within a sandbox, performing scheduling, I/O, and memory management without monitor intervention. This is possible because Quest-V does not multiplex multiple guest virtual machines on the same set of hardware, as is done in conventional hypervisor systems. Instead, the hardware is statically partitioned into separate domains.

Guest OSes in Quest-V can be as simple as a library OS [9], implementing a few basic services. Alternatively, they can be as powerful as a full-featured Linux system. Currently, Quest-V supports Linux guests for legacy services, and Quest native guests for real-time tasks.

VCPU Scheduling. Quest native guests feature a novel

[1]Details are outside the scope of this paper.

virtual CPU (VCPU) scheduling framework [10]. Software threads are mapped to VCPUs, which in turn are mapped to physical CPUs (PCPUs) [2] within the scope of a given sandbox. A VCPU is a logical abstraction, identifying the fraction of time it is allowed to execute on a PCPU in a specific window of real-time. By default, all VCPUs associated with conventional tasks act like sporadic servers [11], and are assigned static priorities.

Each VCPU, V_i, has a budget capacity, C_i, and replenishment period, T_i. Rate monotonic scheduling [12] can then be applied to determine schedulability. For improved utilization it is possible to configure Quest-V to schedule VCPUs according to a dynamic, deadline-based prioritization. However, for this paper, we assume all VCPUs are scheduled according to static priorities, when it is necessary to make the distinction clear.

The idea of managing VCPUs as sporadic servers is based on the observation that tasks issuing I/O requests typically block and wakeup at times that are not guaranteed to be periodic. However, such tasks often have an identifiable minimum interval between when they block and when they need to wakeup. Similarly, interrupts caused by I/O requests often occur at times that are not guaranteed to be periodic. A sporadic server model enables events with minimum inter-arrival times to be analyzed as though they were periodic.

Inter-Sandbox Communication. Inter-sandbox communication in Quest-V relies on message passing primitives built on shared memory, and asynchronous event notification mechanisms using Inter-processor Interrupts (IPIs). IPIs are currently used to communicate with remote sandboxes to assist in fault recovery, and can also be used to notify the arrival of messages exchanged via shared memory channels. Monitors update extended page table mappings as necessary to establish message passing channels between specific sandboxes. Only those sandboxes with mapped shared pages are able to communicate with one another. All other sandboxes are isolated from these memory regions.

A *mailbox* data structure is set up within shared memory by each end of a communication channel. By default, Quest-V currently supports asynchronous communication by polling a status bit in each relevant mailbox to determine message arrival. In this paper, we assume real-time communication is between sandboxes running Quest services, and featuring VCPU scheduling as described above. Message passing threads are bound to VCPUs with specific parameters to control the rate of exchange of information. Likewise, sending and receiving threads are assigned to higher priority VCPUs to reduce the latency of transfer of information across a communication channel. This way, shared memory channels can be prioritized and granted higher or lower throughput as needed, while ensuring information is communicated in a predictable manner. Thus, Quest-V supports real-time communication between sandboxes without compromising the CPU shares allocated to non-communicating tasks.

The lack of both a global clock and global scheduler for all sandboxes creates challenges for a system requiring strict timing guarantees. In the next two sections we elaborate on two such challenges, relating to predictable communication and address space migration.

III. Predictable Communication

For the purposes of predictable communication, we consider a system model as follows:

- A communication channel between a pair of endpoints in separate sandboxes is *half duplex* and has a *single slot*. A single slot has a configurable capacity, B, but is 4KB by default.
- One endpoint acting as a *sender* places up to one full slot of data in the channel when it detects the channel is *empty*.
- A second endpoint acting as a *receiver* consumes one slot of data from the channel when it is *full*.
- A transaction on a channel comprises the exchange of one or more slots of data. A sender sets a *start* flag to initiate a transaction. When the final unit of data is submitted to the channel, the sender sets an *end* of transaction flag.
- Each endpoint executes a thread mapped to a communication VCPU. The sender VCPU, V_s has parameters C_s and T_s, for its budget and period, respectively. Similarly, the receiver VCPU, V_r has parameters C_r and T_r. Both endpoints poll for the arrival of data when not sending, unless a special *out-of-band* signal is used, such as an interprocessor interrupt (IPI).

Consider a sending thread, τ_s, associated with a VCPU, V_s, which wishes to communicate with a receiving thread, τ_r, bound to V_r in a remote sandbox. Suppose τ_s sends a message of N bytes at a cost of δ_s time units per byte. Similarly, suppose τ_r replies with an M byte message at a cost of δ_r time units per byte. Before replying, let τ_r consume K units of processing time to service the communication request. The worst-case round-trip communication delay, Δ_{WC}, between τ_s and τ_r can now be calculated.

Case 1: All messages fit in one channel slot. In this case $N, M \leq B$. To calculate Δ_{WC}, we need to account

[2]We use the term "PCPU" to refer to a processor core, hardware thread, or uniprocessor.

for the time to send a request, process it, and wait for the reply. Let $S(N)$ be the total time taken by τ_s to send a request message of size N. That is:

$$S(N) = \lfloor \frac{N \cdot \delta_s}{C_s} \rfloor \cdot T_s + (N \cdot \delta_s) \bmod C_s$$

This accounts for multiple periods of V_s to send N bytes. At the receiver, we calculate the time, $R(N, M)$, to consume a request of size N, process the request and send a reply of size M as:

$$R(N,M) = \lfloor \frac{[N+M] \cdot \delta_r + K}{C_r} \rfloor \cdot T_r + ([N+M] \cdot \delta_r + K) \bmod C_r \quad (1)$$

The last stage of a communication transaction is consuming a response at the sender, which takes $S(M)$ time units.

Finally, we need to factor the shifts in time between when V_s and V_r are scheduled in their respective sandboxes. In the worst-case, a message is about to be sent when V_s uses up its current budget. This causes a delay of $T_s - C_s$ time units until its budget is replenished. The same situation might happen when V_s tries to consume the response. Similarly, a message arrives at the receiver when V_r has completed its current budget, so it will not be processed for another $T_r - C_r$ time. Consequently, the worst-case round-trip communication delay is:

$$\Delta_{WC}(N,M) = S(N) + (T_s - C_s) + R(N,M) + (T_r - C_r) + S(M) + (T_s - C_s) \quad (2)$$

Case 2: Messages take multiple slots. In this case, $N > B$ and $M \leq N$ [3]. For cases where the request messages take more than one slot, we need to consider Equation 2 each time a request-response channel slot is used. Hence, the multi-slot worst-case response time, Δ'_{WC}, becomes:

$$\Delta'_{WC} = \lceil \frac{N}{B} \rceil \cdot \Delta_{WC}(B, min(M, B)) \quad (3)$$

For the special case where communication is only one-way (e.g., to migrate an address space) of size N, Δ'_{WC} reduces to:

$$\Delta'_{WC} = \lceil \frac{N}{B} \rceil \cdot (S(B) + (T_s - C_s) + R(B,0) + (T_r - C_r)) \quad (4)$$

IV. Predictable Migration

Quest-V supports the migration of VCPUs and associated address spaces for several reasons: (1) to balance loads across sandboxes, (2) to guarantee the schedulability of VCPUs and threads, and (3) for closer proximity to needed resources such as I/O devices that would otherwise have to be accessed by remote procedure calls.

[3]For brevity, we omit the case where $M > N > B$.

Migration is initiated using the *vcpu_migration* interface shown in Listing 1. The *flag* is either 0, `MIG_STRICT` or `MIG_RELAX`. A *time* in milliseconds is used to specify either a deadline or timeout, depending on *flag*. The *dest* argument specifies the sandbox ID of a specific destination, or `DEST_ANY` if the caller wishes to allow the sandbox kernel to pick an acceptable destination.

The migration function is non-blocking. It returns `TRUE` only if the migration request is accepted, and the caller can resume its normal operation. The actual migration will happen at a later time decided by the sandbox kernel. The calling thread can use a flag in its task structure, or retrieve its current sandbox ID, to check whether the migration succeeded or failed.

Listing 1. Predictable Migration User Interface

```
bool vcpu_migration(uint32_t time, int dest,
                    int flag);
```

When *flag* is set to `MIG_STRICT`, the calling thread and its VCPU will be migrated to the destination with the restriction that the migrating VCPU's utilization cannot be affected. The local sandbox kernel must find a suitable time to perform migration to make this guarantee. An optional timeout specified using the *time* argument can be used to avoid indefinite delays before migration can occur. A *time* of 0 disables the timeout deadline.

When *flag* is set to 0, *time* is used to specify a migration deadline. A sandbox kernel will try to migrate the calling thread and its VCPU to the specified destination within the deadline. Unlike the case with `MIG_STRICT` flag, the calling thread's VCPU utilization can potentially be affected during migration. The worst-case down time for the migrating VCPU would be from the time of the request to the specified deadline.

Finally, when *flag* is set to `MIG_RELAX`, the calling thread and its VCPU will be migrated to the destination no matter how long it takes. As with 0 flag, calling *vcpu_migration* with `MIG_RELAX` will potentially affect the migrating VCPU's utilization. However, the VCPU down time is not bounded as with 0 flag.

Notice that the use of different flags in *vcpu_migration* only affects the behavior of the migrating thread and its VCPU. All the other VCPUs running in both the source and the destination sandbox should not be affected. The pseudo-code for *vcpu_migration()* and its integration into the local scheduler are shown in the appendix, in Listings 2 and 3. The major challenges in the implementation are: (1) accurately accounting for migration overheads, and (2) accurately estimating the worst-case migration cost under all circumstances.

A. Predictable Migration Strategy

Threads in Quest-V have corresponding address spaces and VCPUs. The current design limits one, possibly multi-threaded, address space to be associated with a single VCPU. This restriction avoids the problem of migrating VCPUs and multiple address spaces between sandboxes, which could lead to arbitrary delays in copying memory. Migration from one sandbox's private memory requires a copy of an address space and all thread data structures to the destination. Each thread is associated with a `quest_tss` structure that stores the execution context and VCPU state.

Dedicated migration threads and corresponding VCPUs are established within each sandbox at system initialization. A migration thread is responsible for the actual VCPU migration operation. An inter-processor interrupt (IPI) is used by the local sandbox kernel to notify a remote migration thread of a migration request. In our current implementation, only one migration thread and VCPU can be configured for each sandbox. If multiple migration requests occur at the same time, they will be processed serially.

Migration using Message Passing. This approach transfers a thread's state, including its address space and VCPU information, using a series of messages that are passed over a communication channel. The advantage of this approach is that it generalizes across different communication links, including those where shared memory is not available (e.g., Ethernet).

To initiate migration, an IPI is first sent to the migration thread in the destination sandbox. The destination then waits for data on a specific channel. Since the default communication channel size is 4KB, a stream of messages are needed to migrate an address space, along with its thread and VCPU state. This resembles the communication scenario described in *Case 2* of Section III. The destination re-assembles the address space and state information before adding the migrated VCPU to its scheduler queue. An IPI or acknowledgement message from the destination to the source is now needed to signal the completion of migration. If successful, the migration thread in the source sandbox will be able to reclaim the memory of the migrated address space. Otherwise, the migrating VCPU will be put back into the run queue in the source sandbox.

Before a VCPU is migrated, admission control is performed at the destination. This is used to verify the schedulability of the migrating VCPU and all existing VCPUs in the destination. If admission control fails, a migration request is rejected.

At boot time, Quest-V establishes base costs for copying memory pages without caches enabled [4]. These costs are used to determine various parameters used for worst-case execution time estimation.

An estimate of the worst-case migration cost requires: (1) the cost of serializing the migrated state into a sequence of messages (Δ_s), (2) the communication delay to send the messages (Δ_t), and (3) the cost of reassembling the transferred state at the destination (Δ_a). We assume one migration thread is associated with a sender VCPU, V_s, and another is associated with a receiver VCPU, V_r.

$$\Delta_s = \lfloor \frac{\delta_s}{C_s} \rfloor \cdot T_s + \delta_s \bmod C_s + T_s - C_s \qquad (5)$$

Here, δ_s is the execution time of a migration thread to produce a sequence of messages, assuming caches are disabled. Similarly, given δ_a, is the execution time to reassemble a VCPU and address space:

$$\Delta_a = \lfloor \frac{\delta_a}{C_r} \rfloor \cdot T_r + \delta_a \bmod C_r + T_r - C_r \qquad (6)$$

In this case, Δ_t is identical to Δ'_{WC} in Equation 3. Hence, the the worst-case migration cost when message passing is used is:

$$\Delta_{mig} = \Delta_s + \Delta'_{WC} + \Delta_a \qquad (7)$$

Migration with Direct Memory Copy. As shown in Equation 3, the worst-case time to transfer a large amount of state between two sandboxes can span numerous migration VCPU periods. This makes it difficult to satisfy a VCPU migration request using message passing, with the `MIG_STRICT` flag set. Fortunately, for Quest-V sandboxes that communicate via shared memory, it is possible to dramatically reduce the migration overhead.

Figure 2 shows the general migration strategy when direct memory copy is used. An IPI is first sent to the destination sandbox, to initiate migration. The migration thread handles the IPI in the destination, generating a trap into its monitor that has access to machine physical memory of all sandboxes. The migrating address space in the source sandbox is temporarily mapped into the destination. The address space and associated `quest_tss` thread structures are then copied to the target sandbox's memory. At this point, the page mappings in the source sandbox can be removed by the destination monitor.

Similar to the message passing approach, an IPI from the destination to the source sandbox is needed to signal the completion of migration. All IPIs are handled in the sandbox kernels, with interrupts disabled while in monitor mode. The migration thread in the destination

[4]We do not consider memory bus contention issues, which could make worst-case estimations even larger.

can now exit its monitor and return to the sandbox kernel. The migrated address space is attached to its VCPU and added to the local schedule. At this point, the migration threads in source and destination sandboxes are able to yield execution to other VCPUs and, hence, threads.

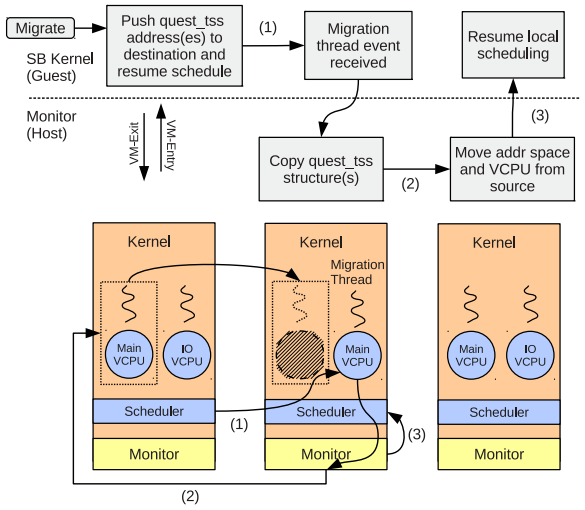

Fig. 2. Migration Strategy

With direct memory copy, the worst-case migration cost can simply be defined as:

$$\Delta_{mig} = \lfloor \frac{\delta_m}{C_r} \rfloor \cdot T_r + \delta_m \bmod C_r + T_r - C_r \qquad (8)$$

Here, C_r and T_r are the budget and period of the migration thread's VCPU in destination sandbox, and δ_m is the execution time to copy an address space and its $quest\_tss$ data structures to the destination.

Migration Thread Preemption. The migration thread in each sandbox is bound to a VCPU. If the VCPU depletes its budget or a higher priority VCPU is ready to run, the migration thread should be preempted. However, if direct memory copy is used, migration thread preemption is complicated by the fact that the thread spends most of its time inside the sandbox monitor, and each sandbox scheduler runs within the local kernel (outside the monitor).

Migration thread preemption, in this case, requires a domain switch between a sandbox monitor and its kernel, to access the local scheduler. This results in costly VM-Exit and VM-Entry operations that flush the TLB of the processor core. To avoid this cost, we limited migration thread preemption to specific preemption points. Additionally, we associated each migration thread with a highest priority VCPU, ensuring it would run until either migration was completed or the VCPU budget expired. Bookkeeping is limited to tracking budget usage at each preemption point. Thus, within one period, a migration thread needs only one call into its local monitor.

Preemption points are currently located: (1) immediately after copying each `quest_tss` structure, (2) between processing each Page Directory Entry during address space cloning, and (3) right before binding the migrated address space to its VCPU, for re-scheduling. In the case of a budget overrun, the next budget replenishment is adjusted according to the corrected POSIX Sporadic Server algorithm [13]. Figure 3 describes the migration control flow.

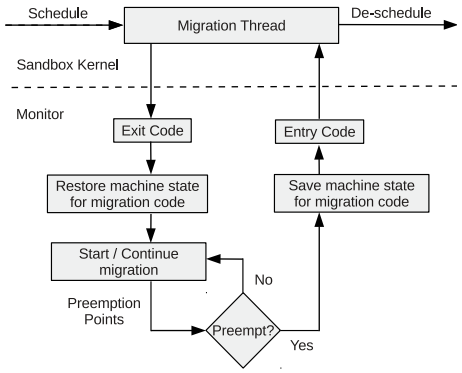

Fig. 3. Migration Framework Control Flow

Clock Synchronization. One extra challenge to be considered during migration is clock synchronization between different sandboxes in Quest-V. Quest-V schedulers use Local APIC Timers and Time Stamp Counters (TSCs) in each core as the source for all time-related activities in the system, and these are not guaranteed to be synchronized by hardware. Consequently, Quest-V adjusts time for each migrating address space to compensate for clock skew. This is necessary when updating budget replenishment and wakeup time events for a migrating VCPU that is sleeping on an I/O request, or which is not yet runnable.

The source sandbox places its current TSC value in shared memory immediately before sending a IPI migration request. This value is compared with the destination TSC when the IPI is received. A time-adjustment, δ_{ADJ}, for the migrating VCPU is calculated as follows:

$$\delta_{ADJ} = TSC_d - TSC_s - 2 * RDTSC_{cost} - IPI_{cost}$$

TSC_d and TSC_s are the destination and source TSCs, while $RDTSC_{cost}$ and IPI_{cost} are the average costs of reading a TSC and sending an IPI, respectively. δ_{ADJ} is then added to all future budget replenishment and wakeup time events for the migrating VCPU in the destination sandbox.

B. Migration Criteria

Quest-V restricts migrate-able address spaces to those associated with VCPUs that either: (1) have currently expired budgets, or (2) are waiting in a sleep queue.

In the former case, the VCPU is not runnable at its foreground priority until its next budget replenishment. In the latter case, a VCPU is blocked until a wakeup event occurs (e.g., due to an I/O request completion or a resource becoming available). Together, these two cases prevent migrating a VCPU when it is runnable, as the migration delay could impact the VCPU's utilization.

For VCPU, V_m, associated with a migrating address space, we define E_m to be the *relative time* [5] of the next event, which is either a replenishment or wakeup.

If V_m issues a migration request with `MIG_STRICT` flag, for the utilization of V_m to be unaffected by migration, the following must hold:

$$E_m \geq \Delta_{mig} \tag{9}$$

Where Δ_{mig} can be calculated by either Equation 7 or 8. Quest-V makes sure that the migrating thread will not be woken up by asynchronous events until the migration is finished. The system imposes the restriction that threads waiting on I/O events cannot be migrated. Similarly, the migration deadline can be compared with Δ_{mig} to make migration decisions when `flag`=0.

V. Experimental Evaluation

We conducted a series of experiments on a Gigabyte Mini-ITX machine with an Intel Core i5-2500K 3.3GHz 4-core processor, 8GB RAM and a Realtek 8111e NIC.

A. Predictable Communication

We first ran 5 different experiments to predict the worst-case round-trip communication time using Equation 2. The VCPU settings of the sender and receiver, spanning two different sandboxes, are shown in Table I.

Case #	Sender VCPU	Receiver VCPU
Case 1	20/100	2/10
Case 2	20/100	20/100
Case 3	20/100	20/130
Case 4	20/100	20/200
Case 5	20/100	20/230

TABLE I
VCPU Parameters

We calculated the values of δ_s and δ_r by setting the message size to 4KB for both sender and receiver (i.e. $M = N = 4$KB) and *disabling caching* of the shared memory communication channel on the test platform. The message processing time K has essentially been ignored because the receiver immediately sends the response after receiving the message from the sender.

Both sender and receiver threads running on VCPUs V_s and V_r, respectively, sleep for controlled time units, to influence phase shifts between their periods

[5] i.e., Relative to current time.

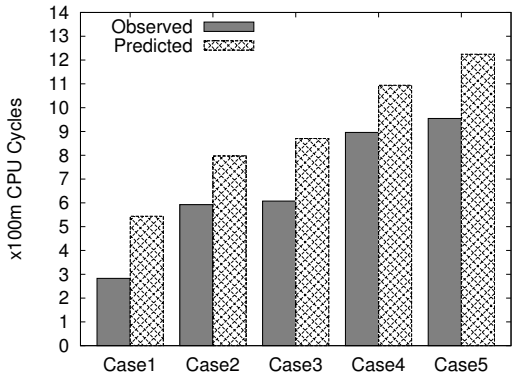

Fig. 4. Worst-Case Round-trip Communication

T_s and T_r. Similarly, the sender thread adds busy-wait delays before transmission, to affect the starting point of communication within its VCPU's available budget, C_s. Figure 4 shows results after 10000 message exchanges are performed for each of the 5 experiments. As can be seen, the observed value is always within the prediction bounds derived from Equation 2.

We next conducted a series of one-way communication experiments to send 4MB messages through a 4KB channel with different VCPU parameters as shown in Table II. Figure 5 again shows that the observed communication times are within the bounds derived from our worst-case estimations. However, the bounds are not as tight as for round-trip communication. We believe this is due to the fact that we used a pessimistic worst-case estimation, which includes leftover VCPU budgets in each instance of the multi-slot communication. Estimation error is reduced when the difference between VCPU budgets and periods is smaller.

Case #	Sender VCPU	Receiver VCPU
Case 1	20/50	20/50
Case 2	10/100	10/100
Case 3	10/100	10/50
Case 4	10/100	10/200
Case 5	5/100	5/130
Case 6	10/200	10/200

TABLE II
VCPU Parameters

B. Predictable Migration

To verify the predictability of the Quest-V migration framework, we constructed a task group consisting of 2 communicating threads and another CPU-intensive thread running a Canny edge detection algorithm on a stream of video frames. The frames were gathered from a LogiTech QuickCam Pro9000 camera mounted on our RacerX mobile robot, which traversed one lap of Boston University's indoor running track at Agganis Arena [6]. To

[6] RacerX is a real-time robot that runs Quest-V.

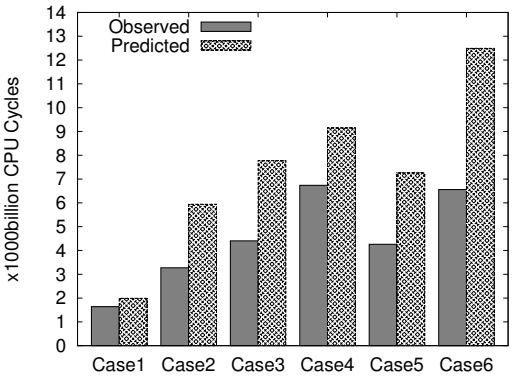

Fig. 5. Worst-Case One-way Multi-slot Communication

avoid variable bit rate frames affecting the results of our experiments, we applied Canny repeatedly to the frame shown in Figure 6 rather than a live stream of the track. This way, we could determine the effects of migration on a Canny thread by observing changes in processing rate while the other threads communicated with one another.

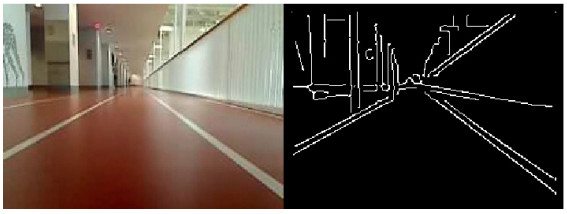

Fig. 6. Track Image Processed by Canny

For all the experiments in this section, we have two active sandbox kernels each with 5 VCPUs. The setup is shown in Table III. The Canny thread is the target for migration from sandbox 1 to sandbox 2 in all cases. Migration is always requested at time 5. A logger thread is used to collect the result of the experiment in a predictable manner. Data points are sampled and reported in a one second interval. For migration with message passing, a low priority migration VCPU (10/200) is used. In the case of direct memory copy, the migration thread is associated with the highest priority VCPU (10/50).

VCPU (C/T)	Sandbox 1	Sandbox 2
20/100	Shell	Shell
10/200 (10/50)	Migration Thread	Migration Thread
20/100	Canny	
20/100	Logger	Logger
10/100	Comms 1	Comms 2

TABLE III
MIGRATION EXPERIMENT VCPU SETUP

Figure 7 shows the behavior of Canny as it is migrated using message passing in the presence of the two communicating threads. The y-axis shows both Canny frame rate (in frames-per-second, *fps*) and message passing throughput (in multiples of a 1000 Kilobytes-per-second). Canny requested migration with the `MIG_RELAX` flag, leading to a drop in frame rate during transfer to the remote sandbox. However, the two communicating threads were not affected.

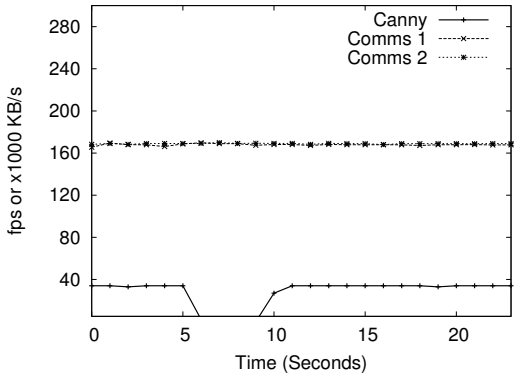

Fig. 7. Migration using Message Passing

Table IV shows the estimated worst-case and actual migration cost. The worst-case is derived from Equation 7. Even though the actual migration cost is much smaller than the estimation, it is still larger than E_m, forbidding migration with the `MIG_STRICT` flag.

Variables	E_m	$\Delta_{mig}, worst$	$\Delta_{mig}, actual$
Time (ms)	79.8	243681.02	4021.18

TABLE IV
MESSAGE PASSING MIGRATION CONDITION

In Figure 8, the same experiment was conducted with direct memory copy and `flag=MIG_STRICT`. Since the migration thread was self-preempted, the right y-axis shows its actual CPU consumption in (millions of, *x1m*) cycles. We can see from this figure that none of the threads have been affected by migration. The sudden spike in migration thread CPU consumption occurs during the migration of the Canny thread.

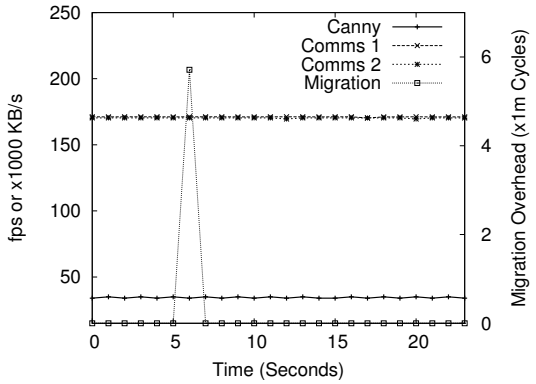

Fig. 8. Migration using Direct Memory Copy

Table V shows the values of variables as defined in Equation 8 and 9. $\delta_m, worst$ is the worst-case time to copy a Canny address space with all caches disabled, including the overhead of walking its page directory. $\delta_m, actual$ is the actual migration thread budget consumption with caches enabled. Both worst-case and actual migration costs satisfy the constraints of Equation 9. This guarantees that all VCPUs remain unaffected in terms of their CPU utilization during migration.

Variables	E_m	$\delta_m, worst$	$\delta_m, actual$	C_r	T_r
Time (ms)	79.8	5.4	1.7	10	50

TABLE V
DIRECT MEMORY COPY MIGRATION CONDITION

In the next experiment, we switched back to `flag=MIG_RELAX` and manually increased the migration cost by adding a busy-wait of $800\mu s$ to the address space clone procedure for each processed Page Directory Entry (of which there were 1024 in total). This forced the migration cost to violate Equation 9. Similar to Figure 7, Figure 9 shows how the migration costs increase, with only the migrating thread being affected. Here, the preemption points within each sandbox monitor prevent excessive budget overruns that would otherwise impact VCPU schedulability.

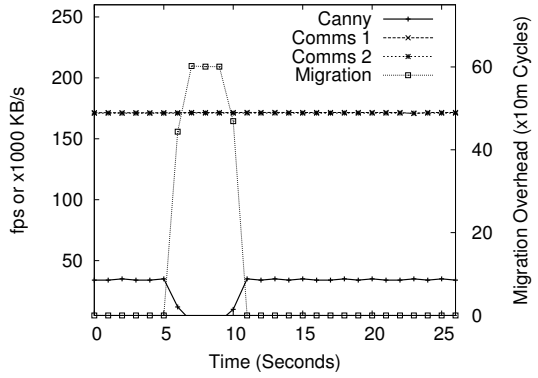

Fig. 9. Migration With Added Overhead

Table VI shows the migration parameters for this experiment. We also measured the budget utilization of the migration thread while it was active. Results are shown in Table VII for the interval [6s,10s] of Figure 9. Migration thread budget consumption peaks at 91.5% rather than 100%, because of self-preemption and accounting overheads. We are currently working on optimizations to reduce these overheads.

Variables	E_m	$\delta_m, worst$	$\delta_m, actual$	C_r	T_r
Time (ms)	79.8	891.4	825.1	10	50

TABLE VI
MIGRATION CONDITION WITH ADDED OVERHEAD

Time (sec)	**6**	**7**	**8**	**9**	**10**
Utilization	67.5%	91.5%	91.5%	91.5%	71.5%

TABLE VII
MIGRATION THREAD SELF-PREEMPTION BUDGET UTILIZATION

For comparison, the same experiment was repeated without a dedicated migration thread. Instead, migration was handled in the context of an IPI handler that runs with interrupts subsequently disabled. Consequently, the handler delays all other threads and their VCPUs during its execution, as shown in Figure 10.

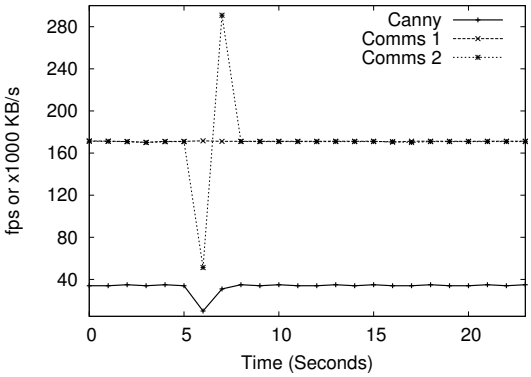

Fig. 10. Migration Without a Dedicated Thread

Table VIII shows the values of the variables used in Equation 8 and 9 when the migration overhead first starts to impact the Canny frame rate. In theory, the minimum δ_m that violates Equation 9 is any value greater than 10ms. However, because $\delta_m, worst$ is a worst-case estimation and the worst-case VCPU phase shift ($T_r - C_r$ in Equation 8) rarely happens in practice, the first visible frame rate drop happens at 26.4ms. At this time, the actual budget consumption of the migration thread is 19.2ms, which is greater than 10ms.

Variables	E_m	$\delta_m, worst$	$\delta_m, actual$	C_r	T_r
Time (ms)	79.8	26.4	19.2	10	50

TABLE VIII
MIGRATION BOUNDARY CASE CONDITION

Finally, as mentioned earlier in Section IV-A, if multiple migration requests to a destination sandbox are issued simultaneously, they will be processed serially. Currently, parallel migration is not supported. The source sandbox kernel has to essentially lock both its own migration thread and the migration thread in the destination before initiating migration. To demonstrate this effect, we conducted an experiment in which two sandboxes issued a migration request at the same time, to the same destination. The VCPU setup is shown in Table IX. In addition to Canny and the 2 communicating threads, we

added another thread in sandbox 3 that repeatedly counts prime numbers from 1 to 2500 and increments a counter after each iteration. Canny and Prime attempt to migrate to sandbox 2 at the same time.

VCPU (C/T)	Sandbox 1	Sandbox 2	Sandbox 3
20/100	Shell	Shell	Shell
10/100	Mig Thread	Mig Thread	Mig Thread
20/100	Canny		
10/100	Logger	Logger	Logger
10/100	Comms 1	Comms 2	
10/100			Prime

TABLE IX
MIGRATION THREAD CONTENTION EXPERIMENT VCPU SETUP

The results of the experiment are shown in Figure 11. The y-axis now also shows the Prime count in addition to Canny frame rate and message passing throughput. Both Prime and Canny request migration to sandbox 2 at some time after 6 seconds. Prime acquires the locks first and starts migration immediately. The migration request of Canny is delayed since the *try_lock* function returns `FALSE` in Listing 3 in the appendix. Because both requests are issued with `flag=MIG_RELAX`, Canny is migrated soon after Prime finishes migration.

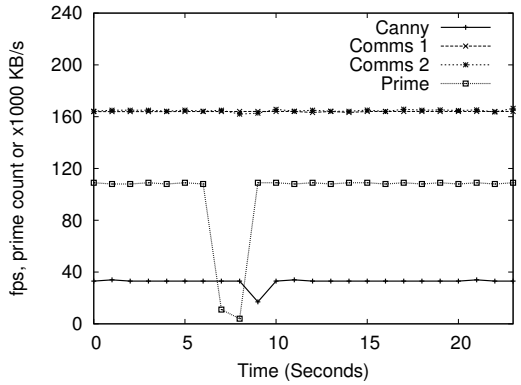

Fig. 11. Migration Thread Contention

By setting the migration start time for Prime to $t_0 = 0$, Table X shows the relative time of: the start of data transfer of Prime (t_1), the end of data transfer of Prime (t_2), the end of Prime migration (t_3), the start of Canny migration (t_4), the start of data transfer of Canny (t_5), the end of data transfer of Canny (t_6) and the end of Canny migration (t_7).

t_0	t_1	t_2	t_3
0	3.15	1903.81	1913.47
t_4	t_5	t_6	t_7
1999.82	2003.67	2402.72	2412.98

TABLE X
MIGRATION TIME SEQUENCE (MILLISECONDS)

VI. RELATED WORK

PikeOS [5] is a separation micro-kernel [6] that supports multiple guest VMs, and targets safety-critical domains such as Integrated Modular Avionics. It uses a virtualization layer to spatially and temporally partition resources amongst guests. Xen [14], Linux-KVM [15], XtratuM [7], the Wind River Hypervisor, Mentor Graphics Embedded Hypervisor, and LynxSecure [16] all use virtualization technologies to isolate and multiplex guest virtual machines on a shared set of physical resources.

In contrast to the above approaches, Quest-V statically partitions machine resources into separate sandboxes, allowing guest OSes to directly manage hardware without involvement of a hypervisor. Interrupts, I/O transfers and scheduling are all directly handled by Quest-V guests without the involvement of a virtual machine monitor.

Quest-V supports the migration of address spaces and VCPUs to remote sandboxes for reasons such as load balancing, or proximity to I/O devices. Quest-V's migration scheme is intended to maintain predictability, even for tasks that may have started executing in one sandbox and then resume execution in another. Other systems that have supported migration include MOSIX [17] and Condor [18], but these do not focus on real-time migration.

In other work, reservation-based scheduling has been applied to client/server interactions involving RPC [19]. This approach is based on analysis of groups of tasks interacting through shared resources accessed with mutual exclusion [20], [21]. A bandwidth inheritance protocol is used to guarantee the server handles each request according to scheduling requirements of the client. We intend to investigate the use of bandwidth inheritance protocols across sandboxes, although this is complicated by the lack of global prioritization of VCPUs.

Finally, models such as RAD-Flows [22] attempt to identify buffer space requirements for communicating tasks to avoid blocking delays. Such techniques will be considered further as we develop Quest-V to use ring buffers for communication between sandboxes. Currently, our work focuses on single-slot communication between pairs of sandboxes hosting bandwidth-preserving VCPUs.

VII. CONCLUSIONS AND FUTURE WORK

This paper focuses on predictable communication and migration in the Quest-V separation kernel. Quest-V partitions machine resources amongst separate sandboxes that operate like traditional hosts in a distributed system. However, unlike a traditional distributed system, communication channels are built on shared memory, which has low latency and high bandwidth.

Quest-V allows threads to migrate between sandboxes. This might be necessary to ensure loads are balanced, and each sandbox can guarantee the schedulability of its virtual CPUs (VCPUs). In other cases, threads might need to be migrated to sandboxes that have direct access to I/O devices, thereby avoiding expensive inter-sandbox communication. We have shown how Quest-V's migration mechanism between separate sandboxes is able to ensure predictable VCPU and thread execution. Experiments show the ability of our Canny edge detector to maintain its frame processing rate while migrating from one sandbox to another. This application bears significance in our RacerX autonomous vehicle project that uses cameras to perform real-time lane detection.

Finally, we have shown how Quest-V is able to enforce predictable time bounds on the exchange of information between threads mapped to different sandboxes. This lays the foundations for real-time communication in a distributed embedded system. Future work will investigate *lazy* migration of only the working set (or *hot*) pages of address spaces. This will likely reduce initial migration costs but incur page faults that will need to be addressed predictably.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1117025. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, M. F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. hua Dai, Y. Zhang, and Z. Zhang, "Corey: An operating system for many cores," in *the 8th USENIX Symposium on Operating Systems Design and Implementation*, pp. 43–57, 2008.

[2] D. Wentzlaff and A. Agarwal, "Factored operating systems (FOS): The case for a scalable operating system for multicores," *SIGOPS Operating Systems Review*, vol. 43, pp. 76–85, 2009.

[3] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The Multikernel: A new OS architecture for scalable multicore systems," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pp. 29–44, 2009.

[4] J. M. Rushby, "Design and verification of secure systems," in *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, pp. 12–21, 1981.

[5] "SYSGO PikeOS." http://www.sysgo.com/products/pikeos-rtos-and-virtualization-concept.

[6] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an OS kernel," in *the 22nd ACM Symposium on Operating Systems Principles*, pp. 207–220, 2009.

[7] A. Crespo, I. Ripoll, and M. Masmano, "Partitioned embedded architecture based on hypervisor: The XtratuM approach.," in *the European Dependable Computing Conference*, pp. 67–72, 2010.

[8] Y. Li, R. West, and E. Missimer, "A virtualized separation kernel for mixed criticality systems," in *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, (Salt Lake City, Utah), March 1-2 2014.

[9] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr., "Exokernel: An operating system architecture for application-level resource management," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pp. 251–266, 1995.

[10] M. Danish, Y. Li, and R. West, "Virtual-CPU scheduling in the Quest operating system," in *Proceedings of the 17th Real-Time and Embedded Technology and Applications Symposium*, pp. 169–179, 2011.

[11] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic task scheduling for hard real-time systems," *Real-Time Systems Journal*, vol. 1, no. 1, pp. 27–60, 1989.

[12] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[13] M. Stanovich, T. P. Baker, A. I. Wang, and M. G. Harbour, "Defects of the POSIX sporadic server and how to correct them," in *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2010.

[14] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pp. 164–177, 2003.

[15] I. Habib, "Virtualization with KVM," *Linux Journal*, vol. 2008, no. 166, p. 8, 2008.

[16] "LynxSecure Embedded Hypervisor and Separation Kernel." http://www.lynuxworks.com/virtualization/hypervisor.php.

[17] T. Maoz, A. Barak, and L. Amar, "Combining Virtual Machine migration with process migration for HPC on multi-clusters and Grids," in *IEEE International Conference on Cluster Computing*, pp. 89 –98, 2008.

[18] M. Litzkow, M. Livny, and M. Mutka, "Condor - a hunter of idle workstations," in *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[19] L. Abeni and T. Prastowo, "Experiences with client/server interactions in a reservation-based system," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, 2012.

[20] D. De Niz, L. Abeni, S. Saewong, and R. Rajkumar, "Resource Sharing in Reservation-Based Systems," in *Proceedings of the 22nd IEEE Real-time Systems Symposium*, pp. 171–180, 2001.

[21] G. Lamastra and L. Abeni, "Task synchronization in reservation-based real-time systems," *IEEE Transactions on Computers*, vol. 53, no. 12, pp. 1591–1601, 2004.

[22] R. Pineiro, K. Ioannidou, C. Maltzahn, and S. A. Brandt, "RAD-flows: Buffering for predictable communication," in *Proceedings of The 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2011.

VIII. Appendix

Listing 2 shows the pseudo code implementation of the *vcpu_migration* function. It simply sets up the migration request and waits for the scheduler to respond. The actual handling of a migration request is part of the *schedule* function shown in Listing 3.

Listing 2. vcpu_migration Pseudo Code

```
bool vcpu_migration (uint32_t time, int dest
    , int flag) {
  if(!valid (dest) || !(valid (flag)))
    return FALSE;

  if(flag == MIG_STRICT) {
    if(time)
      cur_task.mig_timeout = now + time;
    else
      cur_task.mig_timeout = 0;
  } else if(flag == MIG_RELAX) {
    cur_task.mig_dl = MAX_DEADLINE;
  } else {
    if(time)
      cur_task.mig_dl = now + time;
    else
      return FALSE;
  }

  cur_task.mig_status = 0;
  cur_task.mig_flag = flag;
  cur_task.affinity = dest;

  return TRUE;
}
```

Listing 3. Scheduler Pseudo Code

```
void schedule (void) {
  ...

  /* Check migration request when de-
      scheduled */
  if(next_task != cur_task) {
    if(cur_task.affinity != cur_sandbox) {
      if(cur_task.affinity == DEST_ANY)
        cur_task.affinity =
          find_destination();

      /* Lock both source and destination */
      if(try_lock(cur_sandbox, cur_task.
          affinity)) {
        if(!check_utilization_bound()) {
          cur_task.mig_flag = FAIL;
          cur_task.affinity = cur_sandbox;
          goto release;
        }

        /* MIG_RELAX, migrate right now */
        if(cur_task.mig_flag == MIG_RELAX) {
          cur_task.mig_status = SUCCESS;
          do_migration(cur_task);
          goto release;
        }

        /* Calculate migration cost */
        WCET = calculate_wcet(cur_task);

        if(cur_task.mig_flag == MIG_STRICT)
            {
          /* next_event () returns Es */
          cur_task.mig_dl =
            next_event()+now;
        }

        if((now+WCET) > cur_task.mig_dl) {
          cur_task.mig_status = FAIL;
          cur_task.affinity = cur_sandbox;
        } else {
          cur_task.mig_status = SUCCESS;
          do_migration(cur_task);
        }

release:
        unlock(cur_sandbox, cur_task.
            affinity);
      } else {
        /* Destination is busy or we are
            migrating another task */
        if(flag == MIG_STRICT) {
          if((now+next_event()) >=
              cur_task.mig_timeout) {
            cur_task.mig_status = FAIL;
            cur_task.affinity = cur_sandbox;
          }
        } else {
          if((now+next_event()) >=
              cur_task.mig_dl) {
            cur_task.mig_status = FAIL;
            cur_task.affinity = cur_sandbox;
          }
        }
      }
    }
  }

resume_schedule:
  ...
}
```