Time Management in the Quest-V RTOS [*]

Richard West, Ye Li, and Eric Missimer

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

Abstract

Quest-V is a new system currently under development for multicore processors. It comprises a collection of separate kernels operating together as a distributed system on a chip. Each kernel is isolated from others using virtualization techniques, so that faults do not propagate throughout the entire system. This multikernel design supports online fault recovery of compromised or misbehaving services without the need for full system reboots. While the system is designed for high-confidence computing environments that require dependability, Quest-V is also designed to be predictable. It treats time as a first-class resource, requiring that all operations are properly accounted and handled in real-time. This paper focuses on the design aspects of Quest-V that relate to how time is managed. Special attention is given to how Quest-V manages time in four key areas: (1) scheduling and migration of threads and virtual CPUs, (2) I/O management, (3) communication, and (4) fault recovery.

1 Introduction

Multicore processors are becoming increasingly popular, not only in server domains, but also in real-time and embedded systems. Server-class processors such as Intel's Single-chip Cloud Computer (SCC) support 48 cores, and others from companies such as Tilera are now finding their way into real-time environments [18]. In real-time systems, multicore processors offer the opportunity to dedicate time-critical tasks to specific processors, allowing others to be used by best effort services. Alternatively, as in the case of processors such as the ARM Cortex-R7, they provide fault tolerance, ensuring functionality of software in the wake of failures of any one core.

[*]This work is supported in part by NSF Grant #1117025.

Quest-V is a new operating system we are developing for multicore processors. It is designed to be both dependable and predictable, providing functionality even when services executing on one or more cores become compromised or behave erroneously. Predictability even in the face of software component failures ensures that application timing requirements can be met. Together, Quest-V's dependability and predictability objectives make it suitable for the next generation of safety-critical embedded systems.

Quest-V is a virtualized multikernel, featuring multiple sandbox kernels connected via shared memory communication channels. Virtualization is used to isolate and prevent faults in one sandbox from adversely affecting other sandboxes. The resultant system maintains availability while faulty software components are replaced or re-initialized in the background. Effectively, Quest-V operates as a "distributed system on a chip", with each sandbox responsible for local scheduling and management of its own resources, including processing cores.

In Quest-V, scheduling involves the use of *virtual CPUs* (VCPUs). These differ from VCPUs in conventional hypervisor systems, which provide an abstraction of the underlying physical processors that are shared among separate guest OSes. Here, VCPUs act as resource containers [3] for scheduling and accounting the execution time of threads. VCPUs form the basis for system predictability in Quest-V. Each VCPU is associated with one or more software threads, which can be assigned to specific sandboxes according to factors such as per-core load, interrupt balancing, and processor cache usage, amongst others. In this paper, we show how VCPU scheduling and migration is performed predictably. We also show how time is managed to ensure bounded delays for inter-sandbox communication, software fault recovery and I/O management.

An overview of the Quest-V design is described in the next section. This is followed in Section 3 by a description of how Quest-V guarantees predictability in various subsystems, including VCPU scheduling and migration, I/O management, communication and fault recovery. Finally, conclusions and future work are described in Section 4.

2 Quest-V Design

Quest-V is targeted at safety-critical applications, primarily in real-time and embedded systems where dependability is important. Target applications include those emerging in health-care, avionics, automotive systems, factory automation, robotics and space exploration. In such cases, the system requires real-time responsiveness to time-critical events, to prevent potential loss of lives or equipment. Similarly, advances in fields such as cyber-physical systems means that more sophisticated OSes beyond those traditionally found in real-time domains are now required.

The emergence of off-the-shelf and low-power processors now supporting multiple cores and hardware virtualization offer new opportunities for real-time and embedded system designers. Virtualization capabilities enable new techniques to be integrated into the design of the OS, so that software components are isolated from potential faults or security violations. Similarly, added cores offer fault tolerance through redundancy, while allowing time-critical tasks to run in parallel when necessary. While the combination of multiple cores and hardware virtualization are features currently found on more advanced desktop and server-class processors, it is to be anticipated that such features will appear on embedded processors in the near future. For example, the ARM Cortex A15 processor is expected to feature virtualization capabilities, offering new possibilities in the design of operating systems.

Quest-V takes the view that virtualization features should be integral to the design of the OS, rather than providing capabilities to design hypervisors for hosting separate unrelated guest OSes. While virtualization provides the basis for safe isolation of system components, proper time management is necessary for real-time guarantees to be met. Multicore processors pose challenges to system predictability, due to the presence of shared on-chip caches, memory bus bandwidth contention, and in some cases non-uniform memory access (NUMA). These micro-architectural factors must be addressed in the design of the system. Fortunately, hardware performance counters are available, to help deduce micro-architectural resource usage. Quest-V features a performance monitoring subsystem to help improve schedulability of threads and VCPUs, reducing worst-case execution times and allowing higher workloads to be admitted into the system.

2.1 System Architecture

Figure 1 shows an overview of the Quest-V architecture. One sandbox is mapped to a separate core of a multicore processor, although in general it is possible to map sandboxes to more than one core [1]. This is similar to how Corey partitions resources amongst applications [7]. In our current approach, we assume each sandbox kernel is associated with one physical core since that simplifies local (sandbox) scheduling and allows for relatively easy enforcement of service guarantees using a variant of rate-monotonic scheduling [12]. Notwithstanding, application threads can be migrated between sandboxes as part of a load balancing strategy, or to allow parallel thread execution.

A single hypervisor is replaced by a separate monitor for each sandbox kernel. This avoids the need to switch page table mappings within a single global monitor when accessing sandbox (guest) kernel addresses. We assume monitors are trusted but failure of one does not necessarily mean the system is compromised since one or more other monitors may remain fully operational. Additionally, the monitors are expected to only be used for exceptional conditions, such as updating shared memory mappings for inter-sandbox communication [11] and initiating fault recovery.

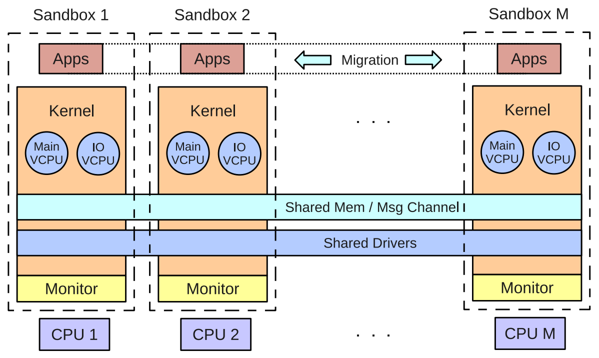

Figure 1. Quest-V Architecture Overview

Quest-V currently runs as a 32-bit system on x86 platforms with hardware virtualization support (e.g., Intel VT-x or AMD-V processors). Memory virtualization is used as an integral design feature, to separate sub-system components into distinct sandboxes. Further details can be found in our complementary paper that focuses more extensively on the performance of the Quest-V design [11]. Figure 2 shows the mapping of sandbox memory spaces to physical memory. Extended page table (EPT [2]) structures combine with conventional page tables to map sandbox (guest) virtual addresses to host physical values. Only monitors can change EPT memory mappings, ensuring software faults or security violations in one sandbox cannot corrupt the memory of another sandbox.

[1]Unless otherwise stated, we make no distinction between a processing core or hardware thread.

[2]Intel processors with VT-x technology support extended page tables, while AMD-V processors have similar support for nested page tables. For consistency we use the term EPT in this paper.

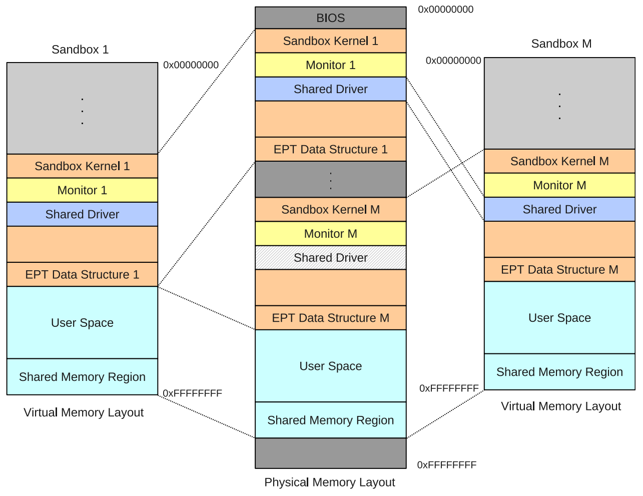

Figure 2. Quest-V Memory Layout

The Quest-V architecture supports sandbox kernels that have both replicated and complementary services. That is, some sandboxes may have identical kernel functionality, while others partition various system components to form an asymmetric configuration. The extent to which functionality is separated across kernels is somewhat configurable in the Quest-V design. In our initial implementation, each sandbox kernel replicates most functionality, offering a private version of the corresponding services to its local application threads. Certain functionality is, however, shared across system components. In particular, we share certain driver data structures across sandboxes [3], to allow I/O requests and responses to be handled locally.

Quest-V allows *any* sandbox to be configured for corresponding device interrupts, rather than have a dedicated sandbox be responsible for all communication with that device. This greatly reduces the communication and control paths necessary for I/O requests from applications in Quest-V. It also differs from the split-driver approach taken by systems such as Xen [4], that require all device interrupts to be channeled through a special driver domain.

Sandboxes that do not require access to shared devices are isolated from unnecessary drivers and associated services. Moreover, a sandbox can be provided with its own private set of devices and drivers, so if a software failure occurs in one driver, it will not necessarily affect all other sandboxes. In fact, if a driver experiences a fault then its effects are limited to the local sandbox and the data structures shared with other sandboxes. Outside these shared data structures, remote sandboxes (including all monitors) are protected by EPTs.

Application and system services in distinct sandbox kernels communicate via shared memory channels. These channels are established by extended page table mappings setup by the corresponding monitors. Messages are passed across these channels similar to the approach in Barrelfish [5].

Main and I/O VCPUs are used for real-time management of CPU cycles, to enforce *temporal isolation*. Application and system threads are bound to VCPUs, which in turn are assigned to underlying physical CPUs. We will elaborate on this aspect of the system in the following section.

2.2 VCPU Management

In Quest-V, *virtual CPUs* (VCPUs) form the fundamental abstraction for scheduling and temporal isolation of the system. Here, temporal isolation means that each VCPU is guaranteed its share of CPU cycles without interference from other VCPUs.

The concept of a VCPU is similar to that in virtual machines [2, 4], where a hypervisor provides the illusion of multiple *physical CPUs* (PCPUs) [4] represented as VCPUs to each of the guest virtual machines. VCPUs exist as kernel abstractions to simplify the management of resource budgets for potentially many software threads. We use a hierarchical approach in which VCPUs are scheduled on PCPUs and threads are scheduled on VCPUs.

A VCPU acts as a resource container [3] for scheduling and accounting decisions on behalf of software threads. It serves no other purpose to virtualize the underlying physical CPUs, since our sandbox kernels and their applications execute directly on the hardware. In particular, a VCPU does not need to act as a container for cached instruction blocks that have been generated to emulate the effects of guest code, as in some trap-and-emulate virtualized systems.

In common with bandwidth preserving servers [1, 9, 14], each VCPU, V, has a maximum compute time budget, C_V, available in a time period, T_V. V is constrained to use no more than the fraction $U_V = \frac{C_V}{T_V}$ of a physical processor (PCPU) in any window of real-time, T_V, while running at its normal (foreground) priority. To avoid situations where PCPUs are idle when there are threads awaiting service, a VCPU that has expired its budget may operate at a lower (background) priority. All background priorities are set below those of foreground priorities to ensure VCPUs with expired budgets do not adversely affect those with available budgets.

Quest-V defines two classes of VCPUs: (1) *Main VCPUs* are used to schedule and track the PCPU usage of conventional software threads, while (2) *I/O VCPUs* are used to account for, and schedule the execution of, interrupt handlers for I/O devices. This distinction allows for interrupts

[3]Only for those drivers that have been mapped as shared between specific sandboxes.

[4]We define a PCPU to be either a conventional CPU, a processing core, or a hardware thread in a simultaneous multi-threaded (SMT) system.

from I/O devices to be scheduled as threads [17], which may be deferred execution when threads associated with higher priority VCPUs having available budgets are runnable. The flexibility of Quest-V allows I/O VCPUs to be specified for certain devices, or for certain tasks that issue I/O requests, thereby allowing interrupts to be handled at different priorities and with different CPU shares than conventional tasks associated with Main VCPUs.

2.2.1 VCPU API

VCPUs form the basis for managing time as a first-class resource: VCPUs are specified time bounds for the execution of corresponding threads. Stated another way, every executable control path in Quest-V is mapped to a VCPU that controls scheduling and time accounting for that path. The basic API for VCPU management is described below. It is assumed this interface is managed only by a user with special privileges.

- *int vcpu_create(struct vcpu_param *param)* – Creates and initializes a new Main or I/O VCPU. The function returns an identifier for later reference to the new VCPU. If the `param` argument is `NULL` the VCPU assumes its default parameters. For now, this is a Main VCPU using a `SCHED_SPORADIC` policy [15, 13]. The `param` argument points to a structure that is initialized with the following fields:

  ```
  struct vcpu_param {
     int vcpuid; // Identifier
     int policy; // SCHED_SPORADIC or SCHED_PIBS
     int mask; // PCPU affinity bit-mask
     int C; // Budget capacity
     int T; // Period
  }
  ```

  The `policy` is `SCHED_SPORADIC` for Main VCPUs and `SCHED_PIBS` for I/O VCPUs. `SCHED_PIBS` is a *priority-inheritance bandwidth-preserving* policy that is described further in Section 3.1. The `mask` is a bit-wise collection of processing cores available to the VCPU. It restricts the cores on which the VCPU can be assigned and to which the VCPU can be later migrated. The remaining VCPU parameters control the budget and period of a sporadic server, or the equivalent bandwidth utilization for a PIBS server. In the latter case, the ratio of C and T is all that matters, not their individual values.

  On success, a `vcpuid` is returned for a new VCPU. An admission controller must check that the addition of the new VCPU meets system schedulability requirements, otherwise the VCPU is not created and an error is returned.
- *int vcpu_destroy (int vcpuid, int force)* – Destroys and cleans up state associated with a VCPU. The count of the number of threads associated with a VCPU must be 0 if the `force` flag is not set. Otherwise, destruction of the VCPU will force all associated threads to be terminated.
- *int vcpu_setparam (struct vcpu_param *param)* – Sets the parameters of the specified VCPU referred to by `param`. This allows an existing VCPU to have new parameters from those when it was first created.
- *int vcpu_getparam (struct vcpu_param *param)* – Gets the VCPU parameters for the next VCPU in a list for the caller's process. That is, each process has associated with it one or more VCPUs, since it also has at least one thread. Initially, this call returns the VCPU parameters at the head of a list of VCPUs for the calling thread's process. A subsequent call returns the parameters for the next VCPU in the list. The current position in this list is maintained on a per-thread basis. Once the list-end is reached, a further call accesses the head of the list once again.
- *int vcpu_bind_task (int vcpuid)* – Binds the calling task, or thread, to a VCPU specified by `vcpuid`.

Functions *vcpu_destroy*, *vcpu_setparam*, *vcpu_getparam* and *vcpu_bind_task* all return 0 on success, or an error value.

2.2.2 Parallelism in Quest-V

At system initialization time, Quest-V launches one or more sandbox kernels. Each sandbox is then assigned a partitioning of resources, in terms of host physical memory, available I/O devices, and PCPUs. The default configuration creates one sandbox per PCPU. As stated earlier, this simplifies scheduling decisions within each sandbox. Sandboxing also reduces the extent to which synchronization is needed, as threads in separate sandboxes typically access private resources. For parallelism of multi-threaded applications, a single sandbox must be configured to manage more than one PCPU, or a method is needed to distribute application threads across multiple sandboxes.

Quest-V maintains a `quest_tss` data structure for each software thread. Every address space has at least one `quest_tss` data structure. Managing multiple threads within a sandbox is similar to managing processes in conventional system designs. The only difference is that Quest-V requires every thread to be associated with a VCPU and the corresponding sandbox kernel (without involvement of its monitor) schedules VCPUs on PCPUs.

In some cases it might be necessary to assign threads of a multi-threaded application to separate sandboxes. This could be for fault isolation reasons, or for situations where one sandbox has access to resources, such as devices, not available in other sandboxes. Similarly, threads may need to be redistributed as part of a load balancing strategy.

In Quest-V, threads in different sandboxes are mapped to separate host physical memory ranges, unless they exist in shared memory regions established between sandboxes. Rather than confining threads of the same application to shared memory regions, Quest-V defaults to using separate process address spaces for threads in different sandboxes. This increases the isolation between application threads in different sandboxes, but requires special communication channels to allow threads to exchange information.

Here, we describe how a multi-threaded application is established across more than one sandbox.

STEP 1: Create a new VCPU in parent process – Quest-V implements process address spaces using `fork/exec/exit` calls, similar to those in conventional UNIX systems. A child process, initially having one thread, inherits a *copy* of the address space and corresponding resources defined in the parent thread's `quest_tss` data structure. Forked threads differ from forked processes in that no new address space copy is made. A parent calling `fork` first establishes a new VCPU for use by the child. In all likelihood the parent will know the child's VCPU parameter requirements, but they can later be changed in the child using `vcpu_setparam`.

If the admission controller allows the new VCPU to be created, it will be established in the local sandbox. If the VCPU cannot be created locally, the PCPU affinity mask can be used to identify a remote sandbox for the VCPU. Remote sandboxes can be contacted via shared memory communication channels, to see which one, if any, is best suited for the VCPU. If shared channels do not exist, monitors can be used to send IPIs to other sandboxes. Remote sandboxes can then respond with *bids* to determine the best target. Alternatively, remote sandboxes can advertise their willingness to accept new loads by posting information relating to their current load in shared memory regions accessible to other sandboxes. This latter strategy is an *offer* to accept remote requests, and is made without waiting for *bid requests* from other sandboxes.

STEP 2: Fork a new thread or process and specify the VCPU – A parent process can now make a special `fork` call, which takes as an argument the `vcpuid` of the VCPU to be used for scheduling and resource accounting. The request can originate from a different sandbox to the one where the VCPU is located, so some means of global resolution of VCPU IDs is needed.

STEP 3: Set VCPU parameters in new thread/process – A thread or process can adjust the parameters of any VCPUs associated with it, using `vcpu_setparam`. This includes updates to its utilization requirements, and also the affinity mask. Changes to the affinity might require the VCPU and its associated process to migrate to a remote sandbox.

The steps described above can be repeated as necessary to create a series of threads, processes and VCPUs within or across multiple sandboxes. As stated in STEP 3, it might be necessary to migrate a VCPU and its associated address space to a remote sandbox. The initial design of Quest-V limits migration of Main VCPUs and associated address spaces. We assume I/O VCPUs are statically mapped to sandboxes responsible for dedicated devices.

The details of how migration is performed are described in Section 3.1. The rationale for only allowing Main VCPUs to migrate is because we can constrain their usage to threads within a single process address space. Specifically, a Main VCPU is associated with one or more threads, but every such thread is within the *same* process address space. However, two separate threads bound to different VCPUs can be part of the same or different address space. This makes VCPU migration simpler since we only have to copy the memory for one address space. It also means that within a process the system maintains a list of VCPUs that can be bound to threads within the corresponding address space. As I/O VCPUs can be associated with multiple different address spaces, their migration would require the migration, and hence copying, of potentially multiple address spaces between sandboxes. For predictability reasons, we can place an upper bound on the time to copy one address space between sandboxes, as opposed to an arbitrary number. Also, migrating I/O VCPUs requires association of devices, and their interrupts, with different sandboxes. This can require intervention of monitors to update I/O APIC interrupt distribution settings.

3 System Predictability

Quest-V uses VCPUs as the basis for time management and predictability of its sub-systems. Here, we describe how time is managed in four key areas of Quest-V: (1) scheduling and migration of threads and virtual CPUs, (2) I/O management, (3) inter-sandbox communication, and (4) fault recovery.

3.1 VCPU Scheduling and Migration

By default, VCPUs act like Sporadic Servers [13]. Sporadic Servers enable a system to be treated as a collection of equivalent periodic tasks scheduled by a rate-monotonic scheduler (RMS) [12]. This is significant, given I/O events can occur at arbitrary (aperiodic) times, potentially triggering the wakeup of blocked tasks (again, at arbitrary times) having higher priority than those currently running. RMS analysis can therefore be applied, to ensure each VCPU is guaranteed its share of CPU time, U_V, in finite windows of real-time.

Scheduling Example. An example schedule is provided in

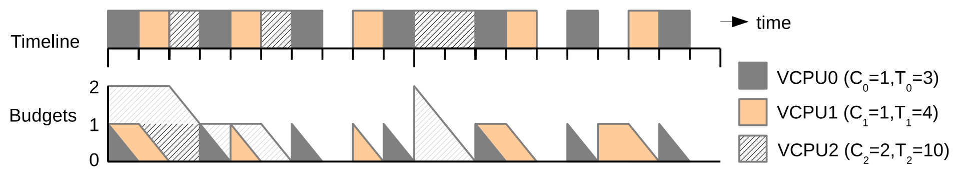

Figure 3. Example VCPU Schedule

Figure 3 for three Main VCPUs, whose budgets are depleted when a corresponding thread is executed. Priorities are inversely proportional to periods. As can be seen, each VCPU is granted its real-time share of the underlying PCPU.

In Quest-V there is no notion of a periodic timer interrupt for updating system clock time. Instead, the system is event driven, using per-processing core local APIC timers to replenish VCPU budgets as they are consumed during thread execution. We use the algorithm proposed by Stanovich et al [15] to correct for early replenishment and budget amplification in the POSIX specification.

Main and I/O VCPU Scheduling. Figure 4 shows an example schedule for two Main VCPUs and one I/O VCPU for a certain device such as a gigabit Ethernet card. In this example, Schedule (A) avoids premature replenishments, while Schedule (B) is implemented according to the POSIX specification. In (B), VCPU1 is scheduled at $t = 0$, only to be preempted by higher priority VCPU0 at $t = 1, 41, 81$, etc. By $t = 28$, VCPU1 has amassed a total of 18 units of execution time and then blocks until $t = 40$. Similarly, VCPU1 blocks in the interval $[t = 68, 80]$. By $t = 68$, Schedule (B) combines the service time chunks for VCPU1 in the intervals $[t = 0, 28]$ and $[t = 40, 68]$ to post future replenishments of 18 units at $t = 50$ and $t = 90$, respectively. This means that over the first 100 time units, VCPU1 actually receives 46 time units, when it should be limited to 40%. Schedule (A) ensures that over the same 100 time units, VCPU1 is limited to the correct amount. The problem is triggered by the blocking delays of VCPU1. Schedule (A) ensures that when a VCPU blocks (e.g., on an I/O operation), on resumption of execution it effectively starts a new replenishment phase. Hence, although VCPU1 actually receives 21 time units in the interval $[t = 50, 100]$ it never exceeds more than its 40% share of CPU time between blocking periods and over the first 100 time units it meets its bandwidth limit.

For completeness, Schedule (A) shows the list of replenishments and how they are updated at specific times, according to scheduling events in Quest-V. The invariant is that the sum of replenishment amounts for all list items must not exceed the budget capacity of the corresponding VCPU (here, 20, for VCPU1). Also, no future replenishment, R, for a VCPU, V, executing from t to $t + R$ can occur before $t + T_V$.

When VCPU1 first blocks at $t = 28$ it still has 2 units of budget remaining, with a further 18 due for replenishment at $t = 50$. At this point, the schedule shows the execution of the I/O VCPU for 2 time units. In Quest-V, threads running on Main VCPUs block (causing the VCPU to block if there are no more runnable threads), while waiting for I/O requests to complete. All I/O operations in response to device interrupts are handled as threads on specific I/O VCPUs. Each I/O VCPU supports threaded interrupt handling at a priority inherited from the Main VCPU associated with the blocked thread. In this example, the I/O VCPU runs at the priority of VCPU1. The I/O VCPU's budget capacity is calculated as the product of it bandwidth specification (here, $U_{IO} = 4\%$) and the period, T_V, of the corresponding Main VCPU for which it is performing service. Hence, the I/O VCPU receives a budget of $U_{IO} \cdot T_V = 2$ time units, and through bandwidth preservation, will be eligible to execute again at $t_e = t + C_{actual}/U_{IO}$, where t is the start time of the I/O VCPU and $C_{actual} \mid 0 \le C_{actual} \le U_{IO} \cdot T_V$ is how much of its budget capacity it really used.

In Schedule (A), VCPU1 resumes execution after unblocking at times, $t = 40$ and 80. In the first case, the I/O VCPU has already completed the I/O request for VCPU1 but some other delay, such as accessing a shared resource guarded by a semaphore (not shown) could be the cause of the added delay. Time $t = 78$ marks the next eligible time for the I/O VCPU after it services the blocked VCPU1, which can then immediately resume. Further details about VCPU scheduling in Quest-V can be found in our accompanying paper for Quest [8], a non-virtualized version of the system that does not support sandboxed service isolation.

Since each sandbox kernel in Quest-V supports local scheduling of its allocated resources, there is no notion of a global scheduling queue. Forked threads are by default managed in the local sandbox but can ultimately be migrated to remote sandboxes along with their VCPUs, according to load constraints or affinity settings of the target VCPU. Although each sandbox is isolated in a special guest execution domain controlled by a corresponding monitor, the monitor is not needed for scheduling purposes. This avoids costly virtual machine exits and re-entries (i.e., VM-Exits and VM-resumes) as would occur with hypervisors such as Xen [4] that manage multiple separate guest OSes.

Temporal Isolation. Quest-V provides temporal isolation

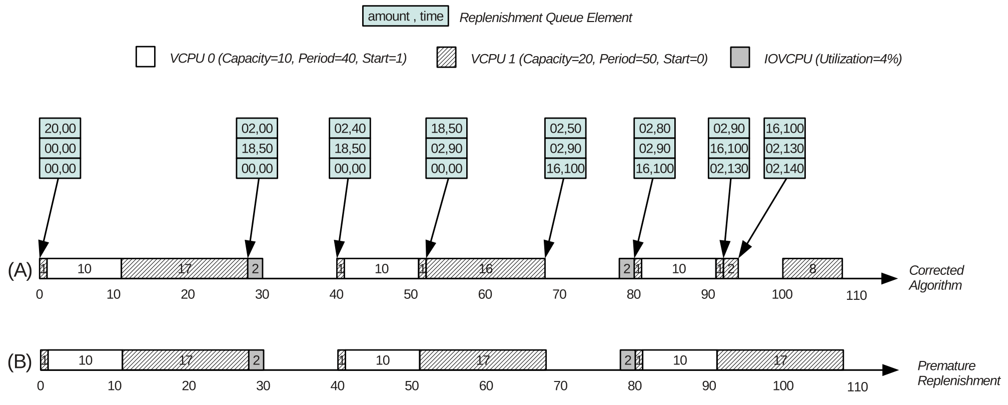
amount , time
Replenishment Queue Element
VCPU 0 (Capacity=10, Period=40, Start=1)
VCPU 1 (Capacity=20, Period=50, Start=0)
IOVCPU (Utilization=4%)
(A)
Corrected Algorithm
(B)
Premature Replenishment

Figure 4. Sporadic Server Replenishment List Management

of VCPUs assuming the total utilization of a set of Main and I/O VCPUs within each sandbox do not exceed specific limits. Each sandbox can determine the schedulability of its local VCPUs independently of all other sandboxes. For cases where a sandbox is associated with one PCPU, n Main VCPUs and m I/O VCPUs we have the following:

$$\sum_{i=0}^{n-1} \frac{C_i}{T_i} + \sum_{j=0}^{m-1} (2 - U_j) \cdot U_j \leq n \left(\sqrt[n]{2} - 1 \right)$$

Here, C_i and T_i are the budget capacity and period of Main VCPU, V_i. U_j is the utilization factor of I/O VCPU, V_j [8].

VCPU and Thread Migration. For multicore processors, the cores share a last-level cache (LLC) whose lines are occupied by software thread state from any of the sandbox kernels. It is therefore possible for one thread to suffer poorer progress than another, due to cache line evictions from conflicts with other threads. Studies have shown memory bus transfers can incur several hundred clock cycles, or more, to fetch lines of data from memory on cache misses [7, 10]. While this overhead is often hidden by prefetch logic, it is not always possible to prevent memory bus stalls due to cache contention from other threads.

To improve the global performance of VCPU scheduling in Quest-V, VCPUs and their associated threads can be migrated between sandbox kernels. This helps prevent co-schedules involving multiple concurrent threads with high memory activity (that is, large working sets or frequent accesses to memory). Similarly, a VCPU and its corresponding thread(s) might be better located in another sandbox that is granted direct access to an I/O device, rather than having to make inter-sandbox communication requests for I/O. Finally, on non-uniform memory access (NUMA) architectures, threads and VCPUs should be located close to the memory domains that best serve their needs without having to issue numerous transactions across an interconnect between chip sockets [6].

In a real-time system, migrating threads (and, in our case, VCPUs) between processors at runtime can impact the schedulability of local schedules. Candidate VCPUs for migration are determined by factors such as memory activity of the threads they support. We use hardware performance counters found on modern multicore processors to measure events such as per-core and per-chip package *cache misses*, *cache hits*, *instructions retired* and *elapsed cycles* between scheduling points.

For a single chip package, or socket, we distribute threads and their VCPUs amongst sandboxes to: (a) balance total VCPU load, and (b) balance per-sandbox LLC miss-rates or aggregate cycles-per-instruction (CPI) for all corresponding threads. For NUMA platforms, we are considering cache occupancy prediction techniques [16] to estimate the costs of migrating thread working sets between sandboxes on separate sockets.

Predictable Migration Strategy. We are considering two approaches to migration. In both cases, we assume that a VCPU and its threads are associated with *one* address space, otherwise multiple address spaces would have to be moved between sandboxes, which adds significant overhead.

The first migration approach uses shared memory to copy an address space and its associated `quest_tss` data structure(s) from the source sandbox to the destination. This allows sandbox kernels to perform the migration without involvement of monitors, which would require VM-Exit and VM-resume operations. These are potentially costly operations, of several hundred clock cycles [11]. This approach only requires monitors to establish shared memory mappings between a pair of sandboxes, by updating extended page tables as necessary. However, for address spaces that are larger than the shared memory channel we effectively have to perform a UNIX *pipe*-style exchange of information between sandboxes. This leads to a synchronous exchange, with the source sandbox blocking when the shared channel is full, and the destination blocking when awaiting more information in the channel.

In the second migration approach, we can eliminate the need to copy address spaces both *into* and *out of* shared memory. Instead, the destination sandbox is asked to move the migrating address space directly from the source sandbox, thereby requiring only one copy. However, the migrating address space and its `quest_tss` data structure(s) are initially located in the source sandbox's private memory. Hence, a VM-Exit into the source monitor is needed, to send an inter-processor interrupt (IPI) to the destination sandbox. This event is received by a remote migration thread that traps into its monitor, which can then access the source sandbox's private memory.

The IPI handler causes the destination monitor to temporarily map the migrating address space into the target sandbox. Then, the migrating address space can be copied to private memory in the destination. Once this is complete, the destination monitor can unmap the pages of the migrating address space, thereby ensuring sandbox memory isolation except where shared memory channels should exist. At this point, all locally scheduled threads can resume as normal. Figure 5 shows the general migration strategy. Note that for address spaces with multiple threads we still have to migrate multiple `quest_tss` structures, but a bound on per-process threads can be enforced.

Migration Threads. We are considering both migration strategies, using special *migration threads* to move address spaces and their VCPUs in bounded time. A migration thread in the destination sandbox has a Main VCPU with parameters C_m and T_m. The migrating address space associated with a VCPU, V_{src}, having parameters C_{src} and T_{src} should ideally be moved without affecting its PCPU share. To ensure this is true, we require the migration cost, $\Delta_{m,src}$, of copying an address space and its `quest_tss` data structure(s) to be less than or equal to C_m. T_m should ideally be set to guarantee the migration thread runs at highest priority in the destination. To ease migration analysis, it is preferable to move VCPUs with full budgets. For any VCPU with maximum tolerable delay, $T_{src} - C_{src}$, before it needs to be executed again, we require preemptions by higher priority VCPUs in the destination sandbox to be less than this value. In practice, V_{src} might have a tolerable delay lower than $T_{src} - C_{src}$. This restricts the choice of migratable VCPUs and address spaces, as well as the destination sandboxes able to accept them. Further investigation is needed to determine the schedulability of migrating VCPUs and their address spaces.

3.2 Predictable I/O Management

As shown in Section 3.1, Quest-V assigns I/O VCPUs to interrupt handling threads. Only a minimal "top half" [17]

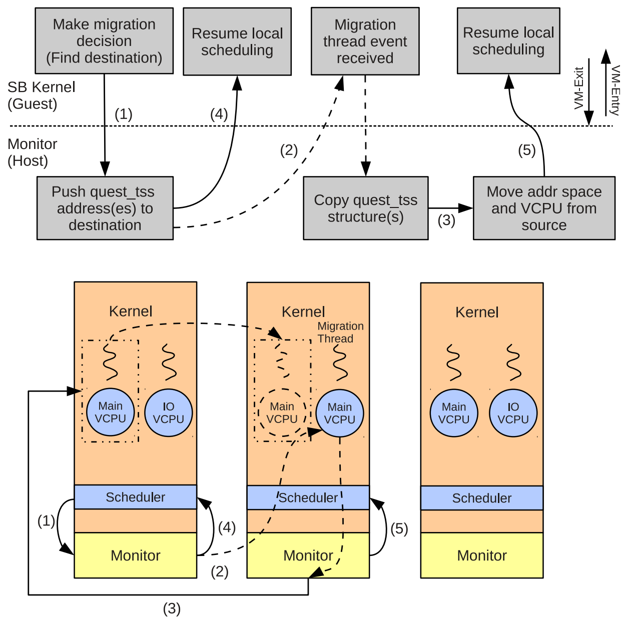
Make migration decision (Find destination)
Resume local scheduling
Migration thread event received
Resume local scheduling
VM-Exit
VM-Entry
SB Kernel (Guest)
Monitor (Host)
(1)
(4)
(2)
(5)
(3)
Push quest_tss address(es) to destination
Copy quest_tss structure(s)
Move addr space and VCPU from source
Kernel
Migration Thread
Main VCPU
IO VCPU
Scheduler
Monitor

Figure 5. Time-Bounded Migration Strategy

part of interrupt processing is needed to acknowledge the interrupt and post an event to handle the subsequent "bottom half" in a thread bound to an I/O VCPU. A worst-case bound can be placed on top half processing, which is charged to the current VCPU as system overhead.

Interrupt processing as part of device I/O requires proper prioritization. In Quest-V, this is addressed by assigning an I/O VCPU the priority of the Main VCPU on behalf of which interrupt processing is being performed. Since all VCPUs are bandwidth preserving, we set the priority of an I/O VCPU to be inversely proportional to the period of its corresponding Main CPU. This is the essence of priority-inheritance bandwidth preservation scheduling (PIBS). Quest-V ensures that the priorities of all I/O operations are correctly matched with threads running on Main VCPUs, although such threads may block on their Main VCPUs while interrupt processing occurs. To ensure I/O processing is bandwidth-limited, each I/O VCPU is assigned a specific percentage of PCPU time. Essentially, a PIBS-based I/O VCPU operates like a Sporadic Server with *one* dynamically-calculated replenishment.

This approach to I/O management prevents live-lock and priority inversion, while integrating the management of interrupts with conventional thread execution. It does, however, require correctly matching interrupts with Main VCPU threads. To do this, Quest-V's drivers support *early demultiplexing* to identify the thread for which the interrupt has occurred. This overhead is also part of the top half cost described above.

Finally, Quest-V programs I/O APICs to multicast device interrupts to the cores of sandboxes with access to those devices. In this way, interrupts are not always directed to one core which becomes an I/O server for all others. Multicast interrupts are filtered as necessary, as part of early demultiplexing, to decide whether or not subsequent I/O processing should continue in the target sandbox.

3.3 Inter-Sandbox Communication

Inter-sandbox communication in Quest-V relies on message passing primitives built on shared memory, and asynchronous event notification mechanisms using Interprocessor 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 shadow 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. 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.

3.4 Predictable Fault Recovery

Central to the Quest-V design is fault isolation and recovery. Hardware virtualization is used to isolate sandboxes from one another, with monitors responsible for mapping sandbox virtual address spaces onto (host) physical regions.

Quest-V supports both local and remote fault recovery. Local fault recovery attempts to restore a software component failure without involvement of another sandbox. The local monitor re-initializes the state of one or more compromised components, as necessary. The recovery procedure itself requires some means of fault detection and trap (VM-Exit) to the monitor, which we assume is never compromised. Remote fault recovery makes sense when a replacement software component already exists in another sandbox, and it is possible to use that functionality while the local sandbox is recovered in the background. This strategy avoids the delay of local recovery, allowing service to be continued remotely. We assume in all cases that execution of a faulty software component can correctly resume from a recovered state, which might be a re-initialized state or one restored to a recent checkpoint. For checkpointing, we require monitors to periodically intervene using a preemption timeout mechanism so they can checkpoint the state of sandboxes into private memory.

Here, we are interested in the predictability of fault recovery and assume the procedure for correctly identifying faults, along with the restoration of suitable state already exists. These aspects of fault recovery are, themselves, challenging problems outside the scope of this paper.

In Quest-V, predictable fault recovery requires the use of *recovery threads* bound to Main VCPUs, which limit the time to restore service while avoiding temporal interference with correctly functioning components and their VCPUs. Although recovery threads exists within sandbox kernels the recovery procedure operates at the monitor-level. This ensures fault recovery can be scheduled and managed just like any other thread, while accessing specially trusted monitor code. A recovery thread traps into its local monitor and guarantees that it can be de-scheduled when necessary. This is done by allowing local APIC timer interrupts to be delivered to a monitor handler just as they normally would be delivered to the event scheduler in a sandbox kernel, outside the monitor. Should a VCPU for a recovery thread expire its budget, a timeout event must be triggered to force the monitor to upcall the sandbox scheduler. This procedure requires that wherever recovery takes place, the corresponding sandbox kernel scheduler is not compromised. This is one of the factors that influences the decision to perform local or remote fault recovery.

When a recovery thread traps into its monitor, VM-Exit information is examined to determine the cause of the exit. If the monitor suspects it has been activated by a fault we need to initialize or continue the recovery steps. Because recovery can only take place while the sandbox recovery thread has available VCPU budget, the monitor must be preemptible. However, VM-Exits trap into a specific monitor entry point rather than where a recovery procedure was last executing if it had to be preempted. To resolve this issue, monitor preemptions must checkpoint the execution state so that it can be restored on later resumption of the monitor-level fault recovery procedure. Specifically, the common entry point into a monitor for all VM-Exits first examines the reason for the exit. For a fault recovery, the exit handler will attempt to restore checkpointed state if it exists from a prior preempted fault recovery stage. This is all assuming that recovery cannot be completed within one period (and budget) of the recovery thread's VCPU. Figure 6 shows how the fault recovery steps are managed predictably.

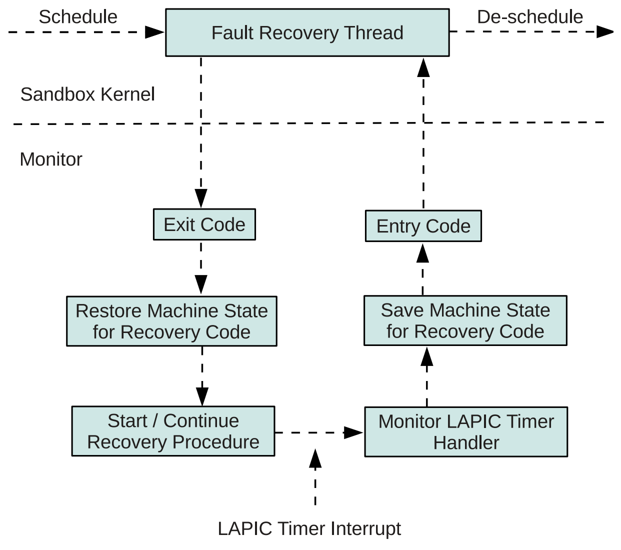
Schedule
Fault Recovery Thread
De-schedule
Sandbox Kernel
Monitor
Exit Code
Entry Code
Restore Machine State for Recovery Code
Save Machine State for Recovery Code
Start / Continue Recovery Procedure
Monitor LAPIC Timer Handler
LAPIC Timer Interrupt

Figure 6. Time-Bounded Fault Recovery

4 Conclusions and Future Work

This paper describes time management in the Quest-V real-time multikernel. We show through the use of virtual CPUs with specific time budgets how several key subsystem components behave predictably. These sub-system components relate to on-line fault recovery, communication, I/O management, scheduling and migration of execution state.

Quest-V is being built from scratch for multicore processors with hardware virtualization capabilities, to isolate sandbox kernels and their application threads. Although Intel VT-x and AMD-V processors are current candidates for Quest-V, we expect the system design to be applicable to future embedded architectures such as the ARM Cortex A15. Future work will investigate fault detection schemes, policies to identify candidate sandboxes for fault recovery, VCPU and thread migration, and also load balancing strategies on NUMA platforms.

References

[1] L. Abeni, G. Buttazzo, S. Superiore, and S. Anna. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-time Systems Symposium*, pages 4–13, 1998.

[2] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 2–13, New York, NY, USA, 2006.

[3] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: a new facility for resource management in server systems. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, 1999.

[4] 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 *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[5] 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*, pages 29–44, 2009.

[6] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova. A case for NUMA-aware contention management on multicore processors. In *USENIX Annual Technical Conference*, 2011.

[7] 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 *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 43–57, 2008.

[8] M. Danish, Y. Li, and R. West. Virtual-CPU Scheduling in the Quest Operating System. In *the 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2011.

[9] Z. Deng, J. W. S. Liu, and J. Sun. A scheme for scheduling hard real-time applications in open system environment. In *Proceedings of the 9th Euromicro Workshop on Real-Time Systems*, 1997.

[10] U. Drepper. *What Every Programmer Should Know About Memory*. Redhat, Inc., November 21 2007.

[11] Y. Li, M. Danish, and R. West. Quest-V: A virtualized multikernel for high-confidence systems. Technical Report 2011-029, Boston University, December 2011.

[12] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[13] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems Journal*, 1(1):27–60, 1989.

[14] M. Spuri, G. Buttazzo, and S. S. S. Anna. Scheduling aperiodic tasks in dynamic priority systems. *Real-Time Systems*, 10:179–210, 1996.

[15] 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.

[16] R. West, P. Zaroo, C. A. Waldspurger, and X. Zhang. Online cache modeling for commodity multicore processors. *Operating Systems Review*, 44(4), December 2010. Special VMware Track.

[17] Y. Zhang and R. West. Process-aware interrupt scheduling and accounting. In *the 27th IEEE Real-Time Systems Symposium*, December 2006.

[18] C. Zimmer and F. Mueller. Low contention mapping of real-time tasks onto a TilePro 64 core processor. In *the 18th IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2012.