Distributed Real-Time Fault Tolerance on a Virtualized Multi-Core System

Eric Missimer, Ye Li and Richard West

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

Chip-level multiprocessors (CMPs) are increasingly being used in real-time and embedded systems, due in part to their power, performance and cost benefits. New opportunities exist to build fault-tolerant, safety-critical systems that leverage the redundancy of separate cores on these processing platforms. In this work, we describe a new chip-level distributed real-time system, called Quest-V.

Quest-V uses hardware virtualization to partition machine resources (processor cores, memory, and I/O devices) amongst separate *sandboxes*. Here, a sandbox is similar to a traditional (guest) virtual machine but, for the most part, it operates independently of an underlying hypervisor (a.k.a., virtual machine monitor, or VMM). In Quest-V, each sandbox is bootstrapped by a corresponding monitor that grants access to specific physical resources. For the most part, resources are statically partitioned amongst sandboxes, so a monitor is not needed to schedule the execution of separate guests on available cores, as is done in traditional hypervisors. A trusted monitor is only needed when shared memory communication channels are established between sandboxes, or fault recovery procedures need to be performed.

Quest-V is intended for use in systems where faults can occur either because of software errors, or partial hardware failures caused by factors such as radiation. If a fault occurs in a safety-critical system, it is essential that all critical services remain operational, or available. Quest-V attempts to maintain high availability by performing real-time, on-line fault detection and recovery. It does this using a consensus protocol to capture the states of replicated processes running in separate sandboxes. Any processes not part of the consensus are deemed to have malfunctioned and are restored to the states of other functional replicas. Recovered processes are *rolled forward* to the most recent checkpointed state of replica processes that are deemed to be working correctly.

Applications requiring high availability use a special `sync` system call, to periodically checkpoint their state and invoke the consensus protocol. When a `sync` call occurs, the user space memory of each application instance is hashed on a per-page basis and the resulting hashes are placed into memory shared between the local sandbox and an *arbitrator* sandbox. A hash is only collected if the page is modified since the last `sync` call. The arbitrator collects the hashes and performs the consensus. It then sends responses to each sandbox, indicating whether or not the corresponding replica process can proceed past its checkpoint. If a replica passes the checkpoint it can free any copied pages it was keeping for remote recovery. If it fails at a checkpoint the arbitrator signals that it needs to recover.

Part of the recovery process involves making a virtual machine call to enter into a sandbox monitor. While in the monitor, pages from a correct process instance in a remote sandbox are copied to the recovering replica. Upon return from the monitor, the virtual machine state is changed so that it is in the `sync` system call, which in turn restores control back to the recovered process.

As long as each replicated process does not receive input that would vary across instances of execution, such as calling a read time-stamp counter function, the memory state across all well-behaving instances should be the same. This approach relies on copying and hashing memory pages but a faulty operating system cannot bring down the entire system, as only monitor code can modify memory in another sandbox.

The Quest-V fault tolerance approach is intended for safety-critical applications in areas such as avionics, automotive systems, robotics, healthcare and manufacturing. We are currently developing an autonomous vehicle system, called RacerX, which uses real-time sensor data inputs to influence path planning and motion control. Other applications include the safe, predictable, and fault-tolerant management of unmanned aerial vehicles (UAVs) and robots deployed in space.

We will demonstrate Quest-V using live examples of autonomous vehicle control, for both simulated and real-world scenarios. Demonstrations will include using Quest-V to manage the AI engine of a TORCS [1] simulated vehicle system. We will show that a vehicle avoids collisions with its surroundings even when faults are injected into the AI engine.

[1] TORCS is "The Open Racing Car Simulator".

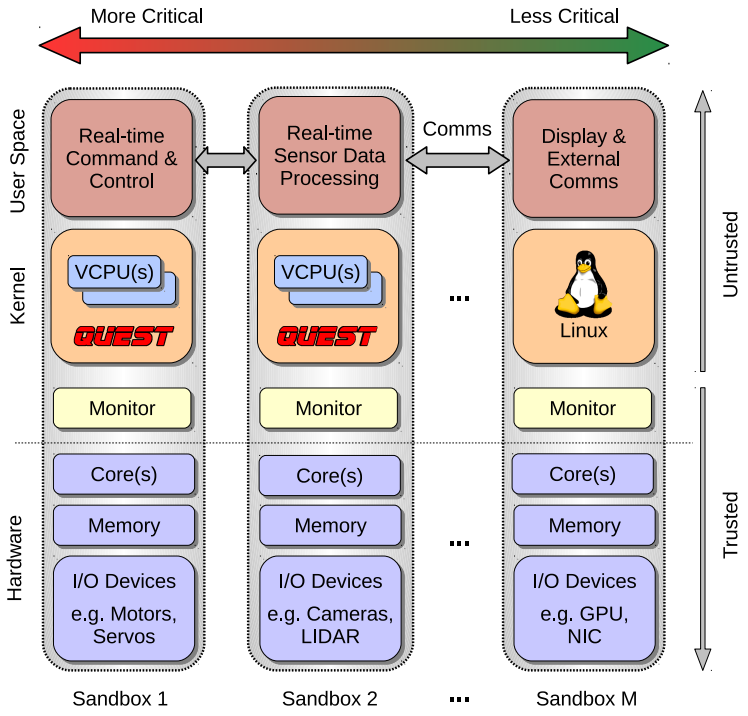

Fig. 1: Example Quest-V Architecture Overview

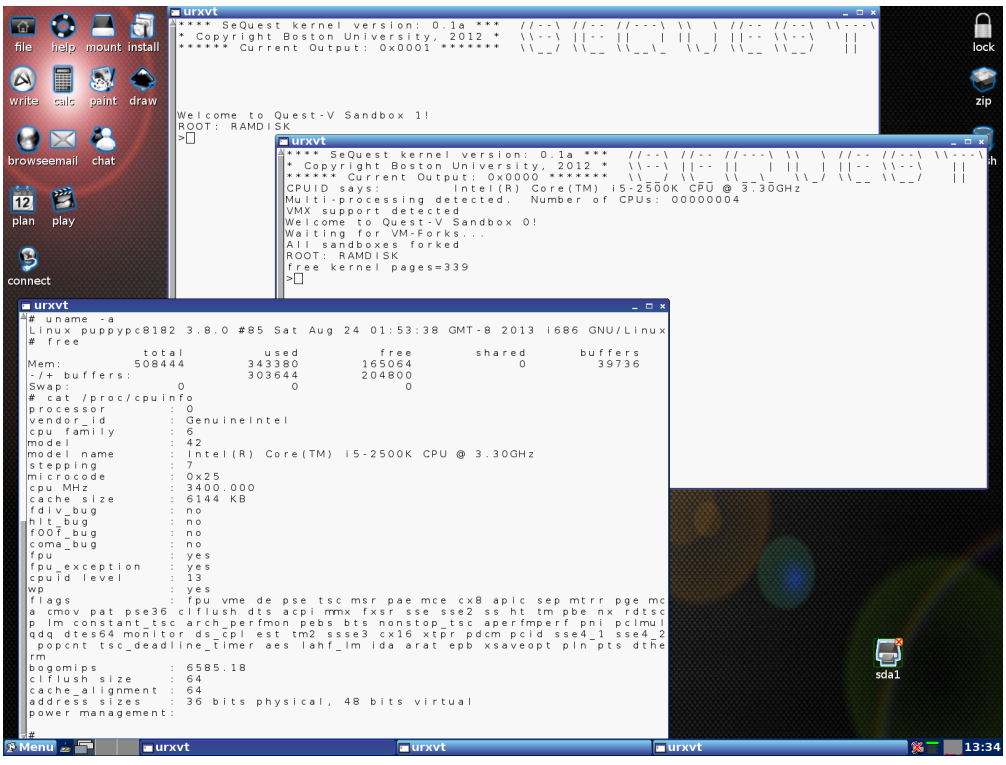

Fig. 2: Quest-V with Linux Front-End