Exploiting Two-Case Delivery for Fast Protected Messages

Bill Kramer
February 13, 2002

The Basic Problem

- How to provide “efficient, direct application access to network hardware …while gaining … the benefits of memory based network interfaces”
  - Direct access to the network interface provides a “fast path” for data to the application
  - Can introduce protection and deadlock issues
- Solution is to transparently switch between different modes of operation
  - Two case delivery
    - One with direct application control of network interfaces
    - One with buffering of messages
      - Use Virtual buffering
        - Efficient use of physical memory
        - Afford VM protection

Why does this work?

- The direct fast path is the common case.
  - The solution is to optimize the common case
  - Make efficient the less common case of buffering
- Have to virtualize and protect the access to the network interface
  - Have to do this with reasonable efficiency
- Uses a light weight messages protocol – User Direct Messaging (UDM) to have every incoming message invoke a user data handler.

Related Work Discussion

- Model
  - Active Messages
  - Remote Queues
- Direct Network Devices
  - CM-5
  - J-Machine
  - *T Interface
  - Alewif
  - CNI
- Memory Based Interfaces
  - Provides low overhead when network hardware not expensive and latency is not an issue
- Hybrids
User Direct Messaging Model

- **Messages and operations**
  - Operations are atomic
    - No partial packets
    - Inject, extract, peek, etc.
  - Messages are asynchronous and unacknowledged
    - Once a message is in the network, it can assume to eventually get delivered
- **Atomicity**
  - Low overhead, virtualized interrupt disable
  - Allows user code to use interrupt driven and/or polling reception
  - Timeout to protect against deadlock and abuse

Two Case Delivery

- **Direct Access Path**
  - Extract either reads from the network device or the buffer depending on where messages are
  - Transparently switched
- **Receipt notification switched and traps or polling**
- **Physical and virtual atomicity**
  - Physical – disable the actual queue
  - Virtual – buffer messages in memory but does not perturb the active process

Protection

- Applied at the receiver
- Uses GID to label messages
  - GID labels groups of processes
  - Hardware enforced
- Interrupt Disable revocable
  - Binds how long a user application controls the device
  - Timer counts how long an application controls the interface
  - Messages that are blocked too long – lack of progress
- Reserved second network to avoid deadlock
  - May be much slower alternative network
  - May be virtual channel reserved on primary physical network

Objectives

- **Transparency** – same semantics as fast case
  - Protection and context switching
  - Three reasons to switch from fast to buffered
    - Page faults in handler
    - Atomicity timeouts
    - Scheduler quantum expirations
  - Management of physical memory
    - Using VM allows virtually unlimited buffer pool
  - Guarantee delivery
    - Avoids deadlock to buffering space
    - Allocates only on demand
- **Overhead is due to extra copies**
  - Buffered null message takes 2.7 times as long as fast path

Virtual Buffering
**Performance and Experiments**

- Used 5 applications
  - 3 assumed to be computational intensive
  - Ran on simulator with 2 nodes
    - Average cycles between messages ran from 615 to 14,200
    - Average cycles in handler 149 to 478
  - Hardware measurements indicated times within 20% of simulation estimates in paper
  - Uses the gang schedule feature of FUGU to introduce “skew”
    - Creates message mismatches and forces buffer mode
- Used one custom benchmark to assess the sensitivity of performance to overhead in the buffered case

**Performance Conclusions**

- Applications tend to avoid buffered mode
  - If there is frequent synchronization of sends and receives
  - Physical memory requirements is low
- Virtual buffering improves performance
  - Avoids unnecessary use of physical memory
- Buffered vs. direct trades of scheduling flexibility
- Buffered case limits throughput of certain program styles
  - Buffered overhead changes the tradeoff point
- Application demand for buffering is relatively small and increases gracefully

**Discussion**

- Are there sufficient mechanisms to assure message protections and deadlock prevention?
- The 5 benchmark are more computational based. Will there be different issues with other workloads?
  - Are you convinced that the majority of applications require only a small number of physical pages?
- Is there sufficient evidence that this implementation will scale to large numbers of processors
  - Performance results done on 2 and 4 processors
- How does one assess this mechanism opposite other mechanisms based on the data provided?