EbbRT: A framework for building per-application library operating systems

Introduction

- A small introduction on library operating systems
    - Operating system level features now implemented at application level
    - Application now has better control of resources and hence can be made more efficient
- Not only dennard scaling is broken, now the size of transistors also reaching limit

Objectives

- MultiLibOS model
- Function offloading, offload functionalities to remote nodes
- IOMMU
    - Have a separate MMU for the device too so that the device can translate the virtual address into physical address itself

System Design

- Distinction between native and hosted runtimes
    - Native - low level
    - Hosted - proces level
- Hosted library is used for the application to make a smooth transition from hosted library to native runtime
- Native runtime sets up a single address space while the applications runs at the highest privilege level
- To provide high degree of customization Ebbs enable developers to modify or extend the entire software stack
- Ebbs are distributed, multicore fragments
- RCU - read, copy, update
    - Reads occur concurrently with updates
    - Contrast this with simple lock - all operations are mutually exclusive, reader-writer lock - concurrent reads are allowed but not in presence of a write
    - Maintains multiple versions of objects and removes stale versions for which there are no readers present (for that version)

    - Reads and writes are done through publish subscribe mechanism
- Non preemptive execution model
    - Non-preemptive does not imply that interrupts are disabled
    - It is just that the interrupt handler returns back to execution and control flow is resumed
- Poll - polls the set of descriptors in a loop
    - Pro - low context switch overheads
    - Con - wastes cpu cycles when idle
    - Used when activity is high
- Select - asks system to notify in case of an event
    - Pro - sleeps when idle
    - Con - context switch overhead present
    - Used when activity is low
- Cooperative threading model
    - Event driven programming may not be a good fit for all applications
    - User saves and restores state at will
    - Needed to support wide applicability

Implementation

- Application can control scheduling decisions by modifying EventManager Ebb. Look at table 1
- A new stack is used for each exception handler
- Spawn method can register an event in any core but executes at a later point in time. This is similar to that of a signal abstraction which internally uses Inter Processor Interrupts at the lowest level to deliver signals to other cores.
- Ebbs use page tables to store references to local representatives
    - Dereferencing an ebb internally walks through page table which is a hash map as a data structure but an efficient one to obtain the reference to local representative
    - Hosted implementations do not have access to page tables and hence use software hash tables to store representatives
- Static dispatch is that function to be computed or data to be accessed is determined at compile time and hence the offsets into objects are also constants determined at compile time
- Dynamic dispatch is different in the sense that function to be executed is determined at runtime. An example can be runtime polymorphism in c++ where the function to be executed depends on nature of object and hence cannot be determined at compile time
- Main disadvantage of interface definition language being is that is used in EbbRT serialization and deserialization takes place irrespective of whether representative is present locally or not. This slows down even the fast path which is undesirable
- Parallel executions are not straight right intuitive and are relatively difficult to develop than their serial counterparts. By adapting higher level programming abstractions like

lambdas and monadic futures into EbbRT we make it relatively easier for the programmer to reason about program

- Lambdas is an abstraction to make construction of continuations easy
- Monadic futures allow programmers to linearly build up futures which is more neat and intuitive in terms of control flow than having nested callbacks which is outright messy
- Depending on programming language monadic futures may also have abstractions which allow programmers to execute functions/lambdas if any of the results are available or all of the results are available depending on the requirement
- Instead of having a single buffer to store all the network data we use a chain of IOBufs assisted by scatter gather interfaces

Evaluation

- In threadtest benchmark, the performance when number of objects allocated in one iteration is large is worse than when number of iterations is large because of overheads caused by memory management logic. Look at Figure 4
- EbbRT is more performant than glibc because of lack of operating system overheads and finer control on memory management
- Glibc does not scale as well as EbbRT because of synchronization overheads which is not present in EbbRT due to its locality and non preemptive nature. Jemalloc scales well since it uses per-thread caches
- A low throughput or high latency may also arise from heavy CPU usage which is generally due to large software stacks. EbbRT applications specialize software stack to improve performance
- Linux VM inefficiency also comes from the fact that Linux is not developed to run on a VM, while EbbRT is specifically designed keeping cloud applications in mind

Conclusion

- EbbRT adapts programming level abstractions to system level software. It uses objects to create software stacks, C++ templates for generic EbbRefs, lambdas for continuations and monadic futures to handle exceptions