Preview of Operating Systems and Hardware Session

Michio Honda  
University of Edinburgh

Sudarsun Kannan  
Rutgers University
Memory Scaling Challenges

→ Scaling application memory capacity without increasing management cost is becoming critical

→ Scaling memory protection and isolation for large address space equally critical

1. Memory tuning is tedious
   Requires extensive application knowledge and requires constant tuning

2. Memory security is challenging
   Hardware memory security non-scalable
Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator

A.H. Hunter  
*Jane Street Capital*

Chris Kennelly  
Google

Paul Turner  
Google

Darryl Gove  
Google

Tipp Moseley  
Google

Parthasarathy Ranganathan  
Google
Memory Allocator Challenges

- Long history of memory allocators designed for specific application needs
  - Concurrency and low fragmentation (e.g., Hoard, jemalloc, TCMalloc)
  - Minimize L1 misses (e.g., Dice), increase locality (e.g., mimallloc)

- However, allocators are not optimized for HugePages
  - HugePages becoming increasingly ubiquitous in large scale applications
  - Could substantially reduce TLB misses by increasing RAM coverage

- Using existing allocators with HugePages could increase fragmentation and are inefficient for warehouse scale systems running several applications
Hugepage-aware user-level allocator using TCMALLOC

Aims at densely packing huge pages grouped into few, saturated bins

Balances memory usage and page allocation costs through adaptive huge page release

Average 6% reduction TLB misses and 26% reduction in memory usage across a fleet of applications
Scalable Memory Protection in the PENGLAI Enclave

Erhu Feng†‡, Xu Lu†‡, Dong Du†‡, Bicheng Yang†‡, Xueqiang Jiang†‡, Yubin Xia†§‡, Binyu Zang†§§, Haibo Chen†§§

† Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
§ Shanghai AI Laboratory
‡ Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China
Hardware Enclaves 101

→ Hardware abstractions and support for trusted execution on untrusted platforms

→ Hardware enclaves: secure boot, on-chip program isolation, protected external memory, execution integrity, and other capabilities
Hardware Enclaves Challenges

→ Non-scalable memory partition/isolation
  • Current hardware supports only 256MB enclaves
  • Some restrict the number of enclaves
  • Require static partitioning

→ Non-scalable memory integrity protection
  • Huge memory overhead to store memory integrity information (e.g., hash)
  • Hardware (e.g., Intel SGX) only supports ~256MB, demands swapping

→ Non-scalable secure memory initialization
  • High-cost secure memory initialization increases enclave setup cost
  • Impractical for serverless applications
PENGLAI Enclave

- Scalable secure memory protection mechanisms for enclaves
- Approach to Scaling: novel *Guarded Page Table* structure
- Guarded Page Table Intuition: map secure and unsecure pages to separate non-secure host page table and secure enclave page table

- Scaling Integrity Protection: Mountable Merkle Tree (MMT), a SubTree structure to reduce both on-die and in-memory storage overhead
(NrOS and nanoPU)
OS evolution over HW generations

- Single core
  - Just protect critical sections from interrupts
  - I/O was also slow
OS evolution over HW generations

- Single core
  - Just protect critical sections from interrupts
  - I/O was also slow
- Multiple CPU cores
  - Giant lock
  - Fine-grained locks
  - Reader-writer locks
OS evolution over HW generations

- **Single core**
  - Just protect critical sections from interrupts
  - I/O was also slow

- **Multiple CPU cores**
  - Giant lock
  - Fine-grained locks
  - Reader-writer locks
OS evolution over HW generations

- **Single core**
  - Just protect critical sections from interrupts
  - I/O was also slow

- **Multiple CPU cores**
  - Giant lock
  - Fine-grained locks
  - Reader-writer locks
OS evolution over HW generations

- Single core
  - Just protect critical sections from interrupts
  - I/O was also slow
- Multiple CPU cores
  - Giant lock
  - Fine-grained locks
  - Reader-writer locks
- Multiple CPU packages (sockets)
  - NUMA-aware memory allocation and scheduling
OS evolution over HW generations

- **Single core**
  - Just protect critical sections from interrupts
  - I/O was also slow

- **Multiple CPU cores**
  - Giant lock
  - Fine-grained locks
  - Reader-writer locks

- **Multiple CPU packages (sockets)**
  - NUMA-aware memory allocation and scheduling

- **Fast I/O**
  - Interrupt mitigation and load-balancing
  - New APIs
    (kqueue/epoll/netmap/io_uring)
OS evolution over HW generations

- **Single core**
  - Just protect critical sections from interrupts
  - I/O was also slow

- **Multiple CPU cores**
  - Giant lock
  - Fine-grained locks
  - Reader-writer locks

- **Multiple CPU packages (sockets)**
  - NUMA-aware memory allocation and scheduling

- **Fast I/O**
  - Interrupt mitigation and load-balancing
  - New APIs (kqueue/epoll/netmap/io_uring)

All of these make kernel code complex and error-prone, but such a kernel is still not scalable!
**NrOS**

<table>
<thead>
<tr>
<th>Design</th>
<th>Synchronization</th>
<th>Kernel programming</th>
<th>Scalability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monolithic</td>
<td>Shared states</td>
<td>Hard</td>
<td>Low</td>
</tr>
<tr>
<td>Multikernel</td>
<td>Message passing</td>
<td>Easy</td>
<td>Low</td>
</tr>
<tr>
<td>NrOS</td>
<td>Operation logs</td>
<td>Easy</td>
<td>High</td>
</tr>
</tbody>
</table>

- Use of shared last-level CPU cache
- Operation logs shared by per-NUMA-node replicas
- Synchronization batching
- NetBSD LibOS
  - POSIX app support

**Abstract**

Writing a correct operating system kernel is notoriously hard and error-prone. A number of concurrent data structures have been designed over the years to achieve thread-level scaling. For example, lock-free algorithms have been developed to provide scalable, read-mostly, cache-friendly, and low-overhead memory access. However, they still require a sophisticated management of shared states.

In this paper, we present NrOS, a new operating system design that addresses the challenges of concurrent programming. NrOS provides a simple and easy-to-use interface for message passing among kernel replicas, allowing for efficient and scalable operation.

---

1. University of Utah
2. University of British Columbia
3. VMware Research
4. PIPPS
**nanoPU**

- **Co-designing NIC and CPU**
  - NIC places receiving data directly in a CPU register file
- **Ultrafast small RPCs (nanoRequests)**
  - High-rate small requests are hard to handle, because most overheads are per-packet or per-request, NOT per bytes
  - nanoPU reduces both average and tail latency

- **Design highlights**
  - Avoid the two latency sources:
    - Host stack
      - Bypass the stack and memory hierarchy
    - Queues in networks
    - Transport protocol in HW
Operating Systems and Hardware Session

Thursday, July 15
7:00 am–8:15 am (PDT)