

Multikernel: Kernel-to-Kernel Isolation with Elastic Resource Management

Cong Wang

Founder and CEO, Multikernel Technologies, Inc.

ROSS Workshop, SC25

Scale Amplifies Problems

```
Time: 0ms    50ms    100ms    150ms    200ms    250ms    300ms
Node 0: [====COMPUTE====] [x] [-] [====COMPUTE====]
Node 1: [====COMPUTE====] [-] [x] [====COMPUTE====]
Node 2: [====COMPUTE====] [x] [-] [====COMPUTE====]
Node 3: [====COMPUTE====] [-] [x] [====COMPUTE====]
        Barrier synchronization delayed by OS jitter →
```

One slow node blocks thousands

HPC Challenge

Modern HPC workloads face a critical dilemma:

- **Predictable performance** for large-scale parallel applications
- **Full Linux compatibility** for tools, productivity, and monitoring
- **Dynamic resource management** for heterogeneous workloads
- **Isolation** between co-located applications

Introducing Multikernel

Core Concept: Run multiple Linux kernels simultaneously without virtualization

Key Benefits:

- Each application gets its own tailored kernel instance
- Hardware resources dynamically allocated between kernels
- Full Linux compatibility, isolation without virtualization overhead

Previous Multikernel Research

Barrelfish (ETH Zurich, 2009)

- Replica-based for single system image
- State replication overhead

Popcorn Linux (Virginia Tech, 2013)

- Process migration between kernels, still single system image
- Focus on heterogeneous ISA

McKernel (RIKEN, 2015)

- Custom lightweight kernel from scratch, ~200 syscalls (44% coverage)
- Two-tier architecture: performance kernel + service kernel

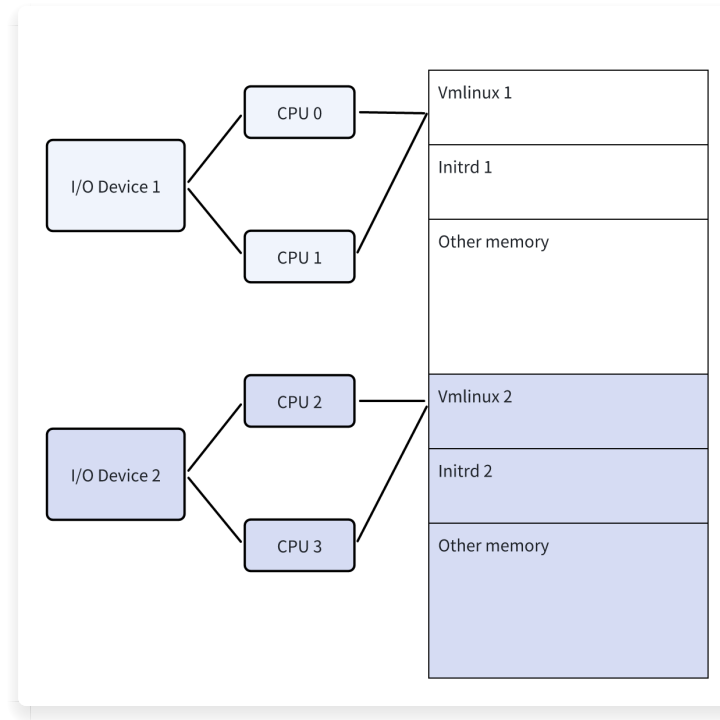
Our Design: Kernel-to-Kernel Isolation

Core Principle:

- *Each application gets its own tailored Linux kernel*
- *With dedicated and elastic hardware resources*

HPC App	ML Training	Data Analytics
Linux Kernel (HPC-tuned)	Linux Kernel (GPU-optimized)	Linux Kernel (I/O-optimized)
CPU Cores DRAM	CPU + GPU HBM	CPU + NVMe Storage

Kernel Isolation Architecture



Tailored Linux Kernel

McKernel's burden:

- Custom kernel from scratch
- ~200 syscalls -> rest offloaded via IPC
- Linux kernel is more than just syscall's (eBPF + kernel modules)

Our approach:

- Configure out unnecessary subsystems (Kconfig)
- Upstream Linux + multikernel features enabled
- Tailored and optimized for each application
- Result: 100% compatible upstream Linux with better performance

Static Partitioning vs Dynamic Allocation

From our perspective:

Static partitioning is merely the default state of dynamic resource allocation, but the reverse is not true.

For HPC/Cloud:

- Dynamic resource management is *not optional*, but essential
- Challenge: achieve dynamism while maintaining isolation

Resource Management

Key insight: Resource management is critical for Multikernel architecture

- Our innovation is using Device Tree for multikernel resource management
- Device Tree is the established standard for hardware resource description in embedded systems
- Device Tree Overlays enable dynamic hardware configuration changes at runtime

Device Tree Example

```
/multikernel-v1/;
/ {
    compatible = "linux,multikernel";

    instances {
        hpc-compute {
            id = <1>;
            resources {
                cpus = <8>;                // 8 cores
                memory-bytes = <0x1000000000>; // 64GB
                devices = <&mlx5_nic>;      // InfiniBand
            };
        };
    };

    mlx5_nic: ethernet@0 {
        pci-id = "0000:03:00.0";
        vendor-id = <0x15b3>; // Mellanox
        device-id = <0x1017>; // ConnectX-5
    };
};
```

```
};  
};
```



Multikernel
TECHNOLOGIES

Multikernel Technologies, Inc.

Device Tree Overlay Example

```
/multikernel-v1/;

/ {
    fragment@0 {
        target-path = "/";
        __overlay__ {
            memory-add {
                mk,instance = "hpc-compute";
                region@0 {
                    reg = <0x0 0x100000000 0x0 0x80000000>; /* 2GB */
                    numa-node = <0>;
                };
            };

            cpu-add {
                mk,instance = "hpc-compute";
                cpu@16 { reg = <16>; numa-node = <0>; };
                cpu@17 { reg = <17>; numa-node = <0>; };
                cpu@18 { reg = <18>; numa-node = <0>; };
                cpu@19 { reg = <19>; numa-node = <0>; };
            };
        };
    };
};
```

```
};  
};  
};  
};
```

Dynamic Resource Allocation

Allocation Process:

1. Userspace generates overlay describing changes
2. Kernel translates to resource update messages
3. Sends via IPI inter-kernel communication
4. Target kernel applies via hotplug operations

Leverages existing Linux infrastructure: CPU/memory/PCI hotplug

Hardware Device Sharing Challenge

Key Challenge: I/O devices are typically fewer compared to CPU cores

SR-IOV:

- Hardware-level isolation requires virtualization
- Limited VF (Virtual Function) count
- Inflexible resource allocation

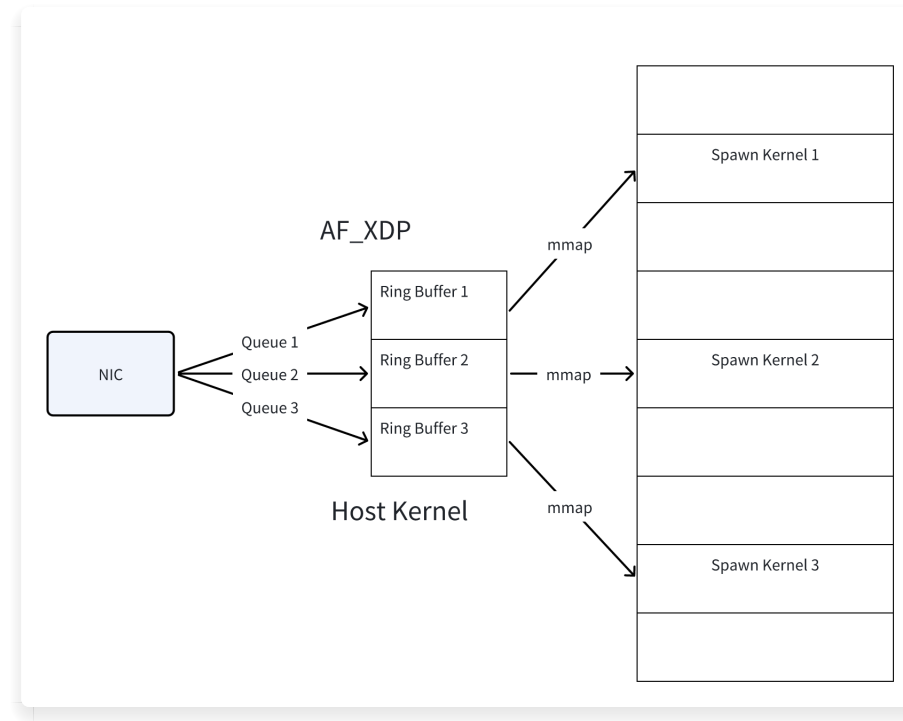
We need hardware isolation without virtualization overhead

Our Solution: Hardware Queue Isolation

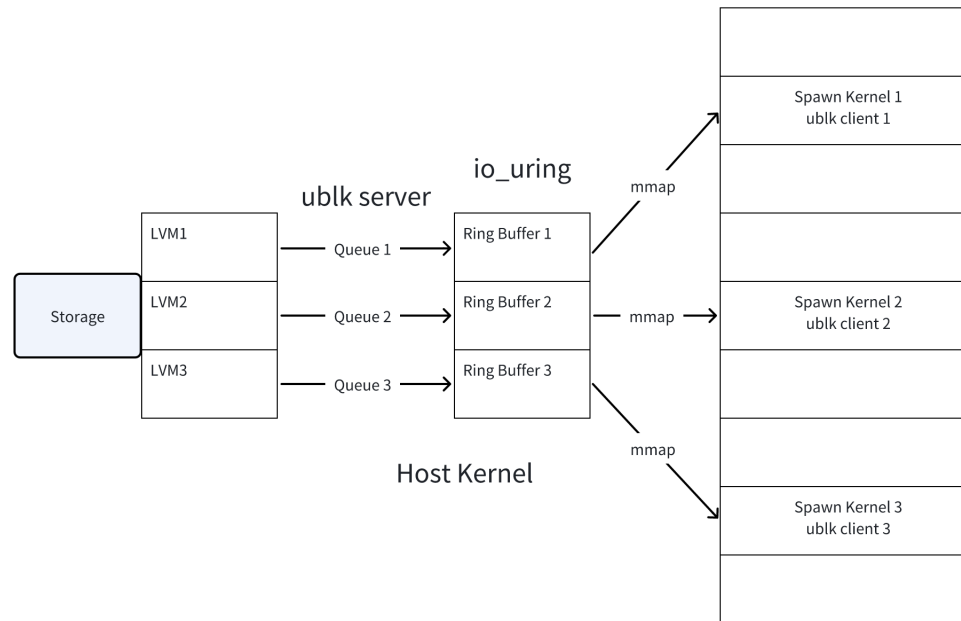
Key insight: Modern I/O devices support multiple hardware queues

- Each kernel gets exclusive access to specific queues
- Leverages `AF_XDP` and `io_uring` ring buffer and zero copy
- With UAPI guarantee, even stronger than kernel compatibility
- Dynamic queue assignment is easier than hotplug

Reuse **AF_XDP** for Networking



Reuse **ublk** for Storage



Hardware Queue Device Tree

Device tree configuration:

```
mlx5_nic: ethernet@0 {  
    pci-id = "0000:03:00.0";  
    vendor-id = <0x15b3>; // Mellanox  
    device-id = <0x1017>; // ConnectX-5  
  
    queue_group_0: queues@0 {  
        queue-ids = <0 1 2 3>; // Queues 0-3  
    };  
  
    queue_group_1: queues@1 {  
        queue-ids = <4 5 6 7>; // Queues 4-7  
    };  
};
```

HPC Use Case 1: Large-Scale MPI

Problem: OS noise amplifies at scale through barrier synchronization

- 8000 nodes × 64 cores = 512,000 MPI ranks
- All on shared Linux → synchronized jitter catastrophe

Solution: One kernel instance per job

- Dedicated resources
- No cross-job interference
- Predictable timing

HPC Use Case 2: Co-located Services

Problem: Traditional HPC workflows require expensive data movement between simulation and analytics phases

Solution: Run simulation and analytics kernels on the same node

- Simulation kernel: Dedicated resources for compute-intensive modeling
- Analytics kernel: Real-time data processing and visualization
- Shared Memory: Zero-copy data exchange via `AF_SMC` transparently

HPC Use Case 3: AI/ML Workload Consolidation

Problem: AI/ML pipelines have conflicting resource requirements

Solution:

- Run training and inference kernels concurrently
- Guaranteed performance boundaries prevent interference
- Adjust CPU/GPU resources based on workload priority

Use Case: Live Kernel Update

Traditional kernel update:

- Node reboot required (minutes of downtime)
- All applications interrupted
- Scheduling disruption across cluster

Google's LUO (Live Update Orchestrator):

- Reduces but doesn't eliminate downtime
- All processes pause during kexec transition
- All-or-nothing state transfer

Our Approach: Parallel Kernel Execution

Run old and new kernels *simultaneously*

1. Boot the new kernel alongside the old one
2. Gradually migrate processes from old to new kernel one by one
3. Retire the old kernel once all processes have moved to the updated version
4. Could rollback at any point

Result: Zero application interruption, *truly* zero downtime

Use Case: Kernel Auto-Healing

Traditional approach:

- Kdump kernel captures vmcore, cannot run real services
- Must reboot *twice* (minutes of downtime)

Our approach: Active-Backup Mode

1. Backup kernel detects via heartbeat timeout
2. Takes over hardware devices, restores process state from saved memory
3. Resumes service in **sub-second time**

HPC Implications

For large HPC clusters:

- Rolling update clusters without job disruption
- Critical security patches deployed immediately
- Reduced operational overhead and maintenance complexity
- Transforms from "fail-stop" to "fail-operational"

Security Model

Implications:

- Each kernel trusted to respect its resource boundaries
- Reduced attack surface through kernel customization

Suitable for:

- Multi-tenant systems within same organization
- HPC centers with controlled environments
- Dedicated devices with strict kernel control

Trade-off: Performance over absolute security isolation

Request for Hardware Enhancements

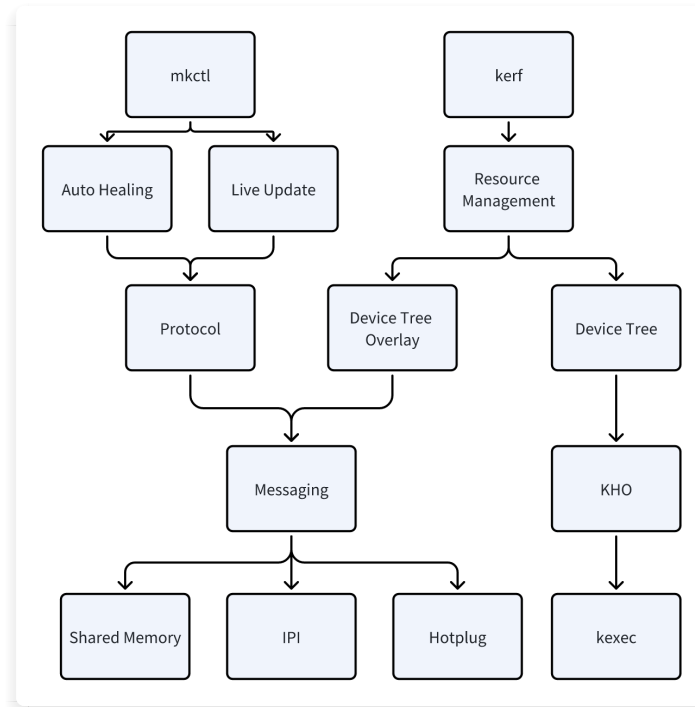
CHERI (Capability Hardware Enhanced RISC Instructions)

- Fine-grained memory protection via hardware capabilities
- Bounded pointers with enforced ranges

Hardware-Filtered IPI

- Currently, any CPU can send IPI to any other CPU
- Hardware-level IPI ACL without software overhead

Putting Together



Multikernel vs Alternatives

	Linux	Containers	VMs	McKernel	Multikernel
Linux Compatibility	Full	Full	Full	Partial	Full
Isolation	None	Low	Strong	Moderate	Moderate
Performance	Native	Native	-20-40%	Native	Native
Customization	No	No	Yes	No	Yes
Dynamic Resources	N/A	Yes	Yes*	Yes	Yes
Zero-Downtime Updates	No	Yes*	Yes*	No	Yes

*With orchestration overhead

Research Opportunities

Open questions we're exploring:

- Automatic kernel tailoring
- Automatic kernel optimizations based on profiles
- Machine learning based kernel optimizations
- Zero-downtime kernel upgrade
- Confidential computing for Multikernel

Call For Collaboration

What you get:

- Direct input on design decisions
- Co-authorship on research papers
- Free consulting during deployment
- Direct open source contributions
- Linux kernel community recognition

Questions?

Email: cwang@multikernel.io

Website: multikernel.io

Projects: github.com/multikernel

Demo: youtube.com/@multikernel-tech

Join the mailing list: multikernel@lists.linux.dev