Improve Load Balancing with Machine Learning Techniques based on *sched_ext* Framework

國立成功大學
National Cheng Kung University

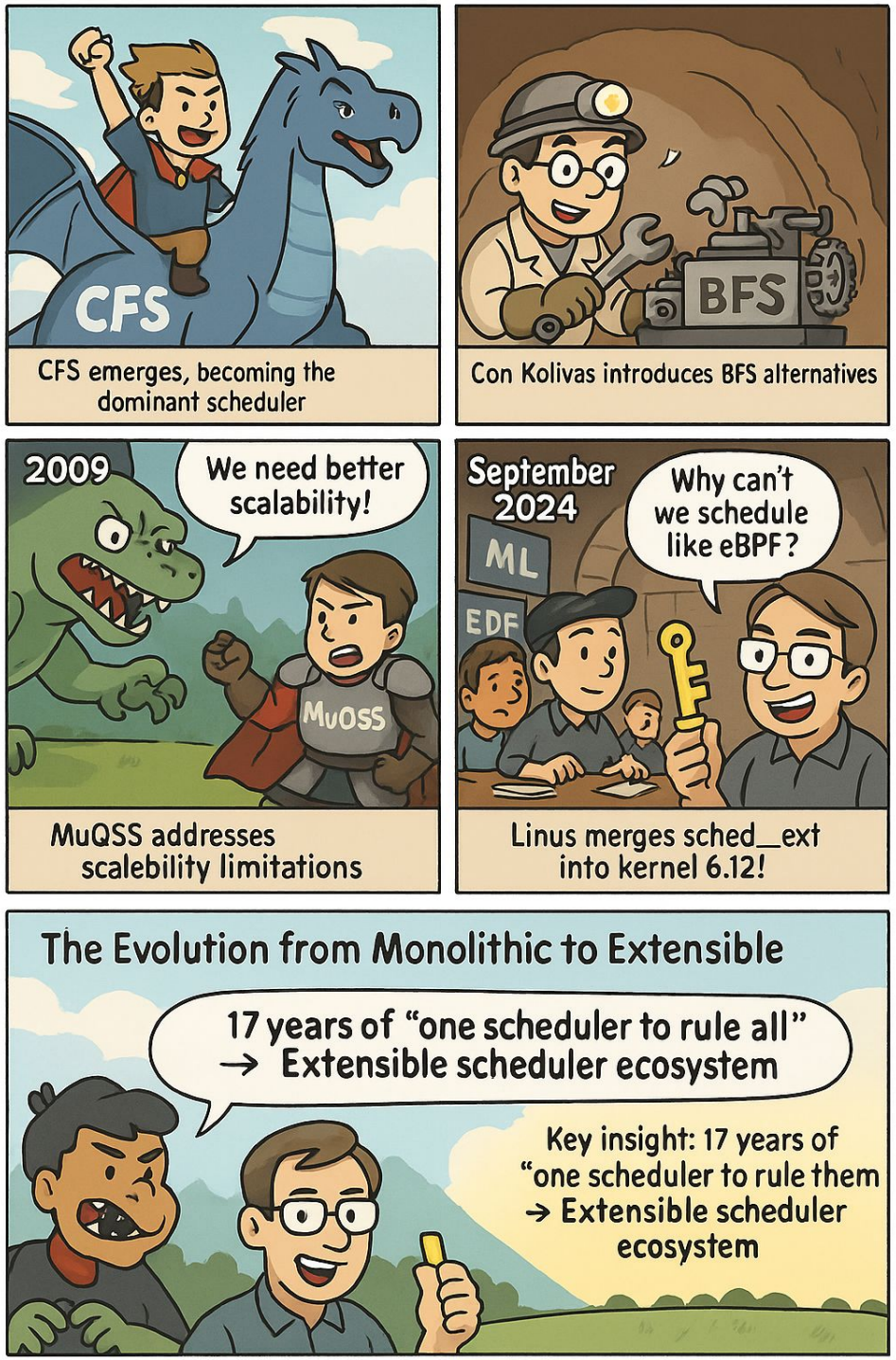
Ching-Chun (Jim) Huang
w/ I Hsin Cheng, Cheng-Yang Chou, Po-Ying Chiu
June 23, 2025
Open Source Summit North America

CPU Scheduler Timeline

The Evolution from Monolithic to Extensible

- **2007**: Ingo Molnar's CFS dominance begins
- **2009**: Con Kolivas introduces BFS alternatives
- **2016**: MuQSS addresses scalability limitations
- **2023**: Peter Zijlstra replaces CFS with EEVDF
- **2022-2024**: sched_ext development journey
- **September 2024**: Linus merges *sched_ext* into kernel 6.12

Key Insight: 17 years of "one scheduler to rule them all" → Extensible scheduler ecosystem



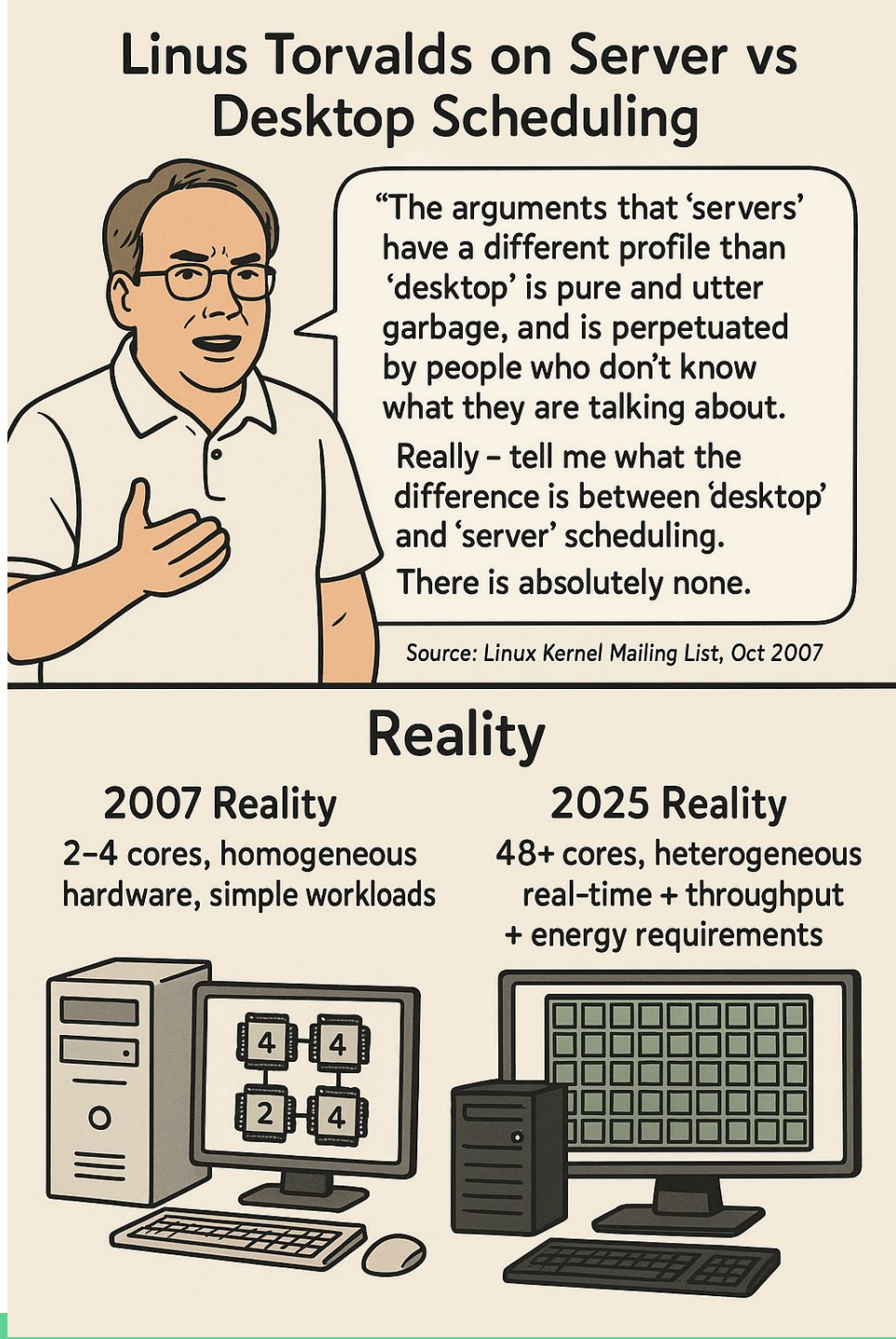
CPU Scheduler Philosophy

Linus Torvalds on *Server vs Desktop Scheduling*

> *"The arguments that 'servers' have a different profile than 'desktop' is **pure and utter garbage**, and is perpetuated by people who don't know what they are talking about."*
>
> *"Really - tell me what the difference is between 'desktop' and 'server' scheduling. There is **absolutely none**."*
>
> Source: Linux Kernel Mailing List, Oct 2007

Reality

- 2007 Reality: 2-4 cores, homogeneous hardware, simple workloads
- 2025 Reality: 48+ cores, heterogeneous, real-time + throughput + energy requirements

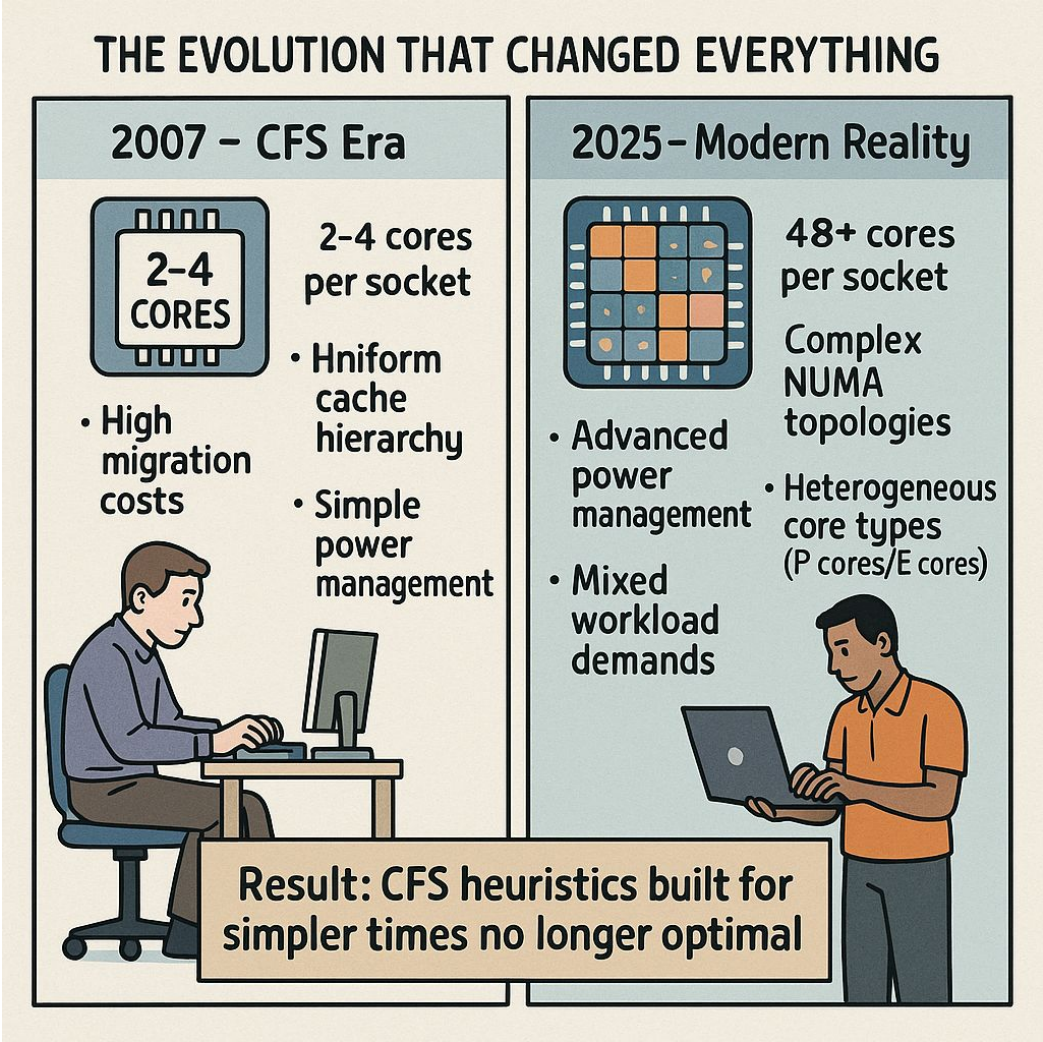


Hardware Complexity: 2007 vs. 2025

The Evolution That Changed Everything

- 2007 - CFS Era
  - 2-4 cores per socket
  - Uniform cache hierarchy
  - High migration costs
  - Simple power management
- 2025 - Modern Reality
  - 48+ cores per socket
  - Complex NUMA topologies
  - Heterogeneous core types (P-cores/E-cores)
  - Advanced power management
  - Mixed workload demands

Result: CFS heuristics built for simpler times no longer optimal



NUMA (Non-Uniform Memory Access)

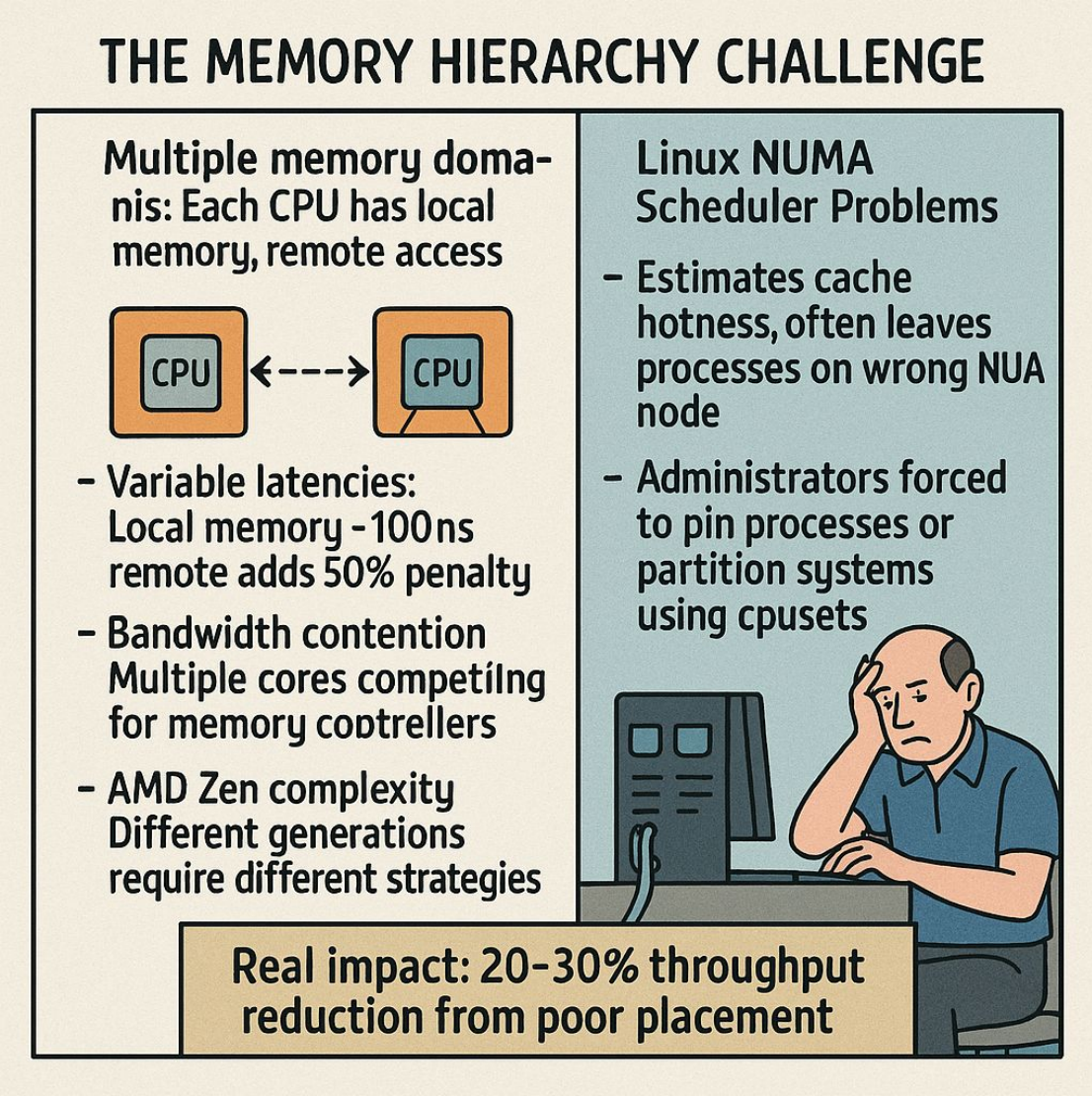
The Memory Hierarchy Challenge

- Multiple memory domains: Each CPU has local memory, remote access slower
- Variable latencies: Local memory ~100ns, remote adds 50% penalty
- Bandwidth contention: Multiple cores competing for memory controllers
- AMD Zen complexity: Different generations require different strategies

Linux NUMA Scheduler Problems

- Estimates cache hotness, often leaves processes on wrong NUMA node
- Administrators forced to pin processes or partition systems using cpusets

Real impact: 20-30% throughput reduction from poor placement



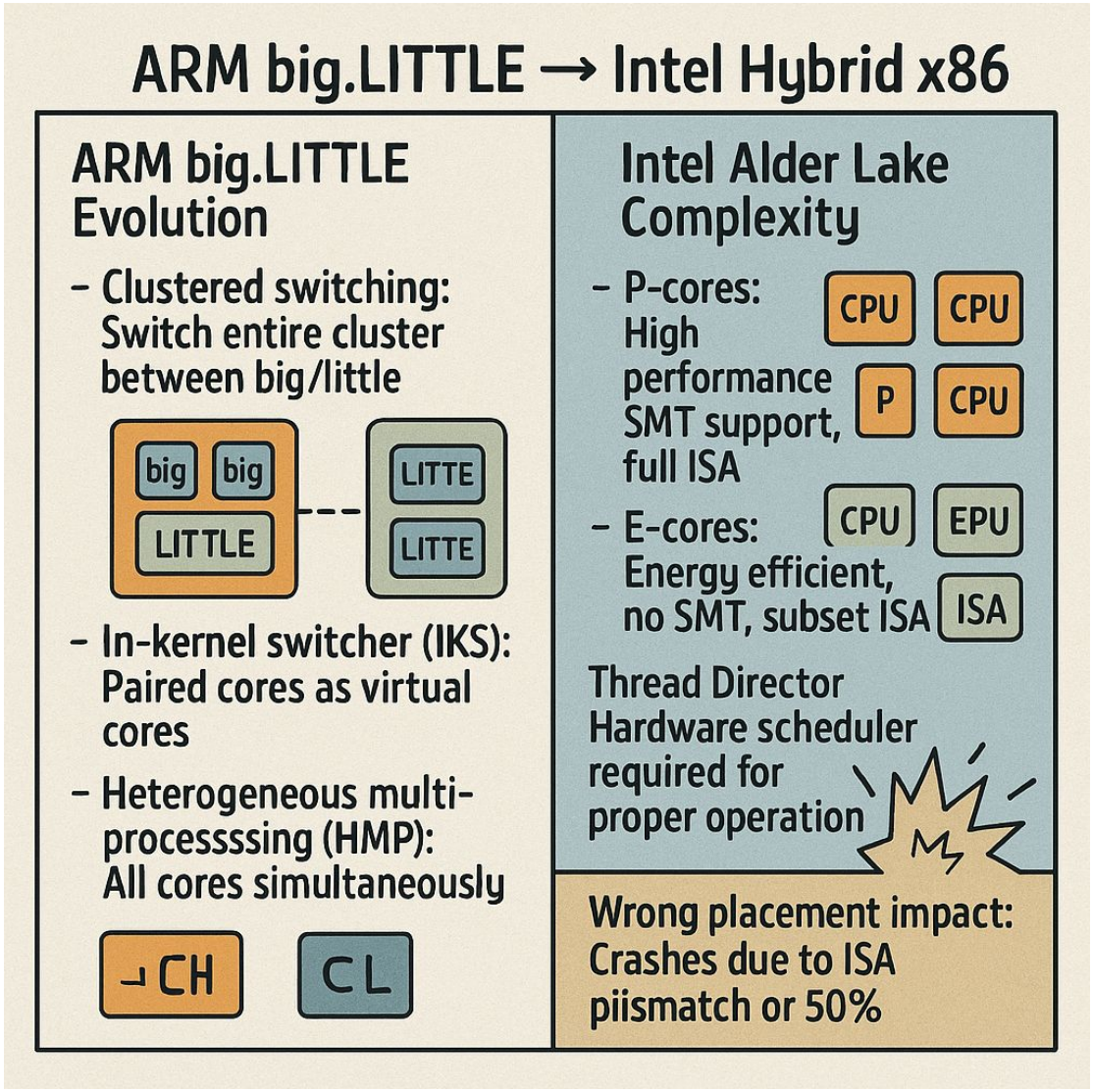
Heterogeneous Computing Crisis

Arm big.LITTLE Evolution

- **Clustered switching**: Switch entire cluster between big/little
- **In-kernel switcher (IKS)**: Paired cores as virtual cores
- **Heterogeneous multi-processing (HMP)**: All cores simultaneously

Intel Alder Lake Complexity

- **P-cores**: High performance, SMT support, full ISA
- **E-cores**: Energy efficient, no SMT, subset ISA
- **Thread Director**: Hardware scheduler required for proper operation
- **Wrong placement impact**: Crashes due to ISA mismatch or 50% performance loss



Mixed Workload Reality

Real-World Scenarios Breaking Traditional Schedulers

Gaming + Streaming

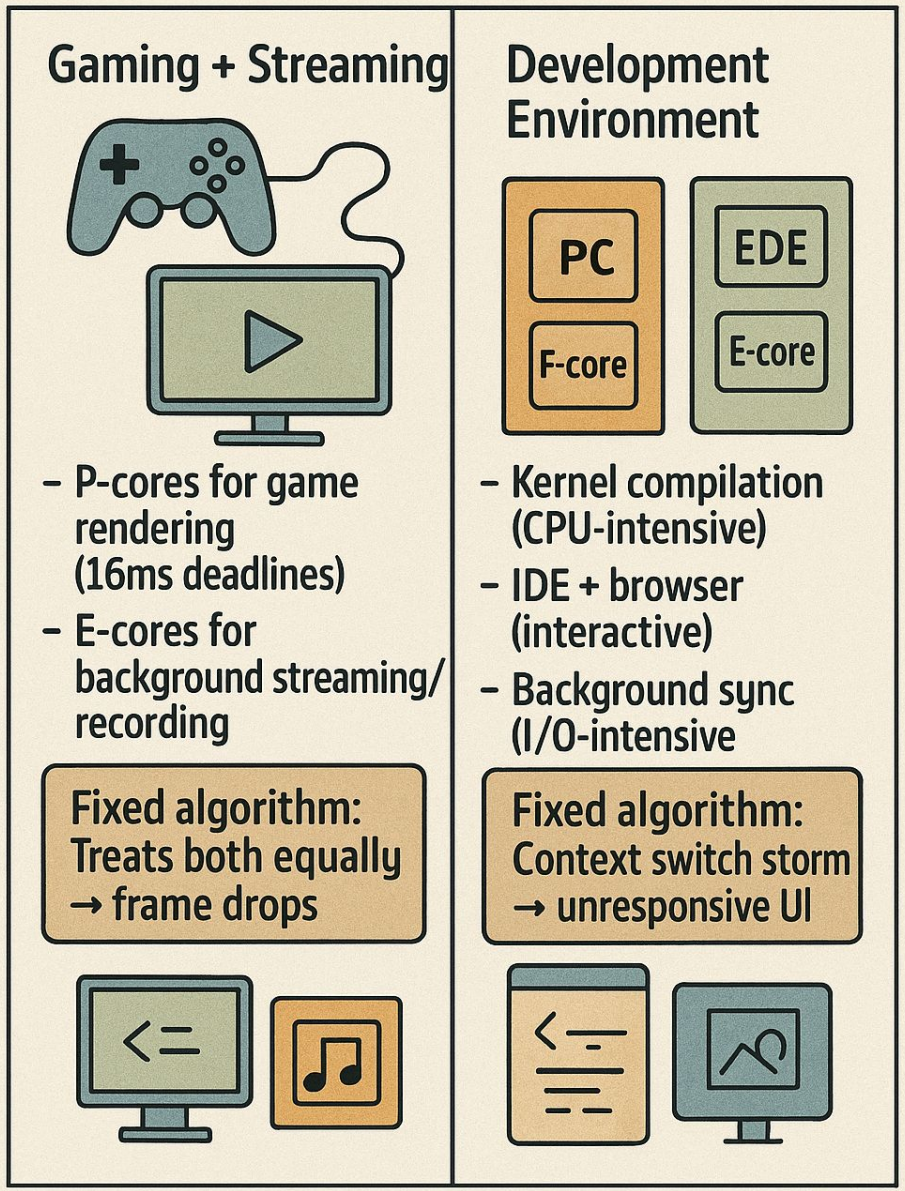
- P-cores for game rendering (16ms deadlines)
- E-cores for background streaming/recording
- Fixed algorithm: Treats both equally → frame drops

5G Network Processing

- RAN packet processing (5ms deadlines)
- Core network signaling (100ms deadlines)
- System monitoring (1s deadlines)
- Fixed algorithm: No deadline awareness → call failures

Development Environment

- Kernel compilation (CPU-intensive)
- IDE + browser (interactive)
- Background sync (I/O-intensive)
- Fixed algorithm: Context switch storm → unresponsive UI



Single Paradigm Problem

Why Fixed Algorithms Are Fundamentally Broken

Every Traditional Scheduler Uses One Fixed Algorithm

- **CFS**: Virtual runtime fairness for everything
- **EEVDF**: Earliest virtual deadline for everything
- **BFS**: Single queue, earliest deadline for everything

Fundamental Limitations

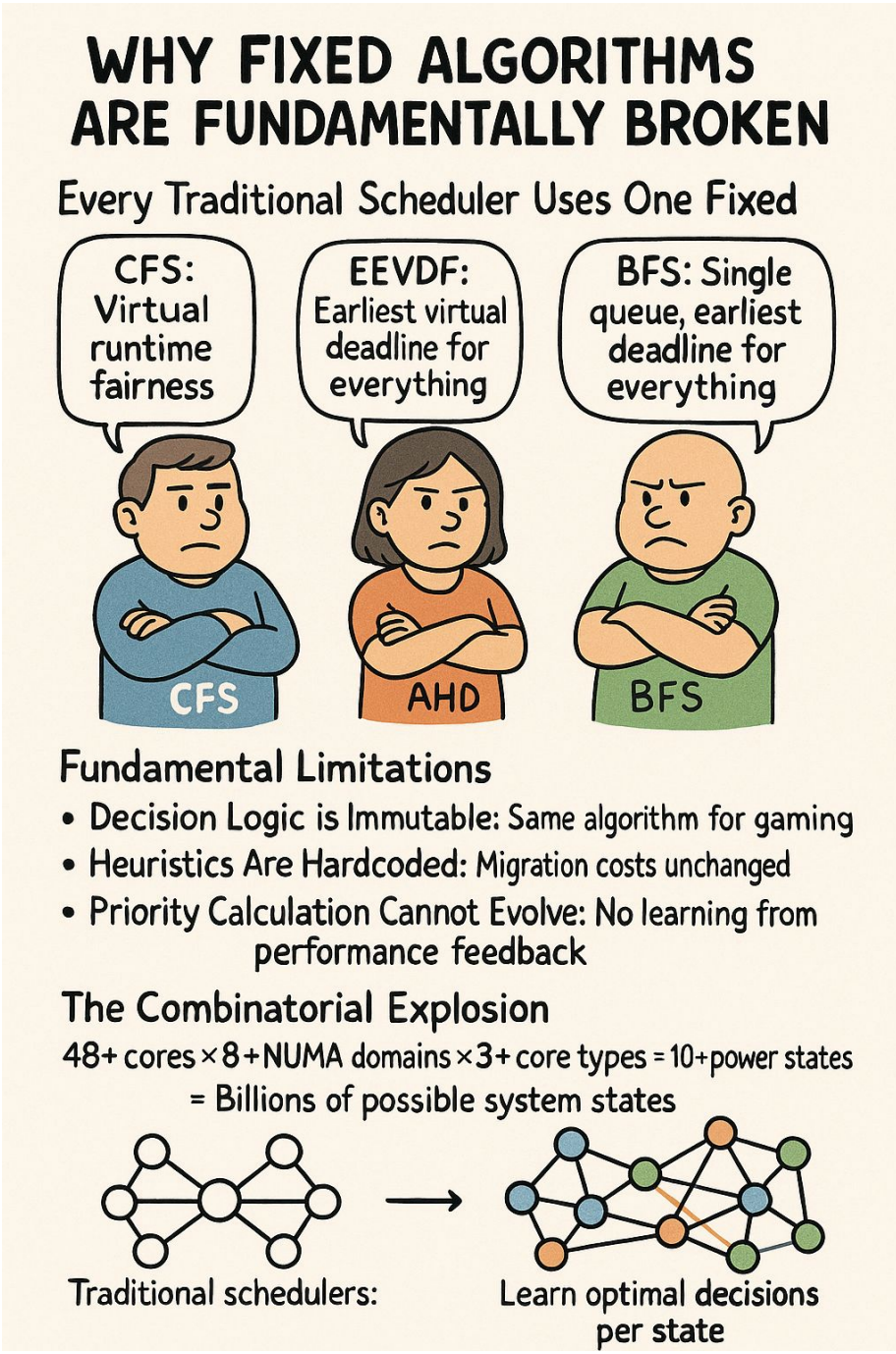
1. **Decision Logic is Immutable**: Same algorithm for gaming and servers
2. **Heuristics Are Hardcoded**: Migration costs unchanged since 2007
3. **Priority Calculation Cannot Evolve**: No learning from performance feedback

The Combinatorial Explosion

48+ cores × 8+ NUMA domains × 3+ core types × 10+ power states = Billions of possible system states

Traditional schedulers: Same decision logic for all states

ML schedulers: Learn optimal decisions per state



Real-World Performance Impact

Why Fixed Algorithms Break

Gaming Performance

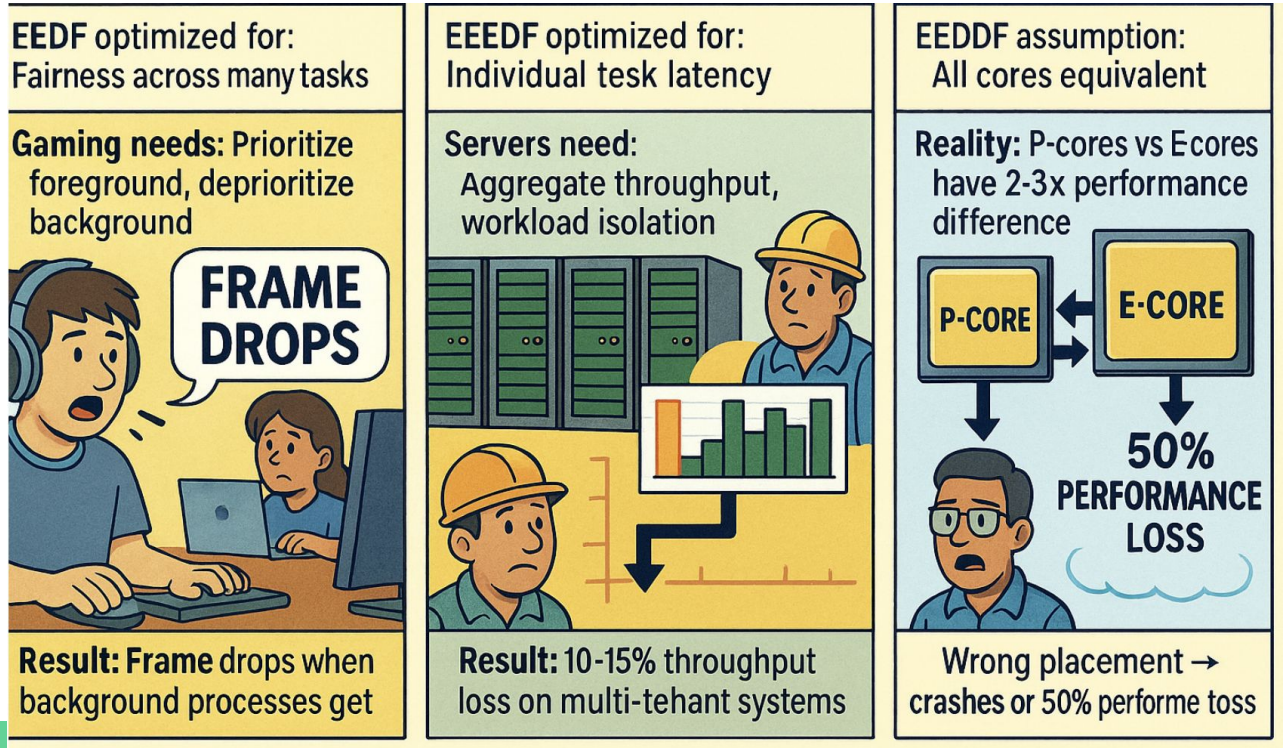
- **EEVDF optimized for**: Fairness across many tasks
- **Gaming needs**: Prioritize foreground, deprioritize background
- **Result**: Frame drops when background processes get equal CPU time

Server Workload

- **EEVDF optimized for**: Individual task latency
- **Servers need**: Aggregate throughput, workload isolation
- **Result**: 10-15% throughput loss on multi-tenant systems

Intel Hybrid Cores

- **EEVDF assumption**: All cores equivalent
- **Reality**: P-cores vs E-cores have 2-3x performance difference
- **Impact**: Wrong placement → crashes or 50% performance loss



Birth of *sched_ext*

From Rejection to Revolution (2022-2024)

The Underground Movement

- **2022-2023**: Meta/Google engineers submit patchsets
- **Initial response**: Rejected by scheduler maintainer Peter Zijlstra
- **Underground culture**: custom scheduler community
- **Production breakthrough**: Meta massive scale + Steam Deck adoption

September 30, 2024: The Victory (Linux v6.12-rc1)

What Changed the Tide

1. **Proven production use**: Large-scale deployments showing real benefits
2. **Safety mechanisms**: BPF verifier ensures no kernel crashes
3. **Innovation enablement**: Rapid scheduler development without reboots
4. **Community momentum**: Growing ecosystem of contributors and users

Technical Architecture of *sched_ext*

Framework Design

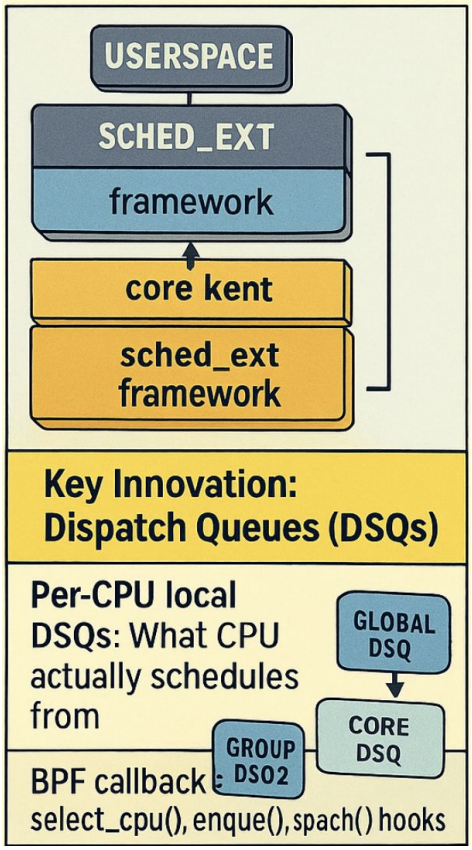
- **New scheduling class**: SCHED_EXT between SCHED_IDLE and SCHED_NORMAL
- **Safety first**: Cannot take over system completely
- **BPF-powered**: Write schedulers as BPF programs
- **Four-layer architecture**: Core kernel → sched_ext framework → BPF scheduler → userspace

Key Innovation: Dispatch Queues (DSQs)

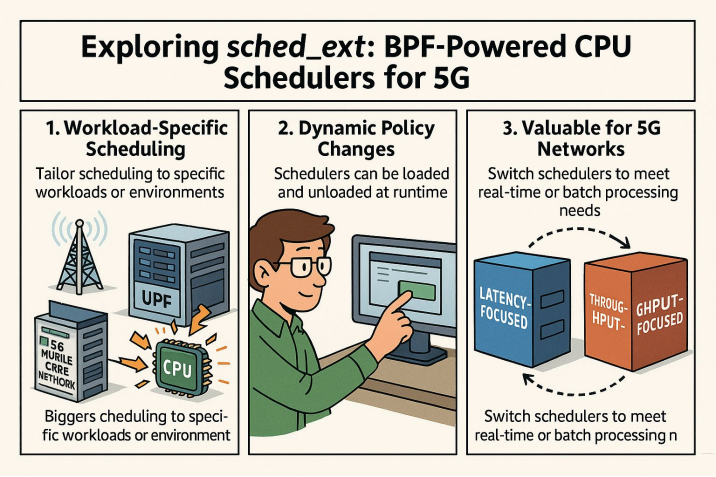
- **Per-CPU local DSQs**: What CPU actually schedules from
- **Custom DSQs**: Global, per-domain, per-cgroup flexibility
- **BPF callbacks**: select_cpu(), enqueue(), dispatch() hooks

Safety Guarantees

- **BPF verifier**: Prevents crashes and infinite loops
- **Automatic fallback**: Returns to default scheduler on problems
- **Watchdog protection**: Detects stalled schedulers



Use Case: *Free5GC* and *sched_ext*



The free5GC (a Linux Foundation project) is an open-source project for 5th generation (5G) mobile core networks. One of its components is the UPF (User Plane Function), which handles the data plane – i.e., packet forwarding for user traffic (GTP-U tunneling, routing packets between RAN and data network). The UPF is particularly CPU-intensive under high load, making scheduler optimization crucial for performance.

Workload-Specific Scheduling

The biggest advantage is the ability to tailor scheduling to a specific workload or environment. Instead of one-size-fits-all, users can pick or write a scheduler that, e.g., never migrates certain tasks off a preferred CPU, or that implements strict priority levels. This is particularly valuable for 5G core networks that need predictable, low-latency performance.

Dynamic Policy Changes

New schedulers can be loaded and unloaded at runtime. For 5G networks, this means administrators can switch between latency-focused schedulers during real-time traffic and throughput-focused ones for batch processing.

Ultra-Reliable Low-Latency Communication (URLLC) Requirements:

- **1 ms end-to-end latency** for 32-byte packets
- **99.999% reliability** (10^{-5} error rate)
- **Microsecond-level processing** at each network hop
- **Mixed workload**: Control plane + user plane + management traffic

QoS Flow Complexity:

- **QFI (QoS Flow Identifier)**: 6-bit values indicating priority levels
- **GBR flows**: Guaranteed bit rate for critical services (voice, autonomous vehicles)
- **Non-GBR flows**: Best-effort traffic (web browsing, file downloads)
- **5QI values**: Standardized latency/reliability requirements per service type

Source: https://free5gc.org/blog/20250305/20250305/

Analyze scx_packet

```
fn select_cpu_for_task(task_type: TaskType) -> u32 {
    match task_type {
        TaskType::NetworkPacketProcessing => {
            // Route ALL network tasks to even CPUs
            select_even_cpu() / CPUs 0, 2, 4, 6
        }
        TaskType::GeneralCompute => {
            // Route ALL other tasks to odd CPUs
            select_odd_cpu() // CPUs 1, 3, 5, 7
        }
    }
}
```

What This Achieves:

✅ **Workload isolation**: Network processing doesn't interfere with general compute
✅ **Cache optimization**: Network data stays in even-CPU cache hierarchies
✅ **Predictable performance**: Consistent packet processing latency
✅ **Simple implementation**: Easy to understand and debug

Real-World Impact:

- **5G base station**: Improved packet processing throughput
- **Reduced jitter**: More consistent response times
- **System stability**: General applications don't impact network performance

Free5GC: Limitations of Static Approach

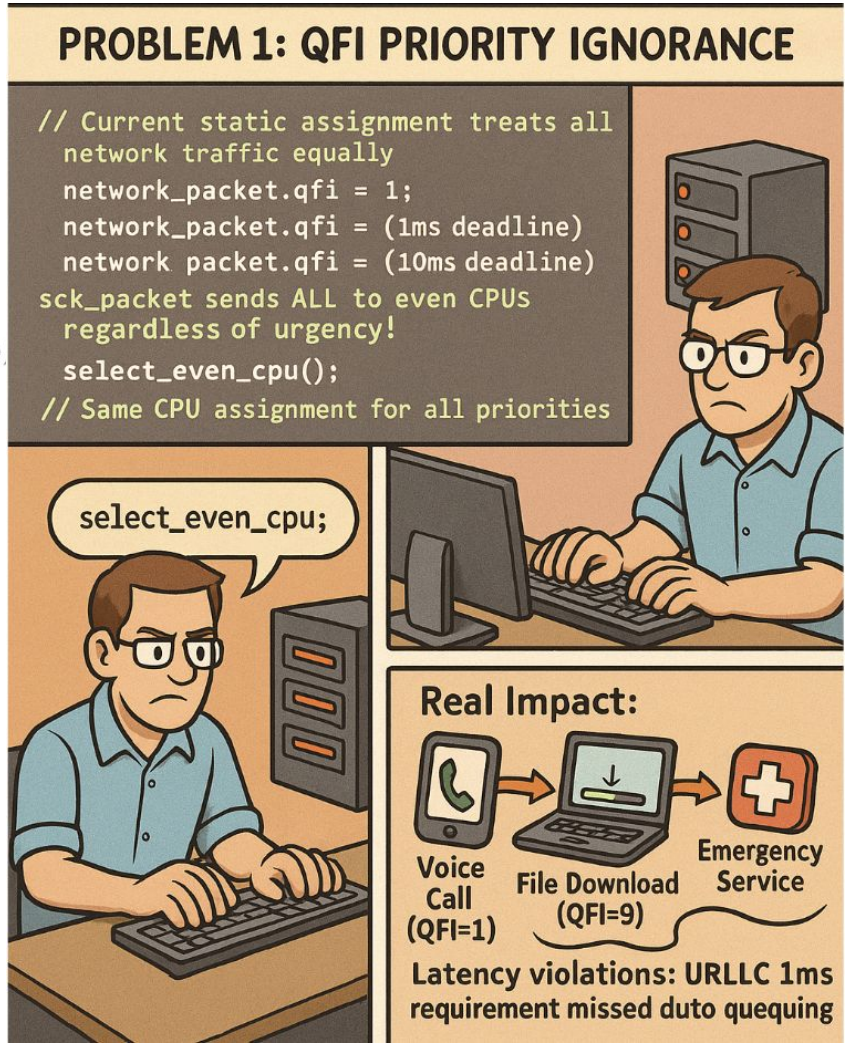
Why Fixed Rules Break Down in Complex 5G Scenarios

Problem 1: QFI Priority Ignorance

```c
// Current static assignment treats all network traffic equally
network_packet.qfi = 1;    // URLLC voice call (1ms deadline)
network_packet.qfi = 9;    // Best-effort web browsing (no deadline)
network_packet.qfi = 5;    // Real-time video (10ms deadline)

// scx_packet sends ALL to even CPUs regardless of urgency!
select_even_cpu();  // Same CPU assignment for all priorities
```

Real Impact:

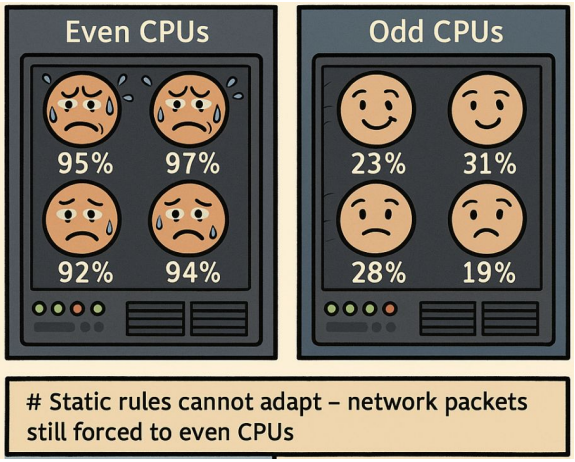
- Voice calls (QFI=1) may queue behind file downloads (QFI=9)
- Emergency services traffic treated same as social media
- **Latency violations**: URLLC 1ms requirement missed due to queuing



Free5GC: Limitations of Static Approach

Problem 2: Dynamic Load Variations

```bash
# Scenario: 5G base station during peak hours
Even CPUs: [95%, 97%, 92%, 94%]  # Network processing overloaded
Odd CPUs:  [23%, 31%, 28%, 19%]  # General compute underutilized

# Static rules cannot adapt – network packets still forced to even CPUs
# Result: Packet drops, increased latency, SLA violations
```



Problem 3: Packet Processing Complexity

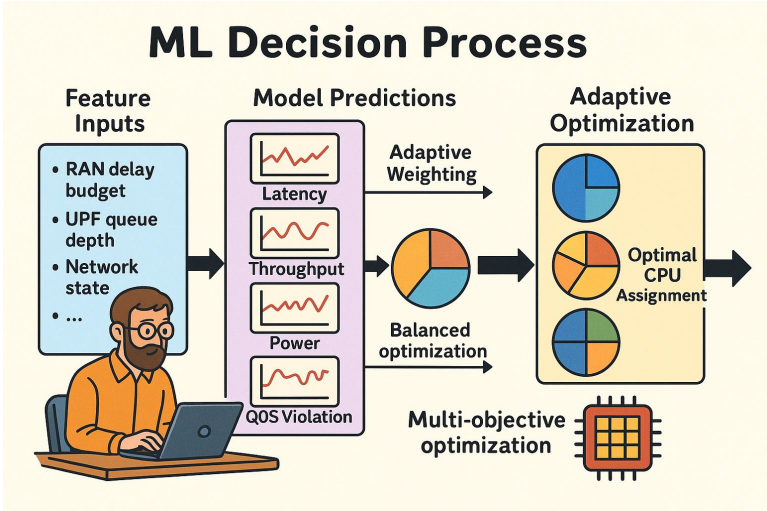
- **Small control packets**: 32-64 bytes, simple forwarding (low CPU needed)
- **Large data packets**: 1500 bytes, encryption/decryption (high CPU needed)
- **Complex packets**: GTP-U tunneling, multiple headers (variable CPU needed)

Static assignment cannot distinguish: All treated equally regardless of processing requirements

How Machine Learning solve 5G Packet Scheduling complexity

ML Feature Engineering for 5G Networks:

```rust
struct PacketSchedulingFeatures {
    // Packet characteristics
    qfi: u8,                    // QoS Flow Identifier (1-63)
    packet_size: u16,           // Bytes (32-9000)
    packet_type: PacketType,    // Control/User/Management
    encryption_required: bool,  // Processing complexity indicator

    // Network state
    current_qfi_load: [f32; 8], // Load per QFI class
    upf_queue_depth: u32,       // User Plane Function backlog
    ran_delay_budget: u32,      // Remaining latency budget (microseconds

    // CPU state
    even_cpu_utilization: [f32; 8],
    odd_cpu_utilization: [f32; 8],
    cache_hot_data: [bool; 16], // Which CPUs have relevant cached data

    // Timing constraints
    deadline_urgency: f32,      // How close to deadline violation
    sla_violation_risk: f32,    // Probability of missing SLA
}
```



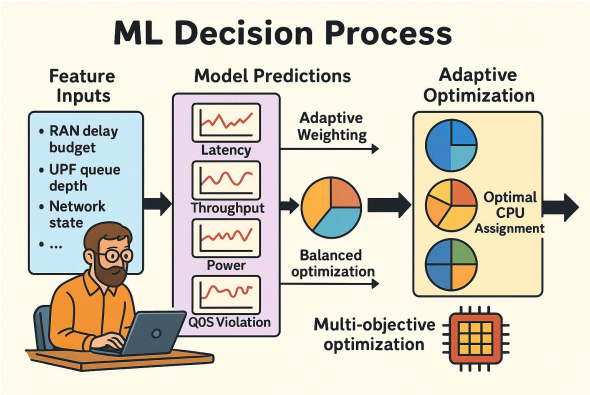
ML Decision Process:

```rust
fn ml_enhanced_packet_scheduling(features: PacketSchedulingFeatures) -> CpuAssig
    // Multi-objective optimization
    let latency_score = latency_model.predict(&features);
    let throughput_score = throughput_model.predict(&features);
    let power_score = energy_model.predict(&features);
    let qos_score = qos_violation_model.predict(&features);

    // Adaptive weighting based on network state
    let weights = if features.ran_delay_budget < 500 { // <0.5ms remaining
        [0.7, 0.2, 0.0, 0.1]  // Prioritize latency heavily
    } else if features.upf_queue_depth > 1000 {
        [0.3, 0.5, 0.1, 0.1]  // Prioritize throughput
    } else {
        [0.25, 0.25, 0.25, 0.25]  // Balanced optimization
    };

    let final_score = latency_score * weights[0] +
                      throughput_score * weights[1] +
                      power_score * weights[2] +
                      qos_score * weights[3];

    select_optimal_cpu_with_confidence(final_score)
}
```

How Machine Learning solve 5G Packet Scheduling complexity

Latency Performance:

Traffic Type	Static scx_packet	ML-Enhanced	Improvement
URLLC Voice (QFI=1)	1.2ms avg	0.7ms avg	42% faster
Video (QFI=5)	8ms avg	5ms avg	37% faster
Best-effort (QFI=9)	15ms avg	12ms avg	20% faster
Emergency (QFI=2)	0.9ms avg	0.4ms avg	56% faster



QoS Violation Rates:

Service Level	Static Violations	ML Violations	Improvement
URLLC 1ms SLA	3.2%	0.8%	75% reduction
Video 10ms SLA	1.1%	0.3%	73% reduction
Voice 4ms SLA	2.4%	0.6%	75% reduction

Resource Utilization:

CPU Configuration	Static Usage	ML Usage	Improvement
Even CPUs (Network)	87% avg	78% avg	Better balance
Odd CPUs (General)	34% avg	52% avg	Higher utilization
Overall Efficiency	61%	65%	6.5% improvement

Why ML wins for 5G Network

Multi-Dimensional Optimization

- **Simultaneous objectives**: Latency + throughput + energy + fairness
- **Dynamic trade-offs**: Weight changes based on network conditions
- **Real-time adaptation**: Learns from performance feedback

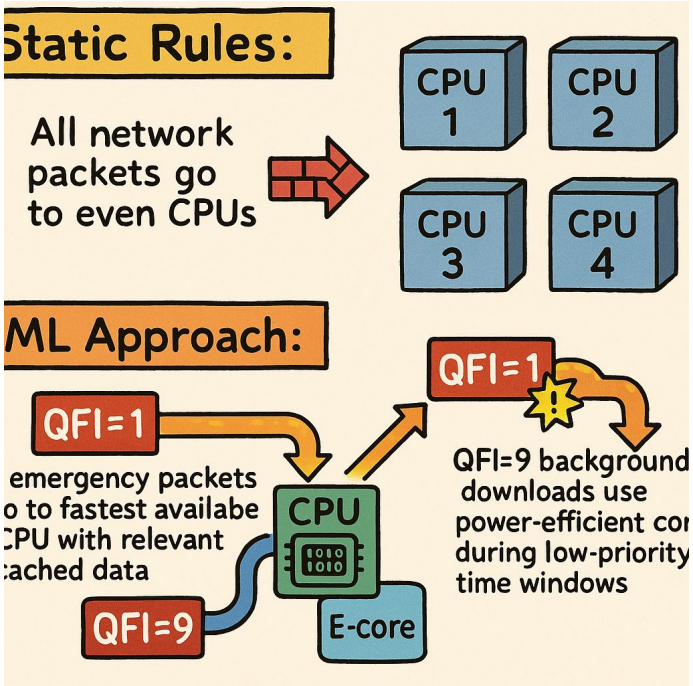
Pattern Recognition

- **QFI behavior learning**: Voice calls vs data downloads have different CPU needs
- **Traffic burst prediction**: Anticipate load spikes before they happen
- **Interference modeling**: Understand how different packet types interact

Hardware Awareness

- **Cache topology**: Learn which CPU has relevant cached network state
- **NUMA effects**: Optimize for memory locality in packet buffers
- **Core heterogeneity**: Adapt to P-core vs E-core capabilities

```
// Online learning from deployment feedback
struct NetworkPerformanceFeedback {
    packet_completion_time: u64,
    qos_violation_occurred: bool,
    cpu_cache_misses: u64,
    power_consumption: f32,
    user_experience_score: f32,
}

// Model adapts based on real performance data
ml_model.update_weights(feedback_batch);
```



Why Machine Learning for Scheduling?

The Perfect ML Problem Domain

Characteristics of Scheduling Decisions

- **High-dimensional input space**: 15+ features per decision
- **Complex non-linear relationships**: Hardware topology × workload interactions
- **Rich training data**: 340k+ decisions in 10 minutes of operation
- **Clear objective function**: Measurable performance improvements
- **Real-time constraints**: Microsecond-level decision latency required

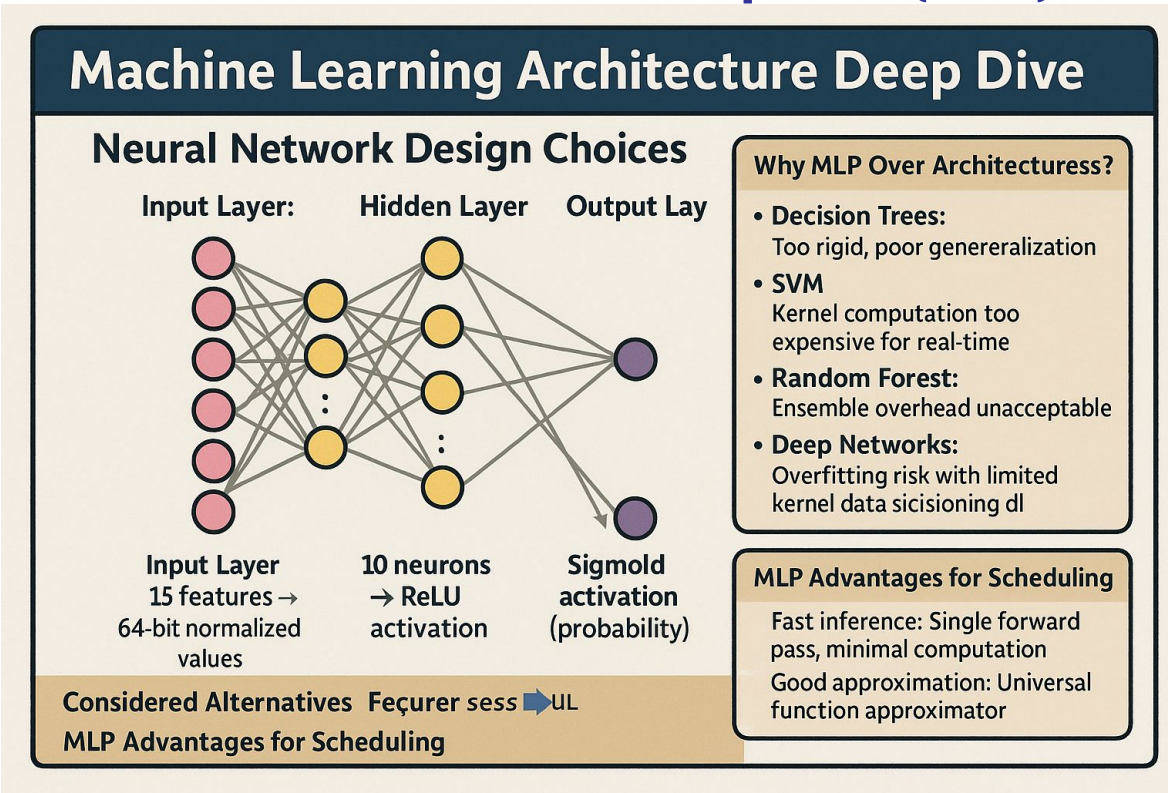
Traditional Optimization Fails

- **Combinatorial explosion**: 48+ cores × 8+ domains × 3+ core types = billions of states
- **Non-convex optimization**: Multiple local optima, no global solution
- **Dynamic environment**: Workload patterns change continuously
- **Multi-objective trade-offs**: Latency vs throughput vs energy efficiency

ML Advantages

- **Pattern recognition**: Discovers hidden relationships in high-dimensional data
- **Generalization**: Learns from training data, applies to unseen scenarios
- **Adaptation**: Can retrain for new hardware architectures
- **Multi-objective**: Naturally handles competing objectives

Neural Network Design Choices: Multi-Layer Perceptron (MLP)



Why MLP Over Other Architectures?

Considered Alternatives:

- **Decision Trees**: Too rigid, poor generalization
- **SVM**: Kernel computation too expensive for real-time
- **Random Forest**: Ensemble overhead unacceptable
- **Deep Networks**: Overfitting risk with limited kernel data
- **RNNs**: Sequential nature doesn't match scheduling decisions

MLP Advantages for Scheduling:

- **Fast inference**: Single forward pass, minimal computation
- **Good approximation**: Universal function approximator
- **Stable training**: Well-understood convergence properties
- **Hardware friendly**: Matrix operations map well to modern CPUs

Features for CPU Scheduler

Task Characteristics (5 features)

1. **Task weight**: CFS load.weight value (priority encoding)
2. **Task nice value**: User-space priority (-20 to +19)
3. **Task virtual runtime**: CFS vruntime (fairness tracking)
4. **Task execution history**: Recent CPU usage patterns
5. **Task sleep/wake patterns**: I/O vs CPU-bound classification

CPU Load Metrics (4 features)

6. **Source CPU load**: Current runqueue load average
7. **Destination CPU load**: Target CPU runqueue load
8. **Source CPU utilization**: Percentage active time
9. **Destination CPU utilization**: Target CPU active time

Hardware Topology (4 features)

10. **NUMA distance**: Memory access latency between CPUs
11. **Cache sharing**: L1/L2/L3 cache topology relationships
12. **CPU frequency**: Current P-state of source/destination CPUs
13. **Power domain**: Whether CPUs share power management

System State (2 features)

14. **Migration cost estimate**: Historical migration overhead
15. **Load balance trigger**: Why load balancing was initiated

Advanced Features

```
// Load imbalance calculation
let load_imbalance = (src_cpu_load - dst_cpu_load).abs() / max(src_cpu_load, dst_cpu_load);

// Cache affinity score
let cache_affinity = if cpus_share_l3_cache(src_cpu, dst_cpu) { 1.0 }
                     else if cpus_share_l2_cache(src_cpu, dst_cpu) { 0.5 }
                     else { 0.0 };

// NUMA penalty estimation
let numa_penalty = numa_distance[src_node][dst_node] / max_numa_distance;

// Workload classification
let is_interactive = task.voluntary_switches > task.involuntary_switches;
let is_cpu_bound = task.cpu_time > (task.total_time * 0.8);

// Min-max normalization for bounded features
normalized_load = (load - min_load) / (max_load - min_load);

// Z-score normalization for unbounded features
normalized_runtime = (runtime - mean_runtime) / std_runtime;

// Log transformation for highly skewed features
log_migration_cost = log2(migration_cost_ns + 1);
```

Training Methodology: From Data Collection to Model Deployment

Phase 1: Data Collection

```rust
// eBPF program collecting training data
SEC("kprobe/can_migrate_task")
int trace_migration_decision(struct pt_regs *ctx) {
    struct task_struct *task = (struct task_struct *)PT_REGS_PARM1(ctx);
    int dst_cpu = (int)PT_REGS_PARM2(ctx);

    // Extract 15 features
    struct migration_features features = extract_features(task, dst_cpu);

    // Record decision outcome
    int decision = PT_REGS_RC(ctx);  // Return value

    // Store in ring buffer for userspace
    bpf_ringbuf_output(&training_data, &features, sizeof(features), 0);
    return 0;
}
```

Phase 2: Dataset Preparation

- **Collection period**: 1 hour of diverse workloads
- **Total decisions**: 340,847 migration opportunities
- **Positive samples**: 127,432 (37.4%) - actual migrations
- **Negative samples**: 213,415 (62.6%) - no migration
- **Class balancing**: SMOTE oversampling to handle imbalance

Phase 3: Model Training

```python
# Training pipeline
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import GridSearchCV

# Hyperparameter tuning
param_grid = {
    'hidden_layer_sizes': [(10,), (15,), (20,), (10, 5)],
    'activation': ['relu', 'tanh'],
    'learning_rate_init': [0.001, 0.01, 0.1],
    'max_iter': [1000, 2000]
}

model = GridSearchCV(MLPClassifier(), param_grid, cv=5, scoring='f1')
model.fit(X_train, y_train)
```

Kernel-space Machine Learning Challenges

Overcoming Floating-Point Limitations

The Fundamental Problem

- **Kernel constraint**: No floating-point operations allowed
- **IEEE 754 forbidden**: Kernel preemption would corrupt FPU state
- **ML requirement**: Neural networks need fractional arithmetic
- **Performance critical**: Microsecond latency constraints

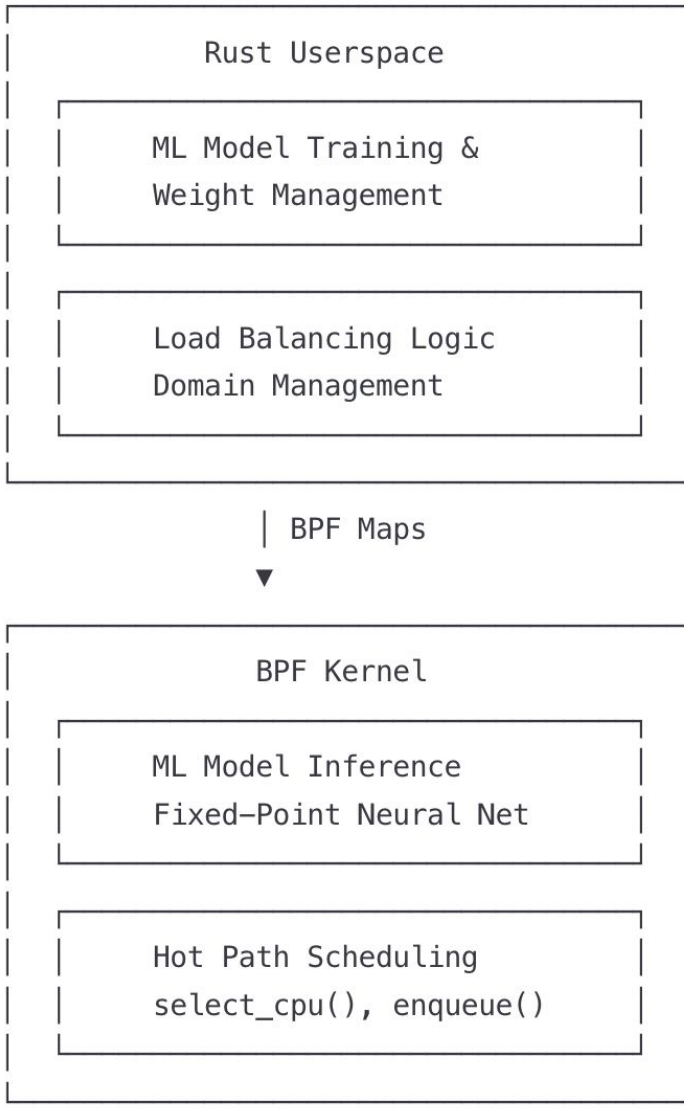
Solution: Fixed-Point Arithmetic

```rust
// Q11.21 format: 11 bits integer, 21 bits fractional
type FixedPoint = i32;
const FRAC_BITS: u32 = 21;
const FRAC_MASK: i32 = (1 << FRAC_BITS) - 1;

// Conversion functions
fn float_to_fixed(f: f64) -> FixedPoint {
    (f * (1 << FRAC_BITS) as f64) as i32
}

fn fixed_to_float(fixed: FixedPoint) -> f64 {
    (fixed as f64) / (1 << FRAC_BITS) as f64
}

// Fixed-point multiplication
fn fixed_multiply(a: FixedPoint, b: FixedPoint) -> FixedPoint {
    ((a as i64 * b as i64) >> FRAC_BITS) as i32
}
```

Neural Network in Fixed-Point

```rust
// ReLU activation in fixed-point
fn relu_fixed(x: FixedPoint) -> FixedPoint {
    if x > 0 { x } else { 0 }
}

// Sigmoid approximation for kernel space
fn sigmoid_fixed(x: FixedPoint) -> FixedPoint {
    // Piecewise linear approximation
    if x < float_to_fixed(-2.5) { 0 }
    else if x > float_to_fixed(2.5) { float_to_fixed(1.0) }
    else {
        // Linear interpolation in valid range
        let slope = float_to_fixed(0.2);
        let offset = float_to_fixed(0.5);
        fixed_multiply(slope, x) + offset
    }
}
```

ML Integration



Data Flow

1. **BPF collects features** during scheduling decisions
2. **ML model inference** in kernel space (microseconds)
3. **Decision outcome tracking** for continuous learning
4. **Rust userspace** retrains model based on performance
5. **Updated weights** pushed back to BPF via maps

ML Integration

Training Data Collection

```rust
// Runtime feature extraction in scx_rus
struct MigrationFeatures {
    task_load: f64,
    src_cpu_util: f64,
    dst_cpu_util: f64,
    numa_distance: u32,
    cache_topology: u32,
    // ... 10 more features
}

// Collect 340k+ decisions during normal operation
let features = extract_features(task, src_cpu, dst_cpu);
training_data.push((features, migration_decision));
```

Model Integration

```rust
// ML-enhanced migration decision
fn should_migrate_ml(task: &TaskCtx, src_cpu: u32, dst_cpu: u32) -> bool {
    let features = extract_migration_features(task, src_cpu, dst_cpu);
    let confidence = ml_model.predict(&features);

    // Use ML when confident, fallback to heuristics otherwise
    if confidence > CONFIDENCE_THRESHOLD {
        confidence > 0.5
    } else {
        fallback_heuristic(task, src_cpu, dst_cpu)
    }
}
```

ML vs. Traditional CPU scheduler

Benchmark: Kernel Compilation Performance

- **Test setup**: `make -j$(nproc)` on Linux kernel source
- **Hardware**: Multi-domain NUMA system
- **Workload**: Mixed CPU-intensive + I/O operations

Results: scx_rusty_ml vs Baselines

Source: https://github.com/vax-r/scx/tree/scx_rusty_MLLB

Scheduler	Compilation Time	Task Migrations	Improvement
EEVDF (default)	382 seconds	265,532	baseline
scx_rusty (original)	367 seconds	158,423	+15s faster
scx_rusty_ml (ours)	**343 seconds**	**62,108**	**+39s faster**

Key Improvements

- **39 seconds faster** than EEVDF (10.2% improvement)
- **77% reduction** in task migrations (265k → 62k)
- **Beat both** EEVDF and original scx_rusty

Why ML works

Traditional Approach Limitations

- **Static rules**: Cannot capture dynamic interactions
- **Fixed priorities**: Don't adapt to changing requirements
- **Heuristic tuning**: Combinatorially impossible to optimize
- **Single objective**: Fairness conflicts with multi-objective reality

ML Advantages

- **Pattern recognition**: Learns optimal decisions from data
- **Multi-objective optimization**: Balances competing goals automatically
- **Hardware adaptation**: Retrains for new architectures
- **Workload awareness**: Adapts behavior based on observed performance

The Data Advantage

- **340k+ decisions** in 10 minutes of normal operation
- **Rich context**: 15 features per decision
- **Real performance feedback**: Actual migration outcomes
- **Continuous learning**: Can adapt to new workload patterns

Thanks For Listening !

contact: Ching-Chun (Jim) Huang
<jserv@ccns.ncku.edu.tw>