

# **CARAT KOP: Towards Protecting the Core HPC Kernel from Linux Kernel Modules**

Thomas Filipiuk, **Nick Wanninger**, Nadharm Dhiantravan, Carson H Surmeier, Alex Bernat, Peter Dinda

# CARAT KOP: Towards Protecting the Core HPC Kernel from Linux Kernel Modules

Thomas Filipiuk<sup>1</sup>, **Nick Wanninger**<sup>1</sup>, Nadharm Dhiantravan<sup>1</sup>, Carson H Surmeier<sup>1</sup>, Alex Bernat<sup>2</sup>, Peter Dinda<sup>1</sup>

Northwestern University<sup>1</sup>, Harvard University<sup>2</sup>



**Applications and Runtimes can be accelerated with custom extensions to the operating system**

## Multiverse: Easy Conversion of Runtime Systems into OS Kernels via Automatic Hybridization

Kyle C. Hale  
Department of Computer Science  
Illinois Institute of Technology  
khale@cs.iit.edu

Conor Heintzel, Peter Dinda  
Department of Electrical Engineering and Computer Science  
Northwestern University  
cib@northwestern.edu, pdinda@northwestern.edu

**Abstract**—The hybrid runtime (HRT) model offers a path towards high performance and efficiency. By integrating the OS kernel, runtime, and application, an HRT allows the runtime developer to leverage the full feature set of the hardware and specialized OS services to the runtime's needs. However, conforming to the HRT model currently requires a part of the runtime to be kernel-level, for example in the Nautilus kernel framework, and this requires knowledge of kernel internals. In response, we developed Multiverse, a system that bridges the gap between a bulk-from-scratch HRT and a legacy runtime system. Multiverse allows unmodified applications and runtimes to be brought into the HRT model without any porting effort whatsoever by splitting the execution of the application between the domains of a legacy OS and an HRT environment. We describe the design and implementation of Multiverse and illustrate its capabilities using the massive, widely-used Hadoop runtime system.

### 1. INTRODUCTION

Runtime systems can gain significant benefits from executing in a tailored software environment, such as our Hybrid Runtime (HRT) [18]. In an HRT, a light-weight kernel framework (called an AeroKernel), a runtime, and an application coalesce into a single kernel-level entity. The OS is this composite of the application, runtime, and AeroKernel. As such, the runtime and application enjoy a base platform of fully privileged access to the underlying hardware, and can also construct task-appropriate abstractions on top of this base, instead of being limited to abstractions provided by a commodity OS. These capabilities demonstrably enhance performance, scalability, and efficiency, particularly for parallel runtime systems running on current NUMA server hardware and next generation high core-count multicore processors. The capabilities also enable forms of adaptation, both during the design process and during execution, that are simply not available to user-level systems.

An AeroKernel facilitates the creation of HRTs by providing core kernel functionality and optional mechanisms whose interfaces are geared to user-level developers instead of kernel developers. An AeroKernel helps ease the migration of user-level code to kernel-level. The motivation for an AeroKernel draws from the reliable performance of light-weight kernels [22], [21], [16], the philosophy regarding kernel abstractions of Exekernel [12], new techniques and ideas developed in multi-core OS research [23], [13], and the simplicity of other experimental OSes from previous decades [20], [28]. In

this paper, we leverage the Nautilus AeroKernel [17], which we describe in more detail in Section II.

Prior to the work and system we describe here, the implementation of an HRT consisted entirely of manual processes. HRT developers needed first to extend an AeroKernel framework such as Nautilus with the functionality the runtime needed. The HRT developers would then port the runtime to this AeroKernel manually. While a manual port can produce the highest performance gains, it requires an intimate familiarity with the runtime system's functional requirements, which may not be obvious. These requirements must then be implemented in the AeroKernel layer and the AeroKernel and runtime combined. This requires a deep understanding of kernel development. This manual process is also iterative: the developer adds AeroKernel functionality until the runtime works correctly. The end result might be that the AeroKernel interfaces support a small subset of POSIX, or that the runtime developer replaces such functionality with custom interfaces.

While such a development model is tractable ([17] gives three examples), it represents a substantial barrier to entry to creating HRTs, which we seek here to lower. The manual porting method is *additive* in its nature. We must add functionality until we arrive at a working system. A more expedient method would allow us to *start* with a working HRT produced by an automatic process, and then incrementally extend it and specialize it to enhance its performance.

The Multiverse system we describe in this paper supports such a method using a technique called *automatic hybridization* to create a working HRT from an existing, unmodified runtime and application. With Multiverse, runtime developers can take an incremental path towards adapting their systems to run in the HRT model. From the user's perspective, a hybridized runtime and application behave the same as the original. It can be run from a Linux command line and interact with the user just like any other executable. But internally, it executes in kernel mode as an HRT.

While in this paper we present one instance of an AeroKernel, Multiverse can work with any AeroKernel (or specialized OS kernel). Such pairings could enable new forms of adaptive computing, both in datacenter and HPC environments. For example, hybridization decisions could be made at runtime to merge an application or runtime system with the most suitable specialized OS kernel in response to e.g., application character-

2474-0756/17/\$31.00 © 2017 IEEE  
DOI 10.1109/ICAC.2017.24

177

Authorized licensed use limited to: Northwestern University. Downloaded on November 01, 2023 at 16:58:51 UTC from IEEE Xplore. Restrictions apply.

## Nautilus/Multiverse

## Argo NodeOS: Toward Unified Resource Management for Exascale

Swann Perarnau<sup>1</sup>, Judicial A. Zoumevo<sup>2</sup>, Matthieu Dreher<sup>1</sup>, Brian C. Van Essen<sup>2</sup>, Roberto Gioiosa<sup>1</sup>, Kamil Ikra<sup>1</sup>, Maya B. Gokhale<sup>2</sup>, Kazutomo Yoshi<sup>1</sup>, Pete Beckman<sup>3</sup>  
<sup>1</sup>Argonne National Laboratory, {swann, mdreher}@anl.gov, {ikra, kazutomo, beckman}@exanl.gov, zoum@linux.com  
<sup>2</sup>Pacific Northwest National Laboratory, Roberts.Gioiosa@pnl.gov  
<sup>3</sup>Lawrence Livermore National Laboratory, {vanessen1, mayaj}@llnl.gov

**Abstract**—Exascale systems are expected to feature hundreds of thousands of compute nodes with hundreds of hardware threads and complex memory hierarchies with a mix of on-package and persistent memory modules.

In this context, the Argo project is developing a new operating system for exascale machines. Targeting production workloads using workloads or coupled codes, we improve the Linux kernel on several fronts. We extend the memory management of Linux to be able to subdivide NUMA memory nodes, allowing better resource partitioning among processes running on the same node. We also add support for memory-mapped access to node-local, FC-attached NVRAM devices and introduce a new scheduling class targeted at parallel runtimes supporting user-level load balancing. These features are unified into *container managers*, a containerization approach focused on providing modern HPC applications with dynamic control over a wide range of kernel interfaces. To keep our approach compatible with industrial containerization products, we also identify continuation points for the adoption of containers in HPC settings.

Each NodeOS feature is evaluated by using a set of parallel benchmarks, minitaps, and coupled applications consisting of simulation and data analysis components, running on a modern NUMA platform. We observe out-of-the-box performance improvements easily matching, and often exceeding, those observed with expert-optimized configurations on standard OS kernels. Our lightweight approach to resource management retains the many benefits of a full OS kernel that application programmers have learned to depend on, at the same time providing a set of extensions that can be freely mixed and matched to best benefit particular application components.

### A. Example Workload

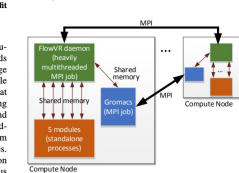


Fig. 1. Process interaction of a coupled application.

Following this roadmap, we argue that the role of a multitasking OS is transitioning from managing access to

Exascale systems are expected to feature hundreds of thousands of compute nodes with hundreds of hardware threads and complex memory hierarchies with a mix of on-package and persistent memory modules. The International Exascale Software Project [1] identified a number of challenges that need to be addressed on such systems. On the operating system (OS) side, the roadmap advocates that interfaces and support for new types of memory must be developed. Additionally, OS software should provide explicit control, from user space, of the resources available on the compute nodes. At the runtime level, parallel languages should transition from straightforward fork-join parallelism to asynchronous overdecomposed approaches, with architecture- and topology-aware load balancing performed by the runtime itself.

Following this roadmap, we argue that the role of a multitasking OS is transitioning from managing access to

## Argo NodeOS

## Palacios and Kitten: New High Performance Operating Systems For Scalable Virtualized and Native Supercomputing

John Lange<sup>\*</sup>, Kevin Pedretti<sup>†</sup>, Trammell Hudson<sup>†</sup>, Peter Dinda<sup>†</sup>, Zheng Cui<sup>†</sup>, Lei Xia<sup>†</sup>, Patrick Bridges<sup>‡</sup>, Andy Gocke<sup>\*</sup>, Steven Jaconette<sup>\*</sup>, Mike Levenhagen<sup>†</sup>, and Ron Brightwell<sup>†</sup>  
<sup>\*</sup> Northwestern University, Department of Electrical Engineering and Computer Science  
Email: {jurlan,pdinda,leixia,agocke,sjaconette}@northwestern.edu  
<sup>†</sup> Sandia National Laboratories, Scalable System Software Department  
Email: {Utpedre,mjlevene,rbrigh}@sandia.gov, hudson@ossresearch.net  
<sup>‡</sup> University of New Mexico, Department of Computer Science  
Email: {zheng,brides}@cs.unm.edu

**Abstract**—Palacios is a new open-source VMM under development at Northwestern University and the University of New Mexico that enables applications executing in a virtualized environment to achieve scalable high performance on large machines. Palacios functions as a modularized extension to Kitten, a high performance operating system being developed at Sandia National Laboratories to support large-scale supercomputing applications. Together, Palacios and Kitten provide a thin layer over the hardware to support full-featured virtualized environments alongside Kitten's lightweight native environment. Palacios supports existing, unmodified applications and operating systems by using the hardware virtualization technologies in recent AMD and Intel processors. Additionally, Palacios leverages Kitten's simple memory management scheme to enable low-overhead pass-through of native devices to a virtualized environment. We describe the design, implementation, and integration of Palacios and Kitten. Our benchmarks show that Palacios provides near native (within 5%), scalable performance for virtualized environments running important parallel applications. This new architecture provides an incremental path for applications to use supercomputers, running specialized lightweight host operating systems, that is not significantly performance-compromised.

**Keywords**—virtual machine monitors; lightweight kernels; parallel computing; high performance computing

### I. INTRODUCTION

This paper introduces Palacios, a new high performance virtual machine monitor (VMM) architecture, that has been embedded into Kitten, a high performance supercomputing operating system (OS). Together, Palacios and Kitten provide a flexible, high performance virtualized system software platform for HPC systems. This platform broadens the applicability and usability of HPC systems by:

• providing access to advanced virtualization features such as migration, full system checkpointing, and debugging;

• allowing system owners to support a wider range of applications and to more easily support legacy applications and programming models when changing the underlying hardware platform;

• enabling system users to incrementally port their codes from small-scale development systems to large-scale supercomputer systems while carefully balancing their performance and system software service requirements with application porting effort; and

• providing system hardware and software architects with a platform for exploring hardware and system software enhancements without disrupting other applications.

Palacios is a "type-F" pure VMM [1] under development at Northwestern University and the University of New Mexico that provides the ability to virtualize existing, unmodified applications and their operating systems with no porting. Palacios is designed to be embeddable into other operating systems, and has been embedded in two so far, including Kitten. Palacios makes extensive, non-optimal use of hardware virtualization technologies and thus can scale with improved implementations of those technologies.

Kitten is an OS being developed at Sandia National Laboratories that is being used to investigate system software techniques for better leveraging multicore processors and hardware virtualization in the context of capability supercomputers. Kitten is designed in the spirit of *lightweight kernels* [2], such as Sandia's Catacomb [3] and IBM's CNK [4], that are well known to perform better than commodity kernels for HPC. The simple framework provided by Kitten and other lightweight kernels facilitates experimentation, has led to novel techniques for reducing the memory bandwidth requirements of intra-node message passing [5], and is being used to explore system-level options for improving resiliency to hardware faults.

978-1-4244-6443-2/10/\$26.00 ©2010 IEEE

Authorized licensed use limited to: Northwestern University. Downloaded on November 01, 2023 at 16:51:02 UTC from IEEE Xplore. Restrictions apply.

## Palacios + Kitten

complexity relies not only on rich features of POSIX, but also on the Linux APIs (such as the `/proc`, `/sys` filesystems, etc.) in particular. Traditionally, lightweight operating systems specialized for HPC followed two approaches to tackle the high degree of parallelism so that scalable performance for bulk synchronous applications can be delivered. In the full weight kernel (FWK) approach [3], [4], [5], a full Linux environment is taken as the basis, and features that inhibit attaining HPC scalability are removed, i.e., making it lightweight. The pure lightweight kernel (LWK) approach [6], [7], [8], on the other hand, starts from scratch and effort is undertaken to add sufficient functionality so that it provides a familiar API, typically something close to that of a general purpose OS, while at the same time it retains the desired scalability and reliability attributes. Neither of these approaches yields a fully Linux compatible environment.

An alternative hybrid approach recognized recently by the system software community is to run Linux simultaneously with a lightweight kernel on compute nodes and multiple research projects are now pursuing this direction [9], [10], [11], [12]. The basic idea is that simulations run on an HPC tailored lightweight kernel, ensuring the necessary isolation for noiseless execution of parallel applications, but Linux is leveraged so that the full POSIX API is supported. Additionally, the small code base of the LWK can also facilitate rapid prototyping for new, exotic hardware features [13], [14], [15]. Nevertheless, the questions of how to share node resources between the two types of kernels, where do device drivers execute, how exactly do the two kernels interact with each other and to what extent are they integrated, remain subjects of ongoing debate.

Figure 1 illustrates the hybrid/specialized LWK landscape highlighting kernel level workload isolation, readability of Linux device drivers, and necessary Linux kernel modifications. It is worth emphasizing that modifications to the Linux kernel are highly undesired since Linux is a rapidly evolving target and keeping patches up-to-date with the latest kernel can pose a major challenge. Generally, the left side of the figure represents tight integration between Linux and the LWK, while progressing to the right gradually erases

complexity relies not only on rich features of POSIX, but also on the Linux APIs (such as the `/proc`, `/sys` filesystems, etc.) in particular. Traditionally, lightweight operating systems specialized for HPC followed two approaches to tackle the high degree of parallelism so that scalable performance for bulk synchronous applications can be delivered. In the full weight kernel (FWK) approach [3], [4], [5], a full Linux environment is taken as the basis, and features that inhibit attaining HPC scalability are removed, i.e., making it lightweight. The pure lightweight kernel (LWK) approach [6], [7], [8], on the other hand, starts from scratch and effort is undertaken to add sufficient functionality so that it provides a familiar API, typically something close to that of a general purpose OS, while at the same time it retains the desired scalability and reliability attributes. Neither of these approaches yields a fully Linux compatible environment.

An alternative hybrid approach recognized recently by the system software community is to run Linux simultaneously with a lightweight kernel on compute nodes and multiple research projects are now pursuing this direction [9], [10], [11], [12]. The basic idea is that simulations run on an HPC tailored lightweight kernel, ensuring the necessary isolation for noiseless execution of parallel applications, but Linux is leveraged so that the full POSIX API is supported. Additionally, the small code base of the LWK can also facilitate rapid prototyping for new, exotic hardware features [13], [14], [15]. Nevertheless, the questions of how to share node resources between the two types of kernels, where do device drivers execute, how exactly do the two kernels interact with each other and to what extent are they integrated, remain subjects of ongoing debate.

Figure 1 illustrates the hybrid/specialized LWK landscape highlighting kernel level workload isolation, readability of Linux device drivers, and necessary Linux kernel modifications. It is worth emphasizing that modifications to the Linux kernel are highly undesired since Linux is a rapidly evolving target and keeping patches up-to-date with the latest kernel can pose a major challenge. Generally, the left side of the figure represents tight integration between Linux and the LWK, while progressing to the right gradually erases

2016 IEEE International Parallel and Distributed Processing Symposium

## On the Scalability, Performance Isolation and Device Driver Transparency of the IHK/McKernel Hybrid Lightweight Kernel

Balazs Gerofi, Masamichi Takagi, Atsushi Hori, Gou Nakamura<sup>†</sup>, Tomoki Shirasawa<sup>†</sup> and Yutaka Ishikawa  
RIKEN Advanced Institute for Computational Science, JAPAN  
<sup>†</sup>Hitachi Solutions Ltd., JAPAN

hirota@riken.jp, masamichi.ta@riken.jp, atsushi.hori@riken.jp, gou.nakamura.y@hitachi-solutions.com, tomoki.shirasawa.h@hitachi-solutions.com, yutaka.ishi@waka.riken.jp

**Abstract**—Extreme degree of parallelism in high-end computing requires low operating system noise so that large scale, bulk-synchronous parallel applications can be run efficiently. Noiseless execution has been historically achieved by deploying lightweight kernels (LWK), which, on the other hand, can provide only a restricted set of the POSIX API in exchange for scalability. However, the increasing prevalence of more complex application constructs, such as in-situ analysis and workflow composition, dictates the need for the rich programming APIs of POSIX/Linux. In order to comply with these seemingly contradictory requirements, hybrid kernels, where Linux and a lightweight kernel (LWK) are run side-by-side on compute nodes, have been recently recognized as a promising approach. Although multiple research projects are now pursuing this direction, the questions of how node resources are shared between the two types of kernels, how exactly the two kernels interact with each other and to what extent they are integrated, remain subjects of ongoing debate.

In this paper, we describe IHK/McKernel, a hybrid software stack that seamlessly blends an LWK with Linux by selectively offloading system services from the lightweight kernel to Linux. Specifically, we are focusing on transparent reuse of Linux device drivers and detail the design of our framework that enables the LWK to naturally leverage the Linux driver code-base without sacrificing scalability or the POSIX API. Through rigorous evaluation on a medium size cluster we demonstrate how McKernel provides consistent, isolated performance for simulations even in face of competing, in-situ workloads.

**Keywords**—operating systems; hybrid kernels; lightweight kernels; system call offloading; scalability

### I. INTRODUCTION

With the growing complexity of high-end supercomputers, it has become indispensable that the current system software stack will face significant challenges as we look forward to exascale and beyond. The necessity to deal with extreme degree of parallelism, heterogeneous architectures, multiple levels of memory hierarchy, power constraints, etc. advocates operating systems that can rapidly adapt to new hardware requirements, and that can support novel programming paradigms and runtime systems. On the other hand, a new class of more dynamic and complex applications are also on the horizon, with an increasing demand for application constructs such as in-situ analysis, workflows, elaborate monitoring and performance tools [1], [2]. This

1530-2075/16/\$31.00 © 2016 IEEE  
DOI 10.1109/IPDPS.2016.80

1041

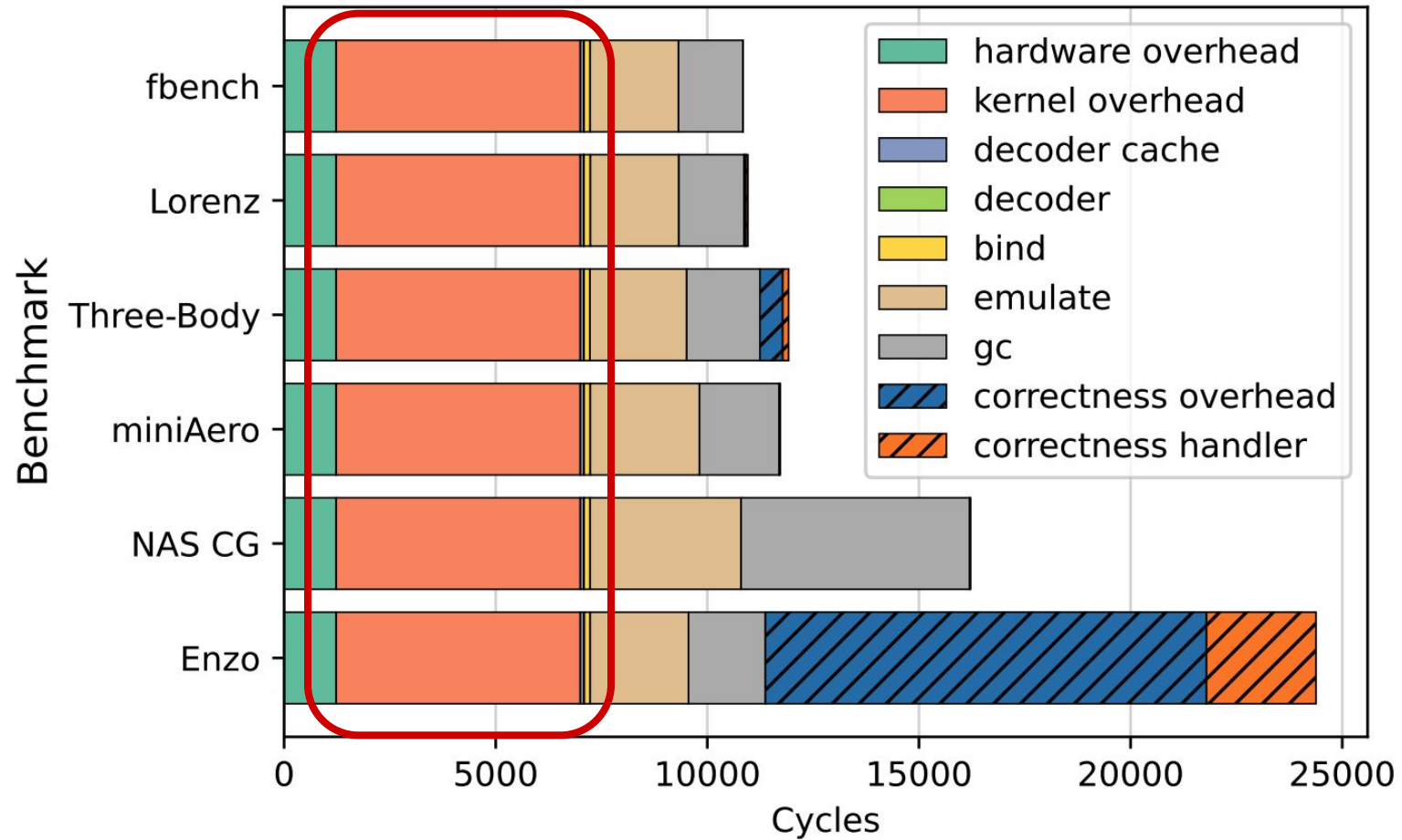
IEEE  
Computer  
Society

Authorized licensed use limited to: Northwestern University. Downloaded on November 01, 2023 at 16:50:00 UTC from IEEE Xplore. Restrictions apply.

## IHK/McKernel

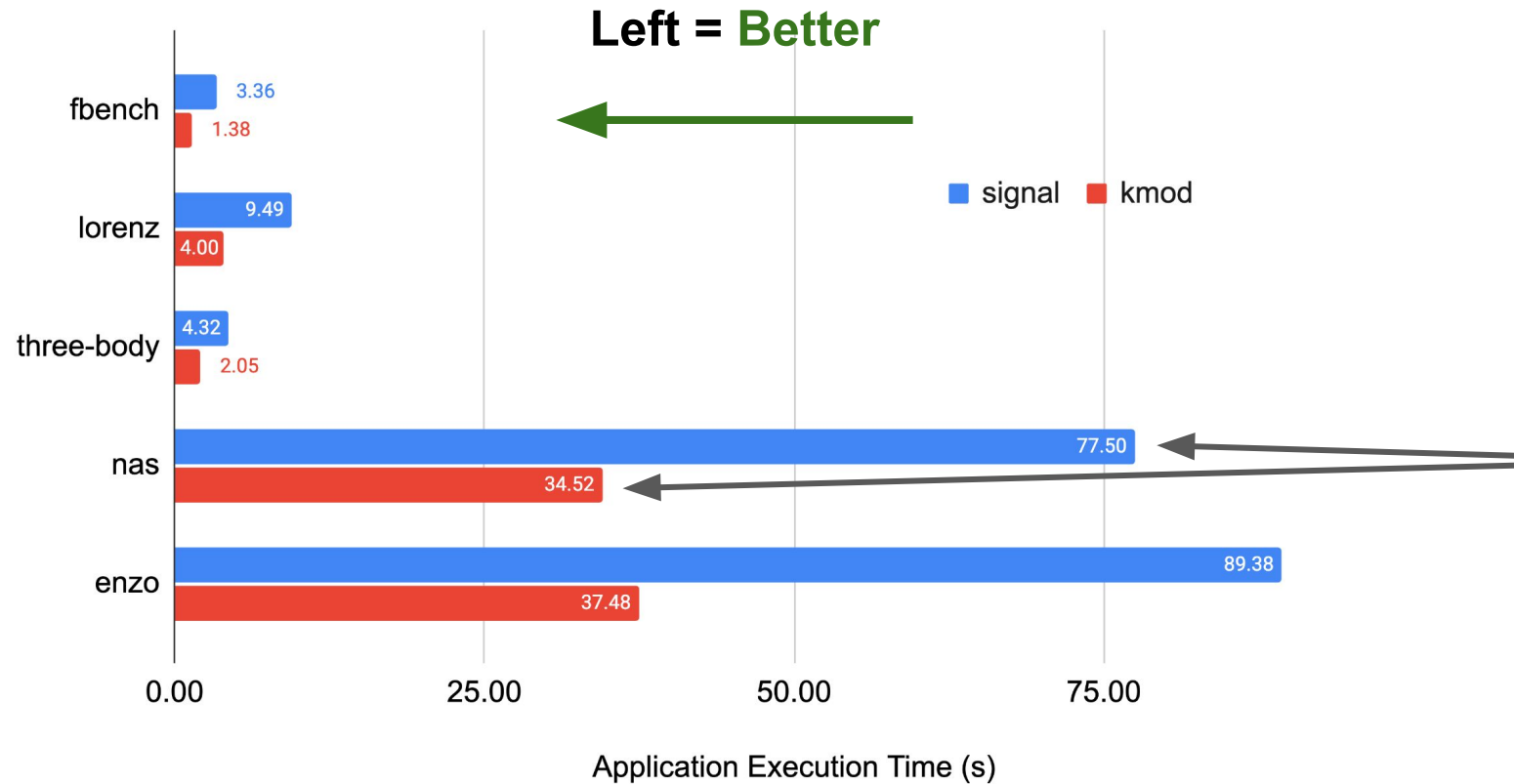
**What if you could extend *Linux*  
with these extensions?**

# A custom exception handler accelerates FPVM<sup>1</sup>



1. FPVM: Towards a Floating Point Virtual Machine. HPDC 2022

# A custom exception handler accelerates FPVM<sup>1</sup>



**>50% reduction**  
**in runtime when**  
**virtualizing**  
**floating point in**  
**SIGFPE**  
**handlers**

1. FPVM: Towards a Floating Point Virtual Machine. HPDC 2022

# Custom timer delivery accelerates Heartbeat<sup>1</sup> scheduling

A thread scheduler for irregular workloads

A custom kernel module for timer  
interrupts instead of signals

lowers delivery overhead by 3-4x

(Recently accepted to ASPLOS'24)

1. Mike Rainey, Ryan R. Newton, Kyle Hale, Nikos Hardavellas, Simone Campanoni, Peter Dinda, and Umut A. Acar. 2021. Task Parallel Assembly Language for Uncompromising Parallelism. PLDI 2021



**So Linux kernel modules can accelerate  
your applications or runtimes!**

**Why isn't everyone doing this?**

Unfortunately...

Vendors *really* don't like you inserting kernel modules

# Why?

**No hardware protections!**

(the module can just turn them off)

**Unrestricted access to all of  
physical memory (+MMIO)**

(The module can just rewrite the page tables)

**Crashes, Data Corruption**

(A bug in the module can bring the whole kernel down.

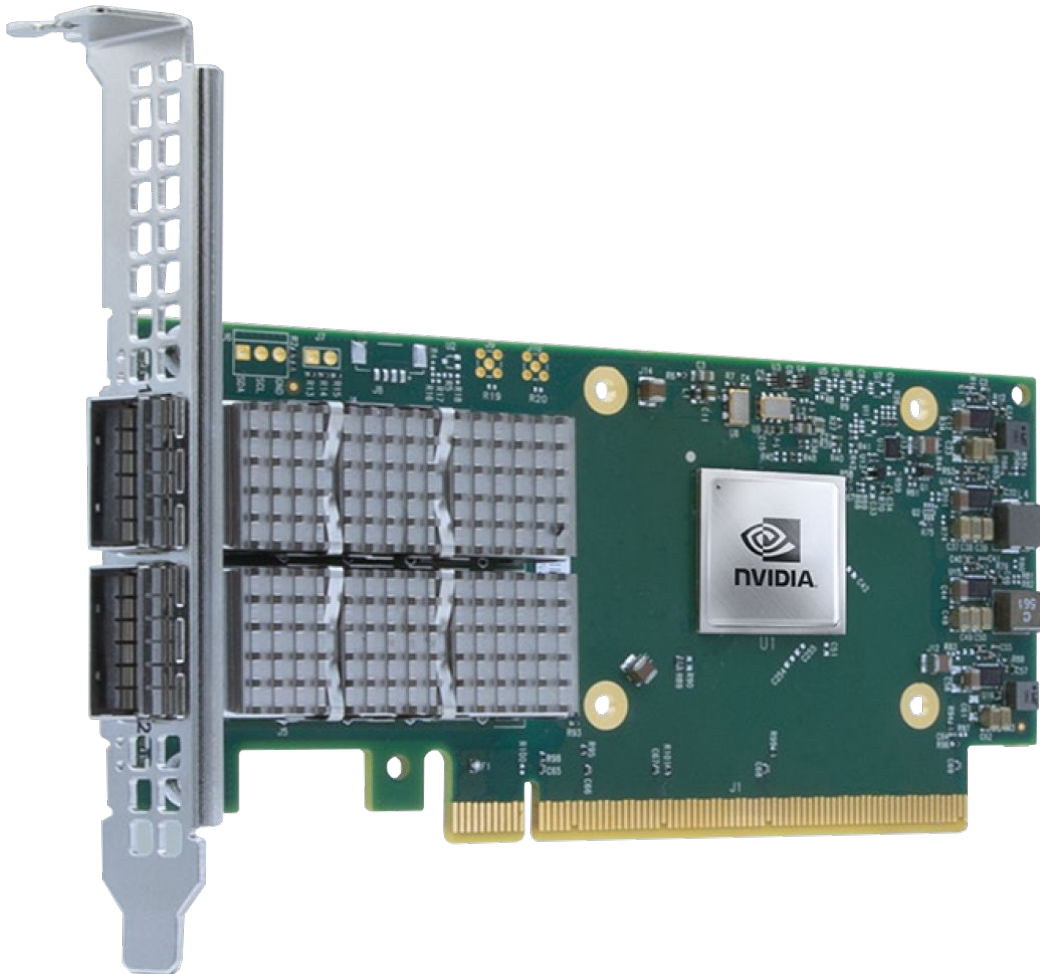
No isolation)

What if **Linux** could be *safely extended*  
with kernel modules?

*Further...*

**Can Linux be extended with *policies* that dictate what a kernel module can and cannot access?**

# For example...



```
struct ib_cq *cq = container_of(iop, struct ib_cq, iop);
struct dim *dim = cq->dim;
int completed;

completed = __ib_process_cq(cq, budget, cq->wc, IB_POLL_BATCH);
if (completed < budget) {
    inq_poll_complete(&cq->iop);
    if (ib_req_notify_cq(cq, IB_POLL_FLAGS) > 0) {
        trace_cq_reschedule(cq);
        inq_poll_sched(&cq->iop);
    }
}

if (dim)
    rdma_dim(dim, completed);

return completed;
}

static void ib_cq_completion_softirq(struct ib_cq *cq, void *private)
{
    trace_cq_schedule(cq);
    inq_poll_sched(&cq->iop);
}

static void ib_cq_poll_work(struct work_struct *work)
{
    struct ib_cq *cq = container_of(work, struct ib_cq, work);
    int completed;

    completed = __ib_process_cq(cq, IB_POLL_BATCH, cq->wc, IB_POLL_BATCH);
    if (completed >= IB_POLL_BATCH)
        ib_req_notify_cq(cq, IB_POLL_FLAGS);
    else if (cq->dim)
        rdma_dim(cq->dim, completed);
}

static void ib_cq_completion(struct ib_cq *cq, void *private)
{
    trace_cq_schedule(cq);
    queue_work(cq->comp_wq, &work);
}

struct ib_cq *ib_cq_pool_get(struct ib_device *dev,
                             int comp_vector_hint,
                             enum ib_poll_context poll_ctx)
{
    static unsigned int default_comp_vector;
    unsigned int vector, num_comp_vectors;
    struct ib_cq *cq, *found = NULL;
    int ret;

    if (poll_ctx > IB_POLL_LAST_POOL_TYPE)
        WARN_ON_ONCE(poll_ctx > IB_POLL_LAST_POOL_TYPE);
    return ERR_PTR(-EINVAL);
}

num_comp_vectors =
min_t(unsigned int, dev->num_comp_vectors,
        /* Project the affinity to the device */
        if (comp_vector_hint < 0) {
            comp_vector_hint =
                (READ_ONCE(default_comp_vector) +
                 WRITE_ONCE(default_comp_vector, comp_vector_hint));
            vector = comp_vector_hint % num_comp_vectors;
        }

/*
 * Find the least used CQ with correct
 * enough free CQ entries
 */
while (!found) {
    spin_lock_irq(&dev->cq_pools_lock);
    list_for_each_entry(cq, &dev->cq_pools[poll_ctx],
                       pool_entry) {
        /*
         * Check to see if we have found a CQ with the
         * correct completion vector
         */
        if (vector != cq->comp_vector)
            continue;
        if (cq->cqe_used + nr_cqe > cq->cqe)
            continue;
        found = cq;
        break;
    }
}

struct list_head id_list;
enum ib_gid_type *default_gid_type;
u8 *default_roce_tos;
};

struct rdma_bind_list {
    enum rdma_ucm_port_space ps;
    struct hlist_head owners;
    unsigned short port;
};

static int cma_ps_alloc(struct net *net, enum rdma_ucm_port_space ps,
                       struct rdma_bind_list *bind_list, int snum)
{
    struct xarray *xa = cma_pernet_xa(net, ps);
    return xa_insert(xa, snum, bind_list, GFP_KERNEL);
}

static struct rdma_bind_list *cma_ps_find(struct net *net,
                                           enum rdma_ucm_port_space ps, int snum)
{
    struct xarray *xa = cma_pernet_xa(net, ps);
    return xa_load(xa, snum);
}

static void cma_ps_remove(struct net *net, enum rdma_ucm_port_space ps,
                          int snum)
{
    struct xarray *xa = cma_pernet_xa(net, ps);
    xa_erase(xa, snum);
}

enum {
    CMA_OPTION_AFONLY,
};

void cma_dev_get(struct cma_device *cma_dev)
{
    refcount_inc(&cma_dev->refcount);
}

void cma_dev_put(struct cma_device *cma_dev)
{
    if (refcount_dec_and_test(&cma_dev->refcount))
        complete(&cma_dev->comp);
}

struct cma_device *cma_enum_devices_by_ibdev(cma_device_filter filter,
                                             void *cookie)
```

# Here's the trouble

How can the sysadmin be sure it is **safe**?

*Who* is going to audit **all** the code in the module?

What if the module has a **hidden bug** that you can't easily see?

```
struct ib_cq *cq = container_of(iop, struct ib_cq, iop);
struct dim *dim = cq->dim;
int completed;

completed = __ib_process_cq(cq, budget, cq->wc, IB_POLL_BATCH);
if (completed < budget) {
    inq_poll_complete(&cq->iop);
    if (ib_req_notify_cq(cq, IB_POLL_FLAGS) > 0) {
        trace_cq_reschedule(cq);
        inq_poll_sched(&cq->iop);
    }
}

if (dim)
    rdma_dim(dim, completed);

return completed;
}

static void ib_cq_completion_softirq(struct ib_cq *cq, void *private)
{
    trace_cq_schedule(cq);
    inq_poll_sched(&cq->iop);
}

static void ib_cq_poll_work(struct work_struct *work)
{
    struct ib_cq *cq = container_of(work, struct ib_cq, work);
    int completed;

    completed = __ib_process_cq(cq, IB_POLL_BATCH, cq->wc, IB_POLL_BATCH);
    if (completed >= IB_POLL_BATCH)
        ib_req_notify_cq(cq, IB_POLL_FLAGS);
    else if (cq->dim)
        rdma_dim(cq->dim, completed);
}

static void ib_cq_completion(struct ib_cq *cq, void *private)
{
    trace_cq_schedule(cq);
    queue_work(cq->comp_wq, &work);
}

/**
 * ib_alloc_cq - allocate a completion queue
 */
struct ib_cq *ib_cq_pool_get(struct ib_device *dev,
                             int comp_vector_hint,
                             enum ib_poll_context poll_ctx)
{
    static unsigned int default_comp_vector;
    unsigned int vector, num_comp_vectors;
    struct ib_cq *cq, *found = NULL;
    int ret;

    if (poll_ctx > IB_POLL_LAST_POOL_TYPE)
        WARN_ON_ONCE(poll_ctx > IB_POLL_LAST_POOL_TYPE);
    return ERR_PTR(-EINVAL);
}

num_comp_vectors =
min_t(unsigned int, dev->num_comp_vectors,
        /* Project the affinity to the device */
        if (comp_vector_hint < 0) {
            comp_vector_hint =
                (READ_ONCE(default_comp_vector) +
                 WRITE_ONCE(default_comp_vector, comp_vector_hint));
            vector = comp_vector_hint % num_comp_vectors;
        }

/*
 * Find the least used CQ with correct completion vector
 * enough free CQ entries
 */
while (!found) {
    spin_lock_irq(&dev->cq_pools_lock);
    list_for_each_entry(cq, &dev->cq_pools[poll_ctx],
                       pool_entry) {
        /*
         * Check to see if we have found a CQ with the
         * correct completion vector
         */
        if (vector != cq->comp_vector)
            continue;
        if (cq->cqe_used + nr_cqe > cq->cqe)
            continue;
        found = cq;
        break;
    }
}

struct list_head id_list;
enum ib_gid_type *default_gid_type;
u8 *default_roce_tos;
};

struct rdma_bind_list {
    enum rdma_ucm_port_space ps;
    struct hlist_head owners;
    unsigned short port;
};

static int cma_ps_alloc(struct net *net, enum rdma_ucm_port_space ps,
                       struct rdma_bind_list *bind_list, int snum)
{
    struct xarray *xa = cma_pernet_xa(net, ps);
    return xa_insert(xa, snum, bind_list, GFP_KERNEL);
}

static struct rdma_bind_list *cma_ps_find(struct net *net,
                                           enum rdma_ucm_port_space ps, int snum)
{
    struct xarray *xa = cma_pernet_xa(net, ps);
    return xa_load(xa, snum);
}

static void cma_ps_remove(struct net *net, enum rdma_ucm_port_space ps,
                          int snum)
{
    struct xarray *xa = cma_pernet_xa(net, ps);
    xa_erase(xa, snum);
}

enum {
    CMA_OPTION_AFONLY,
};

void cma_dev_get(struct cma_device *cma_dev)
{
    refcount_inc(&cma_dev->refcount);
}

void cma_dev_put(struct cma_device *cma_dev)
{
    if (refcount_dec_and_test(&cma_dev->refcount))
        complete(&cma_dev->comp);
}

struct cma_device *cma_enum_devices_by_ibdev(cma_device_filter filter,
                                             void *cookie)
```

# For example...



**The rest of the kernel should be protected from this module**

```
{
  struct ib_cq *cq = container_of(iop, struct ib_cq, iop);
  struct dim *dim = cq->dim;
  int completed;

  completed = __ib_process_cq(cq, budget, cq->wc, IB_POLL_BATCH);
  if (completed < budget) {
    if (completed < budget) {
      if (completed < budget) {

```

```
struct list_head id_list;
enum ib_gid_type *default_gid_type;
*default_roce_tos;

struct {
  space ps;
  ners;
  t;
};

struct net *net, enum rdma_ucm_port_space ps,
list *bind_list, int snum)
ma_pernet_xa(net, ps);
snum, bind_list, GFP_KERNEL);

_list *cma_ps_find(struct net *net,
cm_port_space ps, int snum)
cma_pernet_xa(net, ps);
num);
ove(struct net *net, enum rdma_ucm_port_space ps,
cma_pernet_xa(net, ps);

uct cma_device *cma_dev)
dev->refcount);

uct cma_device *cma_dev)
nd_test(&cma_dev->refcount)
v->comp);

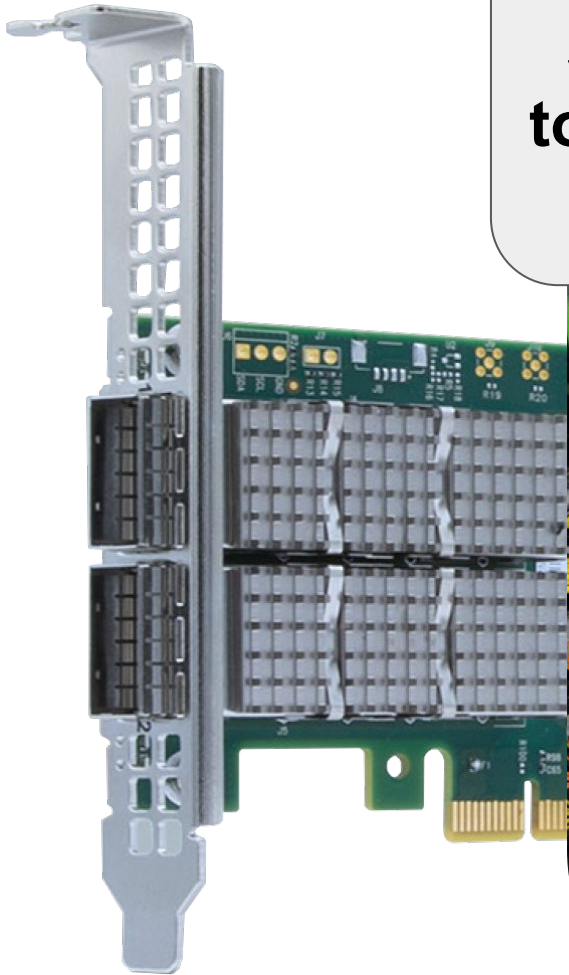
cma_enum_devices_by_ibdev(cma_device_filter filter,
*cookie)
```

```
* correct
+
if (vector != cq->comp_vector)
  continue;
if (cq->cqe_used + nr_cqe > cq->cqe)
  continue;
found = cq;
break;
}
```



# For example...

**A network driver should not be able to change my user id to root!**



```
{
  struct ib_cq *cq = container_of(iop, struct ib_cq, iop);
  struct dim *dim = cq->dim;
  ...
  completed;
  ...
  _ib_process_cq(cq, budget, cq->wc, IB_POLL_BATCH);
  ...
  completed < budget) {
    ...
    _poll_complete(&cq->iop);
    ...
    (ib_req_notify_cq(cq, IB_POLL_FLAGS) > 0) {
      ...
      trace_cq_reschedule(cq);
      ...
      req_poll_sched(&cq->iop);
    }
  }
}
```

```
struct list_head id_list;
enum ib_gid_type *default_gid_type;
u8 *default_roce_tos;
};

struct rdma_bind_list {
  enum rdma_ucm_port_space ps;
  struct list_head owners;
  short port;
};

...
_alloc(struct net *net, enum rdma_ucm_port_space ps,
        bind_list *bind_list, int snum)
...
= cma_pernet_xa(net, ps);
...
, snum, bind_list, GFP_KERNEL);
...
bind_list *cma_ps_find(struct net *net,
                      rdma_ucm_port_space ps, int snum)
...
= cma_pernet_xa(net, ps);
...
, snum);
...
_remove(struct net *net, enum rdma_ucm_port_space ps,
         rdma_ucm_port_space ps, int snum)
...
a = cma_pernet_xa(net, ps);
...
m);
...
LY,
...
(struct cma_device *cma_dev)
...
cma_dev->refcount);
...
(struct cma_device *cma_dev)
...
ec_and_test(&cma_dev->refcount))
...
a_dev->comp);
...
e *cma_enum_devices_by_ibdev(cma_device_filter filter,
void *cookie)
```

**Such a protection mechanism needs...**

**To be Fully  
Automatic**

**Arbitrary Granularity  
Memory protection**

**Extendable with  
Software Policies**

# CARAT KOP

Compiler And Runtime  
Address Translation

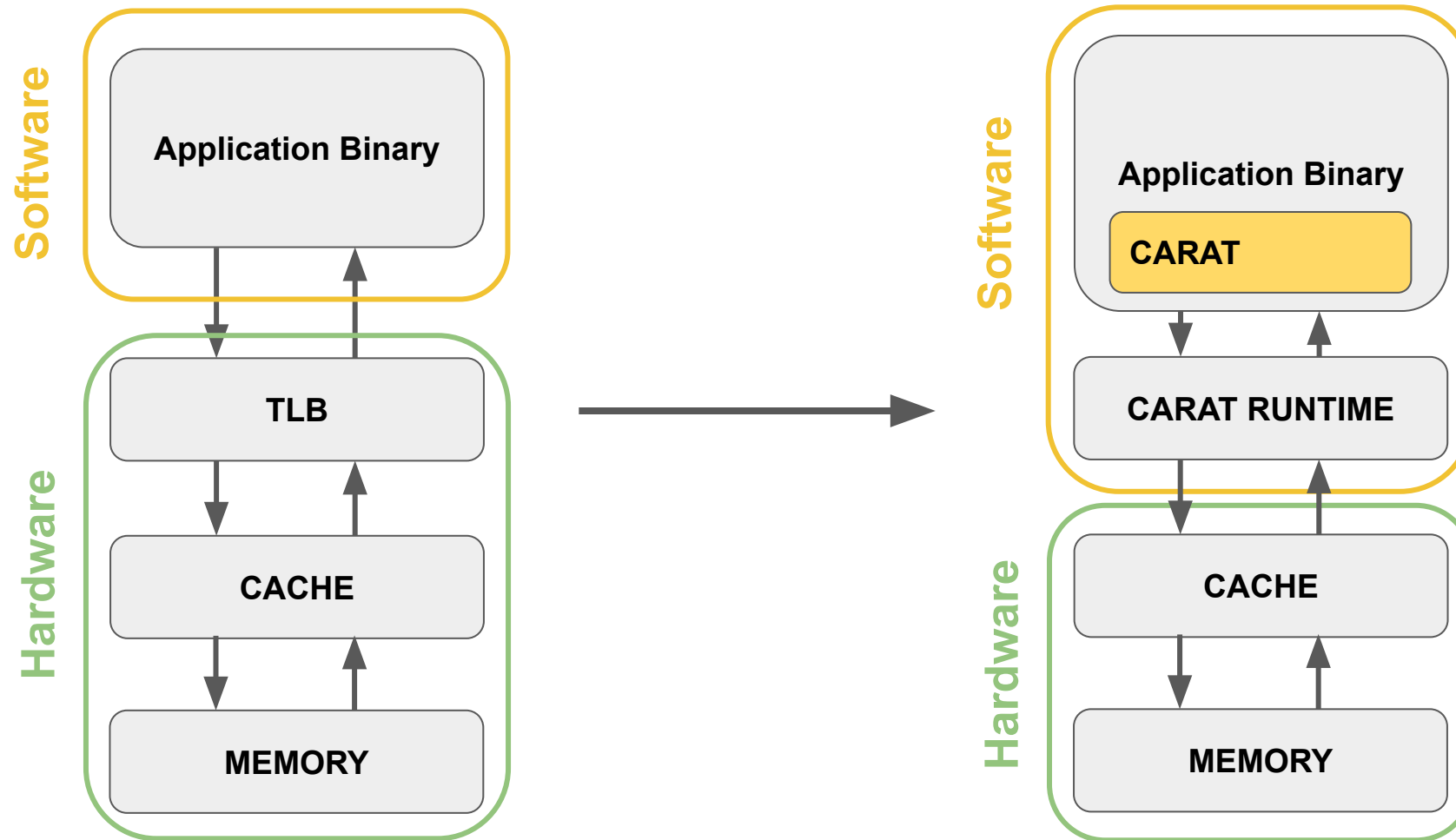
Kernel Object Protection

# CARAT KOP

Compiler And Runtime  
Address Translation

Kernel Object Protection

# CARAT: Compiler- And Runtime-based Address Translation<sup>1,2</sup>



1. Suchy et. al. CARAT: a case for virtual memory through compiler- and runtime-based address translation. PLDI 2022
2. Suchy et. al. CARAT CAKE: replacing paging via compiler/kernel cooperation. ASPLOS '22

# **CARAT replaces paging with a series of capabilities**

**Memory Protections**

**Allocation movement**

**Access pattern tracking**

# CARAT replaces paging with a series of capabilities

**Memory Protections**

We focus on just this

**Allocation movement**

**Access pattern tracking**

# CARAT inserts guards around untrusted memory access

```
struct ib_cq *cq;  
int ret = -ENOMEM;  
  
cq = rdma_zalloc_drv_obj(dev, ib_cq);  
if (!cq)  
    return ERR_PTR(ret);  
  
cq->device = dev;  
cq->cq_context = private;  
cq->poll_ctx = poll_ctx;  
atomic_set(&cq->usecnt, 0);  
cq->comp_vector = comp_vector;
```



```
struct ib_cq *cq;  
int ret = -ENOMEM;  
  
cq = rdma_zalloc_drv_obj(dev, ib_cq);  
if (!cq)  
    return ERR_PTR(ret);  
carat_guard(cq);  
cq->device = dev;  
cq->cq_context = private;  
cq->poll_ctx = poll_ctx;  
atomic_set(&cq->usecnt, 0);  
cq->comp_vector = comp_vector;
```



We experimented  
with a full-system  
designed around  
**CARAT**

In this work, we  
investigate applying  
**CARAT to *linux***



## CARAT CAKE: Replacing Paging via Compiler/Kernel Cooperation

Brian Suchy  
Northwestern University  
Evanston, IL, USA  
brian@briansuchy.com

Souradip Ghosh  
Northwestern University  
Evanston, IL, USA

Drew Kersnar  
Northwestern University  
Evanston, IL, USA

Siyuan Chai  
Northwestern University  
Evanston, IL, USA

Zhen Huang  
Northwestern University  
Evanston, IL, USA

Aaron Nelson  
Northwestern University  
Evanston, IL, USA

Michael Cuevas  
Northwestern University  
Evanston, IL, USA

Alex Bernat  
Northwestern University  
Evanston, IL, USA

Gaurav Chaudhary  
Northwestern University  
Evanston, IL, USA

Nikos Hardavellas  
Northwestern University  
Evanston, IL, USA  
nikos@northwestern.edu

Simone Campanoni  
Northwestern University  
Evanston, IL, USA  
simonec@eecs.northwestern.edu

Peter Dinda  
Northwestern University  
Evanston, IL, USA  
pdinda@northwestern.edu

### ABSTRACT

Virtual memory, specifically paging, is undergoing significant innovation due to being challenged by new demands from modern workloads. Recent work has demonstrated an alternative software-only design that can result in simplified hardware requirements, even supporting purely physical addressing. While we have made the case for this Compiler- And Runtime-based Address Translation (CARAT) concept, its evaluation was based on a user-level prototype. We now report on incorporating CARAT into a kernel, forming Compiler- And Runtime-based Address Translation for Collaborative Kernel Environments (CARAT CAKE). In our implementation, a Linux-compatible x64 process abstraction can be based either on CARAT CAKE, or on a sophisticated paging implementation. Implementing CARAT CAKE involves kernel changes and compiler optimizations/transformations that must work on all code in the system, including kernel code. We evaluate CARAT CAKE in comparison with paging and find that CARAT CAKE is able to achieve the functionality of paging (protection, mapping, and movement properties) with minimal overhead. In turn, CARAT CAKE allows significant new benefits for systems including energy savings, larger L1 caches, and arbitrary granularity memory management.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).  
ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland  
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.  
ACM ISBN 978-1-4503-9265-1/22/02...\$15.00  
<https://doi.org/10.1145/3503222.3507771>

### CCS CONCEPTS

• Software and its engineering → Operating systems, compilers; Runtime environments; • Blended systems;

### KEYWORDS

virtual memory, memory management, kernel, runtime

### ACM Reference Format:

Brian Suchy, Souradip Ghosh, Drew Kersnar, Siyuan Chai, Zhen Huang, Aaron Nelson, Michael Cuevas, Alex Bernat, Gaurav Chaudhary, Nikos Hardavellas, Simone Campanoni, and Peter Dinda. 2022. CARAT CAKE: Replacing Paging via Compiler/Kernel Cooperation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*, February 28 – March 4, 2022, Lausanne, Switzerland. ACM, New York, NY, USA, 17 pages. <https://doi.org/10.1145/3503222.3507771>

### 1 INTRODUCTION

Virtual memory, specifically address translation implemented with paging, is deeply embedded in today's systems at all levels, but particularly within the hardware and the kernel. As we have known since the 1960s [50], virtual memory solves numerous problems. This includes providing a simplifying memory abstraction for programmers, protecting the kernel from processes and processes from each other, and extending physical address space via swapping to/from storage. Its most popular form, paging, also provides a natural unit for memory management.

Unfortunately, paging<sup>1</sup> is not without cost. Paging *requires* hardware/software codesign spanning the hardware directly on the access path to main memory and the deepest levels of the kernel. The hardware structures supporting the traditional address translation model (per-core DTLBs, TLBs, STLBs, separate structures for different page sizes, nested TLBs, quad pagewalkers, walker

<sup>1</sup>And its kissing cousin, segmentation.

# CARAT KOP

Compiler And Runtime  
Address Translation

Kernel Object Protection

# Kernel Object Protection

**Allow a **policy** to dictate  
memory access -  
independent of hardware  
protections**

***Require that **untrusted  
kernel modules** are  
compiled with CARAT,  
ensuring it adheres to this  
**policy**.***

**We aren't sure what this policy  
should look like yet.**

# Maybe a policy looks like this?

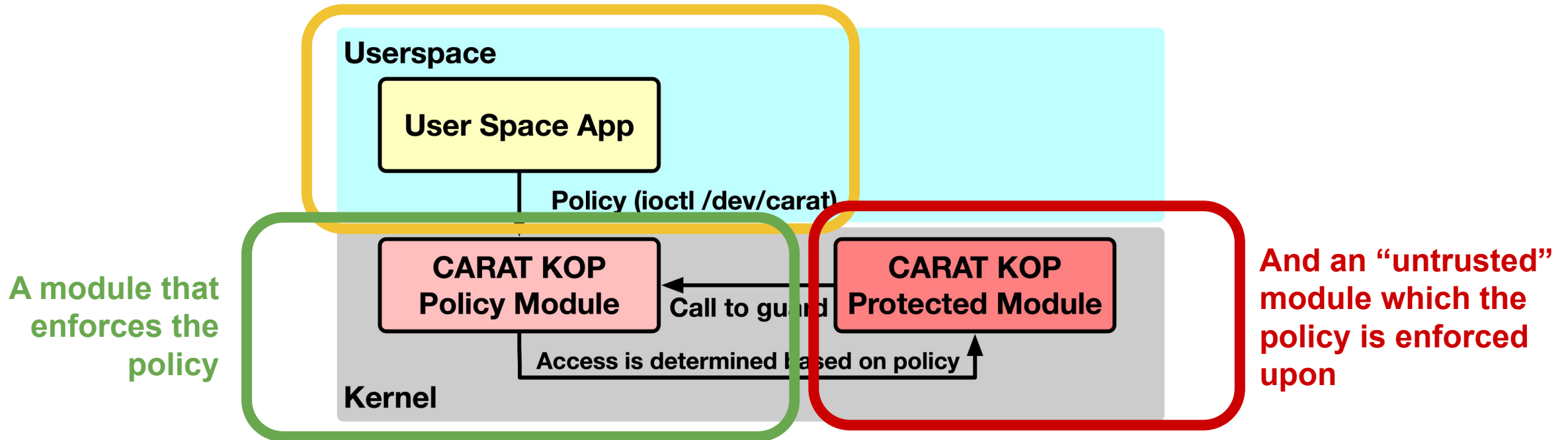
**A network driver  
should not be able  
to change my user id  
to root!**



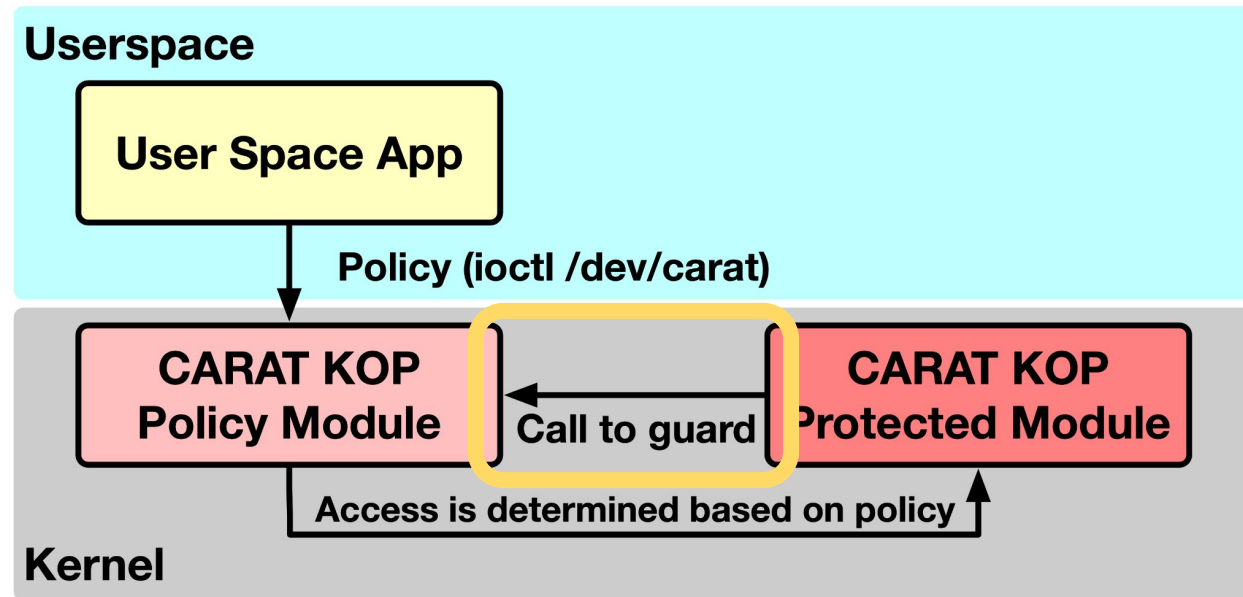
**This notion of guarding  
memory access at any  
granularity is enough to run  
*any* memory policy.**

# Our CARAT KOP Prototype is broken into three parts

A userspace program to configure the policy.



# The **policy** is enforced by calls to `carat_guard()`

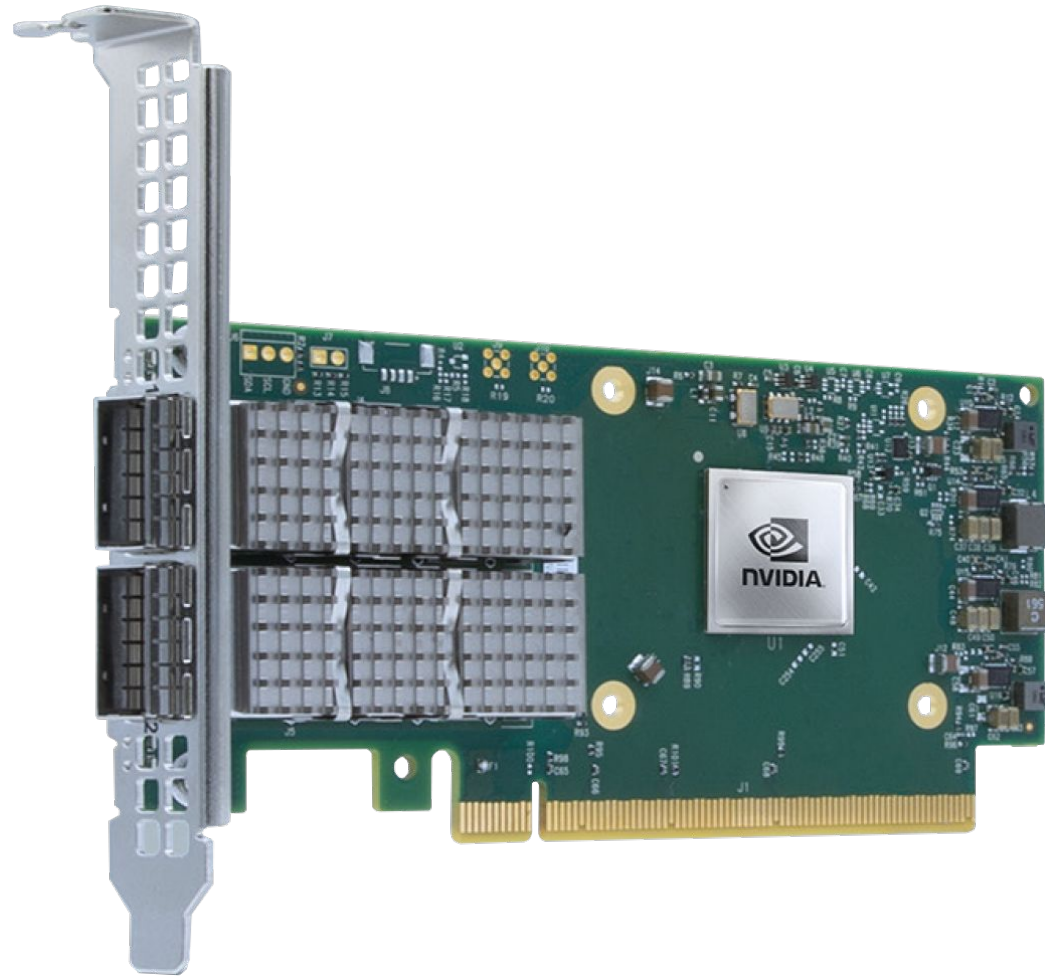




```
struct ib_cq *cq;  
int ret = -ENOMEM;  
  
cq = rdma_zalloc_drv_obj(dev, ib_cq);  
if (!cq)  
    return ERR_PTR(ret);  
carat_guard(cq);  
cq->device = dev;  
cq->cq_context = private;  
cq->poll_ctx = poll_ctx;  
atomic_set(&cq->usecnt, 0);  
cq->comp_vector = comp_vector;
```

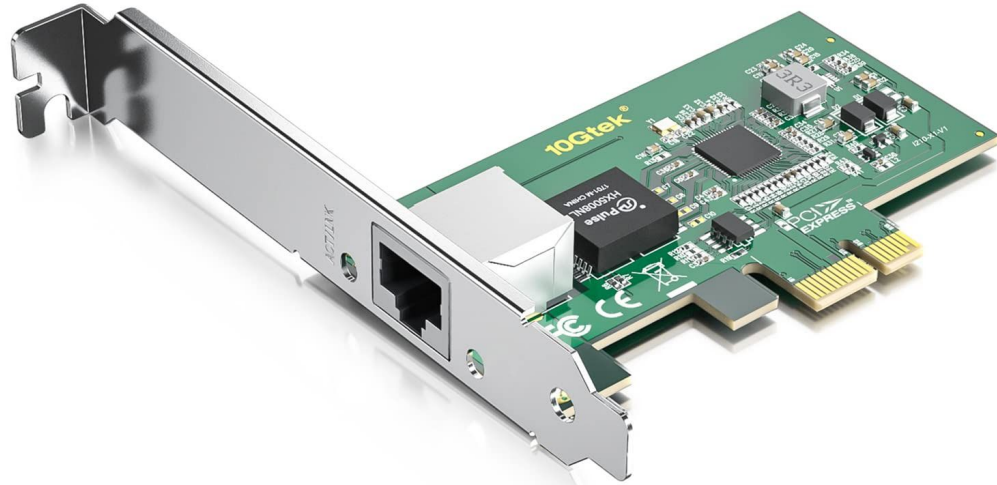
```
memory_region_t *restricted_regions;
void carat_guard(void *ptr) {
    for (long i = 0; i < region_count; i++) {
        if (contains(restricted_regions[i], ptr) {
            printk("No permission to access!\n");
            abort();
        }
    }
}
```

**So what's the performance like?**



**Unfortunately, we don't have access  
to one of these right now...**

**Instead...**



***As a proof of concept,  
we instead applied  
CARAT KOP to an  
unmodified e1000e driver***

**It's not a trivial driver - around 20k loc**

# Compiling the kernel module is easy

```
obj-m += e1000e.o
```

```
ccflags-y := -Xclang -fpass-plugin=$(PWD)/pass/build/CaratKop.so
```

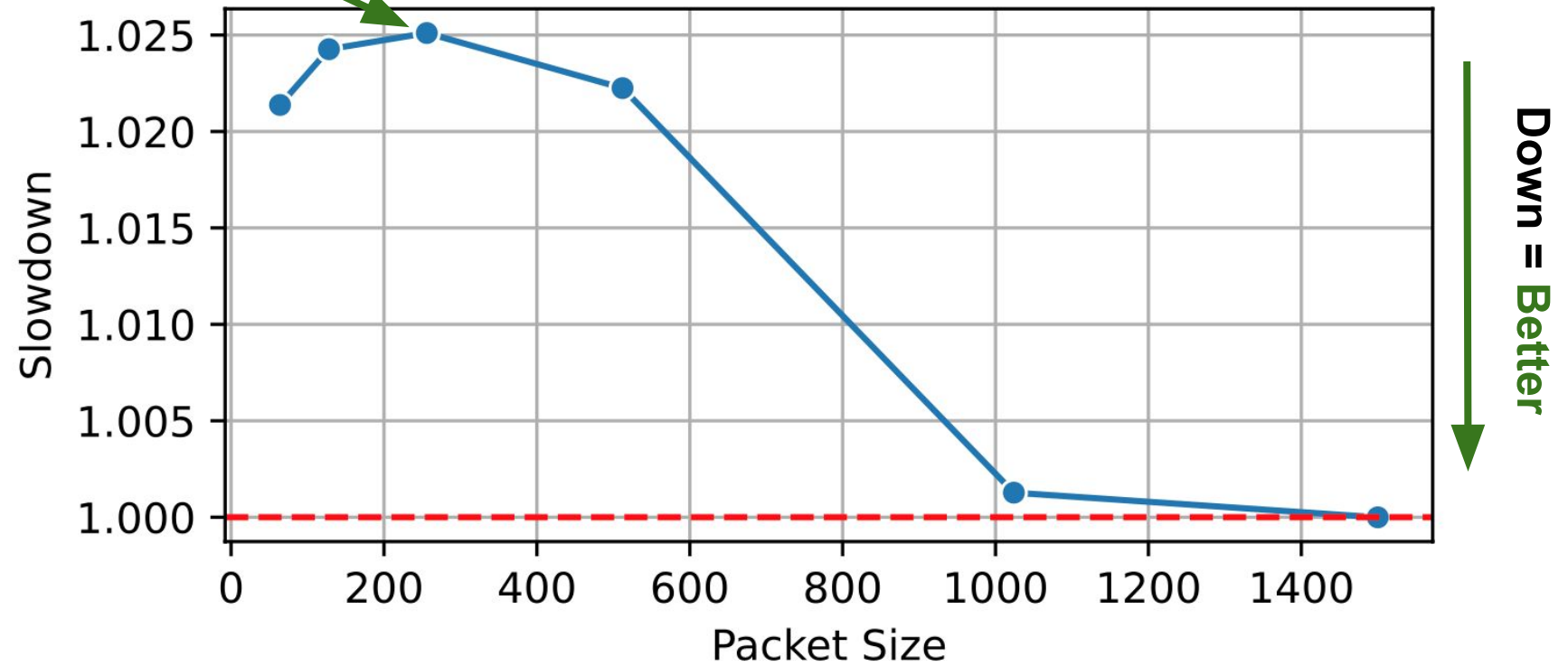
```
e1000e-objs := 82571.o ich8lan.o 80003es2lan.o \  
             mac.o manage.o nvm.o phy.o \  
             param.o ethtool.o netdev.o ptp.o
```

```
all:
```

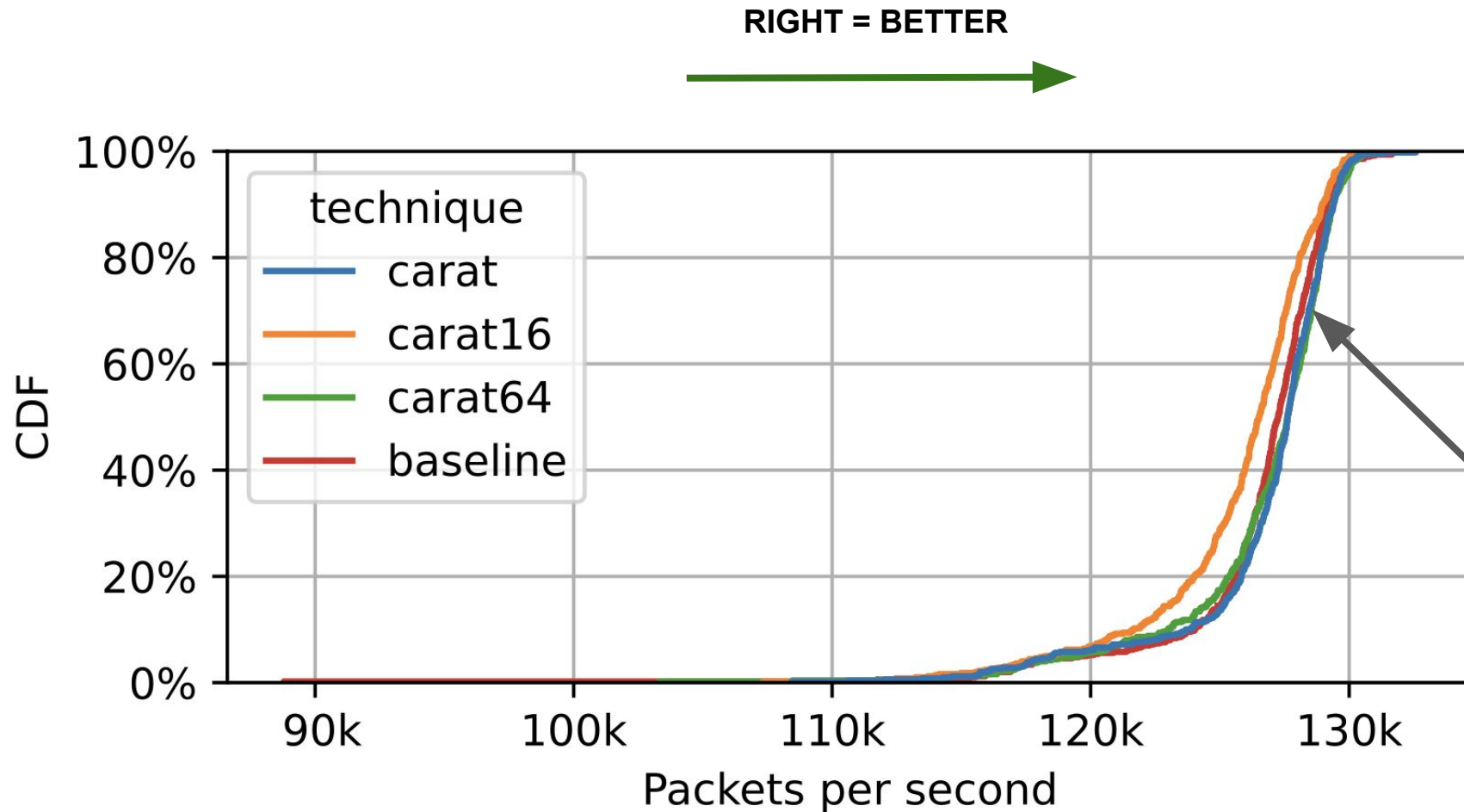
```
make -C $(LINUX) M=$(PWD) modules
```

**2.5%**

slowdown in  
the *worst*  
*case!*



# We swept the complexity of the policy to determine its effect



**What's important:**  
They are all bundled together



# Conclusion

***CARAT KOP*** allows Linux to enforce a  
software **policy** to the memory  
accesses in an unmodified kernel  
module

**With low overhead (~2.5%)**

# Outstanding Questions

- What should these policies look like?
  - CARAT KOP allows *any* policy, it's just code
- What should happen when a module violates a policy?
  - Currently, we halt the kernel
- Should different modules have different policies?
- How do we validate that a module was correctly compiled with CARAT?
  - We built code-signing for CARAT CAKE
- Does CARAT KOP extend to other modules?

# Thank you!

**Contact: [ncw@u.northwestern.edu](mailto:ncw@u.northwestern.edu)**

Thomas Filipiuk, **Nick Wanninger**, Nadharm Dhiantravan, Carson H Surmeier, Alex Bernat, Peter Dinda

