Towards a Machine Learning-Assisted Kernel with LAKE

Henrique Fingler
hfingler@cs.utexas.edu
The University of Texas at Austin
Austin, USA

Isha Tarte
tarteisha@utexas.edu
The University of Texas at Austin
Austin, USA

Hangchen Yu
athy@meta.com
Meta
Menlo Park, USA

Ariel Szekely
arielck@mit.edu
Massachusetts Institute of Technology
Cambridge, USA

Bodun Hu
bodunhu@utexas.edu
The University of Texas at Austin
Austin, USA

Aditya Akella
akella@cs.utexas.edu
The University of Texas at Austin
Austin, USA

Christopher J. Rossbach
rossbach@cs.utexas.edu
The University of Texas at Austin
Katana Graph
Austin, USA

ABSTRACT

The complexity of modern operating systems (OSes), rapid diversification of hardware, and steady evolution of machine learning (ML) motivate us to explore the potential of ML to improve decision-making in OS kernels. We conjecture that ML can better manage tradeoff spaces for subsystems such as memory management and process and I/O scheduling that currently rely on hand-tuned heuristics to provide reasonable average-case performance. We explore the replacement of heuristics with ML-driven decision-making in five kernel subsystems, consider the implications for kernel design, shared OS-level components, and access to hardware acceleration. We identify obstacles, address challenges and characterize tradeoffs for the benefits ML can provide that arise in kernel-space.

We find that use of specialized hardware such as GPUs is critical to absorbing the additional computational load required by ML decisioning, but that poor accessibility of accelerators in kernel space is a barrier to adoption. We also find that the benefits of ML and acceleration for OSes is subsystem-, workload- and hardware-dependent, suggesting that using ML in kernels will require frameworks to help kernel developers navigate new tradeoff spaces. We address these challenge by building a system called LAKE for supporting ML and exposing accelerators in kernel space. LAKE includes APIs for feature collection and management across abstraction layers and module boundaries. LAKE provides mechanisms for managing the variable profitability of acceleration, and interfaces for mitigating contention for resources between user and kernel space. We show that an ML-backed I/O latency predictor can have its inference time reduced by up to 96% with acceleration.

ASPLOS '23, March 25–29, 2023, Vancouver, Canada

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

CCS CONCEPTS

• **Software and its engineering** → **Operating systems**; • **Computing methodologies** → *Machine learning approaches*; • **Computer systems organization** → *Single instruction, multiple data.*

KEYWORDS

ML for systems, systems for ML, accelerators, GPU, operating systems

ACM Reference Format:
Henrique Fingler, Isha Tarte, Hangchen Yu, Ariel Szekely, Bodun Hu, Aditya Akella, and Christopher J. Rossbach. 2022. Towards a Machine Learning-Assisted Kernel with LAKE. In *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS '23).* ACM, New York, NY, USA, 16 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Hardware evolution and diversification is driving an explosion in the complexity of modern operating systems. CPU core counts have grown, new memory technologies such as HBM and NVM and organizations such as NUMA have become commonplace, and new networking and acceleration technologies have emerged, all of which put pressure on OSes for efficient resource management that preserves the promises of the hardware. OS kernels incorporate subsystems to manage these resources, such as memory managers, I/O and process scheduling, and file systems, and currently rely on heuristics to handle complex trade off spaces that are critical to performance. Such heuristics are developed by observing system behavior, incorporating kernel developers' experience and aim to provide reasonable average-case performance.

As hardware and software complexity continue to increase, ML has become an attractive alternative with the potential to better navigate OS tradeoff spaces currently handled by heuristics. Replacing heuristics with ML can enable system-specific solutions trained

using real, dynamically, observed behavior. While ML-backed policies in OS subsystems have been proposed, such as CPU load balancing [15], file system prefetching [11, 22, 42], I/O latency prediction [27], controlling CPU clock and power [52, 72] and others [27, 40, 62, 72, 76], previous works have focused only demonstrating the potential benefit of ML for individual subsystems. We focus instead on the systems challenges that arise from integration of ML decisioning into an OS kernel.

We study five heuristic-based kernel subsystems that can be augmented with ML decisioning, including process scheduling, memory management, and others. We identify a number of important challenges, including the following. **C1** Use of specialized hardware such as GPUs/TPUs can be critical for reducing the performance impact of ML algorithms, but poor accessibility of accelerators in kernel space is a barrier to adoption. Accelerator offload introduces additional overheads when accelerators are I/O attached and creates potential for new forms of contention between user and kernel space for using accelerators. **C2** The benefit of acceleration for ML is subsystem-, workload-, and hardware-dependent, because hardware acceleration must amortize the cost of data transfers. **C3** A fundamental tension exists between abstraction layer boundaries and the need for cross-layer data sharing to expose features for training and inference. We address these challenges in this paper, and share our experience building a **L**earning-assisted, **A**ccelerated **KE**rnel (LAKE).

To address **C1**, LAKE uses API remoting to provide kernel space applications with the vendor-supported accelerator interfaces (e.g. CUDA APIs) and, for applications that require use of libraries that are impractical to port to kernel space, custom, high-level APIs (e.g. TensorFlow). LAKE reduces overheads with zero-copy data movement between kernel applications and user space components. Concurrent use of specialized hardware by user and kernel space introduces contention that LAKE manages with policy callback framework. We find that the same mechanisms required to manage that contention can be repurposed to address **C2**, variable profitability of specialized hardware. LAKE provides a custom policy interface for contention control that allows the kernel to exploit accelerators or fall back to less intensive and/or CPU-based solutions when contention or lack of performance benefit is predicted by the policy. LAKE addresses **C3** with an in-kernel feature store that simplifies the task of instrumenting kernel subsystems to collect data that inform training and inference based on APIs that anticipate challenges like asynchrony and abstraction layer boundaries.

Our experiments show that LAKE provides efficient hardware acceleration for ML-backed subsystems in kernel space, can reduce CPU utilization by the kernel and avoids performance degradation to user space applications through contention management. For example LAKE provides performance benefits for ML-assisted I/O latency prediction, reducing inference time by up to 95%, and ML-driven load balancing inference speedup of up to 3.1×. As our focus is on systems issues arising from in-kernel ML integration, we rely on previous results from the literature demonstrating improvements from ML-backed policy relative to heuristics. However, we present an end-to-end case study for IO-scheduling characterizing the impact of acceleration, finding that ML's benefits are preserved, and that hardware acceleration can enable use of richer models. We find that LAKE's infrastructure can also be used to enable acceleration opportunities outside the ML domain. We evaluate GPU-accelerated file system encryption, finding potential read throughput increase up to 62% relative AES-NI [23] and up to 64% CPU utilization reduction. The contributions of this paper are:

- A framework for exposing ML-focused hardware acceleration in kernel space (§4), with interfaces to manage contention (§4.3) and the variable performance profitability (§4.2) for kernel/user space hardware accelerator sharing.
- A framework and efficient APIs to simplify feature collection and management in different kernel subsystems (§5).
- Evaluation of CPU utilization reduction and performance gains of existing kernel subsystems when powered by LAKE's infrastructure (§7).

2 BACKGROUND

2.1 OS Kernels and ML

Monolithic kernels such as the Linux kernel, increasingly accumulate new features and responsibilities as technology evolves. For example, Linux initially had a simple, greedy and time-sliced scheduling algorithm with a single list of tasks. Hardware evolution, e.g. increased core count, hyper-threading, non-uniform memory access (NUMA) and multiple CPU sockets forced scheduling algorithms to evolve in order to support these features. Currently, Linux's scheduler has a much more complex algorithm that uses self-balancing trees and per-core task lists, and must do complex load balancing across cores to maintain good utilization. This constant increase in problem dimensionality and the fact that systems can have different features complicates the design of efficient, general solutions and leads to heuristics that are inflexible despite the need to address a wide variety of platforms.

The Linux kernel relies on heuristics to make important decisions, such as which page to reclaim and how to balance processes across CPUs. Heuristics are usually a cheap alternative to complex, computationally intensive and sometimes impractical (e.g. NP-hard problems) solutions. The goal of a heuristic is to get a *good enough* (local minima or maxima) solution quickly, instead of spending too much time exploring the solution space for an optimal one. Heuristics used by the kernel are a one-size-fits-all approach, aiming towards average cases. For example, an I/O intensive server and a compute-intensive server that use the same kernel version, will both use the same heuristics; by specializing decisions to each server's workload, performance could be improved [29, 33, 40, 54]. Machine learning is a possible alternative to such fixed heuristics. In file system prefetching, for example, Leap [49] showed that applications have high variation in file access patterns, causing fixed pattern-finding algorithms to perform poorly in many cases. Machine learning can be applied to file system prefetching [11, 14, 43] to improve the shortcomings of heuristics. This can be achieved by learning file accesses pattern online, during execution, and training custom models.

2.2 Accelerators

Specialized accelerators are proliferating: tens of new specific-purpose accelerators and frameworks [74] emerge every year to boost the performance and efficiency of compute-intensive workloads. Deep learning accelerators such as GraphCore IPU [5] and

Google TPU [32] can provide 50× more energy-efficiency than CPUs. Near-data computing and analytics e.g. smartSSDs [34] offload data plane operations to devices, as internal disk bandwidth is much higher than bus bandwidth. General-purpose accelerators like GPUs are widely used in ML, bioinformatics, cryptocurrency, etc [41, 53, 57, 63].

However, current software and system support for accelerators is limited to user-mode programs. Accelerators ship with user libraries and kernel drivers whose interfaces and implementation are proprietary. Although many accelerator virtualization techniques [3, 20, 24, 58, 60, 68, 71, 74] exist (e.g. fixed and mediated pass-through, API remoting) that can provide applications with a virtual GPU, no existing solutions can be used by kernel space applications out of the box.

3 MOTIVATION

Our experience adding ML models to OS kernels motivates us to design infrastructure that simplifies integration and empowers developers to use potentially computationally expensive algorithms. Common infrastructure that can be used by current and future general applications is urgently needed to avoid proliferation of ad-hoc, application-specific solutions. A key challenge is collecting feature data needed by inference, which may require interrogating kernel data structures at different abstracting layers, in different modules with different locking disciplines. We propose an API design to meet this challenge in §5. We also find that accelerators (e.g. GPUs) are vital. Their massive parallelism and high throughput enable more complex and accurate models; CPUs alone are often unable able to meet performance requirements [31].

Unfortunately, accelerator stacks do not, in general, expose kernel space APIs, and typically rely on kernel-bypass designs that factor proprietary high-level API support into user space [74]. Hence, previous kernel acceleration systems [13, 26, 45, 59, 64–66] have used hand-built *up-calls* to enable OS-level interaction with accelerators. General accelerator virtualization techniques, such as API remoting [3, 20, 24, 60, 68, 71, 74] are not sufficient; communication transports used by such systems are either not available or not efficient for data transfer between kernel and user space.

Exposing accelerators to kernel space reveals opportunities and challenges unique to the OS and ML setting. The key challenges unique to this setting include *managing contention* for accelerators between kernel and user space applications and *reducing unnecessary data movement* across the user-kernel boundary, and enabling kernel subsystems to *modulate between CPUs and accelerators according to performance and accuracy profitability.*

Contention and Performance Variability. Kernel ML work can contend with user space work for access to accelerator devices and, unlike cross-user space process contention, no clear mechanism is present to manage that contention. Moreover, acceleration must amortize data transfer costs to be performance profitable, which requires batching of inputs that may be at odds with latency goals of the kernel. Both of these resource management challenges are new to the OS, but the OS has a fallback alternative to use the CPU.

Performance-critical user applications need stable, dependable access to specialized hardware in order to meet strict deadlines. Unmitigated contention for accelerators between user and kernel space can undermine those performance and QoS goals. Figure 1 demonstrates performance pathologies induced by contention when a GPU is shared between an ML-assisted kernel and a compute-bound user process. The user space process is computing data hashes while the kernel uses the GPU to accelerate page warmth classification and I/O latency prediction. As the *Contended* and *Moving Average* lines show, contention between kernel- and user space heavily impacts quality-of-service. Application throughput significantly degrades and destabilizes, decreasing by up to 68%.

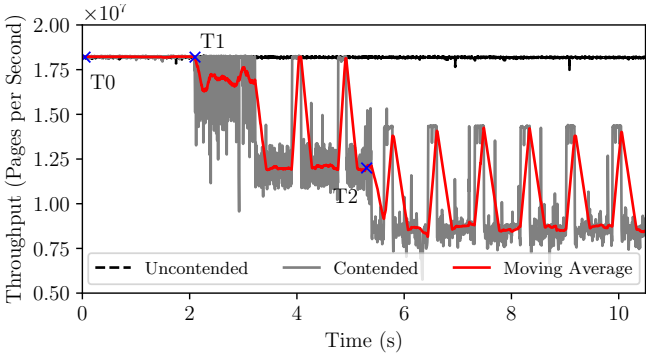

Figure 1: Throughput of a GPU-accelerated user space application which hashes pages, with and without kernel space contention for GPU compute resources. At $T0$, the user space application starts using the GPU. At $T1$, the kernel space ML page warmth classifier starts contending for GPU. At $T2$, the kernel space ML I/O latency predictor starts contending for GPU. The user-space sees a performance drop of up to 68%.

Data movement. Invocation of user space APIs from kernel space (usually done through upcalls) requires data marshalling and copying from the source context into a user space process, and copying results and modified buffers back after completion. This can result in redundant data transfer across the user-kernel boundary and unnecessary synchronization with heavy performance penalties (§6). Because no kernel-level interface to transfer data to the accelerator is present, kernel-level data buffers must first be copied into user space before being copied to/from the accelerator using APIs such as `cudaMemcpy`. Smart combination of kernel mechanisms allows automatic data marshalling and elimination of double buffering for data transfers across the user-kernel boundary.

3.1 Discussion

Why not use accelerators' interfaces directly? While direct support for kernel-level accelerator APIs is possible, frequently-changing internal interfaces and lack of publicly available documentation makes reverse-engineering portions of the accelerator software stack impractical. The opaque nature of accelerator software stacks requires that hardware vendors expose kernel-level APIs themselves. Although NVIDIA recently open-sourced part of its drivers [7], the driver does not expose the necessary high-level APIs to the kernel. We additionally find that ML support is often better served by higher level APIs such as TensorFlow (§7), so more general support for upcalls is necessary.

Could devices manage contention directly? Hardware vendors have shown their willingness to enable some contention management in hardware. A select number of accelerators are Single-Root IO Virtualization (SR-IOV) enabled, and some devices such as SmartNICs and SmartSSDs provide APIs for coarse-grained contention management and rate limiting, or allow developers to express their own policies. However, hardware-based solutions tend to be inflexible. OS kernel developers may wish to select between different contention management policies dynamically. Complex and evolving contention management policies are more easily expressed in software, and our experience is that not every accelerator will support fine-grained contention management policies in hardware. ML requires additional policy support to deal with variable performance profitability that contention management alone does not solve.

Is isolation impacted? OS kernels use address space isolation as their primary memory protection mechanism. We rely on the same mechanism to isolate memory when offloading OS kernel computation to accelerators. In our experience all accelerators support some type of address space isolation. While any approach which offloads OS kernel data to accelerators may expose new side-channels, we leave investigation of side-channel mitigation to future work.

4 KERNEL ACCELERATION WITH LAKE

To allow complex, accelerator-dependent machine learning algorithms to be used in the kernel, LAKE must provide infrastructure that allows for future and current kernel space applications to use accelerators. This is not currently possible because libraries supplied by accelerators' vendors are designed for user space. At the core of enabling accelerator access to kernel space in LAKE is an API remoting system that exposes arbitrary APIs to kernel subsystems. APIs exposed by LAKE are executed, through upcalls, by a process in user space. Figure 2 shows the design of LAKE. We consider a system with Linux as the host OS and at least one accelerator. Although this work focuses on NVIDIA GPUs and CUDA, there is no fundamental issue preventing it to be extended to other accelerators [74].

There are three main components in LAKE: the kernel-side API provider (*lakeLib*), the bulk data kernel-user communication channel (*lakeShm*) and the user side daemon process that realized APIs, *lakeD*. ***lakeLib*** is a kernel module that exposes APIs as such as the vendor's user space library of an accelerator as symbols to kernel space. This module has a function with the same name API it wants to support in kernel space. For example, to support the `cuMemAlloc` CUDA API in kernel space, we must have a function with the same name in *lakeLib*. Each of these functions in *lakeLib* does three things: serialize an API identifier and all of API parameters into a **command**, transmit commands through some communication channel for remote execution in user space and, finally, wait for a response.

The *lakeD* is a user space daemon that listens for **commands** coming from *lakeLib*, deserializes them and executes the requested APIs. This daemon must have access to the vendor's library (e.g. `cudart.so`) to realize APIs requested by *lakeLib*. Continuing the example of the `cuMemAlloc` API, a command for such API includes a field that identifies which API is to executed and its parameters: how many bytes to allocate and a pointer to store the starting address of the new allocation. *lakeD* deserializes the command to obtain these fields, executes the API using the vendor's original library and sends back, through the same channel the initial command came from, the results: the return code and the pointer returned by the API call.

Lastly, *lakeShm* is a kernel module that provides memory allocations to *lakeLib* and LAKE-powered applications. Memory allocated through *lakeShm*'s APIs are optimized for data transfers between kernel space applications and the user space *lakeD*. *lakeShm* works by requesting and mapping a large contiguous memory region from the Linux kernel. When *lakeD* is started, the same region is mapped to its process. While host-to-device transfer is still required, this allows for zero-copy memory movement between kernel space modules and *lakeD*.

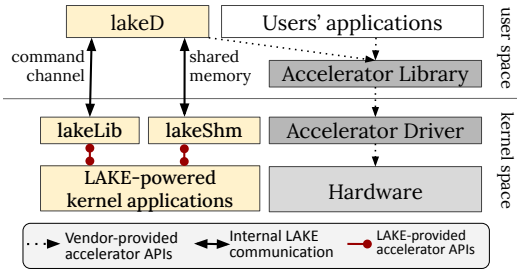

Figure 2: Overview of the architecture of LAKE.

4.1 System Workflow

When a kernel space application calls an API provided by LAKE, a series of mechanisms are activated until it is finally handled by the accelerator. This workflow includes two boundary crossings: from kernel to user and from user to kernel space. Let's consider a simple application that allocates memory locally and on the GPU, copies local data to the GPU and invokes a kernel to do some computation on the GPU. All applications we study exercise these steps.

We classify the operations an application that uses LAKE can do into three categories: **local operations**, **API-remoted operations** and **copiable memory allocations**.

Local operations : these operations include existing kernel functions and memory allocations in kernel space. Such operations do not require remoting and are not modified by LAKE. For example, regular memory allocations can be satisfied by calls to the kernel's memory allocator, e.g. `vmalloc`.

API-remoted operations : LAKE provides the accelerator API to kernel space through *lakeLib*. When the application calls an accelerator API, the execution flow switches to the *lakeLib* module. The buffer for a *command* large enough to hold the API function identifier (e.g. a number) and all function arguments is created. This command is then sent to *lakeD* through a socket-like channel. Once in user space, the command is deserialized and the API requested is executed on the accelerator. When completed, a return command is built with the return value and sent back. Errors caused when executing an API are forwarded to the application, which must do its own error checking.

Copiable memory allocations : memory regions used by applications that will be copied to/from accelerators, should be allocated

using *lakeShm*, which provides a function similar to `malloc`. Memory regions allocated by *lakeShm* are shared, avoiding memory copies between kernel and user space. Using *lakeShm* does not, by itself, yield zero-copy data transfers between the kernel space application and the accelerator. The CUDA API, for example, has a user space API (`cudaMallocHost`) that provides zero-copy transfer from user space to GPU, but LAKE can not integrate this feature since the CUDA runtime library is closed source. For custom high-level APIs provided by LAKE (discussed in §4.4), where kernel space applications call APIs at a much higher level than allocating memory on an accelerator, *lakeShm* removes the only data copying under its domain. API remoting will still work if applications do not user *lakeShm* neither accelerator-specific APIs that reduce data transfers; this will just cause extra data copies.

4.2 Modulating Accelerator Use

Profitability of using an accelerator is not always guaranteed as we show in §7; accelerators' massive parallelism are only advantageous when processing large amounts of data. Accelerators are almost ubiquitous in ML training due its batch processing, but the same is not true for inference. Inference on small batches of inputs is usually faster on CPUs. Typically, there exists a batch size in which accelerators yield better performance (we call this the crossover point).

To provide kernel ML applications with the best performance, LAKE allows on-the-fly switch between execution on CPU and accelerator, at the function call granularity. This is done through custom *execution policies* (an example is given in §4.3). LAKE allows developers to write and install such policies using eBPF [4]. Through callbacks, developers can specify the necessary requirements to consider utilizing an accelerator profitable. The policy is executed automatically by the kernel during the application's execution. Figure 3 shows the pseudocode of a simple policy for CUDA devices that manages variable profitability by falling back to the CPU for batch sizes under a certain threshold.

4.3 Contention Management

We cannot assume that accelerators provided by LAKE will be exclusive to the kernel. User space applications expect performance guarantees from accelerators, and we can not tolerate performance interfere. When the accelerator becomes a contended resource, kernel space applications must reduce or completely stop using the accelerator and fall back to either a simpler, less intensive accelerator implementation or a CPU implementation.

The same policy used for modulating accelerator utilization can be used to manage contention. A policy's toolset includes any OS- or vendor-provided utilities (e.g. NVIDIA's `NVML` API, supported by LAKE), allowing fine-grained information about the current state of the system. Figure 3 shows the pseudocode of a simple contention policy for CUDA devices. The policy rate-limits the query of GPU utilization and using a moving average to keep kernel consumption of GPU compute under a threshold. The developer can specify the policy with two callback functions: the `dev_func` callback usually contains one or more `cuLaunchKernel` invocations, and the `cpu_func` can contain alternative APIs that perform the same computations, but may operate on the CPU or use fewer accelerator resources.

```
policy cu_policy(offload_func_t dev_func, void
    *dev_args, offload_func_t cpu_func, void *
    cpu_args) {
  static nvmlUtilization_t util;
  if ...5 ms elapsed since last check...
    // LAKE-remoted nvml API
    nvmlGetUtilization(dev, &util)
  // compute GPU utilization moving average
  int exec_rate = mov_avg(util.gpu);
  // batch size for profitability threshold
  int batch_sz = get_batch_size(def_args)
  if (exec_rate < exec_threshold
       && batch_sz >= batch_threshold)
    return dev_func(dev_args);
  else
    return cpu_func(dev_args); }
```

Figure 3: An outline of a policy for CUDA devices that uses moving average to manage contention and a batch threshold to use HW acceleration only for batches big enough to enjoy performance benefit.

4.4 High-level APIs

The simplicity of existing machine learning libraries such as Tensorflow [10], which abstracts complex machine learning functionality into high-level APIs, discourage applications from using the CUDA runtime API directly. Although possible, we can not force developers to implement complex and hard-to-optimize algorithms in CUDA. At the same time, porting enormous libraries like Tensorflow to the kernel is impractical; these libraries rely on user space exclusive libraries and have substantial size. Enabling and facilitating the use of machine learning by the kernel is one of the main objective of LAKE, so we must provide mechanisms for applications to use high-level libraries.

LAKE's API remoting system is sufficiently general that it can support manual addition of APIs. This is required to allow kernel space applications to use higher-level APIs without porting them to kernel space. For example, our page warmth predictor(§ 7.2) is based off of Kleio [18], which uses Tensorflow to construct a model with two LSTM layers. While constructing a model is not hard, implementing fast, efficient and correct LSTM inference using the CUDA runtime directly is [8]. Providing high-level APIs to the kernel space requires two things: adding the function's prototype in *lakeLib* and implementing its functionality in *lakeD*. Manual addition of APIs requires developers to design data conversion between raw data in kernel to the libraries' expectation. For example, if NumPy arrays are used as input to TensorFlow, something not available in kernel, the data must be sent in some format (e.g. arrays of numbers) and converted in the *lakeD*. Automatic data serialization between kernel and user space is provided by LAKE.

API	Description
`create_registry(name, sys, schema, window)`	Creates feature registry with capacity
`destroy_registry(name, sys)`	Destroys a feature registry
`create_model(name, sys, path)`	Create a new ML model, saved at path
`update_model(name, sys, path)`	Commit a changed model to the file system
`load_model(name, sys, path)`	Load a model from path into memory
`delete_model(name, sys, path)`	Delete a model from the file system and memory
`register_classifier(name, sys, fn, arch)`	Provide a function pointer for classifiers/inference Note: arch specifies CPU / GPU / XPU
`register_policy(name, sys, fn)`	Provide an eBPF policy for contention/batching (§4.3)
`score_features(name, sys, fvs, num)`	Run inference on a batch, return batch results
`get_features(name, sys, ts)`	Batch retrieves all feature vectors older than ts
`begin_fv_capture(name, sys, ts)`	Starts the creation of a new feature vector. Subsequent calls to capture_feature for name/subsystem will add/overwrite the current value of that feature
`capture_feature(name, sys, key, val, sz)`	Sets feature with key, val on the current vector
`capture_feature_incr(name, sys, key, incrval, sz)`	Update a feature with key by incrementing
`commit_fv_capture(name, sys, ts)`	Commits the current feature vector to the registry.
`truncate_features(name, sys, ts)`	Removes all feature vectors older than ts

Table 1: The LAKE feature registry API.

5 IN-KERNEL FEATURE REGISTRY

LAKE supports in kernel feature registry to manage ML models and feature vector capture, whose API is shown in Table 1. The design goals of the API are **1)** to minimize the performance impact of ML-related functionality, **2)** enable simple, potentially asynchronous feature vector capture in the presence of abstraction and module boundaries and anticipate the needs of multithreaded code (e.g. do the relevant data structures need to be interrogated with locks held, or in interrupt context?) and **3)** simplify the task of invoking inference on *batches* of feature vectors. Generally speaking, the API provides a handful of functions for managing *registries* (named combinations of a model, with an accompanying feature vector schema, associated with kernel subsystems), managing ML models, capturing features, and invoking classifiers/inference.

Design for Performance. The API meets the first goal (minimal overhead) by operating in-kernel and using careful data structure and API design. ML models are committed to the file system and loaded into memory at boot time. Loading and update are infrequent, so file system overheads are acceptable, but at inference time, having the model in memory is critical to performance. Feature vectors are stored in memory in a circular buffer sized according to the window parameter specified, and in general have the format `<numfeatures, kvpair*, ts_begin, ts_end>`. The `kvpair*` is a key-value map from feature keys to values supported by a lock-free hash table. We considered supporting the feature registry in user-space to avoid introducing sensitive code in the kernel, but ultimately decided that kernel crossings for feature capture and for accessing models for inference would put too much overhead on the critical path.

Schema. Each registry has a *schema*, which describes the format of feature vectors: concretely the schema is a map from feature key (name) to a tuple of `<size, entries>`, where `size` is the number of bytes required by the feature type (e.g. 4 bytes for an `int`), and `entries` provides array support for feature vectors that include historical values. LAKE avoids tracking the actual value type for feature vector entries, and instead provides the necessary capacity and treats values as untyped bytes. For most features types, e.g. an integer value, `entries` is 1, meaning the vector includes a single scalar value. When `entries` is greater than 1, the feature is an array of length `entries` where the entry at index 0 is the most recent sample, and the entries at indices $1..(N-1)$ are the historical samples from the last $N-1$ feature vectors. We find features that comprise the last N measurements of a particular value common enough that providing API-level support for the idiom is a worthwhile simplification. An example of the idiom is illustrated in the case study below (§5.1).

Asynchrony and Module Boundaries. To understand design goal 2 above, consider that *synchronous* feature capture (interrogating relevant data structures just before invoking inference) can be impractical because module boundaries and locking disciplines can make access to widely dispersed data impractical. LAKE addresses this an asynchronous API that allows programmers to place simple calls at the code sites where instrumented data are already maintained, building up feature vectors over time. The register relies on lock-free data structures to enable instrumentation calls on arbitrary kernel threads without needing additional locking disciplines. The API supports an idiom where feature capture opened (calling `begin_fv_capture()`): while feature capture is open, individual feature vector values may be captured on any thread using `capture_feature()` which updates the value at the given key in the feature map (`kvpair`). We find there are situations that are significantly simplified for the kernel developer with support for incremental updates for feature values with `capture_feature_incr()`

(see example below: §5.1). Creation of a new feature vector sets a *begin* timestamp (`ts_begin`), while capture is finalized by committing which sets an end timestamp (`ts_end`).

Simplifying Batch Management. Because performance-accuracy profitability of ML is variable, we find that explicit control over batch size is a key parameter to expose to the kernel developer to modulate the use of accelerators. Querying the registry with a timestamp `ts` (get_features()) returns the first feature vector with for which `ts_begin <= ts <= ts_end`. Querying with a null timestamp returns a batch containing all the features in the circular buffer. The API can acknowledge consumption of that batch by calling `truncate_features()`. When the schema for a registry has features that rely on historical samples (`entries > 1` above), LAKE will always preserve the most recent feature vector on truncation to enable the system to correctly populate those feature values. The `score_features()` API invokes a programmer defined callback (specified with `register_classifier()`) to run inference. The policy function specified with `register_policy()` (§4.3) is invoked by the framework to manage accelerator use.

5.1 Feature Registry Case Study

Predicting I/O latency in systems with parallel and redundant storage (e.g. RAID) can improve throughput by rejecting high latency I/Os and re-issuing the same I/O to a different device [27]. We measure this workload in §7 but use it here to illustrate use of the feature registry API. Capturing I/O latency-related features requires inserting code at I/O's boundaries, at code locations that are different from where inference is invoked, making support for asynchronous feature construction necessary. Inference is invoked at I/O issue and the system classifies it as fast or slow based on a feature vector that includes the number of pending I/Os and the completion latency of a fixed number of previous I/Os.

Capturing the number of pending I/Os and latency of an I/O requires developers to insert code on both I/O issue and completion. Listing 4 shows pseudocode added to the `generic_make_request_checks` function, which is called on I/O issue, in order to capture the current state of the system as a feature vector. We store the time this I/O was issued (required to calculate latency), increase the number of pending I/Os in the current feature vector and commit the current state as a feature vector. Then, if either a pre-defined time quantum has passed or we reach a desired batch sized, we retrieve a batch from the registry, perform batch inference, act on the results per I/O, and clear the feature registry ring. Features must also be captured on I/O completion. Listing 5 shows pseudocode added to the `bio_endio` function, which computes how long the current I/O took to complete, decreases the amount of pending I/Os by one and updates the current feature vector.

Latency prediction has clear asynchrony in feature construction, and feature values may be captured conveniently on different threads. I/Os can be handled concurrently by the kernel, and manual state management and construction of a feature vector requires careful concurrency control. LAKE's feature registry eases these issues.

```
// called when issuing a block I/O in Linux
generic_make_request_checks(struct bio *bio)
{
  sys = "bio_latency_prediction"
  // store this I/O's start time
  getnstimeofday(&(bio->io_start_ts));
  // increment pending I/Os on this device
  capture_feature_incr(dev,sys,"pend_ios",1)
  // this I/O becomes a feature vector
  commit_feature_capture(dev, sys, now())
  if(quantum passed or batch > thresh) {
    // get all features vectors in the ring
    fvs = get_features(dev, sys, NULL)
    // do inference on all feature vecs
    scores = score_features(dev, sys, fvs);
    // reject, re-issue or accept I/Os
    ...take action based on scores...
    // reset the feature vector ring
    truncate_features(dev, sys, NULL)
  }
  //start new feature
  begin_fv_capture(dev, sys, now())
  ...
```

Figure 4: Pseudocode of I/O issue code to use LAKE feature registry for I/O latency prediction. Each block device needs its own feature registry (name parameter is the device's name, e.g. sda1).

```
// function called to end a block I/O
void bio_endio(struct bio *bio) {
  sys = "bio_latency_prediction"
  // get latency of this I/O
  lat = get_io_latency(bio->io_start_ts);
  // store this I/O's latency
  capture_feature(dev,sys,io_latencies,lat);
  // one less pending I/O on this device
  capture_feature_incr(dev,sys,pend_ios,-1)
  ...
```

Figure 5: Pseudocode of I/O completion code using LAKE feature registry for I/O latency prediction.

6 IMPLEMENTATION

Our LAKE prototype is based on Linux kernel version 6.0. Floating point operations, required by machine learning algorithms, are not supported in the kernel by default. Code regions that need to use floating points must be wrapped with macros that enable it (`kernel_fpu_begin` and `kernel_fpu_end`).

The LAKE API remoting system provides kernel space with the CUDA driver API version 11.0 as well as TensorFlow 2.4.0 and Keras 2.2.5.

The implementation of LAKE's API remoting system resembles an RPC system: *lakeLib* exports symbols (stubs) to the kernel and *lakeD* is the user space process that handles incoming requests. Commands sent between these two are transmitted through Netlink

sockets due to their low latency. Larger memory transfers are done through a *zero-copy shared memory* mechanism.

Communication Channel. LAKE requires efficient communication channels as applications can be call- or latency sensitive. Linux provides mechanisms for kernel-user communications, such as `ioctl`, system calls, signals, up-call, `mmap`, and sockets. We evaluate the alternatives in Table 2, which summarizes call time and latency to send a doorbell from kernel to user space. All mechanisms except mmap have similar latency, while device read/write and Netlink have additional caching or queuing layers. The `mmap` method is fastest but wastes CPU spinning, so we use Netlink sockets.

	Signal	Device R/W	Netlink	Mmap
Call time (μs)	56	6	11	6
Latency (μs)	56	57	54	6

Table 2: Average call time and latency to send a doorbell message from kernel to user.

Mapped Memory. Bulk data transfers between kernel and user space are done through *lakeShm*, *lakeShm* reserves a contiguous DMA region at load time through `dma_alloc_coherent`. A best-fit based memory allocator algorithm is used. Using mapped memory avoids transferring large data buffers across the kernel-user boundary. Figure 6 shows the round-trip transfer cost of varying sized messages. Transferring larger messages causes large overhead, which can be eliminated by *lakeShm*.

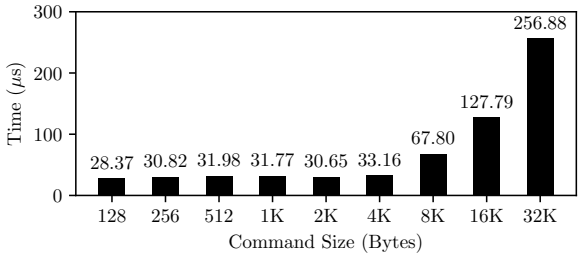

Figure 6: Overhead of sending Netlink messages of different sizes.

6.1 Discussion: Security Implications

LAKE introduces a user-space component to expose accelerators to user space, moving data that is private to the kernel through user space. In LAKE, the user-space daemon is a trusted process, which runs as root, similar to any other user-space daemon that integrates tightly with the kernel (e.g. user-space memory managers, schedulers, and file systems typical of micro-kernels, and user-mode device drivers that are prevalent in modern OSes such as Windows). Address-space separation provides strong security guarantees against leakage, despite the fact that the daemon does not execute in kernel mode. Nonetheless, for additional assurance, the user space daemon (*lakeD*) could be sandboxed and *seccomp* could be used. The *lakeD* daemon's interface with the OS is quite limited (it requires `ioctl` and `mmap` for *lakeShm*, `netlink` sockets for *lakeLib* and the *syscalls* done by the CUDA runtime). While we do not consider side-channels in this work, *lakeD* could be extended to use secure GPU TEEs like Graviton[69] or Telekine[30].

6.2 Source Code

In total, *lakeLib*, *lakeShm* (both kernel space code) and *lakeD* (user space code) consists of approximately 817, 826 and 1072 lines of C/C++ code, respectively, with an extra 769 lines of code for core common functionality. Our neural network for predicting I/O latency and its tooling consists of approximately 4157 lines of code. The other workloads and the modified eCryptfs consists of 1400 and 2925 lines of code, respectively. LAKE is open-source under GPLv3 and is available on GitHub at utcs-scea/LAKE.

7 EVALUATION

We identified several applications in kernel space that are candidates for ML-decisioning and HW-acceleration. We implemented GPU-accelerated versions of them using CUDA and LAKE (Table 3). This section describes each application and analyzes accelerator profitability. Data required by each application can usually be copied to the GPU asynchronously, before its execution. Thus, for some applications, we report execution time with (shown in figures as "LAKE (sync.)") and without synchronous data movement (shown as "LAKE").

Evaluation Scenarios. The benefits of ML on decision quality have been demonstrated in existing literature for all workloads. To validate that those benefits remain in the presence of hardware acceleration and components introduced by LAKE, we perform an end-to-end case study on I/O Latency Prediction, revisiting previous work [27] on more recent hardware, demonstrating additional benefits made possible with acceleration, and characterizing the impact of hardware evolution on ML profitability (§7.1). The remainder of this section evaluates the ability of our infrastructure to provide access to acceleration, make ML decisioning more performant, help manage contention, and simplify manage the variable profitability of acceleration (§7.2–§7.5).

Testbed : All of our evaluation was done on a server with two 16-core Intel Xeon Gold 6226R CPUs, 376 GiB DDR4 RAM, two NVIDIA A100 GPUs and three Samsung 980 Pro 1TB (PCIe 4.0) NVMes. We used Ubuntu 22.04 with our modified Linux kernel based on version 6.0.

7.1 End-to-End Study: I/O Latency Prediction

Predictable latency can be very useful for data-center systems serving interactive applications such as messaging and search [16]. Being able to infer SSD performance at a very fine-grained level, i.e. per I/O, can help these applications achieve performance predictability. If a system predicts an I/O will be slow, the latency penalty can be mitigated by issuing a duplicate I/O request to another storage node [27].

LinnOS [27] shows how an OS can learn and infer per I/O latency using a neural network. In the original paper, LinnOS improves the average I/O latencies by up to 79.6% and its model presents an accuracy of up to 97%. LinnOS classifies I/Os into slow or fast, using a threshold based on the system's state, e.g. number of pending I/Os and latency of most recently served I/Os.

Application	Description	ML Algorithm	Crossover	API	Inference Frequency
I/O Latency Prediction	Predict I/O latencies in storage systems	Neural Network	8	CUDA	Fine grained
Page Warmth	Predict pages hotness to keep in fast memory	LSTM	1	High level	Coarse grained
Load Balancing	Predict benefit of task-stealing between CPUs	Neural Network	256	CUDA	Fine grained
Filesystem Prefetching	Predict next I/O accesses for better read-ahead	Neural Network	64	CUDA	Coarse grained
Malware Detection	Detect malware/attacks	k-NN	128	CUDA	Coarse grained
Filesystem Encryption	Encrypt/Decrypt data to/from storage	-	16/128KB	CUDA	Fine grained

Table 3: Identified applications in kernel space that are candidates for GPU acceleration. Crossover point is the number of inputs (block size for file system encryption) in a batch in which using a GPU becomes profitable.

Trace Name	Avg IOPS	Avg Read/Write I/O size (KB)	Min/Max Arrival Time (us)
Azure	26k	30/19	0/324
Bing-I	4.8k	73/59	0/1.8k
Cosmos	2.5k	657/609	0/1.6k

Table 4: Characteristics of our generated traces based on the traces of LinnOS [27], rerated to double the IOPS.

To validate previous findings of ML benefit in an end-to-end setting with GPU acceleration, we port LinnOS's CPU-only neural network to a LAKE-powered kernel module that uses CUDA. We integrate LinnOS's [27] model into the LAKE kernel and evaluate the original model, as well as more complex models' benefits on our system. The traces used by LinnOS are not available publicly, so we generate traces with similar characteristics based on parameters presented in the paper, using an exponential distribution for inter-arrival time, a lognormal distribution for I/O size and a uniform distribution for I/O offset. The original work adopted a technique of "rerating" traces by reducing inter-arrival time to stress storage devices with different latency characteristics. Since NVMe technology has improved since LinnOS was published, we adopt the same technique, additionally amplifying I/O pressure. We rerate the traces presented as enterprise-level in the original work by doubling the average IOPS of the traces with smaller I/O sizes: Azure and Bing-I. The Cosmos trace was not rerated as it was already sufficiently demanding.

We find that the benefits of reissuing a read I/O to another device if the I/O is predicted as slow is highly dependent on the workload's characteristics and the testbed. For example, our (more recent) NVMe devices can exhibit read latencies up to three times lower than the original work's enterprise grade SSDs, use the PCIe 4.0 interface instead of PCI 3.0 and feature much larger DRAM caches. Larger caches absorb much more of the load, particularly for small I/Os, so the devices do not exhibit significant I/O read latency variance.

Workloads matching the characteristics of traces reported by LinnOS do not stress the NVMes and its caches, and the cost of running a neural network degrades average read latencies regardless of whether GPUs or CPUs are used. Consequently, for the workloads reported in the original paper, there is no benefit in rerouting I/Os. Table 4 shows the I/O properties of each trace. We report measurements for replaying the same trace on each NVMe (the workloads of the original work) and a mixed workload, which replays different traces on different NVMes. Our mixed workload replays each trace with a different default target NVMe, and reissues slow I/Os in round-robin fashion. We rerate all traces to three times their IOPS and combine these into a more intense mixed workload.

The neural network used by LinnOS is small: it contains two layers with 256 and 2 neurons, respectively. Maintaining low CPU utilization and low inference latency is the primary purpose of using such a simple model. We evaluate LAKE using this model, as well as more complex models with one and two additional layers to evaluate LAKE's ability to support richer, more accurate models based on acceleration. The added layers added have the same number of neurons as the first one. We suffix these implementations with *+1* and *+2*. These implementations have three layers with [256,256,2] neurons and four layers with [256,256,256,2] neurons, respectively.

For each workload, we evaluate the average read latency of the kernel's default behavior (baseline, no I/O rerouting), LinnOS's model using a CPU, and LAKE, which targets a CPU or a GPU based on a policy that chooses according to dynamic batch size and average I/O inter-arrival time. We additionally report average read latencies when LinnOS or LAKE use our augmented models. Figure 7 shows the results. We find that for mixed workloads that stress the devices in dissimilar ways, both LinnOS and LAKE provide lower average latency than the baseline. The benefits of ML are preserved using LAKE for GPU acceleration. Using the simple model, the benefit of LAKE is lower relative to LinnOS execution due to latency overheads for dynamic batch formation and data transfer. LAKE performs better with high IOPS workloads like Azure due to increased batching. As model complexity increases, we find that offload to a GPU becomes performance profitable relative to a CPU for all cases.

Our experience with I/O latency prediction underscores our findings reported earlier: the benefit of ML is sensitive to a number of factors including workload, subsystem, and hardware. LAKE's framework is effective for using GPUs only when they will be profitable and falling back to the CPU when batch formation does not produce sufficiently large batches. However, given that even the original CPU-based model actually harms performance when applications do not stress the device, some mechanism to modulate the use of ML even on the CPU is a likely necessity. We believe the same framework LAKE provides for managing contention and selecting between CPU and GPU can be used to implement policies that avoid using ML when it does not help, and will explore this in future work.

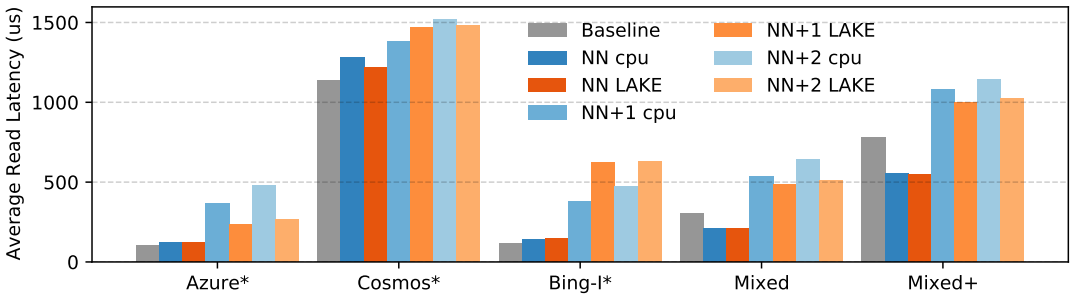

Figure 7: Average latency of our workloads without I/O rerouting (baseline), rerouting through neural networks using cpu, and neural networks using a GPU through LAKE. The suffixes *+1* and *+2* indicate how many extra layers were added to the original two layer neural network. *Traces were generated to reflect characteristics reported in [27].

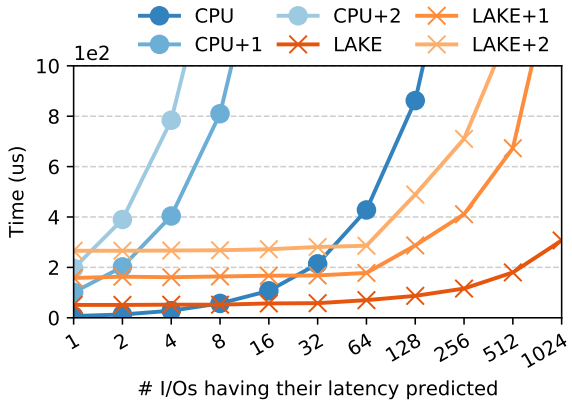

Figure 8: I/O latency prediction time for variable batch sizes using cpu and GPU through LAKE, including data copying latencies. The suffixes *+1* and *+2* indicate how many extra layers were added to the original two layer neural network.

To evaluate batch formation thresholds required for LAKE to be performance profitable, Figure 8 shows inference time of our the original NN and the two augmented NNs on various batch sizes on CPU and GPU using LAKE. For the original NN, using a GPU is profitable for batch sizes greater than 8, which is feasible in systems with high IOPS rate. For instance, a provisioned SSD in Amazon EBS[1] supports 256k IOPS, meaning an average inter-arrival time of I/O requests to be 4μs. In this scenario, with a batch size of 8, doing inference as requests arrive using a CPU would take around 120μs (each inference on CPU takes around 15μs). Instead, we can wait for 8 requests to arrive (28μs) and do inference on a GPU in 58μs, totaling 86μs, a reduction of 28%. For the augmented NNs, with one and two extra layers, using a GPU is profitable for batch sizes larger than 3 and 2, respectively. Profitability increases as batch size increases.

7.2 Page Warmth Classification

Multi-tiered memory systems (available with up to 18TB of memory [2, 6]) combine different memory types (e.g. RAM, NVMe) to expand capacity but face data placement challenges. Placing hot pages (frequently accessed or soon-to-be accessed pages) in a lower memory tier can harm performance. Cold pages stored in high-tier memory wastes precious faster memory. The challenge for this subsystem is to classify pages to inform where they should be stored to optimize performance (also called page scheduling). Recent work [37] shows several limitations in current systems exist.

ML for page scheduling is promising. Kleio [18] simulates different page schedulers and implements a LSTM-based classifier, which makes better decisions than a history based solution [50]. Kleio is implemented using TensorFlow, so we port it to a kernel space module using LAKE. Figure 9 shows the inference time for different sized batch of inputs. We observe significant speedup when using a GPU through LAKE instead of CPUs. There is no significant difference when running Kleio on user or kernel space, since the cost of LAKE's API remoting system is negligible relative to the execution time.

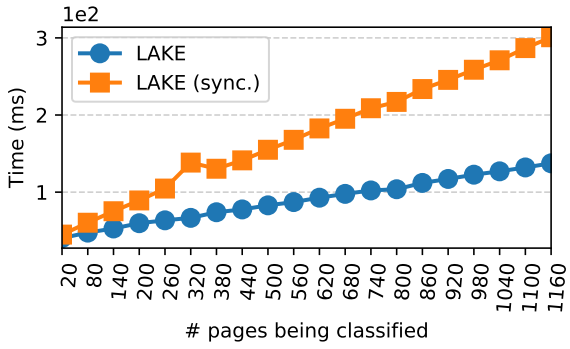

Figure 9: Time taken to predict page warmth through Kleio, a LSTM-based model for variable batch sizes. Data is shown only for LAKE (sync.) because data movement is handled synchronously by TensorFlow.

7.3 Load Balancing

Imbalanced load between CPU cores can cause some CPUs to be overloaded while others are under-utilized, hurting general system performance. The Linux kernel does load balancing using a pull-based, work-stealing mechanism that moves processes' execution

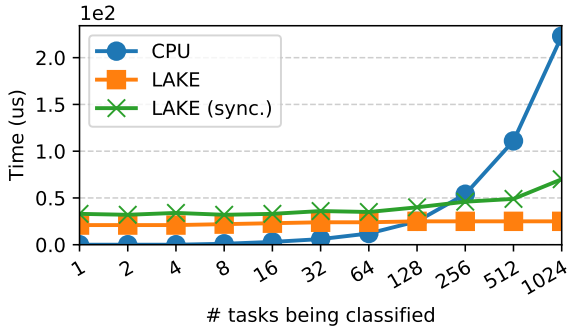

Figure 10: Time taken to predict load balancing decisions using MLLB in variable batch sizes.

between CPUs. Previous work [47] has found that the load balancing heuristic has performance-critical bugs leading to non-optimal choices.

ML is a promising alternative. The difficulty of writing good heuristics for load balancing is aggravated due to the amount of possible system configuration: scheduling group organization, application execution patterns (e.g. many short applications or few long-running), the number of NUMA nodes and their memory distances. The predominant hardship in developing an ML algorithm for load balancing is collecting data in order to compare different algorithm outcomes and estimate a reward to reinforce an ML model.

MLLB [15] used a multi-layer perceptron for load balancing. We port MLLB's model to CUDA and place it in a kernel module using LAKE. The inference time for variable numbers of processes is shown in Figure 10. Using a GPU is only profitable for batches larger than 128 inputs. Current servers with many tens of CPUs and many processes per core can easily exceed this threshold: for example, prior work from 2013 showed that 90% of Google servers loaded with up to 4500 threads concurrently [75].

7.4 Filesystem Prefetching

On-demand reads from storage can be orders of magnitudes slower than reading from memory. Prefetching blocks reduces the time wasted waiting for device storage, and can improve application throughput by up to 50% [61]. Linux's fixed *readahead* prefetch policy sequentially prefetches a configurable amount of data. Applications that are aware their I/Os will not be sequential can advise the kernel not to prefetch.

Predicting non-sequential reads from an application efficiently using heuristics is challenging. Through statistics collection from an application's I/O operations, ML algorithms can learn applications' patterns and perform tailored prefetching. KML [11] uses a pre-trained neural network to classify applications according to I/O patterns, where each pattern has an optimal readahead configuration. KML improves RocksDB throughput by up to 2.3× when using an SSD.

We port KML's NN, originally implemented for CPUs, to a kernel space module that uses CUDA through LAKE. Figure 11 shows the time taken to classify a variable number of inputs. The GPU is profitable more than 64 inputs are batched. We believe this model and the file system can be expanded to classify behavior and configure readahead per-file, as opposed to per-process.

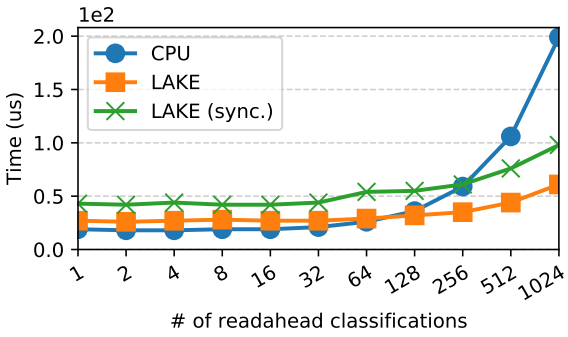

Figure 11: Time to predict file system readahead in variable batch sizes.

7.5 Malware Detection

Prior work [17, 21, 36, 46] can detect malicious software by analyzing performance counters and system call traces using ML classifiers. J. Demme, et al. [17] demonstrates accurate detection of host compromise using a K-Nearest Neighbors (KNN) classifier to achieve >90% accuracy with <10% false positive rate. G. Kim et al. [36] use LSTM to analyze system-call traces for anomaly-based intrusion detection with >95% accuracy at <5.5% false positive rate. We developed a kernel driver which uses a KNN classifier to classify user programs as malicious or benign.

Our KNN classifier operates on feature vectors which can track syscall frequencies and PMU (Performance Monitoring Unit) counters relevant to the target malware. For example, a kernel developer wishing to expose Spectre attacks [39] in the wild may use PMU counters to track microarchitectural state perturbations through cache misses, page faults, and branch mispredictions. Alternatively, a developer who wishes to detect abuse of the syscall API may classify processes based on how frequently they use unusual or suspicious syscall sequences.

Figure 12 shows the average running time for performing 4,096 KNN queries on a database of 16,384 reference points. We vary the number of features in each sample from 1 to 1024, and classify queries based on their 16 nearest neighbors. The GPU implementation of KNN achieves about $1.5k\times$ speedup over the sequential CPU implementation, and the overhead of using CUDA from user space and kernel space through LAKE is negligible: on average 4.2% and at most 5.6%.

7.6 User/Kernel Contention

LAKE enables kernel space access to accelerators without degrading performance of user space processes that also use them. Figure 1 shows that contention can lead to performance degradation and performance instability. Figure 13 demonstrates the impact of the adaptive contention mediation policy shown in Figure 3. Our I/O latency classifier (§ 7.1) avoids competing with user space processes for the GPU. The policy passively monitors accelerator utilization using accelerator APIs. As soon as the user workload

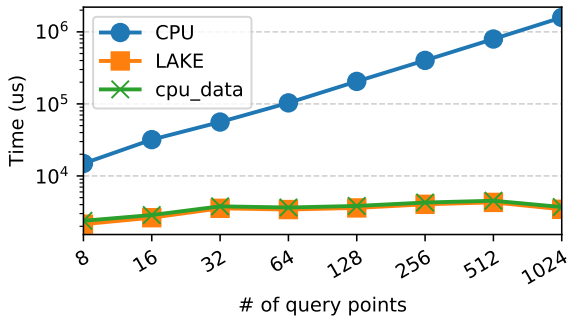

Figure 12: Time taken to predict if a sequence of syscalls could be from malicious software for various input sizes (number of syscalls in feature vector) using 4096 K-Nearest Neighbors (KNN) on a database of 16,384 reference points.

(a GPU-accelerated parallel hashing algorithm) begins to execute, LAKE detects pressure for GPU resources, and switches execution CPU. When the user process terminates, LAKE reclaims the GPU.

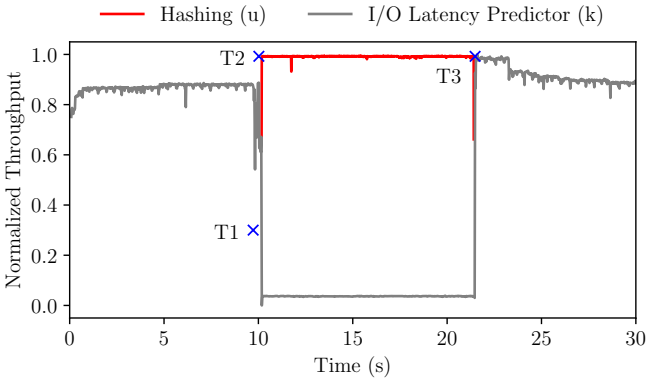

Figure 13: Kernel and user space throughput, normalized against peak throughput, under our adaptive contention-averse policy. At $T0$, our GPU-accelerated I/O latency classifier is running. At $T1$, a user space process that calculates hashes is launched. At $T2$, the user space process starts doing hashed on the GPU. LAKE detects contention for GPU compute resources and falls-back to CPU. At $T3$, the user space process terminates. LAKE detects that the GPU is not contended and switches execution back to the GPU.

7.7 Non-ML acceleration opportunities

File system Encryption eCryptfs [25] is a cryptographic file system that stacks on top of existing file systems and encrypts data-at-rest. We modified eCryptfs to use AES-GCM [35] instead of CBC because it is parallelizable. We also create a Linux crypto API cipher that does AES-GCM encryption and decryption using a LAKE-backed GPU.

Figure 14 shows the bandwidth of the CPU and LAKE implementations of eCryptfs. We clear all page caches and perform sequential reads and (synchronous) writes using varying block sizes. The read-ahead size of the disk is set to the block size, in order to fully overlap the decryption and file system read. The LAKE-supported eCryptfs can achieve 840 MB/s for reading, 6× higher compared to the 142 MB/s achieved using only CPU. Similar performance difference is observed for writing: 836 MB/s vs. 136 MB/s. The CPU eCryptfs has near constant read and write throughput, because the cryptographic operations become the performance bottleneck, instead of the disk.

We measured eCryptfs with AES-NI, a CPU instruction set for accelerated encryption and decryption.AES-NI reaches its peak of read and write performance at around 670 MB/s and 560 MB/s, respectively. The LAKE-powered eCryptfs surpasses AES-NI read performance for blocks larger than 16 KB because read-ahead fetches and decrypts more blocks than requested, creating larger decryption blocks. On the other hand, LAKE only exceeds AES-NI on write performance for blocks larger than 128 KB. By combining LAKE and AES-NI, doing cypher operations concurrently, we increase read and write performance by 31% and 22%, respectively, relative to just using a GPU through LAKE.

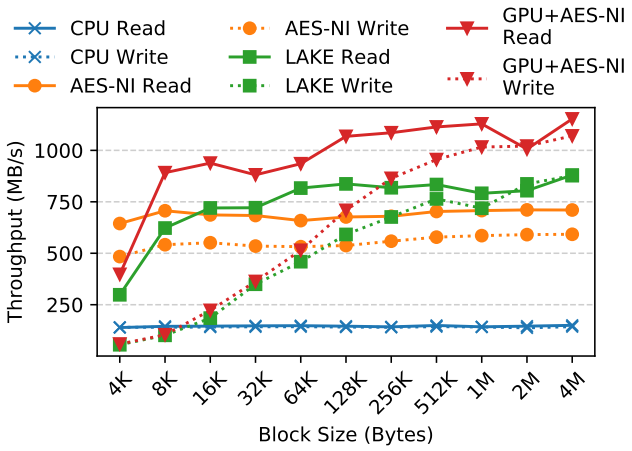

Figure 14: I/O throughput of AES-GCM-based eCryptfs, encrypting/decrypting on CPU, AES-NI, and a GPU through LAKE.

7.8 CPU Utilization

Figure 15 shows the CPU and GPU utilization of decrypting a 2 GB file on the CPU or GPU with a 2 MB block size using the original and our LAKE-powered eCryptfs implementation (§7.7). The CPU and AES-NI were measured using kernel CPU utilization. LAKE is split into kernel CPU utilization (LAKE CPU), user space API handler CPU utilization (LAKE API) and GPU utilization. With AES-NI enabled, there is a short peak of utilization to decrypt all data. LAKE consumes on average 20% of CPU resources, while the original CPU and the AES-NI versions use 56% and 24%, respectively.

8 RELATED WORK

Accelerator virtualization. LAKE is closely related to accelerator virtualization through a core technique: API remoting. Previous API remoting-based virtualization techniques [3, 20, 24, 58, 60, 68, 71] require enormous developer effort to virtualize a single accelerator.

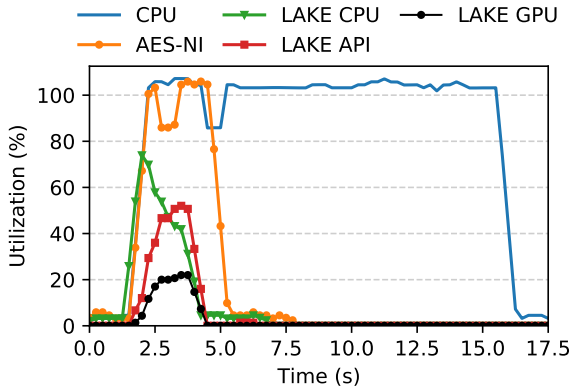

Figure 15: CPU and GPU utilization of reading a 2 GB file sequentially on eCryptfs with a block size of 2 MB, using CPU only, AES-NI and LAKE.

Hardware acceleration in kernel. The majority of existing kernel acceleration systems such as Barracuda, PacketShader and Snap [13, 26, 45, 64–66] use up-calls to enable OS-level interaction with accelerators. They copy kernel data structures into user space for kernel bypass techniques [9, 19, 38, 55, 70]. These systems introduce new and often workload-specific kernel APIs to access user space acceleration services, usually with significant developer effort. LAKE provides a common interface that these systems could use.

Machine learning in kernel. Researchers have proposed to optimize or analyze kernel workloads such as process scheduling, I/O prefetching [44, 51, 67] and others [15, 18, 22, 27, 28, 48, 56, 73] with ML algorithms. KMLib [12] is a recent attempt to bring ML ecosystem to operating systems, but it offers limited functionalities compared to mainstream ML frameworks and no support for accelerators.

9 CONCLUSION

We present LAKE, which gives kernel space applications access to accelerators for ML-assisted decision-making. We identified and evaluated five ML-based kernel space applications that can profit from accelerators, demonstrating the ability of LAKE to provide performant acceleration and simplify the challenge of collecting ML features in the presence of abstraction boundaries, asynchrony, and multi-threading.

10 ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd Hank Hoffmann for feedback and insightful comments and ideas that helped us elevate and improve this paper. This work is supported in part by NSF grants CNS-1846169, CNS-2006943, CNS-2202649, CNS-2207317, the Texas Systems Research Consortium and the National Nuclear Security Administration Award Number DE-NA0003969.

A ARTIFACT APPENDIX

A.1 Abstract

The artifacts of LAKE are, at a high-level, two pieces: our modified Linux kernel 6.0 and the kernel modules used in our evaluation. We provide scripts that produce graphs similar to the ones presented in our evaluation and the data we collected on our testbed. The basic workflow of our evaluation is: while booted in our kernel, enable our kernel space API remoting and our user space daemon and run each workload.

A.2 Artifact check-list (meta-information)

- **Program: Modified Linux kernel, source code of kernel modules, shell and python scripts.**
- **Run-time environment: Linux and root access are required, we suggest Ubuntu 22.04.**
- **Hardware: At least one Nvidia GPU is required. The latency prediction workload requires three storage devices.**
- **Execution: Basic knowledge of *tmux*, shell and python scripts.**
- **Metrics: latency of ML inferences, latency of I/Os, resource utilization and throughput**
- **Output: Output is written to kernel log. Scripts that parse the log are provided.**
- **Experiments: Reproducing of our graphs using collected data.**
- **How much disk space required (approximately)?: 60GB.**
- **How much time is needed to prepare workflow (approximately)?: Three hours.**
- **How much time is needed to complete experiments (approximately)?: Five hours.**
- **Publicly available?: Yes.**
- **Code licenses (if publicly available)?: The modified Linux kernel and parts of the modified eCryptfs code are licensed under GPLv2. Our code is licensed under GPLv3.**
- **Archived?: 10.5281/zenodo.7277139 and 10.5281/zenodo.7277147**

A.3 Description

A.3.1 How to access. Our artifacts can be found on Zenodo (link to archive in §A.2) and GitHub. We recommend retrieving it through GitHub since this is a continuing project. Our modified kernel is available at utcs-scea/LAKE-linux-6.0 and our code base at utcs-scea/LAKE.

A.3.2 Hardware dependencies. We require at least one CUDA enabled Nvidia GPU. Results observed may vary depending on the testbed. Our I/O latency prediction experiment requires three storage devices without important data, since we do writes to arbitrary offsets.

A.3.3 Software dependencies. We recommend Ubuntu 22.04. The system must have a working CUDA runtime installation and Nvidia GPU driver (installed after booting into our kernel). We require the user to have root access.

A.4 Installation

Our artifacts contain a README.md with more detailed steps to install our system. The setup of LAKE consists of two steps: compiling and installing our Linux kernel and our kernel API remoting system.

First, clone both of our repositories: utcs-scea/LAKE-linux-6.0 and utcs-scea/LAKE. Then, the following steps should be taken:

(1) Compile, install and boot into our Linux kernel. A script that automates this process is provided (*full_compilation.sh*).
(2) Add CMA argument (*cma=128M@0-4G*) to the kernel parameters.
(3) Reboot into our kernel.
(4) Install the CUDA runtime (we use version 11.7, but other versions should work) and an Nvidia driver for the GPU. We provide the link to download both in our *README*.

A.5 Experiment workflow

To facilitate evaluating our artifacts, we provide a script that opens a tmux session with pre created panes *tmux.sh*. Kernel output is written to the kernel log. The provided tmux script will show the kernel log on the lower left pane. The top left pane is where the kernel API remoting system will run. The right pane(s) is where the workloads are executed.

The crossover experiments are executed similarly: go into their directory, build with *make*, run with *./run.sh*, go into the *ae_plot* directory and execute *plot.py*. This will generate the graph in pdf format.

The eCryptfs requires building the modules (*make* at the *src/ecryptfs* directory) and executing an automated script (*run.py* at *benchmarks/ecryptfs*). Before executing the script, variables inside the script might have to be changed to set which directory and drive the script will use.

The utilization and contention experiments follow a similar pattern. Their directories are *contention* and *utilization* inside *benchmarks*. Build with *make* and run with *sudo -E python3 run.py*.

The I/O prediction experiments require training, setting weights manually by editing code, choosing and enabling a predictor module and replaying traces so that finally data is produced. We provide detailed instructions on how to replicate this experiment (and previous experiments) in a pdf document at the root of our repository called *ae_experiments.pdf*.

A.6 Evaluation and expected results

We provide scripts that will plot graphs similar to ours and our collected data for easier visual comparison. Raw data can be retrieved from the kernel log.

Our work depends heavily on the tested. We do not expect the results to be the same, but we expect they will display similar trends.

A.7 Experiment customization

Our kernel API remoting currently supports a subset of the CUDA driver API to kernel space. Readers are encouraged to write kernel modules that use the CUDA API in kernel space to realize novel ideas. We recommend our *hello_driver* module as a starting point.

A.8 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

REFERENCES

[1] [n. d.]. Amazon EBS volume types. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-volume-types.html. ([n. d.]).
[2] [n. d.]. Available first on Google Cloud: Intel Optane DC Persistent Memory | Google Cloud Blog. ([n. d.]). https://cloud.google.com/blog/topics/partners/available-first-on-google-cloud-intel-optane-dc-persistent-memory (Last access: June, 2022).
[3] [n. d.]. Bitfusion: The Elastic AI Infrastructure for Multi-Cloud. https://bitfusion.io. ([n. d.]). Accessed: 2019-04.
[4] [n. d.]. eBPF - Introduction, Tutorials & Community Resources. ([n. d.]). https://ebpf.io/ (Last access: June, 2022).
[5] [n. d.]. Graphcore: Accelerating machine learning for a world of intelligent machines. https://www.graphcore.ai. ([n. d.]). Accessed: 2019-12.
[6] [n. d.]. Introducing new product innovations for SAP HANA, Expanded AI collaboration with SAP and more | Azure Blog and Updates | Microsoft Azure. ([n. d.]). https://azure.microsoft.com/en-us/blog/introducing-new-product-innovations-for-sap-hana-expanded-ai-collaboration-with-sap-and-more/ (Last access: June, 2022).
[7] [n. d.]. NVIDIA Releases Open-Source GPU Kernel Modules. ([n. d.]). https://developer.nvidia.com/blog/nvidia-releases-open-source-gpu-kernel-modules/ (Last access: June, 2022).
[8] [n. d.]. Optimizing Recurrent Neural Networks in cuDNN 5. ([n. d.]). https://developer.nvidia.com/blog/optimizing-recurrent-neural-networks-cudnn-5/ (Last access: May, 2022).
[9] Accessed: 2020. DPDK documentation. https://doc.dpdk.org/guides/. (Accessed: 2020).
[10] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, USA, 265–283.
[11] Ibrahim Umit Akgun, Ali Selman Aydin, Aadil Shaikh, Lukas Velikov, and Erez Zadok. 2021. A Machine Learning Framework to Improve Storage System Performance. In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '21)*. Association for Computing Machinery, New York, NY, USA, 94–102. https://doi.org/10.1145/3465332.3470875
[12] Ibrahim Umit Akgun, Ali Selman Aydin, and Erez Zadok. 2020. KMLIB: TOWARDS MACHINE LEARNING FOR OPERATING SYSTEMS.
[13] André Brinkmann and Dominic Eschweiler. 2009. A microdriver architecture for error correcting codes inside the Linux kernel. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 35.
[14] Chandranil Chakraborttii and Heiner Litz. 2020. Learning I/O Access Patterns to Improve Prefetching in SSDs. In *Machine Learning and Knowledge Discovery in Databases: Applied Data Science Track: European Conference, ECML PKDD 2020, Ghent, Belgium, September 14-18, 2020, Proceedings, Part IV*. Springer-Verlag, Berlin, Heidelberg, 427–443. https://doi.org/10.1007/978-3-030-67667-4_26
[15] Jingde Chen, Subho S. Banerjee, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. Machine Learning for Load Balancing in the Linux Kernel. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '20)*. Association for Computing Machinery, New York, NY, USA, 67–74. https://doi.org/10.1145/3409963.3410492
[16] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 153–167.
[17] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. 2013. On the Feasibility of Online Malware Detection with Performance Counters. *SIGARCH Comput. Archit. News* 41, 3 (June 2013), 559–570. https://doi.org/10.1145/2508148.2485970
[18] Thaleia Dimitra Doudali, Sergey Blagodurov, Abhinav Vishnu, Sudhanva Gurumurthi, and Ada Gavrilovska. 2019. Kleio: A Hybrid Memory Page Scheduler with Machine Intelligence. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '19)*. Association for Computing Machinery, New York, NY, USA, 37–48. https://doi.org/10.1145/3307681.3325398
[19] Peter Druschel, Larry L. Peterson, and Bruce S. Davie. 1994. Experiences with a High-Speed Network Adaptor: A Software Perspective. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications (SIGCOMM '94)*. Association for Computing Machinery, New York, NY, USA, 2–13. https://doi.org/10.1145/190314.190315
[20] José Duato, Antonio J Pena, Federico Silla, Juan C Fernandez, Rafael Mayo, and Enrique S Quintana-Orti. 2011. Enabling CUDA Acceleration within Virtual Machines using rCUDA. In *2011 18th International Conference on High Performance*

Computing. IEEE, 1–10.

[21] E. Eskin, Wenke Lee, and S. J. Stolfo. 2001. Modeling system calls for intrusion detection with dynamic window sizes. In *Proceedings DARPA Information Survivability Conference and Exposition II. DISCEX'01*, Vol. 1. 165–175 vol.1. https://doi.org/10.1109/DISCEX.2001.932213

[22] Gaddisa Olani Ganfure, Chun-Feng Wu, Yuan-Hao Chang, and Wei-Kuan Shih. 2020. DeepPrefetcher: A Deep Learning Framework for Data Prefetching in Flash Storage Devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 3311–3322. https://doi.org/10.1109/TCAD.2020.3012173

[23] Shay Gueron. 2009. Intel's New AES Instructions for Enhanced Performance and Security. In *FSE*.

[24] Vishakha Gupta, Ada Gavrilovska, Karsten Schwan, Harshvardhan Kharche, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. 2009. GViM: GPU-accelerated Virtual Machines. In *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*. ACM, 17–24.

[25] Mike Halcrow. 2007. ECryptfs: A stacked cryptographic filesystem. *Linux Journal* 2007 (04 2007).

[26] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. 2011. PacketShader: a GPU-accelerated software router. *ACM SIGCOMM Computer Communication Review* 41, 4 (2011), 195–206.

[27] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S. Gunawi. 2020. LinnOS: Predictability on Unpredictable Flash Storage with a Light Neural Network. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 173–190. https://www.usenix.org/conference/osdi20/presentation/hao

[28] Shifu Hou, Aaron Saas, Lifei Chen, and Yanfang Ye. 2016. Deep4MalDroid: A Deep Learning Framework for Android Malware Detection Based on Linux Kernel System Call Graphs. In *2016 IEEE/WIC/ACM International Conference on Web Intelligence Workshops (WIW)*. 104–111. https://doi.org/10.1109/WIW.2016.040

[29] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. 2021. GhOSt: Fast & Flexible User-Space Delegation of Linux Scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 588–604. https://doi.org/10.1145/3477132.3483542

[30] Tyler Hunt, Zhipeng Jia, Vance Miller, Ariel Szekely, Yige Hu, Christopher J. Rossbach, and Emmett Witchel. 2020. Telekine: Secure Computing with Cloud GPUs. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 817–833. https://www.usenix.org/conference/nsdi20/presentation/hunt

[31] Paras Jain, Xiangxi Mo, Ajay Jain, Harikaran Subbaraj, Rehan Sohail Durrani, Alexey Tumanov, Joseph Gonzalez, and Ion Stoica. 2018. Dynamic Space-Time Scheduling for GPU Inference. In *Thirty-second Conference on Neural Information Processing Systems*.

[32] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/3079856.3080246

[33] Kostis Kaffes, Dragos Sbirlea, Yiyan Lin, David Lo, and Christos Kozyrakis. 2020. Leveraging Application Classes to Save Power in Highly-Utilized Data Centers. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 134–149. https://doi.org/10.1145/3419111.3421274

[34] Y. Kang, Y. Kee, E. L. Miller, and C. Park. 2013. Enabling cost-effective data processing with smart SSD. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*. 1–12.

[35] Emilia Käsper and Peter Schwabe. 2009. Faster and timing-attack resistant AES-GCM. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 1–17.

[36] Gyuwan Kim, Hayoon Yi, Jangho Lee, Yunheung Paek, and Sungroh Yoon. 2016. LSTM-Based System-Call Language Modeling and Robust Ensemble Method for Designing Host-Based Intrusion Detection Systems. (2016). arXiv:cs.CR/1611.01726

[37] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. 2021. Exploring the Design Space of Page Management for Multi-Tiered Memory Systems. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 715–728. https://www.usenix.org/conference/atc21/presentation/kim-jonghyeon

[38] Joongi Kim, Keon Jang, Kyung A Lee, Sangwook Ma, Junhyun Shim, and Sunny Moon. 2015. NBA (network balancing act): A high-performance packet processing framework for heterogeneous processors. *Proceedings of the 10th European Conference on Computer Systems, EuroSys 2015* (04 2015). https://doi.org/10.1145/2741948.2741969

[39] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2019. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1–19.

[40] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 489–504. https://doi.org/10.1145/3183713.3196909

[41] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'12)*. Curran Associates Inc., Red Hook, NY, USA, 1097–1105.

[42] Arezki Laga, Jalil Boukhobza, Michel Koskas, and Frank Singhoff. 2016. Lynx: a learning linux prefetching mechanism for SSD performance model. In *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. 1–6. https://doi.org/10.1109/NVMSA.2016.7547186

[43] Arezki Laga, Jalil Boukhobza, Michel Koskas, and Frank Singhoff. 2016. Lynx: a learning linux prefetching mechanism for SSD performance model. In *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. 1–6. https://doi.org/10.1109/NVMSA.2016.7547186

[44] A. Laga, J. Boukhobza, M. Koskas, and F. Singhoff. 2016. Lynx: a learning linux prefetching mechanism for SSD performance model. In *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. 1–6.

[45] W. Lin, C. Tu, C. Yeh, and S. Hung. 2017. GPU acceleration for Kernel Samepage Merging. In *2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. 1–6.

[46] Ming Liu, Zhi Xue, Xianghua Xu, Changmin Zhong, and Jinjun Chen. 2018. Host-Based Intrusion Detection System with System Calls: Review and Future Trends. *ACM Comput. Surv.* 51, 5, Article 98 (Nov. 2018), 36 pages. https://doi.org/10.1145/3214304

[47] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. 2016. The Linux Scheduler: A Decade of Wasted Cores. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. Association for Computing Machinery, New York, NY, USA, Article 1, 16 pages. https://doi.org/10.1145/2901318.2901326

[48] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. 2020. *Learning-Based Memory Allocation for C++ Server Workloads*. Association for Computing Machinery, New York, NY, USA, 541–556. https://doi.org/10.1145/3373376.3378525

[49] Hasan Al Maruf and Mosharaf Chowdhury. 2020. Effectively Prefetching Remote Memory with Leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 843–857. https://www.usenix.org/conference/atc20/presentation/al-maruf

[50] Mitesh R. Meswani, Sergey Blagodurov, David Roberts, John Slice, Mike Ignatowski, and Gabriel H. Loh. 2015. Heterogeneous memory architectures: A HW/SW approach for mixing die-stacked and off-package memories. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 126–136. https://doi.org/10.1109/HPCA.2015.7056027

[51] Atul Negi and P Kishore Kumar. 2005. Applying machine learning techniques to improve linux process scheduling. In *TENCON 2005-2005 IEEE Region 10 Conference*. IEEE, 1–6.

[52] Rajiv Nishtala, Paul Carpenter, Vinicius Petrucci, and Xavier Martorell. 2017. Hipster: Hybrid Task Manager for Latency-Critical Cloud Workloads. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 409–420. https://doi.org/10.1109/HPCA.2017.13

[53] Marco Nobile, Paolo Cazzaniga, Andrea Tangherloni, and Daniela Besozzi. 2016. Graphics processing units in bioinformatics, computational biology and systems biology. *Briefings in Bioinformatics* 18 (07 2016), bbw058. https://doi.org/10.1093/bib/bbw058

[54] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 361–378. https://www.usenix.org/conference/nsdi19/presentation/ousterhout

[55] I. Pratt and K. Fraser. 2001. Arsenic: a user-accessible gigabit Ethernet interface. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No.01CH37213)*, Vol. 1. 67–76 vol.1.

[56] Yiming Qiu, Hongyi Liu, Thomas Anderson, Yingyan Lin, and Ang Chen. 2021. Toward Reconfigurable Kernel Datapaths with Learned Optimizations. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 175–182. https://doi.org/10.1145/3458336.3465288

[57] Rajat Raina, Anand Madhavan, and Andrew Y. Ng. 2009. Large-Scale Deep Unsupervised Learning Using Graphics Processors. In *Proceedings of the 26th Annual International Conference on Machine Learning (ICML '09)*. Association for Computing Machinery, New York, NY, USA, 873–880. https://doi.org/10.1145/1553374.1553486

[58] Carlos Reaño, Antonio J Peña, Federico Silla, José Duato, Rafael Mayo, and Enrique S Quintana-Ortí. 2012. CU2rCU: Towards the Complete rCUDA Remote GPU Virtualization and Sharing Solution. In *2012 19th International Conference on High Performance Computing*. IEEE, 1–10.

[59] Christopher J Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. 2011. PTask: operating system abstractions to manage GPUs as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 233–248.

[60] Lin Shi, Hao Chen, Jianhua Sun, and Kenli Li. 2012. vCUDA: GPU-Accelerated High-Performance Computing in Virtual Machines. *IEEE Trans. Comput.* 61, 6 (2012), 804–816.

[61] Elizabeth Shriver, Christopher Small, and Keith A. Smith. 1999. Why Does File System Prefetching Work?. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '99)*. USENIX Association, USA, 6.

[62] Warren Smith, Ian Foster, and Valerie Taylor. 2006. *Predicting application run times using historical information.* Vol. 64. 122–142. https://doi.org/10.1007/BFb0053984

[63] Pavel Sukharev, Dmitry Silnov, and Maxim Shishkin. 2019. Determining Optimal Mining Work Size on the OpenCL Platform for the Ethereum Cryptocurrency. *International Journal on Advanced Science, Engineering and Information Technology* 9 (10 2019), 1528. https://doi.org/10.18517/ijaseit.9.5.5820

[64] Weibin Sun and Robert Ricci. 2013. Augmenting operating systems with the GPU. *arXiv preprint arXiv:1305.3345* (2013).

[65] Weibin Sun and Robert Ricci. 2013. Fast and flexible: parallel packet processing with GPUs and click. In *Proceedings of the ninth ACM/IEEE symposium on Architectures for networking and communications systems*. IEEE Press, 25–36.

[66] Weibin Sun, Robert Ricci, and Matthew L Curry. 2012. GPUstore: harnessing GPU computing for storage systems in the kernel. In *Proceedings of the 5th Annual International Systems and Storage Conference*. ACM, 9.

[67] Sukanya Suranauwarat and Hideo Taniguchi. 2001. The Design, Implementation and Initial Evaluation of an Advanced Knowledge-Based Process Scheduler. *SIGOPS Oper. Syst. Rev.* 35, 4 (Oct. 2001), 61–81. https://doi.org/10.1145/506084.506090

[68] Kun Tian, Yaozu Dong, and David Cowperthwaite. 2014. A Full GPU Virtualization Solution with Mediated Pass-Through. In *2014 USENIX Annual Technical Conference (USENIX ATC'14)*. 121–132.

[69] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. 2018. Graviton: Trusted Execution Environments on GPUs. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 681–696. https://www.usenix.org/conference/osdi18/presentation/volos

[70] T. von Eicken, A. Basu, V. Buch, and W. Vogels. 1995. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*. Association for Computing Machinery, New York, NY, USA, 40–53. https://doi.org/10.1145/224056.224061

[71] Lan Vu, Hari Sivaraman, and Rishi Bidarkar. 2014. GPU Virtualization for High Performance General Purpose Computing on the ESX Hypervisor. In *Proceedings of the High Performance Computing Symposium (HPC '14)*. Society for Computer Simulation International, San Diego, CA, USA, Article 2, 8 pages. http://dl.acm.org/citation.cfm?id=2663510.2663512

[72] Yawen Wang, Daniel Crankshaw, Neeraja J. Yadwadkar, Daniel Berger, Christos Kozyrakis, and Ricardo Bianchini. 2022. SOL: Safe on-Node Learning in Cloud Platforms. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2022)*. Association for Computing Machinery, New York, NY, USA, 622–634. https://doi.org/10.1145/3503222.3507704

[73] Zhiyuan Xu, Jian Tang, Chengxiang Yin, Yanzhi Wang, and Guoliang Xue. 2019. Experience-Driven Congestion Control: When Multi-Path TCP Meets Deep Reinforcement Learning. *IEEE Journal on Selected Areas in Communications* 37, 6 (2019), 1325–1336. https://doi.org/10.1109/JSAC.2019.2904358

[74] Hangchen Yu, Arthur Michener Peters, Amogh Akshintala, and Christopher J. Rossbach. 2020. *AvA: Accelerated Virtualization of Accelerators.* Association for Computing Machinery, New York, NY, USA, 807–825. https://doi.org/10.1145/3373376.3378466

[75] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. 2013. CPI2: CPU Performance Isolation for Shared Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*. Association for Computing Machinery, New York, NY, USA, 379–391. https://doi.org/10.1145/2465351.2465388

[76] Yiying Zhang and Yutong Huang. 2019. "Learned": Operating Systems. *SIGOPS Oper. Syst. Rev.* (July 2019), 40–45. https://doi.org/10.1145/3352020.3352027