Toward Reconfigurable Kernel Datapaths with Learned Optimizations

Yiming Qiu
Rice University

Hongyi Liu
Rice University

Thomas Anderson
University of Washington

Yingyan Lin
Rice University

Ang Chen
Rice University

Abstract

Today's computing systems pay a heavy "OS tax", as kernel execution accounts for a significant amount of resource footprint. This is not least because today's kernels abound with hardcoded heuristics that are designed with unstated assumptions, which rarely generalize well for diversifying applications and device technologies.

We propose the concept of *reconfigurable kernel datapaths* that enables kernels to self-optimize dynamically. In this architecture, optimizations are computed from empirical data using machine learning (ML), and they are integrated into the kernel in a safe and systematic manner via an in-kernel virtual machine. This virtual machine implements the *reconfigurable match table (RMT)* abstraction, where tables are installed into the kernel at points where performance-critical events occur, matches look up the current execution context, and actions encode context-specific optimizations computed by ML, which may further vary from application to application. Our envisioned architecture will support both offline and online learning algorithms, as well as varied kernel subsystems. An RMT verifier will check program well-formedness and model efficiency before admitting an RMT program to the kernel. An admitted program can be interpreted in bytecode or just-in-time compiled to optimize the kernel datapaths.

CCS Concepts

• **Computing methodologies** → **Machine learning**; • **Software and its engineering** → **Operating systems**;

HotOS '21, May 31-June 2, 2021, Ann Arbor, MI, USA

ACM ISBN 978-1-4503-8438-4/21/05…$15.00
https://doi.org/10.1145/3458336.3465288

Keywords

Operating system kernels; Machine learning; RMT

ACM Reference Format:
Yiming Qiu, Hongyi Liu, Thomas Anderson, Yingyan Lin, and Ang Chen. 2021. Toward Reconfigurable Kernel Datapaths with Learned Optimizations. In *Workshop on Hot Topics in Operating Systems (HotOS '21), May 31-June 2, 2021, Ann Arbor, MI, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3458336.3465288

1 Introduction

Operating system kernels are being stressed from above and below. As a general-purpose resource manager, the OS kernel needs to support different applications, and it needs to multiplex different types of hardware platforms. As of late, both applications and hardware platforms are diversifying rapidly. On the applications side, for instance, container or microservice workloads are latency-sensitive, while MapReduce-like data processing jobs are throughput-oriented with intensive IO requirements (e.g., for bulk synchronization, checkpointing, or recovery). Home user applications (e.g., document or photo editing software) are yet another class, with their own complex disk IO patterns [25] and frequent interactions with the cloud. This complexity ensures that no one-size-fits-all optimization strategy exists that would simultaneously work well for all scenarios.

Likewise, hardware technologies are developing faster than the software system stack [7], with characteristics that differ from generation to generation, and from vendor to vendor within each generation. The best IO scheduling algorithms for hard disks will inevitably underperform for both SSDs and density-optimized shingled disks. To complicate this picture even further, devices are becoming smarter, enclosing embedded controllers that run proprietary algorithms for local management. Having these uncontrolled, blackbox code running in the devices may confound even the best-tuned kernel optimizations.

The confluence of these two trends calls for a fundamental rethink as to how the OS kernel should *specialize* for a particular scenario in order to perform well, and how these specializations should *generalize* for unseen scenarios that

may arise. Two recent approaches can be viewed as approximating this goal. The kernel bypass approach argues that resource management is best left to the application. Userland applications are given direct access to network cards or disks (e.g., using DPDK/SPDK), and they implement their own optimizations as needed. Alternatively, eBPF allows an application to dynamically inject constrained code into the kernel for customization, aiming to achieve similar effects. However, neither approach answers the question as to *what* optimizations should be implemented *when*. Applications may not have sufficient knowledge about the entire software and hardware stack (or even about their own behaviors) to adequately implement good optimizations, and any changes may be invalidated by new hardware. When individual applications choose their own strategies, the kernel also loses its centralized view needed for cross-application optimizations.

Our vision: Reconfigurable kernel datapaths. In this paper, we advocate for a fundamentally different approach, and provide an answer that draws inspiration from two lines of recent work—the increasingly powerful set of machine learning (ML) techniques, and the efforts in specializing network stacks with reconfigurable match table (RMT). Our key idea is to develop *reconfigurable kernel datapaths*, where the mechanisms are based on an RMT-style architecture in the kernel, and the policies are learned using ML. The OS kernel dynamically discovers the best policies for each scenario in the form of an RMT program, and enforces these policies by configuring the in-kernel virtual machine. By translating this programmable yet lightweight primitive into the OS kernel, we provide an architecture that allows for varied types of adaptivity. By harnessing the power of ML, we can eliminate many best-effort heuristics that abound in today's kernel datapaths, and enable optimizations to generalize to unseen applications, workloads, or hardware platforms.

Application-specific kernel optimizations and extensions were well explored in the 1990s. Exokernel [19] argues for eliminating OS abstractions entirely and leaving their implementations to the applications. SPIN [9], on the other hand, allows applications to inject safe code into the kernel for dynamic extension. They share similar limitations as their modern equivalents with kernel bypass and eBPF injection. In contrast, a key goal of our idea is to automatically identify kernel optimizations via ML-based reconfiguration, so applications no longer have to specialize the kernel in one-off manners.

Research challenges. Realizing our vision of reconfigurable kernel datapaths requires tackling a wide range of challenges: architecting an RMT-style virtual machine into the kernel, developing lightweight in-kernel learning algorithms, and applying the architecture to key kernel subsystems (e.g., scheduling, memory management, file systems, networking). We hope to make a notable dent in reducing the OS tax: it has been reported that kernel execution accounts for 20% of data center CPU cycles [28] while data centers represent 1% of worldwide electricity consumption [1]. Therefore, improving the efficiency of OS kernels has significant implications for a wide range of deployment scenarios.

2 Motivation

Machine learning techniques have produced early but successful results in computer systems, replacing well-tuned index structures for data retrieval [29], predicting hardware device state for better management [24], and managing C++ object memory efficiently [39]. Zhang and Huang [54] have argued that ML should be applied to the OS kernel as well. Our idea is inspired by this work, and it proposes a systematic approach to integrate ML into the kernel via an RMT virtual machine.

2.1 Envisioned benefits

We believe that reconfigurable kernel datapaths has the potential to unleash four classes of benefits that are hard to achieve in today's OS kernels.

#1. Lean monitoring: Operating system kernels employ a large set of runtime monitors, which aim at characterizing current workloads and activating different built-in heuristics. These monitoring events, however, introduce cache pollution, runtime overhead, and in some cases, they work by intentionally causing some performance degradation. An example of the latter is the CPU scheduler on a NUMA machine—in order to detect memory affinity, the scheduler needs to monitor a thread's page-level access pattern; Linux does this by periodically unmapping a process's pages, so that the kernel can trap the page faults and monitor access locations. By introducing ML, we can potentially enable the kernel to reduce the amount of necessary monitoring. For instance, a feature selection process using feature importance ranking [33] may allow the kernel to forego the monitoring of events that contribute little useful information.

#2. Better configurations: The wide range of heuristics and configuration parameters in the OS kernel may not be optimal; tuning kernel parameters to achieve better configurations is also a challenging task. Moreover, heuristics are activated only after a bootstrapping phase (e.g., is this particular thread I/O bound? then increase its scheduling priority). In our design, ML algorithms should be able to explore a broader range of decision making strategies, resulting in better configuration parameters, informed policies, and higher performance. The bootstrapping phase may be shortened or even eliminated if the OS kernel can predict application behaviors, activating a suitable configuration as soon as an application starts. Configuration parameters and policies can also be tuned at runtime as an application runs, instead of being statically configured into the kernel.

#3. Generalization: Another powerful feature of ML is its ability to generalize to unseen data points [38] for certain tasks. Replacing handcrafted, ad-hoc heuristics in the kernel with ML models could lead to more robust decisions. In today's kernels, applications that exhibit new behaviors, not captured by existing heuristics, often have opaque and unpredictable performance. These performance cliffs are only caught and fixed slowly by the kernel development community over time by extensive, and often application-specific, benchmarking.

#4. Cross-application optimization: Moreover, our vision enables the kernel to learn the behaviors of multiple applications, how they relate to each other, as well as opportunities for joint optimizations. These cross-application optimizations will lead to better system-wide resource allocation. As an example, monitoring may detect that tasks exhibit producer-consumer behaviors, and activate optimizations for their efficient communication.

Of course, ML is not a silver bullet—in general, one needs to be judicious in matching the right learning techniques with the right problems [38]. The same principle should hold for the OS kernel: the effectiveness of ML will naturally vary based on the tasks at hand, and in certain cases, well-tuned heuristics may already go a long way. Our position is that ML techniques hold significant promise in the context of the OS kernel, and this paper serves as a call to arms for a more thorough investigation.

2.2 Why RMT?

In order to leverage ML, we need a suitable architecture for its integration into the kernel. Such an architecture must satisfy a set of properties:

- Sufficiently general: We need a generic architecture that can represent different types of reconfiguration requirements, for varied kernel components, and also for different phases of learning (e.g., data collection, training, and inference).
- Restricted: The form of reconfiguration must be restricted, so that one can easily reason about and verify the correctness of a configuration before installing it to the kernel.
- Lightweight: It should enable efficient reconfiguration with small runtime overhead. Ideally, it should be hardware-friendly so that it can be feasibly integrated to CPU architectures, just like how page table walkers have been standardized into hardware.

Our proposed answer is based on *reconfigurable match table* (RMT), a recent development in the networking community to specialize network data planes. An RMT program consists of a pipeline of reconfigurable *tables* where specialized packet processing occurs. The execution of a table

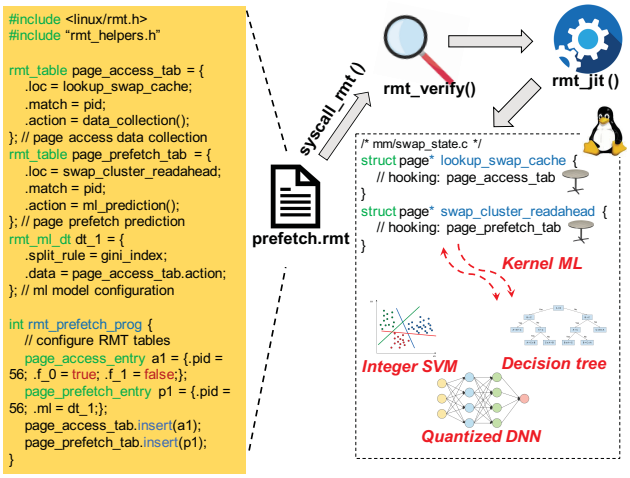

Figure 1: In-kernel RMT virtual machine.

performs *matches* that check one or more packet header fields and triggers *actions* to activate different processings based on the match results. The RMT programming model is restricted yet general enough for a wide range of reconfiguration scenarios, and has been demonstrated to be feasible at high speeds (Tbps). These properties make RMT an attractive candidate for in-kernel reconfiguration, where analogies exist: tables are the decision points (e.g., prefetching), matches check on the current execution environment (e.g., past access patterns), and actions consult an ML model (e.g., predicting the next set of pages to prefetch).

3 Reconfigurable Kernel Datapaths

In this section, we describe our design, its research challenges, and the tentative solutions.

3.1 The in-kernel RMT virtual machine

An RMT program is produced by machine learning from past or current runs, and it is injected to the kernel from userland. The program runs in the virtual machine in interpreted mode or it is just-in-time (JIT) compiled to machine code for efficiency. Many mechanisms are akin to eBPF [49], but RMT programs take a different form than eBPF as they are customized for machine learning.

RMT programs. The key building block of an RMT program is a pipeline of match/action tables. Each table represents a kernel hooking point, which may trigger data collection about the current execution, intercept performance-critical kernel events, or consult ML models based on the execution context. An RMT program can be written in constrained C or a domain-specific language and compiled into machine-independent bytecode, and installed via a system call. A program verifier checks well-formedness and bounded

execution, and it prevents arbitrary kernel calls or data modification. The RMT bytecode can further be JIT compiled directly to machine code for efficiency. At runtime, an RMT program has access to a constrained set of kernel functions that are dedicated to learning and inference. It also has access to kernel memory that stores execution context, historical data, and the ML models themselves.

Tables. Each table represents a key decision point in the kernel datapath—i.e., the critical paths for kernel execution and adaptivity. The number of tables, the types of decisions, and where to install these tables, are configurable. For instance, `rmt_table page_patterns` may be inserted in the `lookup_swap_cache` function in the memory subsystem to collect data about page access patterns in the swap area; later, `rmt_table page_prefetch` is inserted in the `swap_cluster_readahead` function to predict the next set of pages to prefetch. Each table contains a set of match/action entries, which can be statically encoded in the RMT program or dynamically inserted or removed via an API at runtime.

Match/action entries. Each entry represents a decision control flow. For instance, to collect per-file access patterns, new entries are inserted when a file is opened. Another set of entries may monitor per-application patterns, where entries are inserted when applications are created. The match fields of the entry control the pattern matching methods—e.g., inode numbers for per-file entries, and PIDs for per-application entries. Entries may also be aggregates, e.g., per subdirectory or cgroup. We call these match fields the "execution context", and such information is organized in a key/value map of the type `RMT_CTXT` and can be retrieved using a match key. In essence, the execution context is akin to today's kernel monitoring data, but the pattern match strips away unnecessary monitoring and only preserves monitors critical to decision making. This is also constant-time in a system-wide manner without having to walk complex kernel data structures. Under the hood, table matches are compiled into RMT bytecode instructions, such as memory accesses (e.g., `RMT_LD_CTXT`) and compute instructions (e.g., `RMT_MATCH_CTXT`). An action may modify the execution context (e.g., append to access pattern history) using instructions like `RMT_ST_CTXT`, or it may call into an ML model using `CALL` instructions.

Updating RMT entries. The RMT datapath represent decision points, but their policies are reconfigured via the control plane API. This API supports adding, removing, modifying match/action entries and ML models. For instance, the ML training component may periodically update table entries to reflect the latest monitoring data—e.g., by adding extra table entries for newly started applications. Alternatively, the control plane relies on past prediction accuracy to detect workload changes and adjust the table entries. For instance, if the prefetching accuracy falls below a threshold, the control plane will recompute ML decisions to be more conservative in prefetching, and reconfigure the RMT tables to reflect the workload changes.

RMT data structures. The virtual machine also provides an additional set of data structures for in-kernel ML. This includes data structures for monitoring purposes (e.g., akin to different types of eBPF maps), as well as ones for training and inference (e.g., decision trees, NNs). Standard interfaces will be added to these data structures to make them accessible to different kernel subsystems as well as the userspace.

3.2 Lightweight in-kernel ML

As discussed above, the library of ML data structures (e.g., `conv_layer`) and helper functions (e.g., `matrix_multiply`), can help RMT programs to construct more complex ML models (e.g., `action_cnn`). These actions are also triggered from RMT tables, and are compiled into RMT bytecode with a dedicated ML instruction set (e.g., `RMT_VECTOR_LD`, `RMT_MAT_MUL`, `RMT_SCALAR_VAL`), which is patterned after hardware ISA for neural processors [36]. Models can be added to this library, but they must satisfy a set of performance requirements (e.g., the number of NN layers, memory accesses, or floating point operations). The RMT verifier will statically check the model—e.g., by computing the number of floating point operations for a convolutional layer using the height, width and number of channels of the input feature map [41]—before JIT-compiling it to machine code. After computing a prediction result (e.g., page numbers to prefetch), ML-based actions will `EXIT` the RMT pipeline and enter regular kernel execution. Models can also be cascaded using `TAIL_CALL` when needed. Several research challenges exist for in-kernel ML.

ML training. We aim to support both offline and online, real-time training in the kernel. They involve different challenges. Offline training can be done in an asynchronous manner, so it does not incur additional overhead to kernel workloads. However, real-time training in an online manner can better handle rapidly changing workloads and scenarios [43]. In fact, real-time learning [2] is a recent trend in the ML community, with many open problems and ongoing research efforts. Its use in the OS kernel creates even more challenges, especially with regard to latency. For instance, a decision system for self-driving cars may take several milliseconds, but the latency requirement for CPU scheduling is on the order of microseconds. Moreover, offline training can be performed in mature libraries and frameworks, and it can benefit from GPU or TPU support. Online training inside the OS kernel, on the other hand, may require the use of floating point operations, which are by default disabled in kernel execution. As enabling FPUs in-kernel would create high overhead, a promising approach is to rely on lightweight learning models, such as integer-based learning [17, 23, 50, 51]. As another approach, ML training could be performed in real-time in userspace using floating point

operations, with models periodically quantized and pushed to the kernel for inference.

ML inference. Unlike learning, ML inference must be performed in the critical execution path, so it must be very efficient. The overall performance gain will depend on the tradeoff between inference overhead and prediction accuracy. A well-established line of work relies on *knowledge distillation* [16] to convert large "teacher" models to drastically smaller "students" without sacrificing much in accuracy (e.g., simpler NNs or even decision trees). Distillation to interpretable models like decision trees will also elucidate which features are key to decision making, facilitating the goal of "lean monitoring". Feature importance ranking algorithms are also useful for understanding the weights of the features. Quantizing pretrained models for inference [50] has also been shown to have good performance. Depending on the kernel subsystem, inference could be performed locally at CPUs, or in standalone or cache-coherent GPUs if the round-trip time to and from the GPUs is acceptable for that subsystem. If the training takes place in userspace, the model can be periodically updated, quantized, and installed to the kernel. When appropriate, inference results can be cached and reused in a kernel subsystem without incurring repeated queries. Moreover, the RMT program verifier should reason about the efficiency of the ML models [3, 32] before admitting them to the kernel. On-demand *model compression* [37] techniques can also trim a model based on a specified performance goal and resource constraints—e.g., as a subsequent step that can be invoked from the RMT verifier.

Customized ML. When existing ML models cannot be used out of the box, we also need to identify customized models for each subsystem and task. In this direction, *neural architecture search (NAS)* [27] is a method for searching for an appropriate neural network architecture given a certain data sample. It can automatically construct NNs with different depths, widths, and hyperparameters using ML building blocks (e.g., convolution layers) for a given task; such architectures have been shown to have superior performance on a range of tasks [15, 26, 34, 35, 46, 47]. NAS is usually a time-consuming operation, so it is performed in an offline training phase. Once a good neural network architecture has been identified and trained, it can be installed to the kernel for inference. At different RMT tables, *hyper-parameter optimization* [8] techniques will be applied to fine-tune their models and *meta learning* (or "learning to learn") [20] techniques used to identify the best ML models to use. As another form of ML customization, the OS kernel runs atop a diverse range of hardware platforms (e.g., different ISAs, standalone vs. cache-coherent GPUs, or specialized ML accelerators); we should tune or and co-design the ML algorithms based on the underlying platform [31], and automate the construction of platform cost models [53].

3.3 The RMT verifier

Any code that is downloaded into the kernel must be safe. Like eBPF, the RMT virtual machine and verifier are key to address safety concerns, such as ensuring that programs which pass the check will only influence kernel decisions in a constrained manner. While eBPF assumes that the injected code is by nature untrusted user logic, in our case the RMT programs are meant as system-level performance optimizations injected by trusted users or system administrators. Our RMT program verifier also needs to target its reasoning on performance implications and handling the (sometimes) opaque nature of ML models.

Performance interference. The verifier should prevent an RMT program from inducing the kernel to enter undesirable states, e.g., one where resource allocations violate some fair-share policies. For instance, if an RMT program aggressively prefetches disk pages for a certain application, or it always predicts that an application should be given more hugepages, then the verifier may insert additional logic to enforce rate limits.

Model safety. The line of work in adversarial machine learning [11] has repeated shown that the blackbox nature of ML models can sometimes be exploited by an attacker. In this regard, the RMT verifier directly benefits from recent work that aims to verify the correctness of ML models [44] or add guardrails to blackbox inference to prevent worst-case behaviors [40].

Privacy. If ML decisions are made in a cross-application fashion, the verifier may also need to check that privacy goals are met. For instance, it has been shown that the Linux page cache can leak page-level access patterns via side channels [22]. In general, side channels and privacy leakage are difficult to prevent in a shared-resource environment [30]. One idea is to introduce privacy mechanisms that ensure RMT queries only leak data in well-understood ways. For instance, if an RMT query returns some aggregate statistics, we can leverage *differential privacy (DP)* [18] to noise the outputs. In our design, the fact that queries occur at well-defined points (i.e., each RMT table) is helpful for reasoning. The kernel can maintain a "privacy budget", in DP terms, and subtract from this overall budget for each table match.

4 Initial Validation

We present our current progress. We have developed an in-kernel RMT prototype that is hardcoded at specified hook points in Linux kernel v5.9.15, and performed two case studies on page prefetching and CPU scheduling.

Case study #1. The Linux page prefetcher bridges the speed difference between main memory and external disks. The default readahead prefetcher [52] detects sequential page accesses and prefetches the next set of pages. Recent work,

Metric \ Benchmark	OpenCV video resize			Numpy matrix conv		
	Linux	Leap	Ours	Linux	Leap	Ours
Accuracy (%)	40.69	45.40	**78.89**	12.50	48.86	**92.91**
Coverage (%)	65.09	66.81	**84.13**	19.28	65.62	**88.51**
Completion time (s)	24.60	23.02	**17.79**	31.74	17.48	**13.90**

Table 1: Case study: Page prefetching.

Benchmark \ Metric	Full-Featured MLP		Leaner-Featured MLP		Linux
	Acc (%)	JCT (s)	Acc (%)	JCT (s)	JCT (s)
Blackscholes	99.08	19.010	94.0	18.770	18.679
Streamcluster	99.38	58.136	94.3	57.387	57.362
Fib Calculation	99.81	19.567	99.7	19.533	19.543
Matrix Multiply	99.7	16.520	99.6	16.514	16.337

Table 2: Case study: Linux Scheduler.

Leap [4], has extended this to detect striding patterns. To demonstrate the benefits of ML, we have developed a in-kernel integer decision tree that can capture more complex access patterns.

Our RMT pipeline collects page access traces for each process for online training and inference. It trains a new decision tree periodically in the background for each time window, while discarding the old ones. Upon prefetching, another RMT table queries the ML model to predict the next pages to fetch. Figure 1 compares the performance of our infrastructure against Linux as well as Leap, using an OpenCV video resizing application and a Numpy matrix convolution program [12, 45]. The results show that the ML model leads to accuracy improvements of 28%-80% compared to Linux and 23%-44% to Leap, cutting job completion times significantly.

Case study #2. The Linux Completely Fair Scheduler (CFS) periodically migrates tasks across CPUs for load balancing, while taking into account a range of factors to avoid performance regression. A recent project [14] shows that an MLP (Multilayer Perceptron) ML model can mimic Linux CFS decisions effectively. Our next case study investigates this scenario using our infrastructure.

The `can_migrate_task` function in CFS calls into RMT to query the ML model to predict whether or not a task should be migrated. We first replicate the experiment in [14] using our infrastructure for offloading training with quantized models. Using the Blackscholes and other models in the PARSEC benchmark suite [10], as well as matrix multiplication and Fibonacci calculation programs, our infrastructure achieves 99% prediction accuracy in mimicking the Linux CFS decision similar as [14]. Next, we used the *scikit-learn* toolbox to rank and identify two key features for load balancing (out of 15 used in [14]) With this leaner monitoring, our prototype still achieves 94+% accuracy; it achieves competitive results in terms of job completion times. Table 2 compares the performance of ML versus the Linux CFS heuristics.

5 Related Work

ML for systems. ML has found applications in index retrieval [29], bloom filter queries [29], CPU scheduling [14], C++ memory management [39], and many other contexts. Zhang and Huang [54] argue for its use to optimize OS kernels. The DBOS [13] project also proposes to build a data-centric OS where components could be learned using ML. Our project pursues a similar goal but it contributes a concrete proposal to integrate ML into the kernel based upon the RMT architecture.

OS specialization. OS specialization [9, 19] has been a long-standing goal in the community, and recently, eBPF has gained popularity. In Hypercallbacks [6] and Hyperupcalls [5], VMs use eBPF to inject untrusted code to the hypervisor for policy enforcement. LBM [48], on the other hand, injects protection programs into the kernel to defense against malicious peripherals. Verification techniques have also been developed for eBPF programs [21] and their JIT compilers [42] for high assurance of injected code. Our idea is inspired by the eBPF infrastructure, but RMT programs are enhanced with a dedicated ML instruction set and ML models, and their verifier needs to check more advanced properties beyond bounded execution—such as ML model performance and privacy goals.

6 Summary and Future Work

We have argued for a novel *reconfigurable kernel datapaths* architecture based upon RMT that enables efficient machine learning in OS kernels, and presented the research challenges in RMT program design, in-kernel machine learning, and program safety checks. We have also presented some initial results with two case studies. Going forward, much is yet to be done. Customizing ML techniques for different kernel subsystems, striking a good balance between ML overheads and prediction accuracy, when and how to invoke accelerators like GPUs, as well as a full design and implementation of RMT in the kernel, are all interesting avenues of research.

Overall, we believe that the use of in-kernel ML represents an interesting point in the design space. Analogous to SPIN and exokernel, such a design would account for application diversity and the need for specialization. However, instead of modifying the OS and allowing applications to take control over policy decisions, the use of ML could result in more robust kernel policies despite application differences. The use of data-driven approaches also has the potential of placing kernel optimizations on a firmer foundation than what exists today.

Acknowledgments: We thank our shepherd, Ioan Stefanovici, and the HotOS reviewers, for their insightful comments and suggestions. This work was supported in part by NSF grants CCRI-2016727, CNS-1942219, and CNS-1801884.

References

[1] Digitalisation and Energy. https://www.iea.org/reports/digitalisation-and-energy.

[2] The DARPA Real-Time Machine Learning Program. https://www.darpa.mil/program/real-time-machine-learning.

[3] O. Y. Al-Jarrah, P. D. Yoo, S. Muhaidat, G. K. Karagiannidis, and K. Taha. Efficient machine learning for big data: A review. *Big Data Research*, 2(3):87–93, 2015.

[4] H. Al Maruf and M. Chowdhury. Effectively prefetching remote memory with leap. In *Proc. ATC*, 2020.

[5] N. Amit and M. Wei. The design and implementation of hyperupcalls. In *Proc. ATC*, 2018.

[6] N. Amit, M. Wei, and C.-C. Tu. Hypercallbacks: Decoupling policy decisions and execution. In *Proc. HotOS*, 2017.

[7] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: A new os architecture for scalable multicore systems. In *Proc. SOSP*, 2009.

[8] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(2), 2012.

[9] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the spin operating system. In *Proc. SOSP*, 1995.

[10] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[11] W. Brendel, J. Rauber, and M. Bethge. Decision-based adversarial attacks: Reliable attacks against black-box machine learning models. *arXiv preprint arXiv:1712.04248*, 2017.

[12] C. S. Burrus and T. Parks. *DFT/FFT and convolution algorithms: theory and implementation*. John Wiley & Sons, Inc., 1985.

[13] M. Cafarella, D. DeWitt, V. Gadepally, J. Kepner, C. Kozyrakis, T. Kraska, M. Stonebraker, and M. Zaharia. Dbos: A proposal for a data-centric operating system. *arXiv preprint arXiv:2007.11112*, 2020.

[14] J. Chen, S. S. Banerjee, Z. Kalbarczyk, and R. K. Iyer. Machine learning for load balancing in the linux kernel. In *Proc. APSys*, 2020.

[15] L.-C. Chen, M. Collins, Y. Zhu, G. Papandreou, B. Zoph, F. Schroff, H. Adam, and J. Shlens. Searching for efficient multi-scale architectures for dense image prediction. In *Proc. NeurIPS*, 2018.

[16] Y. Coppens, K. Efthymiadis, T. Lenaerts, and A. Nowe. Distilling deep reinforcement learning policies in soft decision trees. In *Proc. IJCAI*, 2019.

[17] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1. arXiv:1602.02830, 2016.

[18] C. Dwork, K. Kenthapadi, F. McSherry, I. Mironov, and M. Naor. Our data, ourselves: Privacy via distributed noise generation. In *Proc. EuroCrypt*, 2006.

[19] D. Engler, F. Kaashoek, and J. O'Toole. Exokernel: an operating system architecture for application-level resource management. In *Proc. SOSP*, 1995.

[20] C. Finn, A. Rajeswaran, S. Kakade, and S. Levine. Online meta-learning. In *Proc. ICML*, 2019.

[21] E. Gershuni, N. Amit, A. Gurfinkel, N. Narodytska, J. A. Navas, N. Rinetzky, L. Ryzhyk, and M. Sagiv. Simple and precise static analysis of untrusted Linux kernel extensions. In *Proc. PLDI*, 2019.

[22] D. Gruss, E. Kraft, T. Tiwari, M. Schwarz, A. Trachtenberg, J. Hennessey, A. Ionescu, and A. Fogh. Page cache attacks. In *Proc. CCS*, 2019.

[23] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. Deep learning with limited numerical precision. In *Proc. ICML*, 2015.

[24] M. Hao, L. Toksoz, N. Li, E. E. Halim, H. Hoffmann, and H. S. Gunawi. Linnos: Predictability on unpredictable flash storage (with a light neural network). In *Proc. OSDI*, 2020.

[25] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A file is not a file: Understanding the I/O behavior of apple desktop applications. In *Proc. SOSP*, 2011.

[26] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan, et al. Searching for mobilenetv3. In *Proc. ICCV*, 2019.

[27] H. Jin, Q. Song, and X. Hu. Auto-keras: An efficient neural architecture search system. In *Proc. KDD*, 2019.

[28] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks. Profiling a warehouse scale computer. In *Proc. ISCA*, 2015.

[29] T. Kraska, A. Beutel, E. H. Chi, and J. Dean. The case for learned data structures. In *Proc. SIGMOD*, 2018.

[30] B. Lampson. A note on the confinement problem. *Communications of the ACM*, 16, 1973.

[31] C. Li, T. Chen, H. You, Z. Wang, and Y. Lin. Halo: Hardware-aware learning to optimize. In *Proc. ECCV*. Springer, 2020.

[32] C. Li, Z. Yu, Y. Fu, Y. Zhang, Y. Zhao, H. You, Q. Yu, Y. Wang, C. Hao, and Y. Lin. Hw-nas-bench: Hardware-aware neural architecture search benchmark. In *Proc. ICLR*, 2021.

[33] J. Li, K. Cheng, S. Wang, F. Morstatter, R. P. Trevino, J. Tang, and H. Liu. Feature selection: A data perspective. *ACM Comput. Surv.*, 50(6), 2017.

[34] C. Liu, L.-C. Chen, F. Schroff, H. Adam, W. Hua, A. L. Yuille, and L. Fei-Fei. Auto-deeplab: Hierarchical neural architecture search for semantic image segmentation. In *Proc. CVPR*, 2019.

[35] H. Liu, K. Simonyan, and Y. Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

[36] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen. Cambricon: An instruction set architecture for neural networks. In *Proc. ISCA*, 2016.

[37] S. Liu, Y. Lin, Z. Zhou, K. Nan, H. Liu, and J. Du. On-demand deep model compression for mobile devices: A usage-driven model selection framework. In *Proc. MobiSys*, 2018.

[38] M. Maas. A taxonomy of ML for systems problems. *IEEE Micro*, 40(5), 2020.

[39] M. Maas, D. Anderson, M. Isard, M. M. Javanmard, K. S. McKinley, and C. Raffel. Learning-based memory allocation for C++ server workloads. In *Proc. ASPLOS*, 2020.

[40] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*, 2017.

[41] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz. Pruning convolutional neural networks for resource efficient inference. In *Proc. ICLR*, 2017.

[42] L. Nelson, J. V. Geffen, E. Torlak, and X. Wang. Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the Linux kernel. In *Proc. OSDI*, 2020.

[43] R. Nishihara, P. Moritz, S. Wang, A. Tumanov, W. Paul, J. Schleier-Smith, R. Liaw, M. Niknami, M. I. Jordan, and I. Stoica. Real-time machine learning: The missing pieces. In *Proc. HotOS*, 2017.

[44] K. Pei, Y. Cao, J. Yang, and S. Jana. Towards practical verification of machine learning: The case of computer vision systems. In *Proc. DeepTest*, 2019.

[45] Q. Shan, Z. Li, J. Jia, and C.-K. Tang. Fast image/video upsampling. *ACM Transactions on Graphics (TOG)*, 27(5):1–7, 2008.

[46] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proc. CVPR*, 2019.

[47] M. Tan and Q. V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *arXiv preprint arXiv:1905.11946*, 2019.

[48] D. J. Tian, G. Hernandez, J. I. Choi, V. Frost, P. C. Johnson, and K. R. B. Butler. LBM–A security framework for peripherals within the Linux

kernel. In *Proc. S&P*, 2019.

[49] M. A. Vieira, M. S. Castanho, R. D. Pacífico, E. R. Santos, E. P. C. Júnior, and L. F. Vieira. Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Computing Surveys (CSUR)*, 53(1):1–36, 2020.

[50] M. Wang, S. Rasoulinezhad, P. H. Leong, and H. K. So. Niti: Training integer neural networks using integer-only arithmetic. *arXiv preprint arXiv:2009.13108*, 2020.

[51] S. Wang, T. Tuor, T. Salonidis, K. K. Leung, C. Makaya, T. He, and K. Chan. When edge meets learning: Adaptive control for resource-constrained distributed machine learning. In *Proc. INFOCOM*, 2018.

[52] Y. Wiseman and S. Jiang. *Advanced Operating Systems and Kernel Applications: Techniques and Technologies: Techniques and Technologies*. IGI Global, 2009.

[53] T.-J. Yang, Y.-H. Chen, and V. Sze. Designing energy-efficient convolutional neural networks using energy-aware pruning. In *Proc. CVPR*, 2017.

[54] Y. Zhang and Y. Huang. "Learned" operating systems. *ACM SIGOPS Operating System Review*, 53, 2019.