Machine Learning for Load Balancing in the Linux Kernel

Jingde Chen
jingdec2@illinois.edu
University of Illinois at Urbana-Champaign

Subho S. Banerjee
ssbaner2@illinois.edu
University of Illinois at Urbana-Champaign

Zbigniew T. Kalbarczyk
kalbarcz@illinois.edu
University of Illinois at Urbana-Champaign

Ravishankar K. Iyer
rkiyer@illinois.edu
University of Illinois at Urbana-Champaign

Abstract

The OS load balancing algorithm governs the performance gains provided by a multiprocessor computer system. The Linux's Completely Fair Scheduler (CFS) scheduler tracks process loads by average CPU utilization to balance workload between processor cores. That approach maximizes the utilization of processing time but overlooks the contention for lower-level hardware resources. In servers running compute-intensive workloads, an imbalanced need for limited computing resources hinders execution performance. This paper solves the above problem using a machine learning (ML)-based resource-aware load balancer. We describe (1) low-overhead methods for collecting training data; (2) an ML model based on a multi-layer perceptron model that imitates the CFS load balancer based on the collected training data; and (3) an in-kernel implementation of inference on the model. Our experiments demonstrate that the proposed model has an accuracy of 99% in making migration decisions and while only increasing the latency by 1.9 µs.

CCS Concepts

• Software and its engineering → Scheduling; **• Theory of computation → Scheduling algorithms**; **• Computing methodologies → Neural networks**.

Keywords

Linux kernel, Machine Learning, Neural Network, Completely Fair Scheduler, Operating System, Load Balancing.

ACM Reference Format:
Jingde Chen, Subho S. Banerjee, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. Machine Learning for Load Balancing in the Linux Kernel. In *11th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '20), August 24–25, 2020, Tsukuba, Japan.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3409963.3410492

1 Introduction

The Linux scheduler, called the *Completely Fair Scheduler* (CFS) algorithm, enforces a fair scheduling policy that aims to allow all tasks to be executed with a fair quota of processing time. CFS allocates CPU time appropriately among the available hardware threads to allow processes to make forward progress. However, the scheduling algorithm of CFS does not take into account contention for low-level hardware resources (e.g., shared functional units or memory bandwidth) other than CPU time. There is an implicit assumption that fair sharing of CPU time implies that other such microarchitectural resources are also divided fairly [2]. That assumption is often false and leaves the scheduling algorithm unaware of hardware resource utilization, thereby leading to shared resource contention and reduced performance.

While prior work has investigated the integration of hardware resource usage in OS load balancing, most attempts have focused only on resources shared at the chip multiprocessor (CMP) level and depend on predefined scheduling policies or on efforts by the user to manually annotate resource usage of the programs [5, 7, 16]. There is not a system-wide solution that works for all general processes, or one that can take advantage of the dynamic contextual information available from tracing and telemetry data from the system. Traditional methods are fundamentally limited in having to effectively quantify relationships between different resources and application performance, which indeed is a hard problem. One solution is to use machine learning (ML) in the kernel to model and tackle the problems described above. Even small gains can translate into big savings at the level of large data centers. An important step in building an ML engine that can outperform the standard CFS Linux load

APSys '20, August 24–25, 2020, Tsukuba, Japan

ACM ISBN 978-1-4503-8069-0/20/08…$15.00
https://doi.org/10.1145/3409963.3410492

balancer is to see whether it is possible to match the current load balancer performance. The insights thus gained, as we will show, can be quite valuable in the next step, namely building an engine that can outperform current techniques.

ML in the Kernel. The ML approach will have to (i) perform load balancing in Linux that emulates original kernel decisions; (ii) include hardware resource utilization models; (iii) train the model to solve specific cases with high resource needs that cause frequent contention; and (iv) apply reinforcement learning to the model to optimize load balancing performance. To exemplify the types of resource contention, consider simultaneous multi-threaded (SMT) threads on a CMP that appear as identical logical CPU cores to the OS, but share most of the computing resources on the chip, including caches and registers. Processes scheduled to run on different SMT threads at the same time contend for the resources on the chip. Similarly, CPUs within the same NUMA node share memory and I/O bandwidth. In the end, we aim to implement an ML module inside the kernel that monitors real-time resource usage in the system, identifies potential hardware performance bottlenecks, and generates the best load balancing decisions.

Contributions. In this paper, we present the results of the first step, i.e., an ML model and an implementation to replace the CFS load balancer. The proposed approach consists of:

(1) Collection of runtime system data via dynamic kernel tracing, using eBPF (described in Section 3.2).
(2) Training of the ML model using collected data to emulate task migration decisions (described in Section 3.3).
(3) Two implementations of the model in the kernel, one with floating-point arithmetic and one with fixed-point arithmetic (described in Section 3.4).

The ML model is trained offline in the user space and is used for online inference in the kernel to generate migration decisions. The performance of the in-kernel ML model is as follows:

(1) 99% of the migration decisions generated by the ML model are correct.
(2) The average latency of the load balancing procedure is 13% higher than the original kernel (16.4 µs vs. 14.5 µs).
(3) During testing, our approach produced a 5-minute average CPU load of 1.773 compared to Linux's 1.758.

The work described in this paper was based on the Linux 4.15 source code. The code used in this paper can be found at https://github.com/keitokuch/MLLB, and the modified kernel source can be found at https://github.com/keitokuch/linux-4.15-lb.

2 Background: The Linux Scheduler

Across its lifetime, the Linux process scheduler has experienced two major innovations. Until version 2.6, the Linux kernel had used a straightforward $O(n)$ scheduling algorithm, which iterated every runnable task to pick the next one to run. In 2003, the $O(n)$ scheduler was replaced with the $O(1)$ scheduler which runs in constant time instead of linear time. The $O(1)$ scheduler managed running tasks with two linked lists for each task priority, one for ready-to-run tasks and the other for expired tasks that have used up their timeshares. Later, Linux abandoned the concept of having predefined time slices. The Completely Fair Scheduler (CFS) was introduced to the kernel in Linux 2.6.23 in 2007 together with the design of the Modular Scheduler Core [13], and has since been the default Linux scheduler. We refer to that latest design as the *Linux scheduler*. The scheduler implements individual scheduling classes to which the Modular Scheduler Core can delegate detailed scheduling tasks. Existing scheduling classes include the `rt` class for real-time processes; the `fair` class which implements the CFS; and the `idle` class, which handles cases when there is no runnable task.

The scheduler comprises a large portion of the kernel source and resides in the `<kernel/sched>` directory and the scheduler core is implemented in `<kernel/sched/core.c>`. The Linux scheduler uses per-CPU allocated runqueues for faster local access, and the load balancing algorithm works by migrating tasks across the per-core runqueues. The primary granularity of work scheduled under Linux is the thread (and not the process). The threads are not scheduled via direct attachment to the per-core runqueues, but as schedulable entities (`sched_entity`) attached to subordinate class runqueues managed by individual scheduling classes.

2.1 The Completely Fair Scheduler

The CFS is the most complicated part of the Linux scheduler and is implemented in `<kernel/sched/fair.c>` with 10,000 lines of code. The CFS schedules threads that have policies of `SCHED_NORMAL` and `SCHED_BATCH`. As mentioned above, the CFS manages its own runqueue structure `cfs_rq` in the per-CPU runqueue. `cfs_rq` sorts the `sched_entity` of all runnable threads on it in a red-black tree (a kind of self-balancing binary search tree) that uses virtual runtimes as keys. When asked to pick the next task to run, CFS returns the task corresponding to the sched_entity with the most insignificant virtual runtime, which is the least executed task. In that way, all tasks in the queue can get a fair chance to run. To reflect task priorities, the virtual runtime accumulates more slowly for high-priority tasks and they get to run longer. With version 2.6.38 of Linux, the Control Group scheduling policy (cgroup feature) was added to enforce fairness between groups of threads. A group can be a process with multiple threads, a user, or a container. Processing time is allocated fairly among the groups on the same level and further divided up by the threads in the groups. That flexible allocation is achieved with the `sched_entity` and `cfs_rq` mechanism. A cfs_rq has its own sched_entity and can be scheduled by another cfs_rq. That hierarchical structure is a nested tree of schedulable entities. The root node is the root cfs_rq attached to the per-CPU runqueue structure; each non-leaf node is a cfs_rq that stands for some control group, and the leaf nodes are the runnable threads.

2.2 CFS Load Balancing

The CFS periodically runs the load balancing code as a software interrupt (softirq) to balance the workload across the cores. The system achieves the balancing by pulling work from the busy cores and moving it to the free cores. Such load balancing decisions are based on the cache and NUMA locality. The CFS partitions computing cores hierarchically into scheduling domains (sched_domain) for each hardware level. During periodic load balancing triggered by the timer interrupt, the workload is balanced within each sched_domain. CPU cores in a sched_domain are separated into scheduling groups (sched_group). Runnable threads are migrated between the groups to ensure that the groups' workloads are balanced within the domain.

We will do a brief walk-through of the CFS load balancing code. The periodic scheduler function scheduler_tick() is called by the timer interrupt handler on each CPU core at a predefined frequency. Function trigger_load_balance() called by scheduler_tick() raises the SCHED_SOFTIRQ software interrupt when it is time to do load balancing. The softirq executes the run_rebalance_domains() function, in which two types of load balancing are triggered: first nohz_idle_balance, in which the current core checks for idle cores whose periodic scheduler ticks were disabled to save energy; and then, the normal load balance, which checks for intra-domain load imbalance.

The real load balancing work then begins. For both types of load balancing, the kernel iterates through the sched_domains the current CPU is in, following the hierarchy from the bottom up, and calls the load_balance() function on each domain to ensure that the domain's workload is balanced. In the load_balance() function, the load balance eligibility of the current CPU is first checked. To avoid unnecessary repeated work, usually only the first idle CPU, or the first CPU in the domain when none is idle, should do the balancing. The CFS then checks for imbalance within the domain, and if a rebalance is needed, it finds the busiest sched_group and the busiest core in the group. To migrate tasks from the cfs_rq of the overloaded CPU core to an eligible destination, load_balance() calls the function detach_tasks() followed by attach_tasks(). detach_tasks() iterates through the runnable threads on the overloaded cfs_rq, calls the can_migrate_task() function to test whether each of the tasks can be moved, and detaches for migration threads up to the load needed to correct the imbalance. Detached tasks are then added to the destination core in attach_tasks(). In addition to applying some hard conditions that would prevent moving, can_migrate_task() decides whether a thread is suitable for migration by doing heuristic estimation on the cache and NUMA locality information of the thread. A thread that is likely to be cache-hot on the current CPU or to be facing a cross-node move is discouraged from migrating. In a nutshell, in periodic execution of the load balancing procedure, in each sched_domain, the load_balance() function is called on an underloaded core to balance the load. It calculates the load imbalance in the domain and arranges for threads to be pulled from a busy core. To ensure system performance, the can_migrate_task() function prevents migrations that are inadvisable based on cache/NUMA locality.

3 Imitation Learning in Linux

To assist the load balancing procedure by using machine learning, we take the can_migrate_task() function as the entry point and use supervised imitation learning to replace part of its internal logic with an ML model. We trained a Multi-Layer Perceptron (MLP) model to emulate the load balancing decisions made by the kernel based on the cache and NUMA locality to determine whether a task should be migrated. We implemented the forward pass of the MLP model in pure C code with floating-point calculations and embedded into a new function should_migrate_task() that will be called by can_migrate_task(). To improve the performance and reduce latency, we developed a second implementation, using fixed-point arithmetic.

3.1 Experiment Setup

The experiments in this paper were conducted on an x86-64 computer that consisted of two NUMA nodes (sockets), each running an Intel Xeon E5 processor with 14 cores and 2 SMT hyper-threads per core. Kernel development and model evaluation were performed inside a QEMU virtual machine (VM) that emulated the system topology of the host machine.

3.2 Data Collection

To emulate the kernel's load-balancing behavior, we needed to collect runtime statistics and corresponding migration decisions made by the kernel. We used eBPF and BCC to collect the migration-related data of the running system through dynamic kernel tracing with kprobes, and prepared the data for model training in Python.

eBPF (extended Berkeley Packet Filter) is a built-in functionality that was added to the kernel for the Linux 3.15 release. It allows the user to attach special user-generated eBPF programs to specific kernel code paths. User-defined eBPF code is compiled into intermediate bytecode and examined by an in-kernel verifier before being attached to kernel events including kprobes, uprobes, tracepoints, and perf_events.

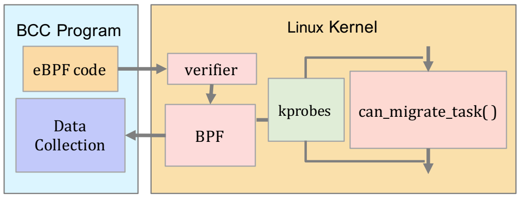

Figure 1: Layout of data collection with BCC and eBPF.

The loaded eBPF program gets executed whenever the kernel event triggers. eBPF programs can use memory-mapped buffers to store data and transport data from the kernel space to the user space, but they can only access the kernel structures and functions defined in the kernel headers.

BCC (BPF Compiler Collection) is a toolkit that uses eBPF for kernel tracing. It provides several convenient kernel performance measurement tools and a Python interface for generating eBPF programs.

kprobes (kernel probes) is a debugging mechanism in Linux for probing kernel code execution. A pair of kprobe handlers are registered for a kernel function of interest. The two handlers are then executed before the entry and after the return of that function.

To achieve runtime data extraction from the kernel, we defined two eBPF functions in a BCC Python program to be compiled and attached to the entry and return of the `can_migrate_task()` function as kprobe handlers. The entry eBPF probe has access to the function parameters, while the return probe has access to the return value. Thus, we store the load balancing related statistics at the entry function in an eBPF hash map that uses the `pid` and CPU number of the current process as key, and then retrieve the stored data by using the same key at the function return and submit the data together with the return value, to the user space BCC program for collection. Since eBPF programs can only access limited kernel functions, the hard condition tests in `can_migrate_task()` cannot be reproduced correctly in the eBPF probes. To resolve the inaccuracies brought by the uncaptured conditions, we modified the kernel and added a test flag to indicate whether the task has passed all the hard condition tests, as we only need cases for which migration is possible. The record entries in which the test flag has not been set are filtered out in the preprocessing stage. The data collection workflow is shown in Figure 1.

Collected migration-related data fields include:

1) Idleness of the target CPU core.
2) Source and destination NUMA node numbers.
3) Preferred NUMA node of the process.
4) Loads of the source and target CPU cores.
5) Lengths of the CFS runqueues of both cores.
6) Number of processes prefer to stay on the source core.
7) Time the process has been running on the source core.
8) Number of NUMA faults the process had on each node.
9) Number of failed balancing attempts.

Table 1: Hyperparameters used for training

Hyperparameter	Opt Value	Min Value	Max Value
Learning Rate	3×10^{-4}	1×10^{-5}	1.5×10^{-3}
Decay	3×10^{-6}	1×10^{-6}	1.5×10^{-4}
Batch Size	64	16	128
Validation Split	0.1	0.1	0.1

3.3 Multi-Layer Perceptron Model

The proposed approach uses a multi-layer perceptron (MLP) model to aid the load balancing process in the kernel. We chose MLP because our current work doesn't require a very complex model and MLP has a relatively simple implementation compared to the other models.

MLP is a type of shallow artificial neural network (ANN). We designed an MLP model with 15 input features, one fully connected hidden layer with 10 nodes, and an output layer with one node. A rectified linear unit (ReLU) activation function is applied after each node in the hidden layer.

Using the approach described in Section 3.2, we built a corpus of 500,000 load-balancer invocations to create the training dataset. The dataset consists of data collected while the system was running various levels of workloads because the distribution of load balancing decisions made by the system is biased on different loads. The `stress-ng` stress-testing program was used to emulate different load levels. We collected the same amount of training data while the system was running 30, 60, and 120 stressor threads, corresponding to approximately 50%, 100%, and 200% average CPU load.

After preprocessing, 15 input features were generated from the 9 collected data fields described above. For example, an input feature that indicates a cross-node migration was derived from the source and destination node numbers, and the idleness of the target CPU was processed into one-hot encoded features. Each entry in the dataset has a 1-D vector of features and a class label that takes value in [0, 1], which is the return value of the `can_migrate_task()` function.

We then used an Adam optimizer to train the model and stored the weights and structure of the trained network for use by the in-kernel implementation. We obtained the optimal values for the hyperparameters after sweeping through a range of values as noted in Table 1. An early-stopping callback was used to stop the training when the model had reached a stable fit. We used the binary cross-entropy as loss function as we were addressing a binary classification problem. Figure 2 illustrates the training process showing the loss of the model as a function of trained batches.

In a ten times Monte Carlo cross-validation, the model was trained ten separate times, each with 10% of the dataset held out randomly as the evaluation set, and we used the remaining 90% of the data for training. The average loss of the

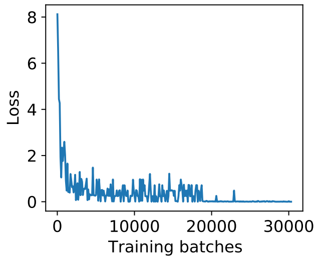

Figure 2: Training loss vs. number of batches.

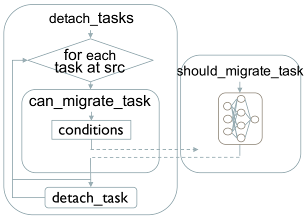

Figure 3: The in-kernel implementation.

ten cross-validation runs was 0.0955, and the average evaluation accuracy was 99.24%. The cross-validation results show that the MLP model can reproduce current kernel migration decisions very well.

3.4 MLP Implementation in Kernel

The forward pass of the MLP model was added to the kernel in a new function `should_migrate_task()` as an optional kernel configuration that replaced the original cache and NUMA locality-checking logic. The feed-forward procedure's matrix multiplication is implemented using floating-point numbers and the weights and biases of the trained model are stored in static memory. Figure 3 provides a schematic diagram of the algorithm design.

However, floating-point operations are generally avoided in the kernel. When floating-point numbers are used in Linux kernel code, all floating-point calculations must be guarded between a pair of macros `kernel_fpu_begin()` and `kernel_fpu_end()`, since the Linux kernel doesn't save and restore floating-point unit (FPU) states at context switches [17]. The Linux kernel does not use floating-point operations because some computer systems might not have an FPU and not having to save and restore FPU states allows faster context switches. As a result, the use of floating-point numbers brings extra overhead to our calculation-heavy application, and that can be avoided if we use fixed-point numbers.

Therefore, we reinvented the original implementation by adding support for fixed-point arithmetic to achieve better robustness and lower latency. The approach will be described in the next section, and as we will see in Section 3.6, the fixed-point implementation reduces the latency by 17%: for the `can_migrate_task()` function, the latency of the fixed-point version is 1.36× that of Linux CFS, and the latency of the floating-point version is 1.64× that of Linux CFS.

3.5 Fixed-Point Implementation

The idea of fixed-point number representation is to use a fixed number of bits to store the integer part of the number and the remaining bits to store the decimal part. Whereas a floating-point number can use a changeable number of bits for the whole part and the decimal part and achieve variable range and precision, a fixed-point number has a fixed decimal point position and immutable range and precision.

A fixed-point representation consists of the total word length (WL), the number of integer bits, and the number of fractional bits in a fixed-point number. In a Q-point notation of the form $Q\langle QI\rangle.\langle QF\rangle$, where QI is the number of integer bits, QF is the number of fractional bits, and $WL = QI + QF$. WL is the total number of bits that can be used to represent the fixed-point number. We use 32-bit integers as fixed-point numbers and have a WL of 32, in a $Q21.11$ representation.

To determine the suitable QI and QF, we considered the maximum range and precision we need. One approach would have been to start with the resolution (decimal precision) we needed in our representation and find QF and then check whether QI was sufficient to cover our required integer range. The resolution of a fixed-point number is $\varepsilon = {}^1/{}_{2^{QF}}$. The QF required by a fixed-point number with a resolution requirement of ε is $QF = \lceil \log_2 ({}^1/{}_\varepsilon) \rceil$.

After experimenting with different resolutions of the model weights, we found that a precision granularity of $\varepsilon \leq 0.0005$ should be used to ensure the model's quality. Hence, $QF = \lceil \log_2 ({}^1/{}_{0.0005}) \rceil = \lceil 10.96 \rceil = 11$ fractional bits are required to achieve a resolution of $\varepsilon \leq 0.0005$. We are left with $QI = WL - QF = 32 - 11 = 21$ integer bits. A fixed-point number α with QI can take from the range of $-2^{QI-1} \leq \alpha \leq 2^{QI-1} - 1$. For $QI = 21$ we have a range of $[-1048576 \leq \alpha \leq 1048575]$ and this range is sufficient for our neural network computations. We established a notation of Q21.11 for our fixed-point numbers which has a finest precision of $\varepsilon = 0.000488281$. To convert a floating-point number to a fixed-point number, the floating-point number is multiplied by a scaling factor: $\alpha_{fxdpt} = \lfloor \beta_{fltpt} \times 2^{QF} \rfloor$

While the scaling can be achieved by bit-shifting the float to the left by QF bits, we first define the fixed-point base as $1 << QF$ and multiply the float by this fixed-point base. The reason is that the left shift of a negative integer is undefined, and the right shift of a negative integer is implementation-dependent. Also, the truncation of a shifted number rounds downwards and causes precision loss. When we convert a constant real number to the fixed-point format we add 0.5 or −0.5 (based on the sign) to the scaled number before typecasting, so that we round to the nearest whole number. The resulting conversion algorithm is:

```
FxdPt = (FltPt * (1 << QF) + (FltPt >= 0 ? 0.5 : -0.5).
```

The addition and subtraction of two fixed-point numbers are the same as those of integer operations. After multiplying two fixed-point numbers directly, we get a result with the fixed-point base factor applied twice. We need to remove the extra scaling factor by dividing the result by the base. Notice that the numbers need to be cast to a type with double WL (`int64`) before the multiplication because otherwise, overflow could happen. Divisions work in a similar way. The

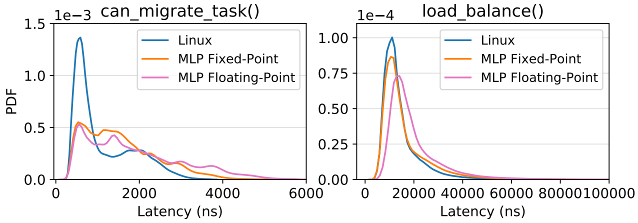

Figure 4: Latency of the evaluated load balancers.

dividend and divisor are first cast to a type with double WL and then back to the original type after the division.

3.6 Evaluation and Results

We evaluate the test accuracy of the model inside the kernel. For both the floating-point and fixed-point implementations, we collected 30,000 lines of logs containing decisions made by the in-kernel MLP model and the original kernel. The decision accuracies of both implementations are over 99%.

We measured the extra runtime overhead brought by the MLP model to the kernel by running a BCC program that tracks the latency between the entry and return of the `can_migrate_task()` function and the `load_balance()` function. Figure 4 shows the distribution of function latency when we ran the original Linux algorithm, the floating-point MLP, and the fixed-point MLP.

As shown in the plots, the in-kernel MLP model brings a moderate amount of extra latency and variance in execution time to the kernel functions. The fixed-point implementation shows an improved latency compared to the floating-point implementation, with an average reduction of 17%. Because of its obvious advantages, we will use the fixed-point implementation to represent the in-kernel ML model in the subsequent evaluations. On average, the ML model with the fixed-point implementation adds an overhead of 0.4μs (36%) to the `can_migrate_task()` function and 1.8μs (13%) overhead to the `load_balance()` function.

The performance of the in-kernel ML model in doing load-balancing work shows whether the extra latency impacts the load balancing ability of the system. We tested it by finding the maximum difference between the length of the runqueues (maximum imbalance of the number of jobs on cores) across the system when the system was running a large number of tasks.

We wrote another BCC program to sample the `cfs_rq` lengths of all CPU cores in the system continuously while different high-load parallel benchmark programs were being run, and we recorded the difference in the number of tasks between the most and least busy cores ten times a second. The same test programs were run for collection on both the original kernel and the new kernel for the same length of time. Figure 5 shows the distribution of job imbalance.

We can see from the plot that both the original kernel logic and the in-kernel ML model exhibited the same frequency of occurrences of maximum job imbalances larger than 11. The occurrences of job imbalance below 11 are different for the two, but further experiments are needed to determine how this affects the two implementations' performance.

Similarly, we plotted the distributions of the total runqueue lengths of both implementations while the system was running a significant number of jobs (see Figure 6). In the figure, the density (y-axis) is displayed in the log scale to show any subtle disparity between the two distributions. The occurrence of runqueues with around 10–22 jobs is higher for the system running the ML model than for the original kernel. The extra load caused by the ML model was reflected in the system's workload, although the effect was tiny.

Last, we evaluated the performance of the modified system in executing real-world programs. Benchmark programs from the PARSEC Benchmark Suite of Princeton University [3] were used as examples of real-world parallel programs for testing. We tested the modified system with the `blackscholes`, `ferret`, `freqmine`, and `swaptions` applications and the `dedup` kernel from the benchmark suite. The benchmarks were run using 80 threads with the native-size inputs. We also implemented a C program that calculates Fibonacci numbers by recursively spawning pthreads to do the computation. That benchmark program, named fibo, was used as an example of programs that rapidly create and destroy a massive number of threads and are heavily affected by the load balancer's performance.

The average runtimes of 20 runs of the benchmark programs when running the new and original systems are shown in Figure 7. There is no significant discrepancy between the two systems in the runtimes of the benchmarks, although they show varying performances for different programs. We conclude that the new kernel with the fixed-point MLP model integrated into the scheduler is doing as well as the original kernel in running real-world programs.

4 Lessons Learned and Future Work

Based on our experience in building and evaluating the proposed system, we list some key insights that will drive the design of future ML-based OSes:

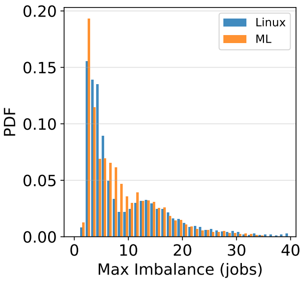

Figure 5: Max. job imbalance across CPU cores.

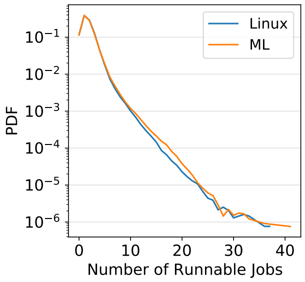

Figure 6: CFS runqueue length of the system.

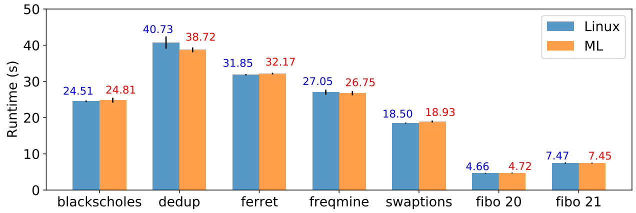

Figure 7: Mean runtimes of benchmark programs.

- **Insight 1:** Models need to capture dynamic and contextual application information. That is, trained models differ based on what the system state was when the training data were collected. For example, a model trained solely with data collected when the system had a high average load, although evaluated to have a 99% accuracy, achieved only 85% accuracy when tested on an idle system. Training the model with a comprehensive set of usage data that contain different classes of workloads can be important in improving performance.
- **Insight 2:** The tools we used to develop the ML model, including Python and TensorFlow, are not available in the kernel. The current tools need to be enhanced and migrated into the kernel space so the ML-based scheduling models can be trained and inferred in the kernel itself, saving development effort.
- **Insight 3:** Hardware performance counters can be integrated as input to our model via the Linux `perf_event` subsystem to provide resource usage information.
- **Insight 4:** Use of deep reinforcement learning can significantly help improve the performance beyond that of the current CFS scheduler, as it can capture dynamic contextual information not currently used.

5 Related Work

OS Scheduling. OS researchers and engineers have been continuously refining existing scheduling algorithms and designing new ones. In [12], Lozi et al. identify four performance bugs that affected the functionality of the Linux CFS and provide solutions to some of them. The bugs were load balancing related and caused by incorrect handling of the NUMA topology, flawed load-tracking metrics, and other issues. In 2009, veteran kernel programmer Con Kolivas developed the BFS [11] as an alternative to the CFS for better desktop responsiveness and more straightforward implementation. In [8], Faggioli et al. described how they implemented an Earliest Deadline First (EDF) scheduling class for the Linux kernel as an addition to the CFS scheduling class, intending to enhance scheduling for time-sensitive applications.

Hardware Resource Management. Different approaches have been studied to optimize the allocation of hardware resources during task scheduling. To resolve the SMT co-scheduling problem, prior work has suggested optimized scheduling policies for SMT cores [7, 15] and an ML-based resource allocation framework [4]. Tillenius et al. [16] proposed a resource-aware load balancing framework in which task resource consumption is annotated by the users and optimized by the scheduling policy.

Machine Learning in Scheduling. There have also been prior efforts to apply machine learning to improve OS task scheduling [6]. A common approach is to train machine learning models to learn the CPU utilization of the processes based on execution history, and then classify them to use different scheduling strategies [9, 14]. In [9], the authors tuned the scheduling policy by setting "nice" values of the processes based on predicted Turn-around-time (TaT). In [14], customized execution time slices were deliberately set in a Linux $O(1)$ scheduler according to predicted CPU utilization of the processes. Previous work that combined machine learning and resource-aware scheduling mostly focused on application scheduling in heterogeneous systems. Successful attempts have been made to train machine-learning models to perform dynamic scheduling of applications with varying workloads in user space on heterogeneous hardware consisting of CPUs, GPUs, and hardware accelerators [1, 2, 10]. In [2], hardware performance counters were used to make measurements of the utilization of system resources as input to a reinforcement learning ML model.

6 Conclusion

In this paper, we explored the application of machine learning to the OS load balancing algorithm of a multiprocessor system. We used imitation learning to incorporate a machine learning model as a system component in the kernel. The evaluation results show that the overhead brought by the in-kernel ML module does not impact system performance. Our experiment results indicate that it is indeed feasible to apply machine learning to tune the load balancing policy in the OS kernel. In future work, we intend to add statistics from hardware performance counters to the model and use deep reinforcement learning to improve the load balancing policy based on hardware resource usages of running processes.

Acknowledgments

We thank J. Applequist for her help in preparing this manuscript. This research was supported in part by the National Science Foundation (NSF) under Grant Nos. CNS 13-37732 and CNS 16-24790; by the IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR), a research collaboration that is part of the IBM AI Horizon Network; and by Intel through equipment donations. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, IBM, or Intel.

References

[1] Subho S. Banerjee, Arjun P. Athreya, Zbigniew Kalbarczyk, Steven Lumetta, and Ravishankar K. Iyer. 2018. A ML-Based Runtime System for Executing Dataflow Graphs on Heterogeneous Processors. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 533–534.

[2] Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. Inductive-bias-driven reinforcement learning for efficient schedules in heterogeneous clusters. In *Proceedings of the 37th International Conference on Machine Learning*. PMLR 119.

[3] Christian Bienia and Kai Li. 2011. *Benchmarking Modern Multiprocessors*. Ph.D. Dissertation. Princeton University.

[4] Ramazan Bitirgen, Engin Ipek, and Jose F. Martinez. 2008. Coordinated Management of Multiple Interacting Resources in Chip Multiprocessors: A Machine Learning Approach. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 41)*. IEEE Computer Society, USA, 318–329.

[5] Francisco J. Cazorla, Alex Ramirez, Mateo Valero, and Enrique Fernandez. 2004. Dynamically controlled resource allocation in SMT processors. In *37th International Symposium on Microarchitecture (MICRO-37'04)*. IEEE, 171–182.

[6] Siddharth Dias, Sidharth Naik, Sreepraneeth Kotaguddam, Sumedha Raman, and Namratha M. 2017. A machine learning approach for improving process scheduling: A survey. *International Journal of Computer Trends and Technology* 43 (2017), 1–4.

[7] A. El-Moursy, R. Garg, D. H. Albonesi, and S. Dwarkadas. 2006. Compatible phase co-scheduling on a CMP of multi-threaded processors. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. IEEE, 10 pp.–.

[8] Dario Faggioli, Fabio Checconi, Michael Trimarchi, and Claudio Scordino. 2009. An EDF scheduling class for the Linux kernel. In *Proceedings of the Eleventh Real-Time Linux Workshop*.

[9] Nikhil N. Jain, Srinivas K. R, and Syed Akram. 2018. Improvising process scheduling using machine learning. In *2018 3rd IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT)*. IEEE, 1379–1382.

[10] Yasir N. Khalid, Muhammad Aleem, Usman Ahmed, Muhammad A. Islam, and Muhammad A. Iqbal. 2019. Troodon: A machine-learning based load-balancing application scheduler for CPU–GPU system. *J. Parallel and Distrib. Comput.* 132 (2019), 79–94.

[11] Con Kolivas. 2010. FAQS about BFS. http://ck.kolivas.org/patches/bfs/bfs-faq.txt Accessed on 2020-06-03.

[12] 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* (London, United Kingdom) *(EuroSys '16)*. Association for Computing Machinery, New York, NY, USA, Article 1, 16 pages.

[13] Ingo Molnar. 2007. Modular Scheduler Core and Completely Fair Scheduler. https://lkml.org/lkml/2007/4/13/180 Accessed on 2020-06-03.

[14] 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.

[15] Suresh Siddha, Venkatesh Pallipadi, and Asit Mallick. 2005. Chip multi processing aware Linux kernel scheduler. In *Linux Symposium*. 193.

[16] Martin Tillenius, Elisabeth Larsson, Rosa M. Badia, and Xavier Martorell. 2015. Resource-Aware Task Scheduling. *ACM Trans. Embed. Comput. Syst.* 14, 1, Article 5 (2015), 25 pages.

[17] Linus Torvalds. 2008. Linux Kernel Documentation v4.15. https://www.kernel.org/doc/Documentation/ Accessed on 2020-06-03.