Exploring Prediction-Based Interrupt Scheduling

JUSTIN DONG, Northwestern University, USA
MICHAEL GAVRINCEA, Northwestern University, USA
EMILY WEI, Northwestern University, USA

Interrupts are a significant source of noise and nondeterminism in computer systems. Moreover, the transition from user to kernel mode for interrupt handling incurs a considerable overhead. We propose a novel approach to predict the arrival times of interrupts in the Nautilus kernel by approaching it as an optimal control problem.

The Nautilus kernel supports an interrupt thread model under which interrupts are deferred to run only when the interrupt thread is active. The interrupt thread runs for s seconds every p seconds. Our approach involves predicting future interrupts and then modifying the interrupt thread configuration parameters s and p to control when interrupts are enabled. We refer to this model as "scheduled interrupts." By dynamically optimizing s and p using our proposed model, we can effectively reduce total interrupt handling overhead, minimize dropped interrupts, while minimizing interrupt latency.

This model motivates the possibility to increase determinism in computer systems by reducing variability in the times when interrupts are executed. It also balances several performance tradeoffs. First, if $\frac{s}{p}$, the percentage of time interrupts are enabled, is too long, then the interrupt thread is left spinning with no interrupts to handle, but if it is too short, then the number of interrupts that we haven't handled will accrue, resulting in dropped interrupts. Second, if $\frac{p}{2}$, the average interrupt latency, is too high, then we risk dropping interrupts, but if it is too low, then context switch overhead is increased.

To test this model, we developed systems for collecting and parsing interrupt traces from applications in Linux on a single core of an x86 system and began developing systems to replay these interrupts in Nautilus and measure performance metrics. Through these methods, we not only setup infrastructure to facilitate future work on interrupt prediction methods, but also show, through preliminary results, the feasibility and possible challenges of such a system.

Additional Key Words and Phrases: interrupts, scheduler, machine learning, linear regression

ACM Reference Format:
Justin Dong, Michael Gavrincea, and Emily Wei. 2023. Exploring Prediction-Based Interrupt Scheduling. In *WLSD '23: Workshop on Low-level Software Development, June 07, 2023, Evanston, IL.* ACM, New York, NY, USA, 10 pages.

1 INTRODUCTION

In modern computer systems, interrupts serve as crucial events that disrupt the regular execution flow and require immediate attention from the operating system. However, interrupts introduce significant noise and non-determinism, which can lead to unpredictable behavior and resource inefficiencies within the system. Moreover, the process of switching from user to kernel mode for interrupt handling incurs a substantial overhead, further impacting system performance. This research paper proposes a novel approach to address these challenges by leveraging interrupt arrival time predictions in the Nautilus kernel to establish an optimal control framework for the interrupt handler, thereby minimizing overhead.

The Nautilus interrupt thread is a real-time thread that restricts the interrupt handler to run only in specific instances in time such that the interrupt thread is run for s seconds every p seconds [1].

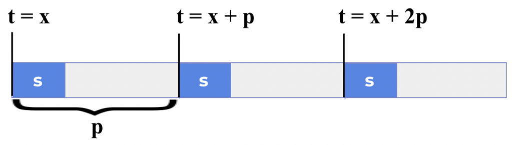


Fig. 1. Visual representation of p and s within the Nautilus interrupt thread. The blue section denotes when the instance when the interrupt thread is enabled.

The approach we propose, known as "scheduled interrupts", involves dynamically adjusting the Nautilus interrupt thread's runtime (s) and period (p) to control the enabling of interrupts. By optimizing the values of s and p using the proposed model, the scheduling of interrupts can be effectively managed. This approach aims to increase performance in computer systems by minimizing the overhead caused by context switching, all while balancing various performance tradeoffs.

One crucial tradeoff to consider is the percentage of time interrupts are enabled, which is denoted by the ratio $\frac{s}{p}$. If the value of $\frac{s}{p}$ is excessively long, the system spends an unnecessary amount of time handling interrupts even when no incoming interrupts are present, diminishing overall system performance as the interrupt thread is left spinning. On the other hand, if $\frac{s}{p}$ is too short, the accumulation of unhandled interrupts over time can lead to a notable increase in the average interrupt latency and possibly dropped interrupts. Additionally, the average interrupt latency $\frac{p}{2}$ plays a significant role in performance. If $\frac{p}{2}$ is too high, there is a risk of incoming interrupts being overwritten and consequently dropped. Conversely, if $\frac{p}{2}$ is minimized without careful consideration, it can result in the interrupt thread more frequently being enabled and disabled, leading to a high number of context switches. Such frequent context switching can worsen system performance, thus making the balance between minimizing average interrupt latency and managing the resulting overhead crucial.

To evaluate the effectiveness of the proposed model, interrupt traces were collected from applications running on a Linux environment in one core using a standard x86-64 Northwestern machine.

WLSD '23, June 07, 2023, Evanston, IL

[1]Note that the Nautilus kernel only guarantees that the interrupt thread will run for s seconds within every p seconds period of time, but does not specify when within that p seconds slice the interrupt thread will run.

The applications used were QEMU compilation, LLVM compilation, and standard NASA Advanced Supercomputing (NAS) Parallel Benchmarks. These traces were then used to generate a prediction model to predict the start time of the Xth interrupt into the future. Through this experimental setup, the research aims to demonstrate the practicality of interrupt predictions and explore the adoption of such techniques in computer systems.

2 BACKGROUND

2.1 Real Time Operating Systems

Real-time operating systems (RTOS) play a critical role in various domains where timing guarantees and deterministic behavior are extremely important, such as aerospace, industrial automation, and robotics. Unlike standard operating systems (OS), which prioritize general-purpose computing and resource sharing, RTOSs are designed to meet strict timing constraints and provide predictable responses to events, including interrupts.

2.2 Interrupts

In both real-time and standard operating systems, interrupts are an essential mechanism for handling external events that require immediate attention from the OS. An interrupt is typically generated by a hardware device or within traps in the code. The interrupt is then typically transferred to an interrupt controller known as an Advanced Programmable Interrupt Controller (APIC) which determines how to route the interrupt for increased efficiency. Once a CPU is chosen the APIC routes the interrupt to the CPU. Upon receiving an interrupt request (IRQ), the processor halts its current execution and transfers control to an interrupt handler routine within the operating system. To efficiently handle interrupts, operating systems rely on data structures and mechanisms, such as the Interrupt Descriptor Table (IDT). The IDT is an array of interrupt descriptors, where each one is associated with a specific interrupt or exception. The IDT provides a mapping between interrupt vectors and their corresponding interrupt handlers. When an interrupt occurs, the CPU consults the IDT to determine the appropriate handler to execute.

2.3 Interrupt Handling

The interrupt handling process involves several steps. First, the processor saves the current state by storing the program counter, registers, and other relevant information onto the stack. Then, the processor switches from user mode to kernel mode, granting the OS privileged access. Next, the interrupt handler is invoked, executing the code specific to the interrupt. After handling the interrupt, the processor restores the saved state from the stack and returns control to the interrupted program. In Nautilus, with the interrupt thread configuration, the CPU can handle interrupts at ideal times (such as after finishing a thread task) to reduce save and restore overhead.

2.4 Interrupt Challenges

While interrupts are crucial for timely event handling, they also introduce sources of noise and nondeterminism. The timing of interrupts can vary widely, leading to unpredictable behavior and performance inconsistencies in both real-time and standard operating systems [3].

Additionally, interrupts invoke a context switch from user mode to kernel mode which incurs considerable overhead, impacting overall system performance for three primary reasons.

1. State preservation requires that operating systems save the registers of the current program so it can continue executing once the interrupt has been handled.
2. The CPU needs to change its privilege level, which can require the control registers to be updated and the TLB to be flushed [2].
3. There is also the possibility of affecting other parts of the operating system, including reducing the efficiency of branch prediction and modifying the cache, resulting in slower memory accesses.

2.5 Nautilus

Nautilus is a real-time operating system (RTOS) known for its emphasis on determinism and predictable behavior. It is designed to meet strict timing constraints and provide reliable responses to interrupts and events. In Nautilus, interrupts are scheduled using a dedicated real-time thread known as the interrupt thread. This thread restricts the execution of interrupt handlers to specific instances in time. The interrupt thread runs for a duration of s seconds every p seconds, controlling when interrupts are enabled and processed by the system. The interrupt thread provides a mechanism for balancing the need for timely event handling with the overhead introduced by interrupt context switching. Given its unique interrupt handling architecture, Nautilus was chosen as the system of choice for evaluating the proposed interrupt scheduling model.

3 MOTIVATIONS

3.1 Nondeterminsim

As described above in section 2.4, there is a significant need to address challenges associated with interrupts and improve the determinism of interrupt handling in the Nautilus kernel, which is the real-time operating system of focus in this paper.

The existing state of interrupt handling in computer systems introduces significant noise and nondeterminism within the system [3]. This has downsides for applications such as safety critical systems.

Therefore, there is a critical need to improve the determinism of interrupt handling and optimize its performance. By proposing a novel approach to predict and schedule interrupt arrival times, our goal is to reduce variability in interrupt execution and increase system determinism.

3.2 Performance

By approaching interrupt prediction as an optimal control problem, our work introduces the concept of "scheduled interrupts." Our objective function, in this case, is to maintain the benefits of interrupts by avoiding dropping interrupts and minimizing handling latency

[2]Note that Nautilus operates fully in kernel mode so this is less of a concern

while simultaneously limiting the overhead that interrupts create, as described in section 2.4.

To accomplish this, our proposed interrupt prediction model dynamically adjusts the interrupt thread's runtime in Nautilus and period to control when interrupts are enabled, thereby achieving a balance between interrupt handling efficiency and system performance.

Through experimentation and analysis of collected interrupt traces, we seek to provide empirical evidence for the effectiveness of the proposed approach and its potential benefits for RTOS and potentially standard OS systems alike.

4 INTERRUPT SCHEDULING MODEL

The proposed approach to interrupt scheduling, referred to as the "scheduled interrupts" model, aims to optimize interrupt handling in the Nautilus kernel by dynamically adjusting the runtime (s) and period (p) of the interrupt thread. This section outlines the methodology used in this model, involving the prediction and scheduling of interrupts based on past interrupt behavior.

4.1 Prediction

To predict the arrival times of interrupts in the Nautilus kernel, the model leverages information from past interrupt occurrences. By analyzing previous interrupt traces, patterns and trends can be identified, allowing for the estimation of the start time of the next interrupt.

The model will be integrated into the Nautilus kernel, where it operates in real-time. As interrupts occur, the model continuously updates its predictions based on the most recent interrupt information. The model will be refit every X number of interrupts, allowing it to adapt to variations in interrupt behavior. Through the knowledge gained from past interrupt behavior, the model provides estimations of the start time of the next few interrupts into the future, offering valuable insights into the timing and occurrence of future interrupt events.

4.2 Scheduling

By leveraging the predictive capabilities of interrupt entry time, the Nautilus kernel can effectively update and optimize the parameters p and s of the interrupt thread, leading to hypothesized significant improvements in system performance. The ability to predict the entry time of interrupts provides valuable insights into future interrupt behavior, allowing for meaningful adjustments to be made in the scheduling of interrupts. This knowledge enables periodic updates to be made to the values of p and s, which respectively dictate the runtime and period of the interrupt thread in the Nautilus kernel.

The dynamic adjustment of these parameters plays a vital role in balancing the trade-offs associated with interrupt handling. By fine-tuning the values of p and s based on the predicted interrupt entry times, the system aims to find an optimal balance between minimizing interrupt handling overhead and maximizing system performance. If the interval between successive interrupts is predicted to be longer, the parameter p can be appropriately adjusted to reduce the frequency of context switching, thereby minimizing the overhead associated with interrupt handling. Similarly, if the predicted interrupt entry times indicate frequent and closely spaced interrupts, the parameter s can be adjusted to allocate a longer runtime to the interrupt thread, ensuring timely and efficient processing of interrupts without unnecessary interruptions. This adaptability allows for a more responsive and efficient utilization of system resources, ensuring that interrupts are handled in a timely manner while minimizing the impact on overall system performance.

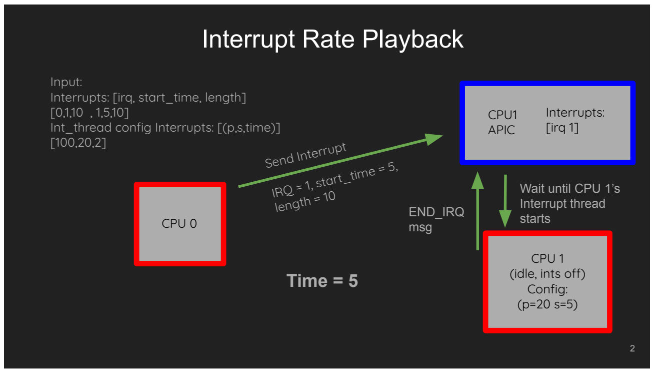
5 INTERRUPT PLAYBACK

5.1 Playback Overview

The complete interrupt playback feature was unfortunately not able to be finished in the timeframe of the project. However, most of it is complete and the implementation details of the final parts (sending p, s configuration changes to the interrupt thread and timing of the interrupt thread) have been thought out. Thus far, we are able to send to interrupts and measure how many are dropped, however, the final parts prior listed still need to be implemented.

The interrupt playback is configured to run on two cpus. The first cpu (cpu 0) runs the main script that handles playback. The second cpu (cpu 1) idles and is made to run the interrupts by sending a signal to the APIC from cpu 0 with the given interrupt we want to run at the correct time.

5.2 Implementation Details


Interrupt Rate Playback

Input:
Interrupts: [irq, start_time, length]
[0,1,10 , 1,5,10]
Int_thread config Interrupts: [(p,s,time)]
[100,20,2]

Send Interrupt
IRQ = 1, start_time = 5, length = 10
CPU 0
CPU1 APIC
Interrupts: [irq 1]
Wait until CPU 1's Interrupt thread starts
END_IRQ msg
CPU 1 (idle, ints off) Config: (p=20 s=5)
Time = 5
2

Fig. 2. Diagram of our interrupt playback mechanism.

Most of the relevant code can be found in `intersched.c` and `intersched.h`. We also demonstrated a visual of how our mechanism works in Figure 2. We first simply registered the shell command to run the overall interrupt scheduling scripts. Once the script starts, it first runs a subset of functions called "setup()" which configure the interrupt decision table and set up the interrupt handler depending on the input data defined at the top of the file. This is done by first reserving a range in the IDT table which depends on the number of unique interrupts in the trace, then assigns their entries. Each entry gets handled by the same interrupt handler, however, each entry has a different state. The states for all the entries are defined in the *interrupt_state* variable, and effectively contains the number of nanoseconds for which the interrupt runs. The handler would then read from this state and spin for the allotted time to simulate "handling" of the interrupt.

Next, we run a small function that times the amount of time it takes to get the time. While a small detail, we found that on average it took about 300 nanoseconds to get the interrupt time. Since we want to ensure that we are sending the interrupt to arrive at the correct time, we use this information to adjust our polling times.

After we get the average timing time, we begin playing back the interrupts. We read from the input data defined at the top of the file the first interrupt IRQ number, its start time, and the interrupt length (the time that the interrupt runs for). We then put the interrupt time into the address that holds that corresponding IRQ's state data. We then poll the time until it is time to send the interrupt, at which point we uses *apic_ipi* to signal the APIC of a new interrupt.

Within the interrupt handler, we first increment a counter, then use $nk\_delay()$ to spin the interrupt handler for the amount of time in the state. The increment counter is compared to the total trace length at the end of the script to determine how many interrupts were dropped.

5.3 Future Interrupt Playback Work

To implement timing of the interrupt threads, we have setup checks in $nk\_shed\_need\_resched()$ in scheduler.c to determine when we are switching into and out of the interrupt thread. Thus, we we add timers here to collect the time we are in the interrupt thread and keep this data by adding a new field within the *nk_sched_thread_state* struct which already keeps track of several other thread statistics. For the most accurate timing we could also keep track of cpu cycles instead of time as it would fit the same purpose as a means of comparing the efficiency of the interrupt thread.

To change the period of slice of the interrupt thread, the function $nk\_sched\_thread\_change\_constraints()$ could be called through an interrupt that we would send from the intersched.c file. We could simply request an extra IDT entry in our setup, and create a separate IRQ handler for it that would read from a different input list that would contain $[slice\_time, period\_time, time\_to\_change]$ and would be read and processed much in the same way of the other interrupts, requiring only minor modification to accomodate given the code already present.

6 METHODS

6.1 Testbed and Setup

We developed and tested our interrupt prediction model on an x86_64, Intel Xeon E5-2603 v2 machine running Linux. It has 8 cores and 32 GB of memory. Notably, while the actual execution and evaluation of the interrupt scheduling model will be performed within the Nautilus kernel, it was necessary to collect interrupt trace data on the Linux platform due to the broader availability of resources offered by Linux to collect interrupts. While obviously collecting interrupts directly from Nautilus would be ideal, this would entail not only building a data collection tool, but also building the benchmarks to collect them as well. Thus, due to time constraints, we opted to use Linux to collect traces. The ftrace tool, integrated within the Linux environment, allowed for the seamless and efficient collection of comprehensive trace data for subsequent analysis. This experimental setup ensured the accuracy and reliability of the collected interrupt trace data.

To collect our traces, we compiled QEMU and LLVM and ran the NAS Parallel Benchmarks (Class W). We ran and traced each of these workloads on a single core.

After collecting our traces, we compiled our linear regression model by using the O3 optimization setting and various floating point optimizations as well.

Lastly, we began developing interrupt playback within Nautilus to test our model.

6.2 Interrupt Trace Collection

6.2.1 Linux Ftrace Tool. The ftrace framework tool, embedded within the Linux kernel, was used to collect interrupt data and investigate the behavior of interrupt handling. In particular, the ftrace function_graph feature was employed to trace and record the start and end times of interrupt handling events with high precision.

Enabling the ftrace function_graph feature involved configuration through the debugfs filesystem, a convenient interface that allows researchers to interact seamlessly with ftrace. Since the interrupts coming through may change with different configurations or hardware, note that f_trace may require further configuration to adapt to other systems. By precisely specifying the desired events or functions to trace and activating the function_graph feature, dynamic code was able to be injected into the kernel at runtime. This code instrumentation captures the function calls and returns, storing the information in a trace buffer for subsequent analysis.

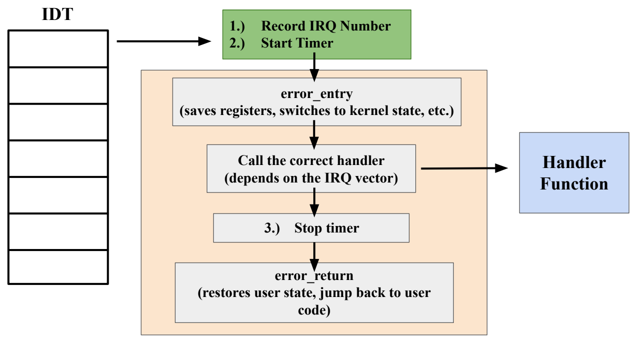

IDT
1.) Record IRQ Number
2.) Start Timer
error_entry
(saves registers, switches to kernel state, etc.)
Call the correct handler
(depends on the IRQ vector)
Handler Function
3.) Stop timer
error_return
(restores user state, jump back to user code)

Fig. 3. Diagram depicting the interal steps that occur during interrupt handling. The function_graph timer is started when the IRQ enters the timer is stopped upon exiting the IRQ handler function.

The function_graph tracer within ftrace provided a mechanism for meticulously tracing the sequence of function calls and returns during interrupt execution. By instrumenting relevant interrupt handling functions, such as the handle_irq_event function(), irq_handler_entry(), and irq_handle_exit function(), we were able to accurately identify the exact entry and exit points of interrupt execution, facilitating an in-depth analysis of interrupt scheduling.

6.2.2 Applications Used. The applications employed in this study encompassed a diverse range of workloads, including QEMU compilation, LLVM compilation, and the NAS Parallel Benchmarks. These applications were selected due to their widespread usage, high interrupt throughput, and generalizability to various other application domains. Their utilization facilitated the extraction of insights that extended beyond specific scenarios, allowing for broader conclusions regarding interrupt behavior.

6.2.3 Data Cleaning and Organization. Upon acquiring the interrupt data traces using the ftrace tool, a Python script was implemented to efficiently cleanse and organize the extensive information obtained. The script parsed the raw trace data, extracting essential details including the IRQ number, start time (measured in seconds), and end time (also measured in seconds). This step facilitated the standardization of the interrupt trace data, establishing a consistent format that allowed for seamless integration with the interrupt prediction model.

6.3 Interrupt Prediction Model

6.3.1 Selecting Model Architecture. For our interrupt prediction model, our goal was to predict the start time of the next N interrupts given the previous N interrupts, such that N is a positive integer. We considered both statistical and machine learning models, including Markov models, autoregressive integrated moving average (ARIMA) models, and neural networks, among others. Markov models operate under the assumption that the probability of future events depends only on the current state of the system (not the past). ARIMA models, on the other hand, use past values to predict future ones while accounting for cyclic trends and past forecast errors.

However, we decided to use a linear regression model for three primary reasons.

1. Significantly lower overhead.
2. Very accurate for interrupt traces.
3. Typically outperforms Markov and ARIMA models during testing.

6.3.2 Linear Regression Model. A linear regression model assumes that an independent and dependent variable exhibit a linear relationship such that it can be represented by the equation

$$y = mx + b, \tag{1}$$

where x is the independent variable and y is the dependent variable.

For our linear regression model, we held set x as the entry time before the computer switches to kernel mode to handle the interrupt. We set y as the index of the interrupt in the trace data. We chose to refit the model every N interrupts, such that N is a positive integer. Each time, we trained our model on data from the past N interrupts.

We implemented our model using the least squares algorithm.

6.3.3 Model Evaluation. To evaluate our model, we compared it with a naive implementation of interrupt prediction: predicting that the entry time of the next interrupt is equal to the entry time of the Nth previous interrupt, such that N is a positive integer.

To test the accuracy of our model's predictions, we use root mean square error (RMSE). We chose to use RMSE as a metric to evaluate our model for two main reasons.

1. Accounts for both the direction and magnitude of error.
2. Expressed in the same units (seconds) as the original data, resulting in improved interpretability.

RMSE represents the average difference between the predicted values and the actual values in the test dataset. The calculation of RMSE involves the following steps.

Let N be the total number of data points in the test dataset, and let $\{y_i\}$ represent the actual values of the interrupt entry times. Similarly, let $\{\hat{y}_i\}$ denote the predicted values of the interrupt entry times.

1. Calculate the residuals, $\{e_i\}$:
$$e_i = y_i - \hat{y}_i.$$
2. Square the residuals:
$$e_i^2 = (y_i - \hat{y}_i)^2.$$
3. Find the sum of the squared residuals:
$$\sum_{i=1}^{N} e_i^2.$$
4. Calculate the mean squared error (MSE):
$$MSE = \frac{1}{N}\sum_{i=1}^{N} e_i^2.$$
5. Calculate the RMSE:
$$RMSE = \sqrt{MSE}.$$

Lower values for the RMSE indicate better predictions and increased accuracy. On the other hand, higher values correspond to greater errors and worse predictions.

However, RMSE is a metric that can be relatively more sensitive to outliers in the data because it squares the residuals. This risk is mitigated by our use of mean absolute percentage error as a metric to help confirm that our RMSE results are representative.

7 RESULTS

7.1 Model Overhead

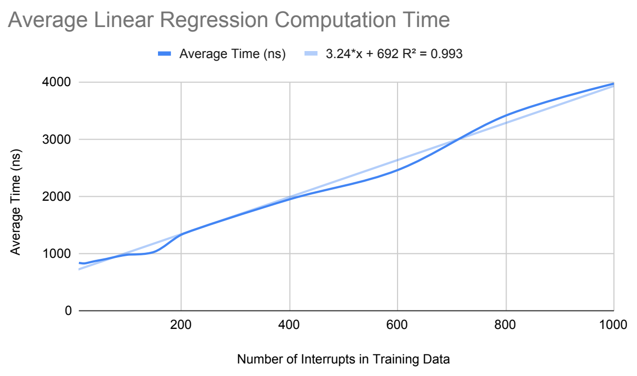


Fig. 4. Graph of the average overhead of the linear regression model compared with the number of training data points.

Our linear regression model has a relatively low overhead, even at the timing granularity of interrupts. Let n be the number of samples in the training data and let m be the total number of samples, such that $n \leq m$. We achieve the following theoretical time and space complexities in the code, such that train time represents fitting the linear regression model and test time represents calculating the predictions.

$$\begin{aligned} \text{Train Time Complexity} &= O(n) \\ \text{Test Time Complexity} &= O(n) \\ \text{Space Complexity} &= O(m) \end{aligned} \tag{2}$$

To test this empirically, we implemented a version of our linear regression model in C using the least squares method and timed it using the `clock_gettime()` function from the standard C library to achieve nanosecond level precision. The model was implemented in serial but used several compiler optimizations.

In Figure 4, we demonstrate that even with 1,000 training data points for the linear regression model, we still have roughly only 4 microseconds of computation time, demonstrating the possibility for this model to integrate in an operating system kernel without significantly hurting performance.

7.2 Model Accuracy

7.2.1 Accuracy. Overall, our linear regression prediction model greatly outperformed the naive prediction implementation.

In Figure 5, our model achieves an RMSE of 0.192, compared to the naive model's RMSE of 0.702 seconds when predicting interrupt entry times for LLVM compilation and refitting the model every 20 interrupts. Additionally, in Figure 6, our model achieves an RMSE of 0.373 seconds, compared to the naive model's RMSE of 3.641 seconds when predicting interrupt entry times for QEMU compilation and refitting the model every 100 interrupts.

7.2.2 Increasing the Number of Interrupts Between Refits. We also study several of the NAS Parallel Benchmarks (NPB) to confirm these results. In particular, we found the Block Tri-diagonal solver (BT) and Lower-Upper Gauss-Seidel solver (LU) benchmarks interesting because they generated higher levels of interrupts than the other benchmarks. Both workloads generated very apparent linear trends in their interrupts over time, as shown in Figure 7, further motivating the use of our linear regression model.

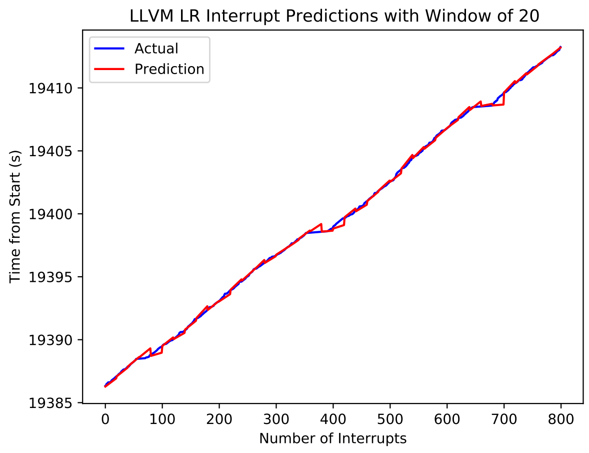
Figure 8 demonstrates one of the primary advantages of our linear regression model: its error grows very slowly over time as the number of interrupts between each model refit increases.

Hence, we demonstrate the predictability of interrupts and the viability of our linear regression model for various applications and benchmarks, even as the number of interrupts between each model refit increases.

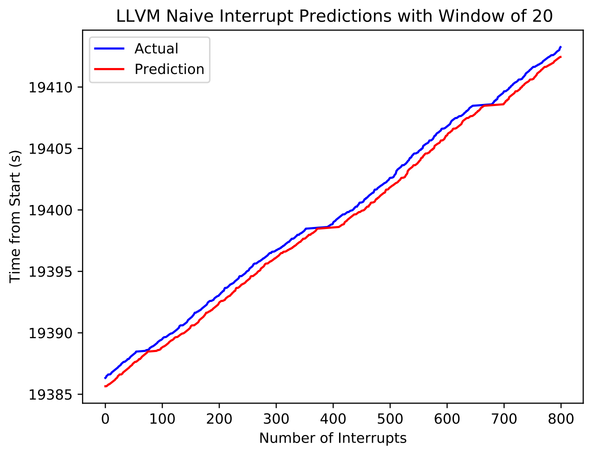
8 LIMITATIONS

8.1 Single Core

A limitation of the interrupt trace data collection process in this study is the use of a single core for running the QEMU, LLVM, and NAS benchmarks. By confining the workload to a single CPU, the interrupt behavior and patterns may not fully represent those of a multi-core system. Interrupts generated by other cores and their interactions could not be captured, potentially limiting the generalizability of the findings. To mitigate this limitation in future



(a) Linear Regression Model for LLVM Compilation



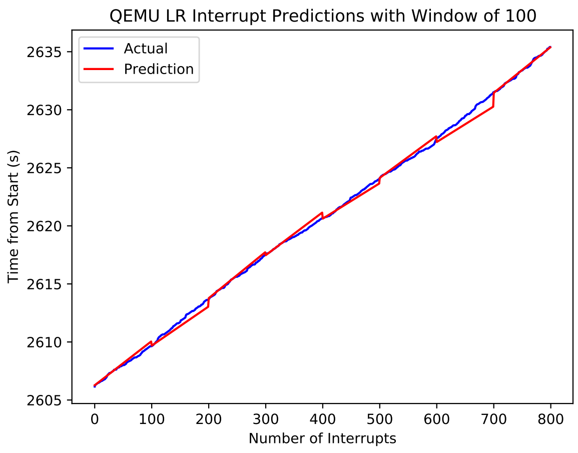
(b) Naive Model for LLVM Compilation

Fig. 5. Graphs of our models to predict the last 800 interrupts traced during LLVM compilation, such that we refit our model every 20 interrupts.

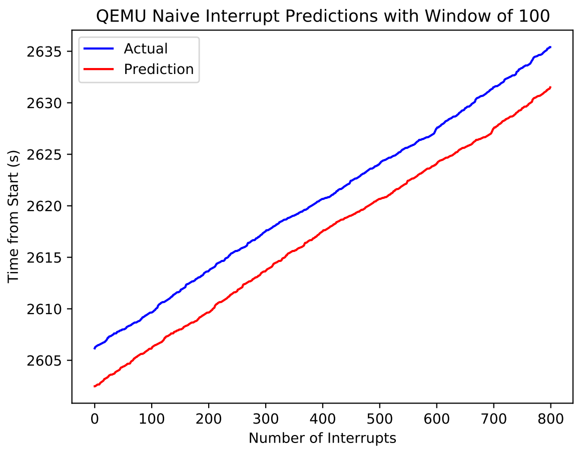
research, expanding the experimental setup to include multiple cores would be beneficial. By distributing the workload across multiple CPUs, a more realistic and comprehensive analysis of interrupt behavior can be achieved. This would provide insights into the impact of interrupt scheduling and handling on system performance in a multi-core environment, enabling a more accurate evaluation of the proposed interrupt scheduling model.

8.2 Poor Prediction on Task Changes

One important limitation is that our predictions perform well only under these long-term workloads. Even within these workloads, towards the end of the workload out predictions become less accurate. For instance, in Figure 9, after the core finishes processig the BT and LU benchmarks and switches to idle, the interrupts become significantly less predictable. We strongly suspect that this is because

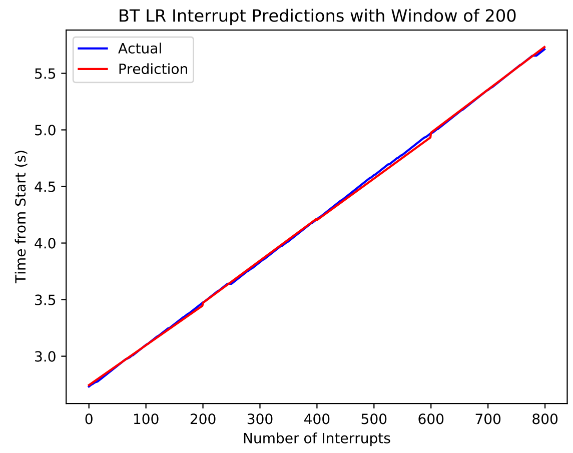


(a) Linear Regression Model for QEMU Compilation

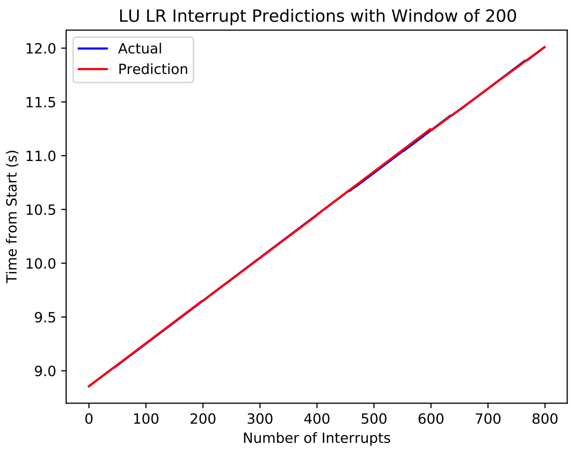


(b) Naive Model for QEMU Compilation

Fig. 6. Graphs of our models to predict the last 800 interrupts traced during QEMU compilation, such that we refit our model every 100 interrupts.



(a) Linear Regression Model for BT Benchmark



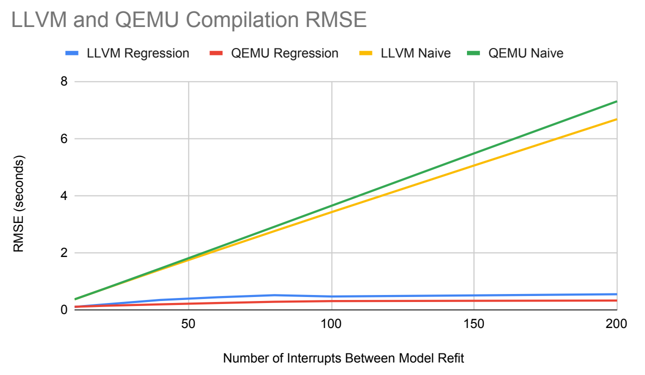
(b) Linear Regression Model for LU Benchmark

Fig. 7. Graphs of our models to predict the last 800 interrupts traced for the BT and LU NPB benchmarks, such that we refit our model every 200 interrupts.

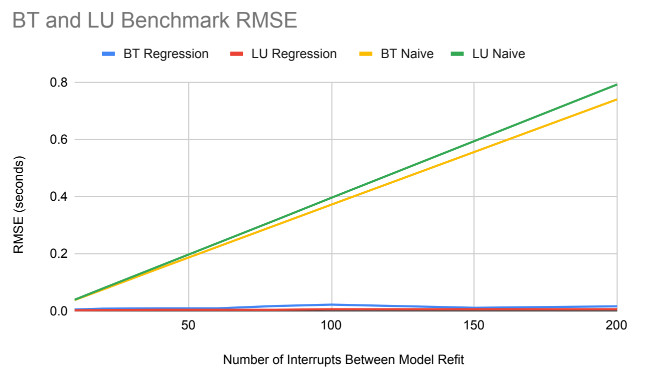
any changes in the system tasks will inevitably result to rapidly changing interrupt patterns which cannot be well predicted even by our more advanced models even for short times into the future. This is supported empirically by our results: the RMSE for the BT benchmark increases from 0.016 seconds to 0.295 seconds and the RMSE for the LU benchmark increases from 0.006 seconds to 0.124 seconds.

Thus, it is likely that interrupt prediction will be most useful to systems running at relatively constant loads with constant tasks changing at intervals no shorter than several seconds. A good example of something that would likely benefit from interrupt prediction would be a server machine running large computational models. Even from our preliminary search, it is evident that for highly interactive systems, interrupts would likely not be predictable with simple statistical models to an extent to be useful in making decisions on interrupt handling. Conversely, a system in which interrupt prediction is only activated when high constant workload and low user interactivity is detected may also prove somewhat useful (for example cloud computing where user interactivity occurs but is minimal).

It is possible that more complex models using machine learning may be able to more accurately predict interrupts in more interactive systems, however, this would also bring significant overhead to prediction plus the additional overhead from having to much more rapidly change system configurations to match.

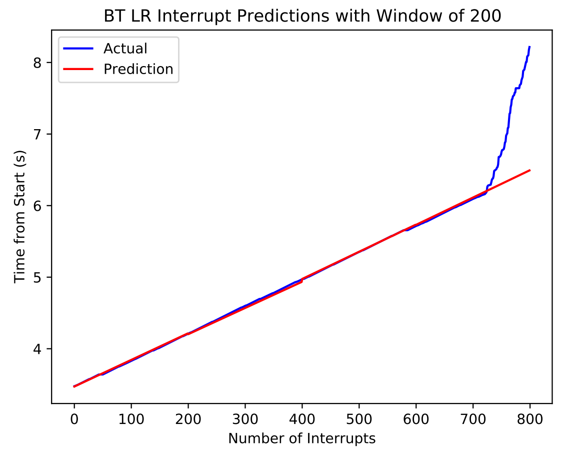


(a) RMSE of Naive and Linear Regression Models for LLVM and QEMU Compilation

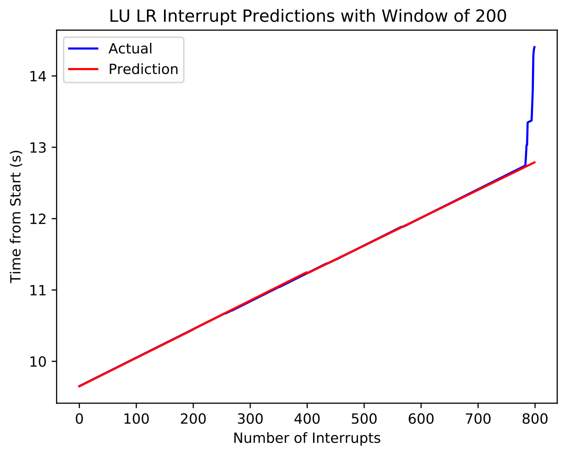


(b) RMSE of Naive and Linear Regression Models for BT and LU Benchmarks

Fig. 8. Graphs of the RMSE metrics of the naive and linear regression models to demonstrate differences in their accuracy for increasing numbers of interrupts between each model refit.



(a) Linear Regression Model for BT Benchmark Switching to Idle



(b) Linear Regression Model for LU Benchmark Switching to Idle

Fig. 9. Graphs of our models to predict the last 800 interrupts for the BT and LU NPB benchmarks, **include idle time after the end of the benchmark**, such that we refit our model every 200 interrupts.

8.3 Translating Prediction to Optimal Configuration

A big non-trivial consideration is how to determine an optimal configuration for the interrupt thread given a relatively accurate prediction. If the slice is set too stringently, then if the prediction is even slightly longer, you run the risk of not finishing all the expected interrupts. Determining an optimal period is even more difficult as there is no straightforward way to determine how frequently we would need to run the interrupt thread to avoid dropping interrupts because we don't know the frequency of each individual interrupt based off of the prediction. Additional and more complex models would be needed to determine each interrupts individual frequencies which would likely have too much overhead to be useful.

Furthermore, to get accurate translations we would also need to take into account additional nuances of the interrupt thread such as the fact that it could run for its designated period of time at any point with p time (not necessarily every p time), and the ordering in which it processes the deferred interrupts. Lastly, if we were to generalize the model to multiple cores, then we would also need to take into consideration how interrupts could be processed by different CPUs, further adding to the complexity of an optimal configuration as each CPU would have its own interrupt thread with its own configuration.

9 FUTURE WORK

9.1 Complete Interrupt Playback Feature

In terms of future work for this project, the first item to pursue would be finalizing interrupt playback as referenced previously. With interrupt playback built, we could compare handling interrupts with our model and a naive optimal configuration mapping to determine period and slice to a standard playback a constant period and slice. This way we could determine whether our model benefits being updated during an individual task and how frequently that update would occur. Looking back at our model results, we can see that the actual predictions are relatively constant in terms of how long the next x interrupts will take. Additionally, if we find that updating during a task if non-beneficial, then we can attempt to collect traces with tasks changing every few seconds and see if, as hypothesized in *Limitations: Poor Prediction on Task Changes*, such a scenario would instead prove of large benefit. In conclusion, playback would simply allow us to fully test different models and determine concrete number of performance increase.

9.2 Calculate Optimal Parameters

After finalizing interrupt playback, the next task would be determining some mapping from predictions to optimal parameters. One method for tackling this problem would be to use playback to fine tune parameters for different traces on different tasks and then use that information to try to determine first if we can determine a relatively efficient mapping only from knowing the timing of the next x interrupts, or if additional models (for example one that predicts the IRQ numbers of the next interrupts or the frequency of specific IRQs) would be needed and/or could help.

9.3 Integrate Model and Scheduler

After determining a relatively efficient mapping, the next task would be creating a full implementation by directly implementing the models and mappings into the scheduler. This would provide the most accurate information into how well interrupt prediction would realistically work.

9.4 Nautilus Benchmarks

After implementation, some benchmarks in Nautilus need to be created in order to optimally test the improvements garnered by interrupt prediction. These benchmarks would naturally need to be IO heavy to fully test the system. Based on the benchmarks, mappings determined earlier could be fined tuned to the real system.

9.5 Multiple Cores

The last task would be to implement compatibility with multiple-cores. This could be as simple as having each core have its own model. However, to get optimal efficiency, it is likely that the model would communicate with the APIC in order to assist in determining not only interrupt thread parameters for different cores, but also the optimal core to send a given interrupt to. In other words, the additional information from the models would be able to assist the APIC in determining which core to send interrupts to.

10 RELATED WORK

There has been pre-existing work on predicting the next interrupt to enhance the performance and efficiency of the Linux kernel [2]. Those approaches use statistics and mathematics to predict a *single interrupt* into the future.

Our linear regression model differs by predicting the entry time of interrupts several seconds into the future. We also used ftrace to collect our interrupt data, whereas previous work has directly modified the Linux kernel to capture timings.

Additionally, there has been prior work done on scheduling interrupts [1]. However, this work focuses more on efficiently scheduling higher priority interrupts, as opposed to using predictions to inform interrupt scheduling.

Previous studies have also highlighted I/O, periodic, preemption, and scheduling interrupts as significant origins of operating system disturbances, although page faults can contribute considerably to the non-deterministic behavior of numerous applications [3]. Nevertheless, hardware interrupts remain a substantial source of noise, thus driving our proposition of our scheduled interrupts model as an approach to mitigate nondeterminism.

11 CONCLUSION

This research paper proposes a novel approach to address the challenges posed by interrupts in computer systems. Interrupts are a significant source of noise and non-determinism, and the transition from user to kernel mode for interrupt handling incurs overhead. The proposed approach, called "scheduled interrupts," treats interrupt prediction as an optimal control problem in the Nautilus kernel.

By dynamically adjusting the runtime and period of the interrupt thread, the scheduling of interrupts can be effectively managed. This approach aims to increase determinism in computer systems by reducing variability in interrupt execution. It also considers the tradeoffs between interrupt handling efficiency and system performance.

We demonstrate that interrupt entry times are predictable during various workloads using a linear least-squares regression model with little overhead. In under 4 microseconds, we can predict the entry times of future interrupts within a few tenths of a second (depending on the parameters of our model and the workload given).

More work still needs to be done to integrate our interrupt prediction model into the Nautilus kernel scheduler. However, we have developed interrupt prediction and are finalizing interrupt playback, which motivate the continued development of such a feature. The research contributes to the field of interrupt scheduling and offers insights into improving determinism and optimizing performance. By leveraging interrupt prediction and dynamic adjustment of scheduling parameters, the proposed approach shows promise in mitigating the challenges posed by interrupts in modern computer systems.

ACKNOWLEDGMENTS

- To Professor Peter Dinda, for teaching COMP_SCI 446-0 and mentoring us throughout the project.
- To Kirill Nagaitsev, for explaining interrupt traces and behaviors.
- To Nadharm Dhiantravan, for demonstrating how to use ftrace to collect interrupt traces.

REFERENCES

[1] Minsub Lee, Juyoung Lee, Andrii Shyshkalov, Jaevaek Seo, Intaek Hong, and Insik Shin. 2010. On Interrupt Scheduling Based on Process Priority for Predictable Real-Time Behavior. *SIGBED Rev.* 7, 1, Article 6 (Jan. 2010), 4 pages. https://doi.org/10.1145/1851166.1851174

[2] Daniel Lezcano and Georges Da Costa. 2023. O/S level interrupt prediction for performance and energy management on Android. *IEEE Transactions on Mobile Computing* (March 2023), 1–12. https://doi.org/10.1109/TMC.2023.3253798

[3] Alessandro Morari, Roberto Gioiosa, Robert W. Wisniewski, Francisco J. Cazorla, and Mateo Valero. 2011. A Quantitative Analysis of OS Noise. In *2011 IEEE*

International Parallel and Distributed Processing Symposium. 852–863. https://doi.org/10.1109/IPDPS.2011.84