A Prediction System Service

Zhizhou Zhang
UC Santa Barbara
Santa Barbara, CA, USA
zhizhouzhang@ucsb.edu

Alvin Oliver Glova
UC Santa Barbara
Santa Barbara, CA, USA
aomglova@ece.ucsb.edu

Timothy Sherwood
UC Santa Barbara
Santa Barbara, CA, USA
sherwood@cs.ucsb.edu

Jonathan Balkind
UC Santa Barbara
Santa Barbara, CA, USA
jbalkind@ucsb.edu

ABSTRACT

To better facilitate application performance programming we propose a software optimization strategy enabled by a novel low-latency Prediction System Service (PSS). Rather than relying on nuanced domain-specific knowledge or slapdash heuristics, a system service for prediction encourages programmers to spend their time uncovering new levers for optimization rather than worrying about the details of their control. The core idea is to write optimizations that improve performance in specific cases, or under specific tunings, and leave the decision of how and when exactly to apply those optimizations to the system to learn through feedback-directed learning. Such a prediction service can be implemented in any number of ways, including as a shared library that can be easily reused by software written in different programming languages, and opens the door to both new software optimization patterns and hardware design possibilities.

As a demonstration of the utility of this approach, we show that three very different application-targeted optimization scenarios can each benefit from even a very straightforward perceptron-based implementation of the PSS as long as the service latency can be held low. First, we show that PSS can be used to more intelligently guide hardware lock elision with resulting speedups over a baseline implementation by 34% on average. Second, we show that a PSS can find good configuration parameters for PyPy's Just-In-Time (JIT) compiler resulting in 15% speedup on average. Last, we show PSS can guide the page reclamation task within a kernel memory management subsystem to reduce the average memory latency by 33% on average. In all three cases, this new optimization pattern with service support is able to meet or beat the best-known handcrafted methods with a fraction of the complexity.

CCS CONCEPTS

• **Computer systems organization** → **Architectures**; **Real-time operating systems**; • **Software and its engineering** → **Software performance**; • **Computing methodologies** → **Perceptron algorithm**.

KEYWORDS

software optimization, runtime optimization, perceptron, Operation System, hardware lock elision, Just-In-Time compiler, memory management

ACM Reference Format:
Zhizhou Zhang, Alvin Oliver Glova, Timothy Sherwood, and Jonathan Balkind. 2023. A Prediction System Service. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3575693.3575714

1 INTRODUCTION

When the low-hanging fruit of obviously inefficient implementation has been stripped away and the performance of an application is still a critical concern, capable performance programmers often find themselves attempting to navigate a complex set of trade-offs. When is it faster to just lock this data structure versus wrapping it optimistically in a transaction? When should I just execute the unoptimized version of this function versus investing the time to make it faster? When should the operating system pull this resource so it can go to a better use somewhere else? Answering such questions is a matter of balancing a set of conflicting forces. As our applications, systems, and hardware grow increasingly complex, it is hard to understand (or even characterize) all of the forces relevant to good decision making – and even more difficult to navigate those forces with simple ad-hoc heuristics.

Of course, the fact that *machine learning* has proved particularly capable of navigating exactly this type of complex optimization space is not something that has been lost on application researchers. Machine learning techniques have been demonstrated for optimizing data structures [41], implementing state-of-the-art recommendation systems [14], improving anomaly detection [58], and learning the structure and optimal access of databases [40]. In the case of TVM [13], the optimising compiler could produce machine learning kernels that beat human optimisers', leading to significant performance improvements.

This style of optimization is bound to become increasingly common in the coming years and it makes little sense for each and every application to roll out their own internal embedded ML framework for dynamically controlling a few parameters. Such an approach requires each application to support their own machine learning code base and elides opportunities for sharing of memory or exploiting hardware resources. Instead, it is time to consider the question of what new *abstractions* are necessary to lower the barrier to entry and sustainably support this important style of optimization.

While the process of learning a good response from noisy examples is well covered in the machine learning literature, actually deploying the ability to make predictions in a manner useful for software optimization requires some innovation. Because these predictions are often (by the nature of targeting performance-critical code) directly on the critical path, their utility is a function of both their accuracy *and* their latency. A prediction service must be both cheap (in terms of computational overhead) and it must be good enough (providing enough performance benefit to be comparable to or better than a hand-tuned approach).

In this paper, we argue that it is possible and worthwhile to introduce a common, simple, shared prediction mechanism to a variety of runtime tasks and that the right location for this mechanism is as a system service. A system-wide prediction service can operate usefully with as few as two API functions and can be made to both be easily reusable across the software stack and allow for additional innovation on both sides of the interface. By functioning as a service, the operating system can enable sharing of training information across user applications when desired or restrict usage according to system policy. The service can also be provided within the kernel for use by runtime services that otherwise rely on domain-specific heuristics to make performance decisions.

To explore the potential of a PSS to enable optimization, we prototype this change to a full operating system and examine, both qualitatively and quantitatively, the capability of our nascent service to ease optimization across three different scenarios calling back to the questions at the beginning of the introduction: transactional lock elision, JIT parameter tuning, and page reclaim. These scenarios exercise the interface in user and system mode, across multiple languages, and in both aiding online decisions and parameter tuning settings. Specifically, Transactional Lock Elision [62] via Hardware Transactional Memory (HTM) is a classic example of a fastpath-slowpath heuristic employed by software; we will show how our predictor guides this decision on when to use HTM (fastpath) and when to fallback to locks (slowpath). Just-in-Time (JIT) compilation always has a tension between high compile time if highly optimized and low code quality if not well optimized while offering a whole search space of possible optimization parameters; we will illustrate a way to employ PSS to quickly arrive at optimization parameters that improve program speed as compared to the existing parameter tuning solution provided by the human-optimised PyPy runtime. Page reclaim in the Linux kernel under high congestion relies on a careful, heuristic-driven consideration of memory usage and storage device utilisation in order to maintain global performance; we show that introducing PSS to the kernel can significantly outperform human-optimised heuristics developed within the last year. Specifically we:

- Introduce the novel concept of prediction as a system service
- Demonstrate that an exceedingly simple interface providing only `predict`, `update`, and `reset` is all that is required to be useful for software optimization.
- Develop a complete proof-of-concept implementation capable of providing Linux processes with useful and actionable predictions in 4.19 ns.
- Evaluate the effect of these prediction-driven optimizations across a variety of both user and kernel mode applications and demonstrate the resulting system performance improves Transactional Lock Elision by 34% on average, PyPy JIT parameter tuning by 15% on average over microbenchmarks and 12% over macrobenchmarks, and provides a 33% average latency reduction for page reclaim.

We begin with more description of the concepts and requirements of Prediction as a Service in Section 2 followed by details of our prototype implementation and the reasoning behind our latency-optimized software architecture in Section 3. Section 4 describes the application use scenarios in detail and is followed by a more detailed quantitative evaluation, related work, and conclusions in Sections 5, 6, and 7 respectively.

2 PREDICTION AS A SERVICE

If one were to take a careful catalog of all of the performance optimization techniques available, there is no question that a particularly large chapter would be required for those driven by prediction. Operating systems can predict the next set of disk pages required by applications and speculatively bring them into main memory [35, 38]. Memory access patterns of CPU cores or OS threads can be learned and the OS can automatically migrate page frames from a remote NUMA socket to a local socket to reduce latency [16]. Lock implementations can have a spin-and-then-block [20, 37] logic which spins for a set time before falling back to heavyweight OS-facilitated blocking. Transactional memory [31] (both in software and hardware) can dynamically and speculatively adjust to observed contention [17, 66, 72].

While these techniques rely on a prediction, most are not *explicit* about the predictive nature of their ability to achieve a speedup. Instead, most hide their predictive nature in the choice of a parameter or in a set of criteria used to make a selection. Unfortunately: **1)** *Parameter choices are often ad-hoc*, relying on limited use cases and/or hard-won domain-specific expertise making such approaches fragile and hard to scale. **2)** *Even when well informed by data, most parameter choices are still static*, meaning they are unable to adapt to the changing machine state or objectives. Profiles can help gather information on effective parameters, but profiling requires either well-understood use-cases or the ability to gather useful information in production with low overhead. Both of these are possible, but **3)** *Complex dynamic approaches for either prediction or profiling increase application complexity* which, in turn, makes the system harder to support across multiple platforms and increases the code footprint significantly. Finally **4)** *There is no effective way to share developments.* Programmers can spend non-trivial amounts of time optimizing the code in one specific language given a predetermined interface, but as we cross languages, as we have collections of smaller services, and as we seek to exploit hardware to help in the process, there is little opportunity for reuse.

In contrast, an ideal system would be **straightforward** to understand and simple to use. Users should only need to specify a target function, candidate solutions, and feedback. A prediction service would generate a prediction (informing, for example, which equivalent code path take) and update the model based on feedback. To be effective the prediction service must be **low overhead** both in terms of training and inference. The service will need to provide useful predictions as early as possible, to avoid long warm-up overheads, and provide those predictions with very low latency, to avoid eating into all of the potential performance improvements such predictions might provide. The prediction service should also be suitably **general purpose**, allowing it to be applied to a wide range of applications, possibly written in multiple programming languages. It should not only work with one or a few domain specific scenarios.

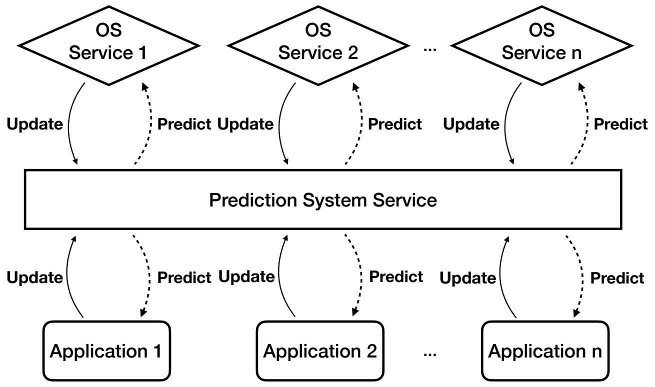


Figure 1: Design of Prediction System Service

An effective target for such an optimization, in turn, needs to be both *measurable* (meaning that it is possible to determine the "goodness" of the prediction to inform learning) and *correctness preserving* (meaning paths under all possible predictions are equally correct even if not equally desirable).

3 DESIGN AND IMPLEMENTATION OF A PSS

Informed by the requirements above, a Prediction System Service (PSS) provides a standard interface and straightforward prediction and update procedures. At a high level, the PSS takes input of the programmer's choice and returns the value of the prediction. In our proof of concept we limit ourselves to predictions along a single dimension where the return values can be interpreted as "predict true" when positive and "predict false" when negative and the magnitude of the return value shares some degree of confidence in the prediction (particularly useful when the costs of mispredictions are asymmetric or when true and false are used iteratively to narrow in on some balance point). The system then attempts to optimize its predictions over time based on feedback in the form of updates.

3.1 System Interface

PSS can be implemented with two core functions, `predict` and `update`, and one state management function, `reset`, with behavior as follows:

Predict: Given input features and stored weights, `predict` generates a binary result prediction which determines which path to take. The format of the input features can be different depending on the prediction scenario. The function signature is

```
int predict(int* features, int len)
```

where the input is an array of user-specified features for predict with length of `len` and the returned prediction value is an integer. The number and value of features can be changed by users for different scenarios.

Update: Based on the predicted and observed results, PSS will `update` the stored model parameters accordingly depending on whether the prediction was correct or not. The function signature can be viewed as

```
void update(int* features, int len, bool dir)
```

where the input parameter contains a feature array and its length like `predict` and one Boolean variable to indicate whether the prediction is correct or not.

Reset: This function allows the users to initialize the stored PSS data, either by section or in totality. It can be called if certain environment parameters of the prediction have been changed or to completely wipe the PSS data. As an example, when some data need to be reused without initializing all data used by PSS for prediction, we can use this function to selectively clear only some data. The function signature can be viewed as

```
void reset(int* features, int len, bool all)
```

where the input feature array and length are similar to the previous two cases and there is an additional boolean variable to indicate whether to wipe out the entire PSS data or clean a specific entry.

3.2 Prediction Unit Design

While there are many possible implementations of a PSS, for our proof of concept implementation we wanted to pick a design that we knew would have consistently low latency and that would help us test our hypothesis that even relatively simple predictions would be an important step beyond the state of the art in many potential optimizations. As such, for this effort, we limit our evaluation to an online perceptron predictor [34]. Given an input feature vector, the predictor simply calculates the weighted sum of the input and compares it with a threshold value. If the sum surpasses the threshold, the result will be regarded as positive, otherwise the return value will be negative. During `update`, the prediction from the perceptron is compared to the actual outcome. If the prediction is correct, the weight will be increased. Otherwise, the weight will be decreased as a penalty.

The hash-based perceptron predictor has been proven to be highly versatile yet can be both executed and updated in very short order (in either software or hardware) [7, 55, 68]. While more sophisticated predictor designs are possible to consider with hardware support in the future, we prioritize low latency software implementations in this work.

Currently, PSS is designed to support up to 16 features with 1024 entries for each feature. The feature data is hashed to reduce the chance of conflict with other features and stored in a weight matrix. Once the predicted result is obtained, it can be compared with a threshold to generate binary decisions. If the value surpasses the threshold, the prediction will be regarded as true, otherwise the prediction is considered false.

3.2.1 *Predictor Model Extensibility.*

Since the system interface is not tied to the implementation, the underlying predictor model can be replaced easily if the users have specific needs. When low latency is preferred, other relatively simple models can be used, such as decision trees [52], linear regression [23], and naive Bayes algorithm [77]. On the other hand, if accuracy is prioritized more complicated model can be deployed, including XGBoost [12], k-nearest neighbors (KNN) [22], and neural networks [32].

3.2.2 *Parameter Types.*

The API described above mainly focuses on numeric parameters. But PSS can accept categorical parameter types after some preprocessing or transformation. For example, if those categories exist in some sort of embedded space then they can be exposed to a predictor through hierarchy or projection.

3.3 Reduced Latency Predictions with vDSO

In modern Operating Systems (OS), a user-level application cannot touch the kernel's memory space directly for a whole variety of reasons. Instead, interactions with the kernel are typically supported by system calls (syscall) — unfortunately, syscalls carry with them a significant amount of context switch overhead which conflicts directly with our stated goal of achieving low latency.

Fortunately, we are not the first to grapple with such a limitation and there are now multiple different mechanisms to build from, most notably virtual system calls (vsyscall) and virtual dynamic shared objects (vDSO). A vDSO is a Linux kernel mechanism that allows a portion of kernel memory space to be accessible in user space via a small shared library. The system presents to user space a map to the corresponding kernel data and programs such that it can access that memory directly. This facilitated direct read-out means there is no context switch involved in satisfying a vDSO read request which, in turn, leads to significant speedups [27]. In our experiments, this reduces the latency by more than a factor of 16x (from 68ns with syscall down to 4.19 ns) and, even more importantly, translates to real and noticeable improvements in application runtime.

Of course, vDSOs have their own limitations. By definition, it can be only used in a read-only manner since user mode cannot modify the kernel memory without a syscall and we must provide data as part of the update process.

Therefore, we design PSS in a way that combines a mix of syscalls and vDSOs. Specifically, we implement `predict` via vDSO since no writing to kernel data is involved. For `update`, we choose a syscall as the means to modify PSS model within kernel space. To further reduce the syscall overhead from `update` calls, we adopt a batch update mechanism that pools together multiple `update` calls into a single system call. A local buffer aggregates updates and allows us to amortize the boundary crossing.

Advantages of a System Service. One of the most interesting aspects of a system-service approach to prediction is that learning can happen across application invocations, a feature we demonstrate in application studies. While this is technically possible in application space, it requires the system to save and restore application-level files which is a poor match for the model of increasingly short-lived processes called in reaction to dynamic events. A system library has the additional advantage of being able to be used across kernel-space applications. Lastly, by utilising a vDSO that connects to kernel space, system policy can be enforced around the use of PSS, for example, to restrict which users or which programs can use the service and how information is shared across those programs.

4 USE-CASE SCENARIOS

To demonstrate the usefulness of the services described above, in this section, we present the application of PSS in three different scenarios chosen to demonstrate the generality of the service.

```
1 void TxLock(mutex *m) {
2   int * features = {perf_cnt, remain_retry} // 2 features
3   if (predict(features, /*len=*/ 2) == USE_HTM) {
4     tryingHTM = true
5     while(m->isLocked()) ; // spin
6     slowPath = false;
7     for (int i := 0; i < MAX_RETRIES; i++) {
8       if (tx_begin() == SUCCESS) {
9         if (m->isLocked()) {
10          tx_abort(); //abort
11        }
12        // transaction started
13        return;
14      }
15    }
16  } else {
17    tryingHTM = false
18  }
19  slowPath = true;
20  m->lock(); // slow path
21 }
22 void TxUnlock(mutex *m) {
23   int * features = {perf_cnt, remain_retry} // 2 features
24   if (!slowPath) {
25     tx_commit();
26     update(features,/*len*/2, /*reward*/+1)
27   } else {
28     m->unlock();
29     if (tryingHTM)
30       update(features,/*len*/2, /*reward*/-1)
31   }
32 }
```

Listing 1: Hardware Lock Elision with PSS.

4.1 PSS in Hardware Lock Elision

Synchronizing accesses to shared variables is a critical performance limiter in shared-memory multicore systems. While locks are one of the most frequently used mechanisms to safely manage sharing among many threads, locking is an inherently pessimistic method of synchronization where execution is potentially serialized and locking and unlocking costs are paid whether or not concurrent executions conflict in accessing data. In contrast, Transactional Memory [18, 19, 31, 64, 67] (TM) allows threads to execute through a set of guarded transactions optimistically and relies instead on the run-time detection of conflicts with an accompanying roll back when serialization is determined to be required. If one wishes to keep to the semantics of critical sections assumed by locks, TM can still be useful in allowing the system to speculatively execute through lock-protected critical sections through a class of techniques known as Hardware Lock Elision [36, 57, 79].

Of course, there is a balance to be struck between optimism and pessimism. Each lock under different use scenarios may benefit from a different approach and it is not straightforward to achieve good performance in practice due to the high costs of both rollback and of overly pessimistic locking.

Listing 1 presents a typical method for eliding locks using HTM. The original code is shown with a white background color and the additions we made to patch with PSS are highlighted in a gray background. There are two functions at the heart of the eliding lock implementation: `TxLock`, which is called at the beginning of a critical section, and `TxUnlock`, which is called at the end of the critical section. The input to both functions is a mutex object with lock/unlock/isLocked operations on it that can potentially be replaced by HTM. For interoperability with the lock, the HTM path is not tried until the lock is held (Line 5).

The transaction starts at Line 8 and the lock status is checked again to ensure it was not taken in the meantime by another thread and an explicit abort (Line 10) is issued if that is the case. The successful start of the transaction (`tx_begin()`) will result in a return from the function and will allow execution to continue into the critical section. Any failure due to conflict, capacity, explicit abort, or unsupported instruction, will cause the `tx_begin()` to return a non-success return code. On failure, retries are made a fixed number of times after which the algorithm falls back to the slow path of taking the underlying lock at Line 19. A special flag `slowPath` is set to indicate the corresponding action at the end of the critical section. This design performs well when most transactions succeed. However, in reality, it may not be known ahead of time whether lock-elision for a critical section is beneficial. For various reasons, the transactions can fail: notably due to increased contention and increased conflicts, due to increased memory footprint that may not fit within the HTM implementation's capacity, or due to the execution path using unsupported instructions.

Listing 1 shows the minor modifications (gray background) to the baseline that are required to enable PSS to guide the HTM vs. lock decision at runtime. At a high level, the idea is to utilize HTM if it is likely to succeed and rollback to lock if the transaction is likely to fail. Instead of having a fixed trial number mechanism, PSS allows the system to easily make the lock/HTM decisions at runtime. The output of the prediction directly informs the path through the code taken. In order to make reasonable predictions from the PSS, we use two parameters. The first is a thread-level performance counter from past transactions. We use an integer to store the past performance and each bit represents one transaction attempt. A value of '1' means the transaction finished successfully whereas '0' suggests the transaction failed. The second parameter is the number of retries left before hitting the maximum retry number (*MAX_RETRIES*).

The same two features will be used for the calls to `predict` and `update`. The first argument to `predict` is this feature vector and the second argument is the feature length (2). If the result of `predict` is above the threshold (`USE_HTM`), the program attempts the HTM path, otherwise, it falls back to using the underlying lock without trying the HTM. The feedback to the prediction is given after the critical section ends, in the `TxUnlock()` function. If the perceptron recommended taking the HTM path (`tryingHTM` is true), then a successful fastpath rewards the perceptron by invoking the `update` API function with +1 (Line 9); however, if the perceptron recommended the HTM path but the HTM failed, we penalize it with a negative reward of -1 (Line 30). To avoid the perceptron becoming trapped in only the lock path after several failed predictions, a predetermined threshold is also set.

4.2 Page Reclaim and Congestion Wait

When memory gets tight, the operating system memory management subsystem starts to reclaim already used pages for later use. During the reclaim process, pages with modified contents need to be written out before the reclaim can occur. However, if the devices that the pages will be written out are already congested with other traffic, there is very limited benefit to adding extra I/O requests.

To mitigate congestion problems in the Linux kernel, a tracking mechanism for block devices was proposed in 2002 [48] which was adopted in v2.5.39 [47]. If the devices are congested, the memory management sub-system would not create any new I/O requests before the congestion is resolved. This idea has been extended in various ways and such a mechanism still exists in Linux kernel 5.15 as `congestion_wait()` [46].

Unfortunately, over the years developers have found that there are several limitations to the congestion-wait mechanism. First, congestion tracking suffers from an inherent race condition as the degree of device congestion can change before the query returns to the caller. Second, accurate tracking of congestion has become more difficult as storage devices have come to support longer command queues. As a result, *congestion_wait()* is used in practice only when the timeout expires, which is not at all what it was originally intended to do [51].

To overcome the limitation mentioned above, in 2021 it was proposed that all instances of congestion-wait in the source code should be completely eliminated [49]. The proposed new design reclassifies the original congestion wait into three sub categories and handles each one differently:

- When there are too many dirty or writeback pages, sleep until enough pages are cleaned or a timeout expires
- When there are too many isolated pages, sleep until enough of them are put back into the LRU system or reclaimed
- When there is no progress in page reclaim, the direct reclaim task sleeps until another reclaim task proceeds with some acceptable efficiency

The third point specifically measures the efficiency of another reclaim task by dividing the number of pages reclaimed by the number of pages scanned: $\frac{nr\_reclaimed}{nr\_scanned}$. In the most recent patch, the efficiency threshold is set at a fixed value of 12.5%. However, as the proposer of this technique rightly points out, the fixed threshold value may not work for all scenarios. Here we see yet another opportunity to apply PSS to optimize control of the system, in this case dynamically optimizing the sleep condition instead of relying on a fixed ratio.

We input several parameters into PSS and dynamically decide if the current reclaim task should sleep or not. The parameters include the rounded values of nr_reclaimed and nr_scanned as well as the ratio of $\frac{nr\_reclaimed}{nr\_scanned}$. Since PSS only takes integer inputs currently, we use the reciprocal of the ratio and rounded to the closest integer, i.e. $floor(\frac{nr\_scanned}{nr\_reclaimed})$ If the returned result is greater than or equal to 0, the task will not go to sleep.

For this use case `update` is not as straightforward as `predict` — how does one know that the prediction was "wrong"? While we don't have direct access to ground truth, we can instead assume that entering the page claim throttle function is a negative sign for the last decision since overall page reclaim is a procedure that we want to minimize. Therefore, we keep a timer via *ktime_get()* to measure the timestamp of the last entrance of the function and the duration between two entrances. If the duration becomes longer, it means that the page reclaim has been invoked less frequently and we will reward the weights that lead to such a decision. Otherwise, it suggests that the reclaim happens more often and we will penalize the weights accordingly. In the end, even though we are inferring prediction and misprediction indirectly, we are able to limit the scope of our code changes to only the original function, *consider_reclaim_throttle*.

Table 1: List of selected PyPy JIT parameters.

parameters	Default	Descriptions
decay	40	amount to regularly decay counters by
function_threshold	1619	number of times a function must run for it to become traced from start
loop_longevity	1000	a parameter controlling how long loops will be kept before being freed
threshold	1039	number of times a loop has to run for it to become hot
trace_eagerness	200	number of times a guard has to fail before we start compiling a bridge
trace_limit	6000	number of recorded operations before we abort tracing with ABORT_TOO_LONG

4.3 JIT Parameter Tuning for PyPy

Python is one of the most popular languages because of its simplified syntax and dynamic features. However, the default interpreter implementation (CPython) suffers from slow execution speed brought by the extra interpreter layer. To recover some of that performance, Just-In-Time (JIT) compilation can be used to translate frequently executed code snippets into machine code that can be executed directly. PyPy is one of the most popular tools for doing so, in part because of its efficient tracing-based JIT compiler [8].

Unlike a method-based JIT that compiles an entire method at a time, a trace-based JIT only considers the frequently executed code paths (a.k.a., "hot path") within a method.

In PyPy specifically, there is a critical parameter named *threshold*, which decides whether a loop is *hot* or not. The default value of *threshold* happens to be 1039, meaning a loop will trigger the JIT tracing mechanism on the code path only after the loop has been executed 1039 times.

Like *threshold*, there are 16 other parameters in PyPy that control the tracing and compilation mechanism [61]. We detail the subset of the parameters we utilise and their default values in Table 1.

While it is well understood that these parameters are critical, most prior work seeks to find an single static set that strikes that balance. One of the most commonly used methods of achieving such tuning is genetic algorithm (GA). For instance, Yu et al. [80] use GAs to optimize parameters for Spark while Li and Jiang [44] show that GAs can find PyPy parameters that can significantly outperform the default JIT parameters. However, GA and other static parameter tuning approaches require both expensive upfront overhead and a set of "representative" programs while to training. A large amount of data, machine and load dependence of results, and the significant design space exploration time required to make improvements all potentially limit the applicability of such an approach.

Using PSS, we can tune the JIT parameters on-the-fly without additional data collection and model training cost. Inspired by Li and Jiang [44], we choose the parameters within a set of prefixed values. The default value is multiplied by $\frac{1}{4}$, $\frac{1}{2}$, 2, and 4 to get the 4 new settings. The only exception is trace_limit of $4X$, which is set to 16000 instead of 24000 because of a range limit.

Listing 2 sketches the use of PSS in the PyPy JIT. After each iteration, we record the performance including the number of instructions and the execution time. More counter information can also be used if available, such as branch prediction and cache performance. We then feed this information into the perceptron as input features. The perceptron returns the decision on whether more aggressive optimization should be used or not and the JIT configurations will be set accordingly.

```
def main():
    features = {performance counters}
    for i in range(iteration):
        run the workload
        if predict(features, len) == True:
            set more aggressive JIT parameter
        else:
            set more conservative JIT parameters
        if curItrTime < prevItrTime:
            update(features, +1)
        else:
            update(features, -1)
```

Listing 2: Integration of PSS with PyPy JIT

Once the timing information is collected with the new parameters, we compare it with the duration from the previous iteration. If the new parameters speed up the execution, the corresponding weight will be increased; otherwise it will be decreased.

The input feature for the use case includes detailed information from PAPI [69] like the number of instructions and potentially different cache levels' hit rates. To better utilize them, we round the raw values before passing them to the perceptron. The rounding keeps only the most significant figures of a given integer. For example, 1234 will be rounded to 1000, 6276 will be rounded to 6000, and 1999 will be rounded to 2000. Rounding allows the perceptron to learn common input and prediction patterns.

5 EVALUATION

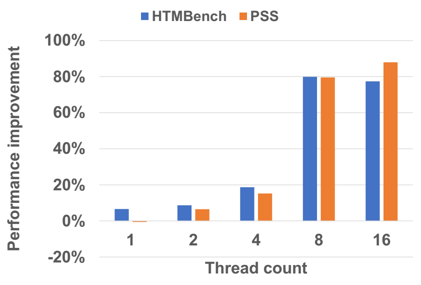
We evaluate PSS on an 8-core (×2-way SMT [71]) Intel Coffee Lake CPU with a total 32GB memory, running Linux `5.15.0`. The CPU has 32KB L1I and L1D cache, 256KB L2 cache, and 16MB L3 cache.

For each applications in the three examples, we use STAMP/HTMBench [54, 74] as the HTM workload, MMTests for page reclaim benchmarks [28], and PolyBenchPython [5] and python-macrobenchmarks [3] as the PyPy JIT benchmarks.

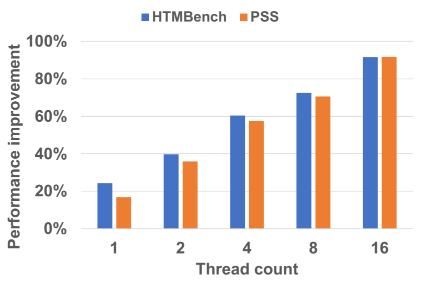
In this section, we define the word *iteration* to describe the number of a dividable subroutine internally repeated in a benchmark program. And we use *benchmark run* to refer to one whole run of a benchmark.

5.1 Hardware Lock Elision Results

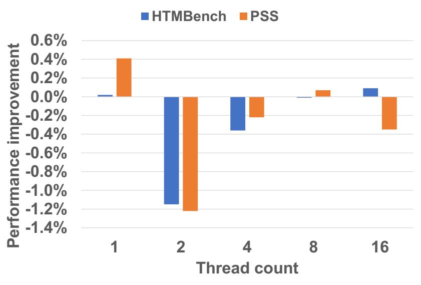
We choose Stanford Transactional Applications for Multi-Processors (STAMP) [54] as the workload for HLE. STAMP is a collection of applications targeted for transactional memory research. The description of the benchmark programs can be found in Table 2. We use the recommended parameters for simulation setup.



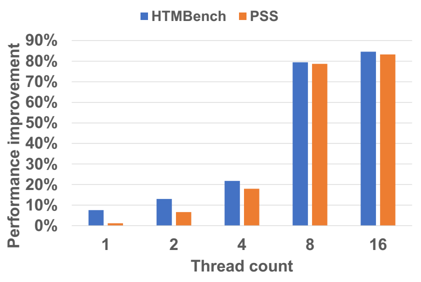
(a) genome



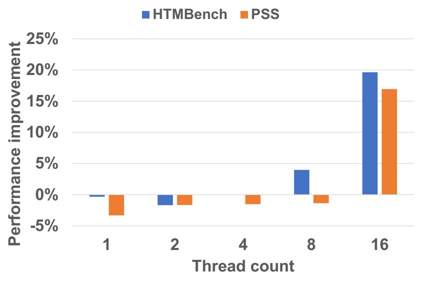
(b) SSCA2



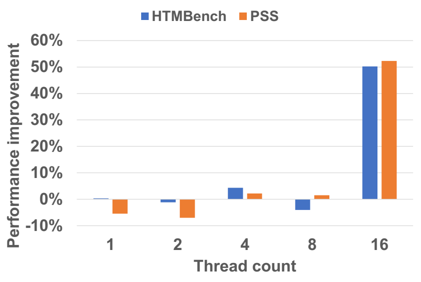
(c) labyrinth



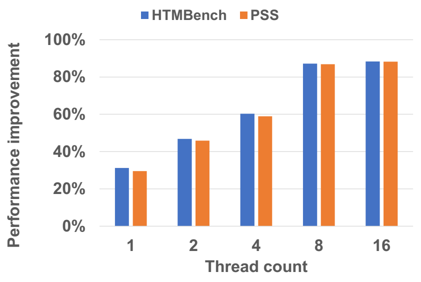
(d) intruder



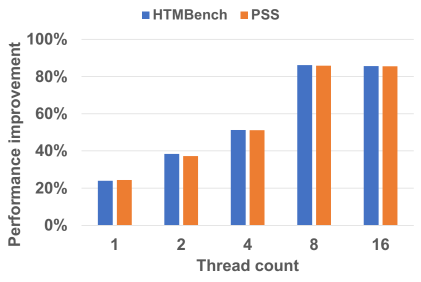
(e) kmeans-low



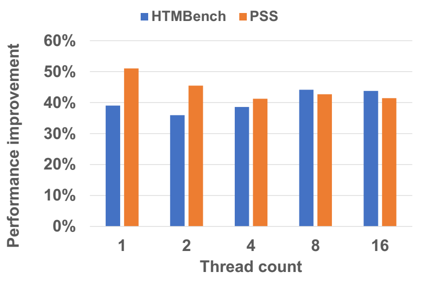
(f) kmeans-high



(g) vacation-low



(h) vacation-high



(i) yada

Figure 2: Performance of HTMBench and PSS HLE normalised to vanilla STAMP.

Table 2: Benchmarks used from STAMP/HTMBench.

Benchmark	Description
intruder	Network intrusion detection
labyrinth	Maze routing
yada	Delaunay mesh refinement
SSCA2	Graph kernel
vacation	Travel reservation system
kmeans	K-means clustering
genome	Gene sequencing

HTMBench is the state-of-the-art benchmark suite of HTM and it is implemented using Intel’s TSX [33]. It provides an efficient profiler to analyze HTM and offers optimizations that generate nontrivial speedups. We compare our PSS implementation against HTMBench [74] and vanilla STAMP with HTM support as the baselines. We vary core count over 1, 2, 4, 8, and 16 cores. We run each program five times and report the median value of the results.

The result of STAMP is plotted in Fig 2, which shows the performance improvement of HTMBench and PSS over vanilla STAMP. Overall, the overhead of using PSS is relatively low. The most slowdown comes from 1 thread setting for kmeans-high in Fig. 2f, where PSS optimized code generates 7.02% performance degradation. In most of the other cases, the slowdown is less than 5%. On the other hand, PSS optimized code can clearly show benefits over the vanilla baseline or even HTMBench in selected cases like Fig. 2a and 2i. For instance, PSS leads to 87.62% of improvement for 16 threads setup in genome, which is 11% higher than HTMBench.

In terms of overhead, HTMBench has state-of-the-art implementations of STAMP after extensive profiling and optimization. On the other hand, baseline code patched with PSS is only trained a few

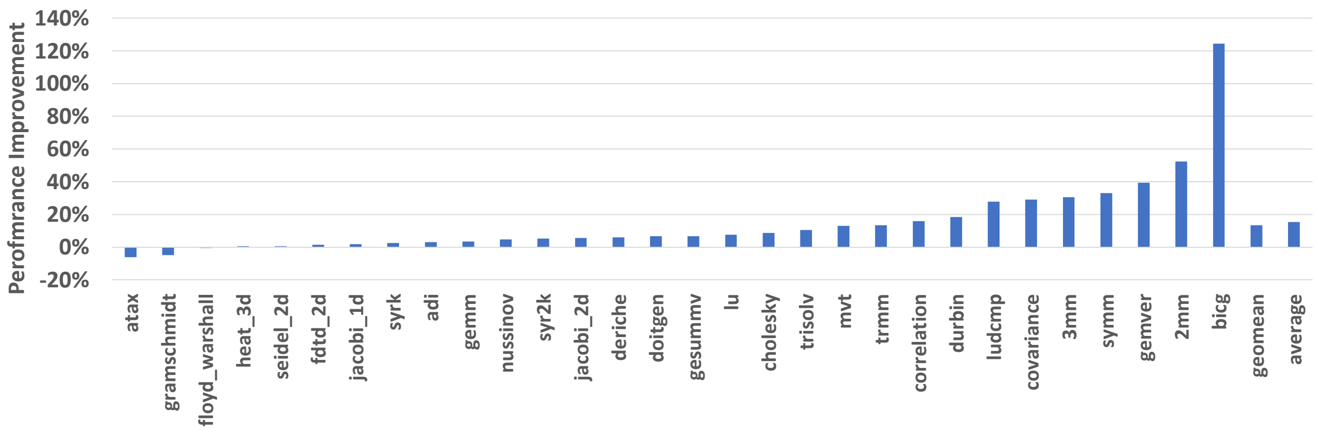


Figure 3: Performance improvement of PSS with 20 iterations on PolyBenchPython

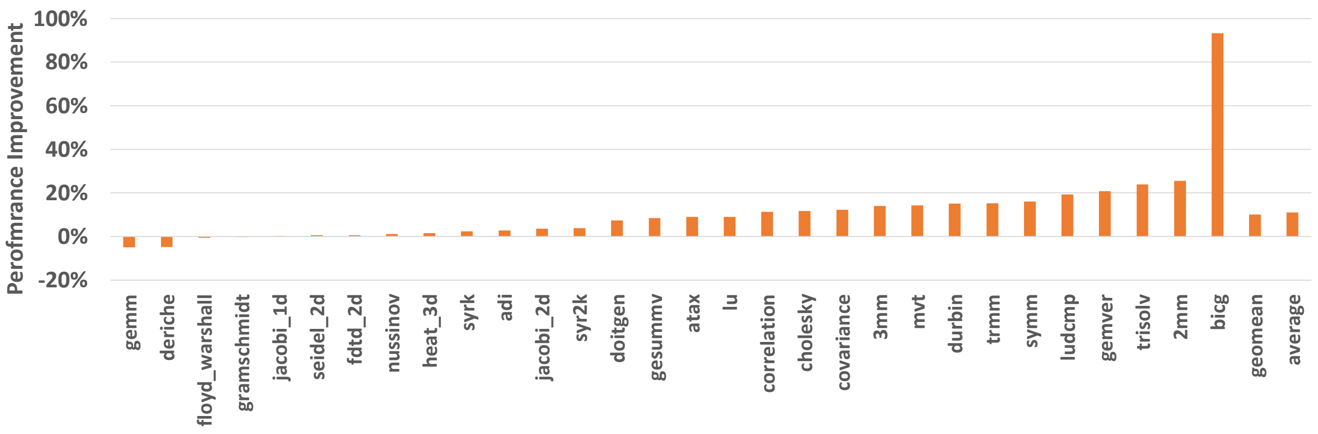


Figure 4: Performance improvement of PSS with 50 iterations on PolyBenchPython

hours and it performs very close to HTMBench or even outperforms it in several cases.

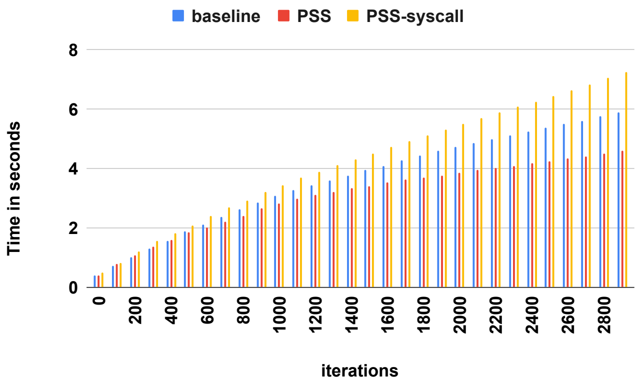
5.2 PyPy JIT Results

5.2.1 Benchmark Setup. We use version 7.3.3 of PyPy as the JIT compiler and PolyBenchPython [5] with python-macrobenchmarks [3] as the workloads. PolyBenchPython is a benchmark suite with 30 commonly used kernels for scientific computing and it is representative as microbenchmarks. On the other hand, python-macrobenchmarks contains some of the most popular python applications on a macro-level, including Flask [4], Django content management system (CMS) [1], Gunicorn [2] and more.

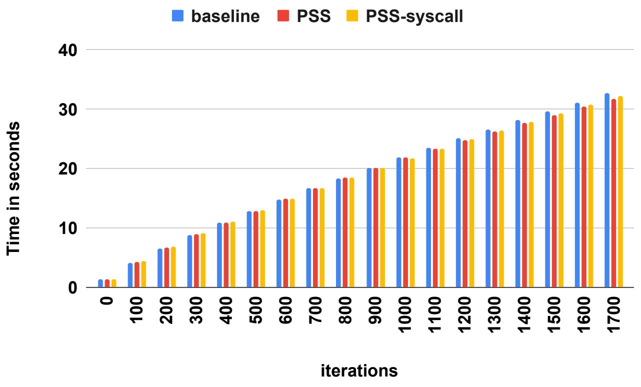
For PolyBenchPython, we run the benchmark using the default list implementation of the array and MINI as the input data size. Since the original PolyBenchPython already uses PAPI [76] counters, we include some of them as input features to PSS. Specifically, we use the execution time and the ratio between L1D hit and L1D miss as parameters for PSS. Each benchmark is executed 10 times and we report the time spent in the first 20 and 50 iterations. The baseline is the program with the default JIT setting while the modified JIT is the program patched with PSS, dynamically changing the JIT configuration parameters as we described in Section 4.3.

5.2.2 PolyBenchPython Results. The result of PyPy JIT parameter tuning is presented in Figure 3 and 4. On average, PSS can improve the performance of the 30 programs by 15.38% and 11.11% for 20 and 50 iterations, respectively. For the first 20 iterations, the largest improvement is over 120% while the largest slowdown is only around 6%. For 50 iterations, the largest performance gain and loss are smaller since most of the commonly executed code is already jitted in the late iterations. However, the improvement is still significantly larger than the slowdown. We believe this setup of optimization can be potentially useful for Function-as-a-Service (FaaS) applications, which tend to run short computation tasks over and over.

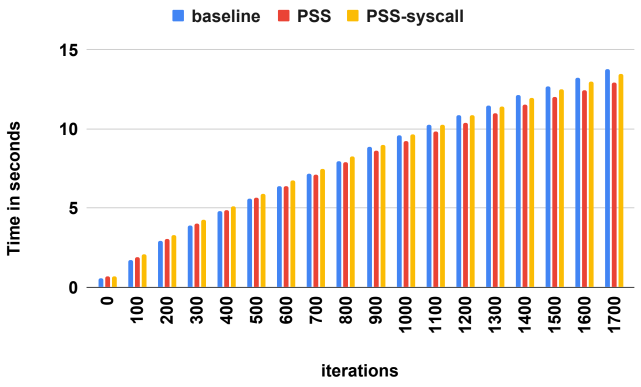
5.2.3 Macrobenchmark Result. The result of the macro benchmark is plotted in Fig 5. We choose 4 benchmarks that can easily demonstrate performance iteration-wise and we simply use the iteration-wise runtime as the parameter for PSS. We run 3000 iterations



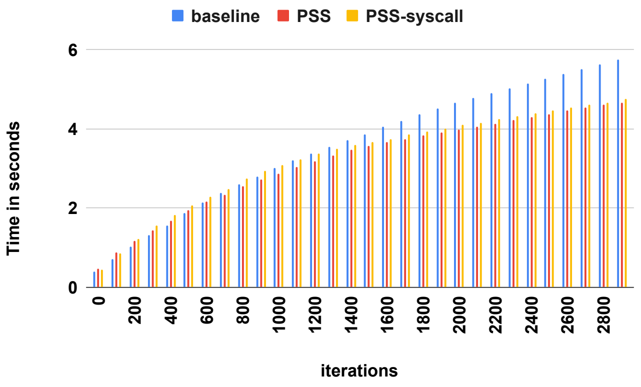
(a) aiohttp



(b) djangocms



(c) flaskblogging



(d) gunicorn

Figure 5: Result of macrobenchmarks.

for aiohttp and gunicorn and 1800 iterations for djangocms and flaskblogging. Each benchmark runs 5 times and we plot the averaged result iteration-wise.

It is clear that PSS can speed up the macro-benchmark with better dynamic parameter tuning. For the four benchmarks, the performance improvements are 22.17%, 2.54%, 6.3%, and 18.66%, respectively. From the two sets of benchmarks, we demonstrate the functionality and performance benefits of PSS for both micro and macro-benchmarks of Python.

5.2.4 Latency-Sensitive Applications. Figure 5 also contains the result of using syscall as prediction instead of vDSO. From the results, it is clear that for the latency sensitive applications, implementation using vDSO performs better than syscall. The syscall-based results either have less speedup as shown in Figures 5b, 5c, and 5d) or generate significant slowdown as shown in Figure 5a.

5.3 Page Reclaim and Memory Management

5.3.1 Benchmarks and Methodology. We follow the experiments mentioned in the original patch [50]. We ran mmtests [28] on the original 5.15.0-rc3 kernel, the patched kernel [29], and the kernel with dynamic control from PSS. MMtests is a benchmark framework aimed at performance testing of the Linux kernel. Specifically, we ran a test named `stutterp`, which sweeps a different number of "worker" processes and inspects the impact of the direct reclaim. There are four types of workers in `stutterp`:

- One "anon latency" worker: creates `mmap` mappings then measures the duration to fault the mapping.
- X file writers: flexible I/O tester (`fio`) that randomly writes X files. The total size of the files equals the preset *dirty_ratio*.
- Y file readers: `fio` that randomly reads small files.
- Z anon memory hogs: continually map memory with the ratio $(100 - dirty\_ratio)\%$.

The total estimated working set size (WSS) is $(100 + dirty\_ration)\%$ of memory. The motivation of `stutterp` is to maximise the total WSS with file and anonymous memory. During execution, some anonymous memory has to be swapped and it is very likely that dirty/writeback pages reach the end of the LRU.

5.3.2 Results. The result of `stutterp` is plotted in Fig. 6. It shows the performance improvement compared with 5.15.0 vanilla kernel. The number after `mmap` indicates the number of the worker threads

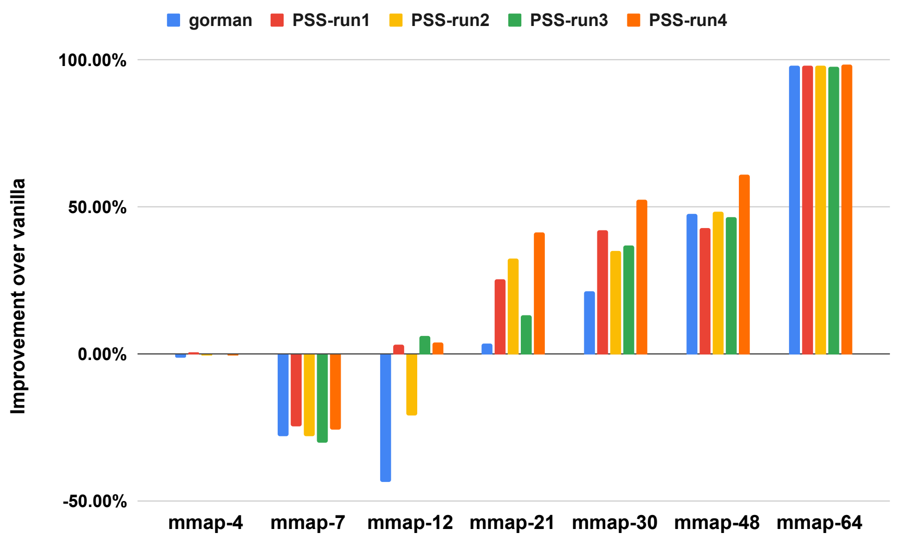


Figure 6: Average latency of MMTests. Each mmap-N represents a run with N worker threads. The larger the number, the higher the memory pressure.

mentioned above and larger worker counts means higher memory pressure for the system.

From the plot it is clear that PSS can outperform the baseline implementation now merged into the kernel. The improvement is much higher for the 21, 30, and 48 workers setups and PSS achieved slight improvement where the baseline suffers significant performance loss for 12 workers. For 7 workers, all the implementations perform worse than the vanilla code, but the slowdown is less for PSS code after several iterations.

Another benefit we can observe from the figure is that the performance of PSS is improving over multiple benchmark runs. It does not show a monotonic increase, but shows improvement as the general trend over time. On the other hand, we tried to run the baseline version multiple times and we did not observe any noticeable improvement.

6 RELATED WORK

Since prediction is a key feature of the system software stack, there have been many different implementations which have been demonstrated to take advantage of common system operations and communication patterns for prediction to improve performance and other system metrics. Kraska et. al. [39] and Mitzenmacher et. al. [56] survey recent work which make use of prediction and machine learning for systems.

The most relevant work to our proposed system service for prediction is SmartChoices [9]. Similar to PSS, it also proposes a set of interface functions that software can use to make predictions, as well as ability to do on-the-fly learning. However, their proposed system is based on Reinforcement Learning which requires significantly more resources for training and incurs higher latency compared to a simple perceptron-based predictor. Thus, it has limited applicability in resource-limited systems which only require simple and fast predictions.

Other than the work mentioned on prediction memory access and synchronization mentioned in Section 2, there have been several other synchronization algorithms which have a fastpath/slowpath or other variants [11, 17, 42, 45, 78, 81, 82] and the decision to dynamically choose the correct variant is predicted based on the past behavior and current conditions.

Low-level runtime systems for dynamically adjusting to power and energy consumption employ lightweight prediction mechanisms [6, 26, 53, 63]. Esmaeilzadeh et al. present a learning-based technique to accelerate approximate programs [21]. In their work, programmers can label a code region to approximate and then a NN model will be trained to emulate the region. Once the training is complete, the original code region will be replaced by the invocations to a low-power Neural Processing Unit via an ISA extension. Furthermore, system failure prediction [24, 25, 30, 60, 65, 75] has also attracted a lot of attention in recent times due to very large scale systems and increased failure rate. Finally, searching for the set of compiler optimizations and their order of application employs various prediction techniques based on past learnings and behaviors [10, 15].

There are many studies focused on how to automatically tune the configuration settings for different kinds of software systems. In general, those studies can be classified into two groups. The first group utilizes a certain type of search-based algorithm, including hill-climbing [73], genetic algorithms [80], and ParamILS [43]. The second group tries to find the optimal configurations by reduction, including the iterative experiment [70] and similarity measurement [59].

Our proposed system service prediction is flexible enough to be used and bring performance benefits in most of these prediction scenarios and avoid the complications of parameter tuning. Compared to traditional approaches, our proposed service offers advantages in terms of lower effort and resources needed with on-the-fly tuning and better reusability.

7 CONCLUSION

The effective end of processor frequency scaling and the continued drive for higher performance and lower energy utilization means that application-targeted software optimization will only continue to grow in importance in the field. While there are sure to be many application specific optimizations that do not rely on prediction, a surprisingly diverse class of optimizations, from hot-path/cold-path, to parameter tuning, to resource optimization, and beyond are more easily and readily enabled through support from a simple to call and low-latency software service. The move to a new abstraction that is useful in the process of optimization helps us step away from both the fragile heuristics so common in production code today while avoiding inheriting the complexity of complete application-embedded prediction frameworks. A system service for prediction has the potential to enable performance optimizers to spend their time worrying more about the discovery of new opportunities for specialization and tuning, and less about how exactly one should navigate the space of trade offs such opportunities live in. Even if there are times when such a service might not be appropriate for a final deployment, a prediction service can still be helpful in the development process by speeding up the sorting of promising optimization opportunities from those that will offer little gain even with well-crafted heuristic control. A core idea of prediction as a service is the decoupling of the creation of optimizations and the specific decision of how and when exactly to apply those optimizations.

We demonstrate the utility of Prediction as a System Service across three application-targeted optimization scenarios, and in all three cases find performance improvements. As to be expected such an approach is highly latency sensitive, but we are able to demonstrate a creative new use of vDSOs that can allow applications to extract predictions in an average of 4.19ns. In all of the cases we examined this new optimization pattern with operating system service support is able to meet or beat the best known hand-crafted methods with a fraction of the complexity of existing hardware. We believe this approach would be of particular interest to the ASPLOS community as it also opens the door for new and creative uses of architectural support for assisting in prediction with low latency, language level opportunities for the exploitation of prediction services, and further innovation in the system-level abstractions appropriate to more fully support dynamic control of software optimization.

ACKNOWLEDGEMENT

This material is based upon work supported by the National Science Foundation under Grants No. 2006542 and 1763699. We thank the anonymous reviewers for their feedback. We also thank Milind Chabbi and Timothy Hayes for the discussion and assistance.

REFERENCES

[1] [n.d.]. django-cms/django-cms: The easy-to-use and developer-friendly enterprise CMS powered by Django. https://github.com/django-cms/django-cms. (Accessed on 06/17/2022).

[2] [n.d.]. Gunicorn - Python WSGI HTTP Server for UNIX. https://gunicorn.org/. (Accessed on 06/17/2022).

[3] [n.d.]. pyston/python-macrobenchmarks: A collection of macro benchmarks for the Python programming language. https://github.com/pyston/python-macrobenchmarks. (Accessed on 06/17/2022).

[4] [n.d.]. Welcome to Flask — Flask Documentation (2.1.x). https://flask.palletsprojects.com/en/2.1.x/. (Accessed on 06/17/2022).

[5] Miguel Á Abella-González, Pedro Carollo-Fernández, Louis-Noël Pouchet, Fabrice Rastello, and Gabriel Rodríguez. 2021. PolyBench/Python: benchmarking Python environments with polyhedral optimizations. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*. 59–70.

[6] Bradley J. Barnes, Barry Rountree, David K. Lowenthal, Jaxk Reeves, Bronis de Supinski, and Martin Schulz. 2008. A Regression-Based Approach to Scalability Prediction. In *Proceedings of the 22nd Annual International Conference on Supercomputing* (Island of Kos, Greece) *(ICS '08)*. Association for Computing Machinery, New York, NY, USA, 368–377. https://doi.org/10.1145/1375527.1375580

[7] Eshan Bhatia, Gino Chacon, Seth Pugsley, Elvira Teran, Paul V Gratz, and Daniel A Jiménez. 2019. Perceptron-based prefetch filtering. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1–13.

[8] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*. 18–25.

[9] Victor Carbune, Thierry Coppey, Alexander Daryin, Thomas Deselaers, Nikhil Sarda, and Jay Yagnik. 2019. SmartChoices: hybridizing programming and machine learning. *Reinforcement Learning for Real Life (RL4RealLife) Workshop in the 36th International Conference on Machine Learning (ICML)* (2019).

[10] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. P. O'Boyle, and O. Temam. 2007. Rapidly Selecting Good Compiler Optimizations using Performance Counters. In *International Symposium on Code Generation and Optimization (CGO'07)*. 185–197. https://doi.org/10.1109/CGO.2007.32

[11] Milind Chabbi and John Mellor-Crummey. 2016. Contention-Conscious, Locality-Preserving Locks. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Barcelona, Spain) *(PPoPP '16)*. Association for Computing Machinery, New York, NY, USA, Article 22, 14 pages. https://doi.org/10.1145/2851141.2851166

[12] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. ACM, New York, NY, USA, 785–794. https://doi.org/10.1145/2939672.2939785

[13] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 579–594.

[14] Xinshi Chen, Shuang Li, Hui Li, Shaohua Jiang, Yuan Qi, and Le Song. 2019. Generative adversarial user model for reinforcement learning based recommendation system. In *International Conference on Machine Learning*. PMLR, 1052–1061.

[15] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. 1999. Optimizing for Reduced Code Space Using Genetic Algorithms. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems* (Atlanta, Georgia, USA) *(LCTES '99)*. Association for Computing Machinery, New York, NY, USA, 1–9. https://doi.org/10.1145/314403.314414

[16] Redhat Corp. [n.d.]. Automatic NUMA Balancing. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/virtualization_tuning_and_optimization_guide/sect-virtualization_tuning_optimization_guide-numa-auto_numa_balancing.

[17] Dave Dice, Alex Kogan, Yossi Lev, Timothy Merrifield, and Mark Moir. 2014. Adaptive Integration of Hardware and Software Lock Elision Techniques. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures* (Prague, Czech Republic) *(SPAA '14)*. Association for Computing Machinery, New York, NY, USA, 188–197. https://doi.org/10.1145/2612669.2612696

[18] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. 2009. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*. 157–168.

[19] Dave Dice, Ori Shalev, and Nir Shavit. 2006. Transactional locking II. In *International Symposium on Distributed Computing*. Springer, 194–208.

[20] Dice Dave. 2015. waiting policies for locks : spin-then-park . https://blogs.oracle.com/dave/waiting-policies-for-locks-:-spin-then-park.

[21] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Neural acceleration for general-purpose approximate programs. In *2012 45th Annual*

IEEE/ACM International Symposium on Microarchitecture. IEEE, 449–460.

[22] Evelyn Fix and Joseph Lawson Hodges. 1989. Discriminatory analysis. Nonparametric discrimination: Consistency properties. *International Statistical Review/Revue Internationale de Statistique* 57, 3 (1989), 238–247.

[23] David A Freedman. 2009. *Statistical models: theory and practice*. cambridge university press.

[24] Song Fu and Cheng-Zhong Xu. 2007. Exploring event correlation for failure prediction in coalitions of clusters. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. 1–12.

[25] Ana Gainaru, Franck Cappello, Marc Snir, and William Kramer. 2013. Failure prediction for HPC systems and applications: Current situation and open issues. *The International journal of high performance computing applications* 27, 3 (2013), 273–282.

[26] Neha Gholkar, Frank Mueller, and Barry Rountree. 2019. Uncore Power Scavenger: A Runtime for Uncore Power Conservation on HPC Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 27, 23 pages. https://doi.org/10.1145/3295500.3356150

[27] GitHub. 2018. GitHub - nlynch-mentor/vdsotest: Utility for testing and benchmarking a Linux VDSO. https://github.com/nlynch-mentor/vdsotest. (Accessed on 04/08/2022).

[28] Mel Gorman. 2011. gormanm/mmtests: MMTests: Benchmarking framework primarily aimed at Linux kernel testing. https://github.com/gormanm/mmtests. (Accessed on 12/01/2021).

[29] Mel Gorman. 2021. kernel/git/mel/linux.git - Candidate patch series by Mel Gorman. https://git.kernel.org/pub/scm/linux/kernel/git/mel/linux.git/commit/?h=mm-reclaimcongest-v5r4&id=a2f8f6191574311e28d0c3609394937533ec490c. (Accessed on 12/01/2021).

[30] Saurabh Gupta, Tirthak Patel, Christian Engelmann, and Devesh Tiwari. 2017. Failures in large scale systems: long-term measurement, analysis, and implications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[31] Maurice Herlihy and J Eliot B Moss. 1993. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on Computer architecture*. 289–300.

[32] John J Hopfield. 1982. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences* 79, 8 (1982), 2554–2558.

[33] Intel. 2022. Intel 64 and IA-32 Architectures Optimization Reference Manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[34] Daniel A Jiménez and Calvin Lin. 2001. Dynamic branch prediction with perceptrons. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*. IEEE, 197–206.

[35] R. Karedla, J. S. Love, and B. G. Wherry. 1994. Caching strategies to improve disk system performance. *Computer* 27, 3 (1994), 38–46. https://doi.org/10.1109/2.268884

[36] Tomas Karnagel, Roman Dementiev, Ravi Rajwar, Konrad Lai, Thomas Legler, Benjamin Schlegel, and Wolfgang Lehner. 2014. Improving in-memory database index performance with Intel® Transactional Synchronization Extensions. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 476–487.

[37] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2017. Scalable NUMA-aware Blocking Synchronization Primitives. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 603–615. https://www.usenix.org/conference/atc17/technical-sessions/presentation/kashyap

[38] Tracy Kimbrel, Andrew Tomkins, R Hugo Patterson, Brian Bershad, Pei Cao, Edward W Felten, Garth A Gibson, Anna R Karlin, and Kai Li. 1996. A trace-driven comparison of algorithms for parallel prefetching and caching. In *OSDI*. 19–34.

[39] Tim Kraska. 2021. Towards Instance-Optimized Data Systems. *Proc. VLDB Endow.* 14, 12 (jul 2021), 3222–3232. https://doi.org/10.14778/3476311.3476392

[40] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Jialin Ding, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. 2019. SageDB: A Learned Database System.

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

[42] Leslie Lamport. 1987. A Fast Mutual Exclusion Algorithm. *ACM Trans. Comput. Syst.* 5, 1 (Jan. 1987), 1–11. https://doi.org/10.1145/7351.7352

[43] Philipp Lengauer and Hanspeter Mössenböck. 2014. The taming of the shrew: increasing performance by automatic parameter tuning for java garbage collectors. In *Proceedings of the 5th ACM/SPEC international conference on Performance engineering*. 111–122.

[44] Yangguang Li and Zhen Ming Jack Jiang. 2019. Assessing and optimizing the performance impact of the just-in-time configuration parameters-a case study on PyPy. *Empirical Software Engineering* 24, 4 (2019), 2323–2363.

[45] Beng-Hong Lim and Anant Agarwal. 1994. Reactive Synchronization Algorithms for Multiprocessors. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS VI)*. Association for Computing Machinery, New York, NY, USA, 25–35. https://doi.org/10.1145/195473.195490

[46] Linux. 2021. backing-dev.c - mm/backing-dev.c - Linux source code (v5.15-rc6) - Bootlin. https://elixir.bootlin.com/linux/v5.15-rc6/source/mm/backing-dev.c. (Accessed on 04/05/2022).

[47] LWN. 2002. Development kernel 2.5.39 released [LWN.net]. https://lwn.net/Articles/11130/. (Accessed on 04/05/2022).

[48] LWN. 2002. infrastruture for monitoring request queue congestion [LWN.net]. https://lwn.net/Articles/9519/. (Accessed on 04/05/2022).

[49] LWN. 2021. [PATCH v5 0/8] Remove dependency on congestion_wait in mm/ [LWN.net]. https://lwn.net/ml/linux-kernel/20211022144651.19914-1-mgorman@techsingularity.net/. (Accessed on 04/05/2022).

[50] LWN. 2021. [PATCH v5 0/8] Remove dependency on congestion_wait in mm/ [LWN.net]. https://lwn.net/ml/linux-kernel/20211022144651.19914-1-mgorman@techsingularity.net/. (Accessed on 12/01/2021).

[51] LWN. 2021. Replacing congestion_wait() [LWN.net]. https://lwn.net/Articles/873672/. (Accessed on 04/05/2022).

[52] Oded Z Maimon and Lior Rokach. 2014. *Data mining with decision trees: theory and applications*. Vol. 81. World scientific.

[53] Aniruddha Marathe, Peter E. Bailey, David K. Lowenthal, Barry Rountree, Martin Schulz, and Bronis R. de Supinski. 2015. A Run-Time System for Power-Constrained HPC Applications. In *High Performance Computing*, Julian M. Kunkel and Thomas Ludwig (Eds.). Springer International Publishing, Cham, 394–408.

[54] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. 2008. STAMP: Stanford transactional applications for multi-processing. In *2008 IEEE International Symposium on Workload Characterization*. IEEE, 35–46.

[55] Samira Mirbagher-Ajorpaz, Gilles Pokam, Esmaeil Mohammadian-Koruyeh, Elba Garza, Nael Abu-Ghazaleh, and Daniel A Jiménez. 2020. PerSpectron: Detecting Invariant Footprints of Microarchitectural Attacks with Perceptron. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1124–1137.

[56] Michael Mitzenmacher and Sergei Vassilvitskii. 2021. *Algorithms with Predictions*. Cambridge University Press, 646–662. https://doi.org/10.1017/9781108637435.037

[57] Takuya Nakaike, Rei Odaira, Matthew Gaudet, Maged M Michael, and Hisanobu Tomari. 2015. Quantitative comparison of hardware transactional memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8. *ACM SIGARCH Computer Architecture News* 43, 3S (2015), 144–157.

[58] Duc Tam Nguyen, Zhongyu Lou, Michael Klar, and Thomas Brox. 2019. Anomaly detection with multiple-hypotheses predictions. In *International Conference on Machine Learning*. PMLR, 4800–4809.

[59] Takayuki Osogami and Sei Kato. 2007. Optimizing system configurations quickly by guessing at the performance. In *Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. 145–156.

[60] Teerat Pitakrat, Dušan Okanović, André van Hoorn, and Lars Grunske. 2018. Hora: Architecture-aware online failure prediction. *Journal of Systems and Software* 137 (2018), 669–685.

[61] PyPy. 2021. JIT help — PyPy documentation. https://doc.pypy.org/en/latest/jit_help.html. (Accessed on 04/05/2022).

[62] Ravi Rajwar and James R. Goodman. 2001. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution *(MICRO 34)*. IEEE Computer Society, USA, 294–305.

[63] Barry Rountree, David K. Lowenthal, Bronis R. de Supinski, Martin Schulz, Vincent W. Freeh, and Tyler Bletsch. 2009. Adagio: Making DVS Practical for Complex HPC Applications. In *Proceedings of the 23rd International Conference on Supercomputing* (Yorktown Heights, NY, USA) *(ICS '09)*. Association for Computing Machinery, New York, NY, USA, 460–469. https://doi.org/10.1145/1542275.1542340

[64] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L Hudson, Chi Cao Minh, and Benjamin Hertzberg. 2006. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. 187–197.

[65] Felix Salfner, Maren Lenk, and Miroslaw Malek. 2010. A survey of online failure prediction methods. *ACM Computing Surveys (CSUR)* 42, 3 (2010), 1–42.

[66] William N. Scherer and Michael L. Scott. 2005. Advanced Contention Management for Dynamic Software Transactional Memory *(PODC '05)*. Association for Computing Machinery, New York, NY, USA, 240–248. https://doi.org/10.1145/1073814.1073861

[67] Michael F Spear, Maged M Michael, and Christoph Von Praun. 2008. RingSTM: scalable transactions with a single atomic instruction. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*. 275–284.

[68] Elvira Teran, Zhe Wang, and Daniel A Jiménez. 2016. Perceptron learning for reuse prediction. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–12.

[69] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. 2010. Collecting performance data with PAPI-C. In *Tools for High Performance Computing 2009*. Springer, 157–173.

[70] Risi Thonangi, Vamsidhar Thummala, and Shivnath Babu. 2008. Finding good configurations in high-dimensional spaces: Doing more with less. In *2008 IEEE international symposium on modeling, analysis and simulation of computers and telecommunication systems*. IEEE, 1–10.

[71] D. M. Tullsen, S. J. Eggers, and H. M. Levy. 1995. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings 22nd Annual International Symposium on Computer Architecture*. 392–403.

[72] Takayuki Usui, Reimer Behrends, Jacob Evans, and Yannis Smaragdakis. 2009. Adaptive Locks: Combining Transactions and Locks for Efficient Concurrency. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques (PACT '09)*. IEEE Computer Society, USA, 3–14. https://doi.org/10.1109/PACT.2009.20

[73] Kewen Wang, Xuelian Lin, and Wenzhong Tang. 2012. Predator—An experience guided configuration optimizer for Hadoop MapReduce. In *4Th IEEE international conference on cloud computing technology and science proceedings*. IEEE, 419–426.

[74] Qingsen Wang, Pengfei Su, Milind Chabbi, and Xu Liu. 2019. Lightweight hardware transactional memory profiling. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. 186–200.

[75] Yukihiro Watanabe, Hiroshi Otsuka, Masataka Sonoda, Shinji Kikuchi, and Yasuhide Matsumoto. 2012. Online failure prediction in cloud datacenters by real-time message pattern learning. In *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*. IEEE, 504–511.

[76] Vincent M Weaver, Matt Johnson, Kiran Kasichayanula, James Ralph, Piotr Luszczek, Dan Terpstra, and Shirley Moore. 2012. Measuring energy and power with PAPI. In *2012 41st international conference on parallel processing workshops*. IEEE, 262–268.

[77] Geoffrey I Webb, Eamonn Keogh, and Risto Miikkulainen. 2010. Naïve Bayes. *Encyclopedia of machine learning* 15 (2010), 713–714.

[78] Jae-Heon Yang and James H. Anderson. 1995. A Fast, Scalable Mutual Exclusion Algorithm. *Distrib. Comput.* 9, 1 (March 1995), 51–60. https://doi.org/10.1007/BF01784242

[79] Richard M Yoo, Christopher J Hughes, Konrad Lai, and Ravi Rajwar. 2013. Performance evaluation of Intel® transactional synchronization extensions for high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–11.

[80] Zhibin Yu, Zhendong Bei, and Xuehai Qian. 2018. Datasize-aware high dimensional configurations auto-tuning of in-memory cluster computing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 564–577.

[81] M. Zhang, H. Chen, L. Cheng, F. C. M. Lau, and C. Wang. 2017. Scalable Adaptive NUMA-Aware Lock. *IEEE Transactions on Parallel and Distributed Systems* 28, 6 (2017), 1754–1769. https://doi.org/10.1109/TPDS.2016.2630695

[82] Zhizhou Zhang, Milind Chabbi, Adam Welc, and Timothy Sherwood. 2021. Optimistic Concurrency Control for Real-world Go Programs. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 939–955.

Received 2022-07-07; accepted 2022-09-22