Toward Efficient Interactions between Python and Native Libraries

Abstract

Python becomes a popular programming language because of its excellent programmability. Many modern software packages utilize Python to drive the high-level algorithm design and depend on native libraries written in C/C++/Fortran for efficient computation kernels. Interaction between Python code and native libraries introduces performance losses because of the abstraction lying on the boundary of Python and native libraries. On the one side, Python code, typically run with interpretation, is disjoint from its execution behavior. On the other side, native libraries do not include program semantics to understand algorithm defects.

To understand the interaction inefficiencies, we extensively study more than 100 Python software packages and categorize them according to the root causes of inefficiencies. We further extract two inefficiency patterns that are common in interaction inefficiencies. Based on these patterns, we develop PIEPROF*, a lightweight profiler, to pinpoint interaction inefficiencies in Python applications. The principle of* PIEPROF *is to measure the inefficiencies in the native execution and associate inefficiencies with high-level Python code to provide a holistic view. Guided by* PIEPROF*, we optimize 17 real-world applications, yielding significant speedups.*

1. Introduction

Many large software packages in the domains of machine learning, cloud computing, and scientific computing use Python to design high-level algorithms and manage various dependencies. The success of Python is largely due to its simplicity, extensibility, and flexibility. Like most scripting languages, Python typically uses interpretation-based execution, which suffers from low performance. Thus, modern Python applications usually integrate native libraries written in C/C++/Fortran for computation kernels, as shown in Figure 1. Such libraries include Numpy [39, 61], Scikit-learn [42], Tensorflow [1], and PyTorch [41] to name a few. Therefore, modern software packages can enjoy the simplicity and flexibility from Python as well as performance from native library.

However, such programming model complicates the software stack, which results in new performance inefficiencies. Figure 1 shows an abstraction across the boundary of Python runtime and native library, which logically splits the entire software stack. On the upper level, Python applications are disjoined from their execution behaviors because Python runtime (e.g., interpreter and GC) hides most of the execution details. On the lower level, the native libraries lose most of program semantic information, resulting in a challenge of associating

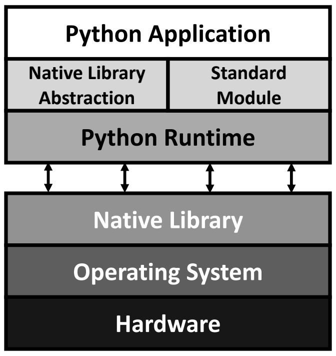


Figure 1: The typical stack of production Python software packages. Python applications usually rely on native libraries for high performance but introduce an abstraction across the boundary of Python runtime and native libraries.

```
def CEC_4(solution=None, problem_size=None, shift=0):
    ...
    for i in range(dim - 1):
        res += 100 * np.square(x[i]**2-x[i+1]) + np.square(x[i]-1)
    ...
```

Listing 1: Code snippet from Metaheuritics [35, 36]. Interaction inefficiencies exist in referencing Numpy arrays.

execution behaviors with high-level algorithms. This knowledge gap leads to a new type of performance losses, which we refer to as *interaction inefficiencies*.

Let us examine a concrete example. See Listing 1. This code snippet is from Metaheuristics [35, 36], which implements the-state-of-the-art meta-heuristic algorithms. It iterates a Numpy array `x` in a `for` loop. Here, no inefficiencies can be easily identified by examining the python code. However, massive redundant computation is revealed when we inspect the native Numpy libraries. The Numpy index operator `[]` invokes the following function in the native library:

```
PyObject *array_subscript_asarray(
                        PyArrayObject *array,
                        PyObject *index)
```

In this function, the first argument points to an array and the second argument points to an index. Besides returning the indexed array element, this function performs extra work, such as checking boundary safety and loading raw data region[1]. This extra work is redundancy on the array `x` when referencing different element of `x` with `[]` across different iterations. We are able to eliminate most of the redundant work by applying the slice notation available in Python, as shown in Listing 6.

[1]Numpy array objects keep a pointer that points to the raw data region, where the real values are stored.

This optimization yields a 27.3× function-level speedup and a 6.3× speedup for the entire application.

Missing the usage of slice notation is the tip of an iceberg of the interaction inefficiencies, which can be in various forms such as API misuse, inefficient algorithms, redundant function invocations, and many others. However, pinpointing interaction inefficiencies and providing optimization guidance is challenging because performance tools need to measure the entire software stack to bridge the gap of high-level Python code and low-level native libraries.

Existing profilers lack of this capability: *(i) Existing Python profilers*, such as cProfile [18], guppy3 [68], PySpy [20], PyFlame [60], PyInstrument [49], PyCallGraph [22], PProfile [43], MemoryProfiler [46] and Austin [59], monitor the execution with Python interpreter APIs. They produce profiles by obtaining Python runtime information only. They attribute various metrics (e.g., execution time and memory consumption) to Python constructs such as methods or objects, but provide limited insights into the behavior of native library internals, which interact with OS and hardware. *(ii) Existing native profilers* working on native binary executables, such as VTune [48], Perf [15], Toddler [37], HPCTookit [2], Witch [65], RedSpy [64], DeadSpy [8], LoadSpy [55], provide low-level binary execution details. These tools use hardware performance monitoring units (PMUs) or binary instrumentation framework to analyze inefficient code generation but they lack the semantics from high-level Python code.

Both existing Python and native profilers fail to identify the interaction inefficiencies. Motivated by the need to obtain holistic profiles from both Python code and native libraries, we propose PIEPROF (Python Interaction inEfficiency PROFiler), a lightweight, insightful profiler to pinpoint interaction inefficiencies in Python programs. The key novelty is to leverage PMUs and other hardware facilities available in commodity CPU processors to monitor native execution and associate the analysis with Python semantics.

Scope. First, we target only interaction inefficiencies between Python codes and native libraries, and measuring inefficiencies in pure Python or pure native codes is out of the scope. Second, we design PIEPROF as a dynamic profiler that pinpoints inefficiencies in codes, but it is the responsibility of human programmers to investigate the profilers and optimize the codes. Third, PIEPROF is input dependent; to ensure that it produces representative profiles, we recommend using typical inputs to study the given Python application.

Our contributions. We made three contributions.

- We are the first to thoroughly study the interaction inefficiencies between Python codes and native libraries. We categorize the interaction inefficiencies by their root causes.
- We design and implement PIEPROF, the first profiler to identify interaction inefficiencies and provide intuitive optimization guidance. PIEPROF works for production Python software packages in commodity CPU processors without modifying the software stack.
- We utilize PIEPROF to study more than 100 Python applications. We identify interaction inefficiencies in 17 real-world applications and optimize them for nontrivial speedups.

Organization. Section 2 covers background and related work; Section 3 performs interaction inefficiency characterization; Section 4 describes the design and implementation of PIEPROF; Section 5 explains the evaluation; Section 6 presentscase studies; Section 7 concludes.

2. Background and Related Work

2.1. Python Runtime

Python is an interpreted language with dynamic features. When running a Python application, the interpreter translates Python source code into stack-based bytecode and executes the bytecode on the Python virtual machine (PVM), which has different implementations such as CPython [12], Jython [29] and PyPy [56]. We study CPython, which is the most popular Python implementation. PVM maintains the call stack during the application execution, which consists of a chain of `PyFrame` objects as known as function frames. Each `PyFrame` object includes the executing context of the corresponding function call, such as local variables, last call instruction, source code file, and current executing code line, which can be leveraged by performance or debugging tools.

Python supports multi-threaded programming, where each Python thread has an individual call stack. Because of the global interpreter lock (GIL) [19], Python threads' concurrent executions are emulated as regular switching threads by the interpreter. It means that, for one interpreter instance, only one Python thread is allowed to execute at any time.

Python allows calling native functions for high performance. The Python runtime passes the `PyObject`[2] or its subclass objects to the native function. The Python runtime treats the native functions as blackboxes—the Python code is blocked from execution until the native function returns.

2.2. Existing Tools vs. PIEPROF

As the first to thoroughly characterize the interaction inefficiencies, we are not aware any related characterization or benchmarking work in this domain, to the best of our knowledge. In this section, we only review the tools that analyze inefficiencies in Python and native codes to distinguish PIEPROF.

Python Performance Analysis Tools PyExZ3 [27], PySym [14], flake8 [13], and Frosted [58] analyze Python source code and employ multiple heuristics to identify code issues statically [25]. XLA [57] and TVM [10] apply compiler techniques to optimize deep learning applications. Harp [67] detects inefficiencies in Tensorflow and PyTorch applications based on computation graphs. These static approaches may

[2]`PyObject` is the super class of all objects in Python.

omit Python dynamic information, resulting in inaccurate analysis.

Dynamic profilers are a complementary approach. cProfile [18] performs Python deterministic profiling, which provides executing frequency/time of specific code region. Guppy [68] employs object-centric profiling, which associates metrics such as allocation frequency, allocation size, and cumulative memory consumption with each Python object. PyInstrument [49] and Austin [59] capture Python call stack frames periodically to identify executing/memory hotspots in Python code. PySpy [20] is able to attach to a Python process and pinpoint function hotspots in real time. Unfortunately existing Python profilers mainly focus on Python codes, with no insights into the native libraries. Thus, unlike PIEPROF, they fail to identify interaction inefficiencies in Python software packages.

Native Performance Analysis Tools There exist various native profiling tools and we only review most related ones that can identify performance inefficiencies. Toddler [37] identifies redundant memory loads across loop iterations. The follow-up tool, LDoctor [51] reduces Toddler's overhead by applying dynamic sampling and static analysis. DeadSpy [8], RedSpy [64], and LoadSpy [55] analyze dynamic instructions in the entire program execution to detect useless computations or data movements. Unlike PIEPROF, these tools utilize heavy-weight binary instrumentation, resulting in high measurement overhead. Moreover, they do not work directly on Python programs.

Perhaps, the most related work is Witch [65], which leverages hardware performance monitoring units (PMUs) and debug registers to identify redundant memory loads and stores in consecutive memory accesses to the same memory location. The basic idea is to use PMUs to sample memory loads or stores and use debug registers to trap the next accesses to the same memory location. If the values from the two loads or two stores are the same, the second load or store is redundant. PIEPROF relies on a similar approach to obtain the inefficiencies in native libraries. Unlike Witch, PIEPROF integrates the semantics from Python runtime, which pioneers the analysis in interaction inefficiencies in Python applications.

3. Interaction Inefficiency Characterization

Interaction inefficiencies exist due to the knowledge gap between Python semantics and native libraries. We have extensively studied more than 100 Python applications, finding that interaction inefficiencies pervasively exist in production Python software packages. Selecting Python applications for investigating in this paper follows the following rule: 1) they are open-sourced Python projects on Github; 2) they are highly ranked in Github trending system [23]; and 3) they have interactions with native libraries.

In this section, we categorize interaction inefficiencies according to their root causes and summarize two common patterns to serve as indicators to pinpoint these inefficiencies. This is the first work on thorough characterization of interaction inefficiencies.

```python
def train(self, trainData, maxEpochs, learnRate):
    ...
    for j in range(self.nh):
        delta = -1.0 * learnRate * ihGrads[i,j]
        self.ihWeights[i, j] += delta
    ...
```

Listing 2: Interaction inefficiencies in IrisData due to the iteration on Numpy arrays within a loop.

```python
def train(self, trainData, maxEpochs, learnRate):
    ...
    self.ihWeights[i, 0:self.nh] += -1.0 * learnRate * ihGrads[i, 0:self.nh]
    ...
```

Listing 3: Optimized IrisData code with slice notation.

3.1. Interaction Inefficiency Categorization

We classify interaction inefficiencies into five categories. We give each category an example using real applications, analyze the root causes, and provide a solution for the fix.

Slice Underutilization Listing 2 shows an example code from IrisData [53], a back-propagation algorithm implementation on Iris Dataset [17]. A loop iterates two multidimensional arrays `ihGrads` and `ihWeights` with indices `i` and `j` for computation. In Python, iterating arrays with a loop is inefficient. Because arrays are supported by native libraries such as Numpy and PyTorch/TensorFlow, indexing operations (i.e., `[]`) in a loop trigger native function calls to perform repeated array safety checks, such as boundary and type verifications, resulting in significant repeated work [38].

To fix this problem, we apply the slice notation, as shown in Listing 3, which removes most repeated native function calls for array indexing, and leverages the accesses to consecutive memory elements, leading to more efficient interactions between Python code and native libraries. For this example, our optimization yields a 2× speedup for the entire program execution.

Repeated Native Function Calls with the Same Augments Functions from native libraries typically have no side effects, so passing the same arguments to a native function results in the same return value, which introduces redundant computation. Listing 4 shows an example of a code from Matplotlib [26], a comprehensive library for creating static, animated, and interactive visualizations in Python. In the code, the argument `theta` for the `rotate` function (rotate angle) is usually the same across different invocations because many machine learning applications rotate images on batches and images in each batch share the same rotate angle. Thus, `Pyobjects` returned from native functions `np.cos()`, `np.sin()` and `np.array()` in lines 2-4 have the same values across images that share the same input `theta`.

Our solution is to apply the memorization technique [16,

```
def rotate(self, theta):
    a = np.cos(theta)
    b = np.sin(theta)
    rotate_mtx = np.array([[a, -b, 0.0], [b, a, 0.0], [0.0, 0.0,
         1.0]], float)
    self._mtx = np.dot(rotate_mtx, self._mtx)
    ...
```

Listing 4: Interaction inefficiencies in Matplotlib due to the same input `theta`.

```
def lars_path(X, y, Xy=None, ...):
    ...
    for i in range(ii, n_active):
        X.T[i], X.T[i + 1] = swap(X.T[i], X.T[i + 1])
        indices[i], indices[i + 1] = indices[i + 1], indices[i]
    ...
```

Listing 5: Interaction inefficiencies in Scikit-learn due to the inefficient algorithm.

34]. Before calling the native functions, we check whether `theta` is the same as in the last invocation. If the same, we reuse the return values from the last invocations of `np.cos()`, `np.sin()` and `np.array()`. Our optimization saves redundant native function calls, and yields a 2.8× speedup to the `rotate` function.

Inefficient Algorithms Listing 5 shows an example of algorithmic inefficiencies from Scikit-learn [42], a widely used machine learning package. The code works on `X`, a two-dimensional Numpy array. It calls the native function `swap` from BLAS [5] library to exchange two adjacent vectors. In each iteration, `swap` returns two `PyObject`s and Python runtime assigns these two `PyObject`s to `X.T[i]` and `X.T[i+1]` respectively. The loop uses `swap` to move the first element in the range to the end position. Because it requires multiple iterations to move `X.T[i]` to the final location, inefficiencies occur.

Instead of using `swap`, our solution is to directly move each element to the target location. We apply the similar optimization to the `indices` array as well. Our improvement yields a 6.1× speedup to the `lars_path` function.

API Misuse in Native Libraries Listing 1 in Section 1 shows an example of API misuse. The code accumulates the computation results to `res`. Since the computation is based on Numpy arrays, the accumulation operation triggers one native function call in each iteration, resulting in many inefficiencies.

Our solution is to use the efficient `sum` API from Numpy as shown in Listing 6, which avoids most of native function invocations by directly operating on the Numpy arrays. This optimization removes most of interaction inefficiencies, and yields a 1.9× speedup to the entire program.

Loop-invariant Computation Listing 7 shows code example code snippet from Deep Dictionary Learning, which seeks multiple dictionaries at different image scales to capture complementary coherent characteristics [33], implemented with TensorFlow. Lines 1-3 indicate the computation inputs `A`, `D`, and `X`. Line 4-5 define the main computation. Line 6-7 actually execute the computation with the actual parameters `D_` and `X_`. The implementation can be shown as following pseudo-code:

```
def CEC_4(solution=None, problem_size=None, shift=0):
    ...
    res += np.sum(100 * np.square(x[0:dim-1]**2 -  x[1:dim]) + np.
         square(x[0:dim-1] - 1))
    ...
```

Listing 6: Optimized Metaheuritics code for Listing 1, with appropriate native library API.

```
1 A = tf.Variable(tf.zeros(shape=[N, N]), dtype=tf.float32)
2 D = tf.placeholder(shape=[N, N], dtype=tf.float32)
3 X = tf.placeholder(shape=[N, N], dtype=tf.float32)
4 R = tf.matmul(D, tf.subtract(X, tf.matmul(tf.transpose(D), A)))
5 L = tf.assign (A, R)
6 for i in range(Iter):
7     result = sess.run(L, feed_dict={D: D_, X: X_})
```

Listing 7: Interaction inefficiencies in Deep Dictionary Learning [33] due to loop-invariant computation.

```
for $i \leftarrow 1$ to $Iter$ do
    $A = D(X - D^T A)$
```

where `D` and `X` are loop invariants, and if we expand the computation, DX and DX^T can be computed outside the loop and reused among iterations, shown as pseudo-code:

```
$t_1 = DX$
$t_2 = DD^T$
for $i \leftarrow 1$ to $Iter$ do
    $A = t_1 - t_2 A$
```

This optimization yields a 3× speedup to the entire execution [67].

3.2. Common Patterns in Interaction Inefficiencies

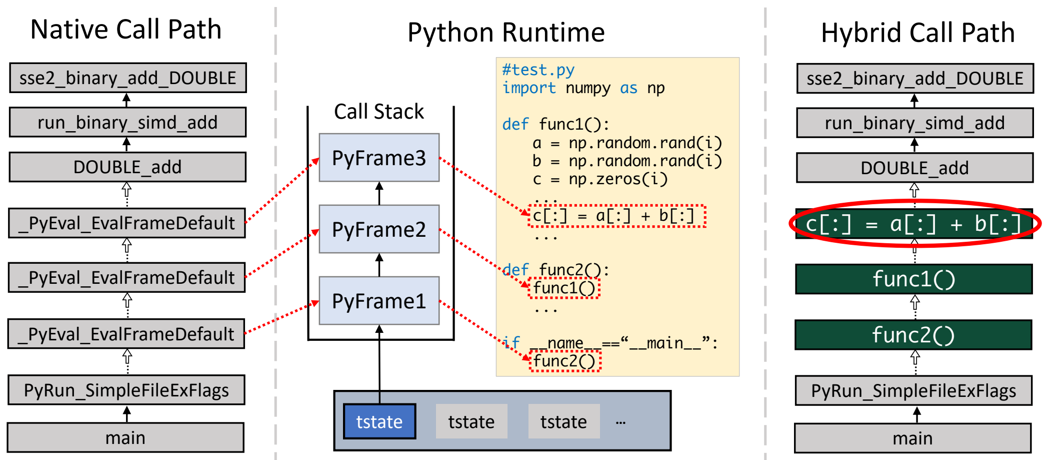
From our studies, we find that almost all interaction inefficiencies involve redundant operations when calling native functions. The redundancies include (1) repeatedly reading the same `PyObject`s of the same values and (2) repeatedly returning `PyObject`s of the same values. However, developing a tool to identify redundant `PyObject`s is difficult and costly because it requires heavyweight Python instrumentation and modification to Python runtime.

With further analysis, we observe that `PyObject` redundancies reveal the following two patterns during the execution from the hardware perspective.

- *Redundant loads:* If two adjacent memory load instructions read the same value from the same memory location, the second load is a redundant load. Repeatedly reading `PyObject` of the same value can result in redundant loads.

Inefficiency Pattern	Inefficiency Category
Redundant Loads	Slice Underutilization
	Inefficient Algorithms
	API Misuse in Native Libraries
Redundant Stores	Loop-invariant Computation
	Repeated Native Function Calls with the same Arguments
	Inefficient Algorithms
	API Misuse in Native Libraries

Table 1: Redundant loads and stores detect different categories of interaction inefficiencies.



```
#test.py
import numpy as np

def func1():
    a = np.random.rand(i)
    b = np.random.rand(i)
    c = np.zeros(i)
    ...
    c[:] = a[:] + b[:]
    ...
def func2():
    func1()
    ...
if __name__=="__main__":
    func2()
```

Figure 2: Constructing a hybrid call path across Python runtime and native libraries. White arrows in call paths denote a series of elided call frames in PVM. The red circle in hybrid call path shows the boundary of Python and native frames, where interaction inefficiencies occur.

- *Redundant stores:* If two adjacent memory store instructions write the same value to the same memory location, the second store is a redundant store. Repeatedly returning `PyObject` of the same value can result in redundant stores.

We use the redundant loads and stores to serve as indicators of interaction inefficiencies. Table 1 shows different categories of interaction inefficiencies, which show up as redundant loads or stores. Thus, we design PIEPROF to monitor redundant load and store instructions and associate them with Python structures to pinpoint interaction inefficiencies.

4. Design and Implementation

To identify interaction inefficiencies, PIEPROF first pinpoints the redundant loads and stores with performance monitoring units (PMUs) and debug registers available in commodity CPU processors. Such information only is too low level to understand Python semantics for optimization guidance, so we develop a new approach, leveraging combined calling contexts from both Python code and native libraries, to identify interaction behaviors. In the rest of this section, we elaborate on the mechanisms of pinpointing redundancies and determining the combined calling contexts. We then describe how we address some implementation challenges.

4.1. Pinpointing Redundant Loads/Stores

PIEPROF adopts PMUs and debug registers to identify redundant loads and stores in an instruction steam. PIEPROF configures PMUs to sample memory accesses and captures the instruction pointer and effective address upon each sample. PIEPROF uses debug registers to watch the effective address and trap on the next access to the same effective address. By analyzing the sampled and trapped instructions, PIEPROF can compare the values and determine whether they are a pair of redundant memory accesses. We leverage the existing approach [65] to address the shortage of debug registers (only 4 in x86 architectures) via reservoir sampling [62]. PIEPROF interacts with Python runtime to correlate the low-level behavior in an instruction steam with high-level Python code. The workflow of PIEPROF consists of following steps:

1. PIEPROF subscribes to the precise PMUs events E for each thread before the thread's initialization. PMUs deliver samples of subscribed events to PIEPROF periodically.
2. When PIEPROF receives a sample, it snapshots the current Python runtime state. Combining the sample and runtime state, PIEPROF builds hybrid call path C_1, and extracts the effective address M and the value V_1 stored at M.
3. PIEPROF sets watchpoint to monitor M with access type T, and resumes runtime execution.
4. Memory accessing to M triggers a watchpoint to trap.
5. PIEPROF snapshots the state of runtime again and builds the hybrid call path C_2, and inspects the current value V_2 stored at M.
6. PIEPROF compares V_1 and V_2, and reports $< C_1, C_2 >$ as an inefficiency pair if V_1 equals V_2.

When detecting redundant stores, PIEPROF sets the hardware event E as memory store event and sets T as memory store type, while PIEPROF sets the hardware event E as a memory load event and sets T a as memory load type for detecting redundant loads.

4.2. Constructing Hybrid Calling Contexts

Combining the call paths of Python code and native libraries at a given point of the execution provides unique insights into the interaction inefficiencies. We use different techniques to determine the calling contexts for native libraries and Python codes. For native libraries. we unwind the call stack to determine the call path. PIEPROF leverages libunwind [50] to obtain a chain of procedure frames on the call stack.

However, call stack unwinding is not directly applicable to Python code because of the abstraction introduced by Python virtual machine (PVM). The frames on the stack are from

PVM, not Python codes, as the native call path shown in Figure 2. Thus, PIEPROF inspects the dynamic runtime to extract the call path of Python code on the fly. As shown in Figure 2, each Python thread maintains its call stacks in a thread local object `PyThreadState` (i.e., `tstates`). To obtain the calling context at any point, PIEPROF first calls `GetThisThreadState()` to get the `PyThreadState` object of the current thread. Then PIEPROF obtains the bottom `PyFrame` object in the PVM call stack from the `PyThreadState` object. All `PyFrame` objects in the PVM call stack are organized as a singly linked list from bottom to top. Each `PyFrame` object contains rich information about the current Python frame, such as source code files and line numbers, which PIEPROF uses to correlate a `PyFrame` to a Python method. In the example, `PyFrame1`, `PyFrame2` and `PyFrame3` are for Python methods `main`, `func2` and `func1`, respectively.

To construct the hybrid call paths between Python code and native libraries, PIEPROF needs to map PVM-related frames on the native call path to the Python methods (denoted as `PyFrame`). From our observation, the frame `_PyEval_EvalFrameDefault` on the native call path 1-to-1 maps to a `PyFrame`. Based on this observation, PIEPROF uses the following algorithm to construct the hybrid call paths:

1. PIEPROF scans all the frames in the native call paths and obtains the frames from PVM.
2. PIEPROF leverages the instruction pointer stored in the frame to identify `_PyEval_EvalFrameDefault` frames in the native call path and drop other unnecessary PVM frames.
3. PIEPROF replaces `_PyEval_EvalFrameDefault` frames with `PyFrame` frames according to the call relationship obtained from the Python runtime.

Figure 2 illustrates the procedure of hybrid call path is constructed from native and Python call paths upon each PMU sample and debug register trap. The boundary of Python and native frames indicates the location where interaction inefficiencies potentiall occur.

Maintaining Hybrid Calling Contexts PIEPROF applies compact calling context tree (CCT) [4, 3] to represent the profile. Figure 3 overviews the structure of a CCT produced by PIEPROF. Internal nodes represent native or Python function calls and leaf nodes presents the sampled memory loads or stores. Logically, each path from a leaf node to the root represents a unique call path.

It is challenging to handle CCT operations efficiently. Previous efforts [8, 64, 55, 7, 54] produce a CCT for each thread-/process. The memory overhead of these approaches grows linearly with more threads/processes in use. In contrast, PIEPROF creates a single CCT for the entire program execution. PIEPROF uses a lock-free skip-list [44] to maintain the children of a CCT node to enjoy fast and thread-safe operations. Theoretically, the lookup, insert, and delete operations have $\mathcal{O}(log_n)$ time complexity. Typically, Skip-list with more

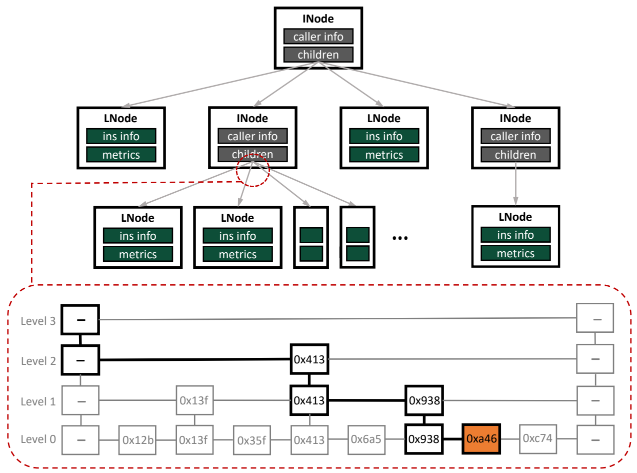


Figure 3: A calling context tree constructed by PIEPROF. Each parent node applies skip-list to organize children. Red box shows searching `0xa46` in the example skip-list.

layers has higher performance but higher memory overhead. In a CCT, the nodes closer to the root will be more frequently accessed. For a good tradeoff, PIEPROF proportionally adjusts the number of layers in the skip-lists according to the depth of the CCT. It sets a larger number to skip-lists that are closer to the root to maximize the performance, and sets a smaller number to skip-lists that are closer to the leaf to minimize the memory consumption.

4.3. Interfering with Python Runtime

Safe Sampling for Python runtime PMUs sampling or watchpoint trap send signals to process, which pause Python runtime and execute PIEPROF's code procedure. Pausing Python runtime in specific states triggers exceptions. Some possible exceptions are: 1) the dead lock caused by handling samples when Python runtime is executing memory management APIs; 2) the interference to profiling result caused by garbage collection; 3) invalid sampling when its instruction pointer falls in PIEPROF's code region.

PIEPROF performs safe sampling by applying both instruction-level (fine-grained) and function-level (coarse-grained) protections. Fine-grained protection has a block list of virtual address ranges *(Block V List)*, which contains the memory range of pre-defined block functions/libraries. Through fine-grained protection, PIEPROF ignores the samples whose instruction pointers are in the *Block V List*. Coarse-grained protection has a block list of functions *(Block F List)*, that will be redefined by mocking functions in a new shared library. PIEPROF preloads mocking functions at the beginning of program execution to overload block functions. When a function on *Block F List* is called, it executes the function wrapper. The wrapper function first pauses PIEPROF, then executes the real function by function pointer, and finally resumes PIEPROF's profiling.

Handling Python Garbage Collection Python runtime applies reference counting and generational garbage collector to perform the garbage collection (GC). GC interferes PIEPROF from three aspects: 1) frequent increments/decrements of references trigger a great amount of memory access, occupying limited debug registers; 2) generational garbage collector reads `PyObject`s, resulting in misreporting of redundant loads; 3) when PIEPROF is monitoring a memory address, deallocating corresponding `PyObject` by GC incurs unexpected errors.

Python runtime does not provide APIs to investigate GC's behaviors. PIEPROF has to employ multiple techniques to guarantee the correct measurement when handling Python garbage collection. First, all `PyObject`s share the same header structure that contains the reference number, so PIEPROF does not monitor the memory address belonging to the header of a `PyObject`. Second, PIEPROF uses coarse-grained protection to ignore generational garbage collecting functions, avoiding the interference with generational garbage collector. Finally, when PIEPROF monitors memory addresses, it increases corresponding `PyObject`'s reference to avoid the deallocation, and decreases the reference at the end of monitoring to prevent memory leak.

Dynamic Query of Native Function Addresses PIEPROF needs to dynamically query native function addresses for calling context construction, *Block V List* and *Block F List* initialization and calculating watchpoint's precise instruction pointer. Therefore, PIEPROF maintains a function range map that has all native functions existing in application memory space. A function and its basic information (e.g., function name, library name, and virtual memory address range) are stored in the function range map for further usage.

We perform some optimizations to update the function range map, because the `/proc/self/maps` file changes frequently during runtime. Previous work [9] suggests periodic checking, but it is difficult to choose an appropriate checking interval. Long checking interval makes the function range map inconsistent with the runtime state, while short checking interval results a high overhead. PIEPROF uses the "lazy" strategy, which triggers an updating operation when a query fails, to balance functionality and performance. Function range map is also subject to collisions when the same file is mapped/unmapped or the same virtual address is reused multiple times. To prevent collisions, PIEPROF uses incremental version numbers for faster detection of outdated address ranges, deleting the corresponding functions from the map. PIEPROF equips readers-write lock to protect the map because read operations are more frequent than write operations.

5. Evaluation

We evaluate PIEPROF on a 14-core Intel Xeon E7-4830 v4 machine clocked at 2GHz running Linux 3.10. The machine is equipped with 256 GB main memory and four debug registers. PIEPROF is complied with `GCC 6.2.0 -O3`, and CPython (version 3.6) is built with `-enable-shared` flag. PIEPROF subscribes hardware event `MEM_UOPS_RETIRED_ALL_STORES` for redundant stores detection and `MEM_UOPS_RETIRED_ALL_LOADS` for redundant loads detection, respectively.

Python does not have standard benchmarks, so PIEPROF is evaluated on three high-stars GitHub projects — Scikit-learn [42], Numexpr [45] and NumpyDL [63] that consist of benchmark programs from scientific computing, numerical expression, and deep learning domains. Only the evaluation result of the first half of scikit-learn benchmark set is reported due to the space limitation, and three programs (`varying-expr.py` from Numexpr, `cnn-minist.py` and `mlp-minist.py` from NumpyDL) are excluded due to either huge memory consumption variances of vanilla runs (`varying-expr.py`), or runtime errors of vanilla runs (`cnn-minist.py`, and `mlp-minist.py`).

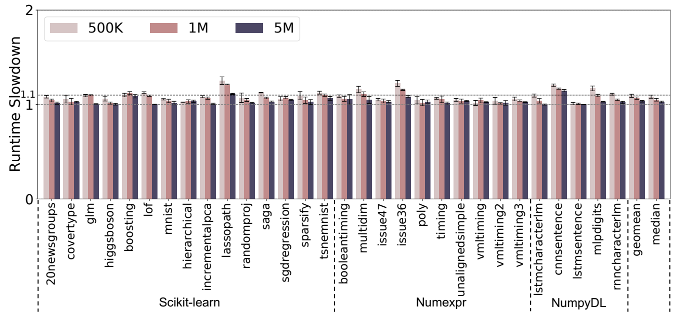
Each experiment runs three times. We report the average overhead and error bars. Furthermore, the overhead of PIEPROF is evaluated with three sampling frequencies.

5.1. Overhead

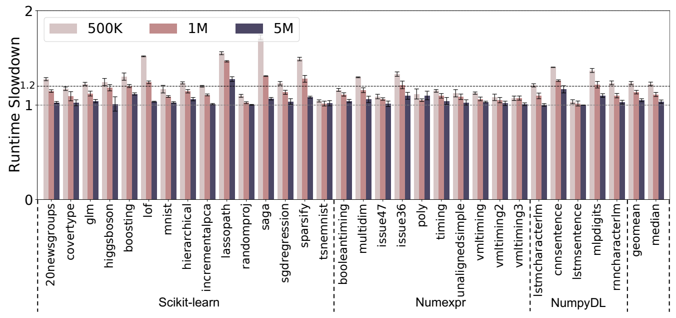
This section reports the runtime slowdown and memory bloating of programs caused by PIEPROF. Runtime slowdown is measured by the ratio of program execution time with PIEPROF enabled over its vanilla execution time. Memory bloating shares the same measuring method but with the peak memory usage.

PIEPROF overhead is evaluated with three commonly-used sampling rates (500K, 1M and 5M). Figure 4a shows the runtime slowdown of redundant stores detection. The geo-means are 1.09×, 1.07×, and 1.03× under the sampling rates of 500K, 1M, and 5M, respectively, and the medians are 1.08×, 1.05×, and 1.03×, respectively. Similarly, Figure 4b shows the runtime slowdown of redundant loads detection. The geo-means are 1.22×, 1.14×, and 1.05×, under the sampling rates of 500K, 1M, and 5M, respectively, and the medians are 1.22×, 1.11×, and 1.04×, respectively. The runtime slowdown drops as sampling rate decreases, because more PMUs samples incur more frequent profiling events, such as inspecting Python runtime, querying the CCT, and arming/disarming watchpoints. Redundant loads detection incurs more runtime slowdown compared to redundant stores detection, because programs usually have more loads than stores. Another reason is that PIEPROF sets `RW_TRAP` for the debug register to monitor memory loads (x86 does not provide trap on read-only facility) which traps on both memory stores and loads. Even though PIEPROF ignores the traps triggered by memory stores, monitoring memory loads still incurs extra overhead.

Figure 5a reports memory bloating of redundant stores detection. The geo-means are 1.25×, 1.24×, and 1.23× under the sampling rates of 500K, 1M, and 5M, respectively, and the medians are 1.18×, 1.18×, and 1.16×, respectively. Similarly, Figure 5b reports memory bloating of redundant loads

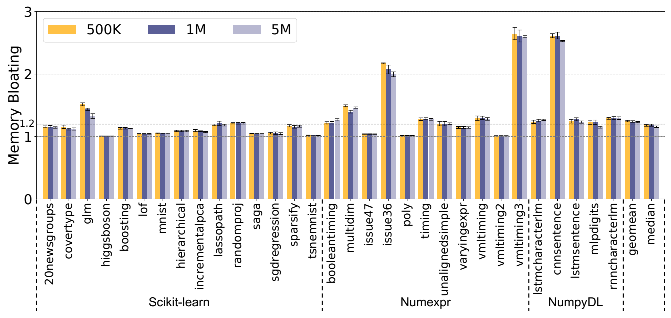


(a) Redundant Stores Detection

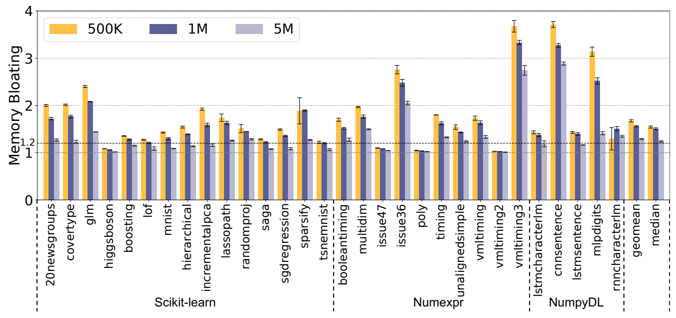


(b) Redundant Loads Detection

Figure 4: Runtime slowdown of PIEPROF on Scikit-learn, Numexpr and NumpyDL with sampling rates of 500K, 1M and 5M. The y-axis denotes slowdown ratio and the x-axis denotes program name. The error bars indicate 95% confidence intervals, calculated by repeating each trial 3 times.



(a) Redundant Stores Detection



(b) Redundant Loads Detection

Figure 5: Memory bloating of PIEPROF on Scikit-learn, Numexpr and NumpyDL with sampling rates of 500K, 1M and 5M. The y-axis denotes slowdown ratio and the x-axis denotes program name. The error bars indicate 95% confidence intervals, calculated by repeating each trial 3 times.

detection. The geo-means are 1.67×, 1.56×, and 1.29×, respectively, under the same sampling rates to the above, and the medians are 1.52×, 1.51×, and 1.24× respectively. Memory bloating shows a similar trend to runtime slowdown with varied sampling rates and between two kinds of inefficiency detection. The extra memory consumption is caused by the larger CCT required for the larger number of unique call paths. `issue36`, `vmltiming2`, and `cnnsentence` suffer the most severe memory bloating due to the small memory required by their vanilla runs. PIEPROF consumes a fixed amount of memory because of some static structures that are irrelevant to the testing program. Thus, a program has a higher memory bloating ratio if it requires less memory for a vanilla run. `mlpdigits` consumes much more memory for redundant loads detection than redundant stores detection. This is because `mlpdigits` (a deep learning program) contains a two-level multilayer perceptron (MLP) that has more memory loads than stores.

Although lower sampling rate leads to less overhead, it increases the probability of missing some subtle inefficiencies. To achieve a better trade-off between overhead and detecting ability, we empirically select 1M as sampling rate.

5.2. Effectiveness

To confirm the effectiveness of PIEPROF, we investigate more than 100 real-world Python applications. 19 inefficiencies are identified, of which 18 are newly identified performance bugs (see Table 2 for a summary). Table 2 also reports the performance speedup over the whole application (`AS`, denoting application-level speedup) and over the function (`FS`, denoting function-level speedup), respectively, by following PIEPROF's optimization guidance. `AS` is defined as the ratio of the whole application execution time with the original library over the execution time with the optimized library. `FS` is defined as the ratio of the total execution time of inefficient function before optimization over its total execution time after optimization.

Table 2 shows that the optimizations following PIEPROF's optimization guidance lead to 1.02× to 6.3× application-level speedup (`AS`), and 1.05× to 27.3× function-level speedup (`FS`), respectively. According to Amdahl's law, `AS` approaches `FS` as the function increasingly dominates the overall execution time. It is worth noting that the five inefficiency categories defined before (see Section 3.1) widely exist in real-world applications, and PIEPROF can detect with redundant loads/stores detection. It proves PIEPROF's strong effectiveness.

Program Information			Inefficiency		Optimization	
Applications	Library	Problem Code	Category	Pattern	AS	FS
Ta [40]	Ta	volatily.py(45)/trend.py(536, 549, 557, 571, 579)	Slice Underutilization	L	1.1×	16.6×
NumPyCNN [21]	Numpy [39, 61]	numpycnn.py(161)	Loop-invariant Computation	S	1.8×	2.04×
Census_main	NumpyWDL [52]	ftrl.py(60)	Loop-invariant Computation	S	1.03×	1.1×
Lasso	Scikit-learn [42]	least_angle.py(456, 458)	Inefficient Algorithms	S	1.2×	6.1×
IrisData [53]	Numpy	nn_backprop.py(222, 228, 247, 256, 263, 271, 278)	Slice Underutilization & API Misuse	L	2×	2.02×
Network	Neural-network-from-scratch	network.py(103-115)	Repeated NFC with the same Augments	L	1.03×	1.05×
Cnn-from-scratch [66]	Numpy	conv.py(62)	Slice Underutilization	L	2.5×	3.9×
Metaheuristics [35, 36]	Numpy	FunctionUtil.py(374)	API Misuse	L	1.4×	1.9×
		FunctionUtil.py(270)	Slice Underutilization	L	6.3×	27.3×
		FunctionUtil.py(309, 375)	Loop-invariant Computation	S	1.04×	1.4×
		FunctionUtil.py(437)	Repeated NFC with the same Augments	L	1.02×	1.1×
		EPO.py(40)	Loop-invariant Computation	S	1.1×	1.1×
LinearRegression [31]	LinearRegression	LinearRegression.py(49, 50)	Repeated NFC with the same Augments	L	1.4×	1.5×
Pytorch-examples [28]	PyTorch [41]	adam.py:loop(66)	Loop-invariant Computation	L	1.02×	1.07×
Cholesky [67]	PyTorch	cholesky.py(76)	Slice Underutilization	L	3.2×	3.9×
GGNN.pytorch [11]	PyTorch	model.py(122, 125)	Loop-invariant	S	1.03×	1.07×
Network-sliming [32]	Torchvision [47]	functional.py(164)	Slice Underutilization	L	1.1×	1.7×
Pytorch-sliming [32]					1.04×	1.7×
Fourier-Transform [30]	Matplotlib [26]	transforms.py(1973)	Repeated NFC with the same Augments	S	1.02×	2.8×
Jax [6]					1.04×	2.8×
Autograd [24]					1.05×	2.8×

Table 2: Overview of performance improvement guided by PIEPROF. AS denotes application-level speedup, FS denotes function-level speedup. L refers to redundant loads, S refers to redundant stores.

```
1 def backprop(self, d_L_d_out, learn_rate):
2     d_L_d_filters = np.zeros(self.filters.shape)
3     for im_region, i, j in self.iterate_regions(self.last_input):
4         for f in range(self.num_filters):
5             d_L_d_filters[f] += d_L_d_out[i, j, f] * im_region
```

Listing 8: Interaction inefficiency in CNN-from-Scratch due to slice underutilization.

```
1 def backprop(self, d_L_d_out, learn_rate):
2     d_L_d_filters = np.zeros(self.filters.shape)
3     for im_region, i, j in self.iterate_regions(self.last_input):
4         new_im_region = np.repeat(im_region[np.newaxis,:,:], 8,
              axis = 0)
5         tmp = d_L_d_out[i, j, 0:self.num_filters]
6         d_L_d_filters[0:self.num_filters] += tmp[:,None,None] *
              new_im_region
```

Listing 9: Optimized code of Listing 8, eliminates inefficiencies by performing slice notation.

```
--------------------------------------------------------------------------------
prepare_index                     numpy/core/_multiarray_umath.so       0x2b728f71ead1
array_subscript                   numpy/core/_multiarray_umath.so       0x2b728f72094b
d_L_d_filters[f] += d_L_d_out[i, j, f] * im_region                      conv.py:62
_PyFunction_FastCall              libpython3.6m.so.1.0                  0x2b7282e99040
call_function                     libpython3.6m.so.1.0                  0x2b7282e9a061
gradient = conv.backprop(gradient, lr)                                  cnn.py:55
_PyEval_EvalCodeWithName          libpython3.6m.so.1.0                  0x2b7282e99aac
call_function                     libpython3.6m.so.1.0                  0x2b7282e99d74
l, acc = train(im, label)                                               cnn.py:82
_PyEval_EvalCodeWithName          libpython3.6m.so.1.0                  0x2b7282e99aac
PyEval_EvalCodeEx                 libpython3.6m.so.1.0                  0x2b7282e9a0be
PyEval_EvalCode                   libpython3.6m.so.1.0                  0x2b7282e9a0eb
PyRun_FileExFlags                 libpython3.6m.so.1.0                  0x2b7282ecf392
PyRun_SimpleFileExFlags           libpython3.6m.so.1.0                  0x2b7282ecf505
main                              PieProf/bin/main                       0x400bc7
**************************************** killed by ****************************************
prepare_index                     numpy/core/_multiarray_umath.so       0x2b728f71ead1
array_subscript                   numpy/core/_multiarray_umath.so       0x2b728f72094b
d_L_d_filters[f] += d_L_d_out[i, j, f] * im_region                      conv.py:62
_PyFunction_FastCall              libpython3.6m.so.1.0                  0x2b7282e99040
call_function                     libpython3.6m.so.1.0                  0x2b7282e9a061
gradient = conv.backprop(gradient, lr)                                  cnn.py:55
_PyEval_EvalCodeWithName          libpython3.6m.so.1.0                  0x2b7282e99aac
call_function                     libpython3.6m.so.1.0                  0x2b7282e99d74
l, acc = train(im, label)                                               cnn.py:82
_PyEval_EvalCodeWithName          libpython3.6m.so.1.0                  0x2b7282e99aac
PyEval_EvalCodeEx                 libpython3.6m.so.1.0                  0x2b7282e9a0be
PyEval_EvalCode                   libpython3.6m.so.1.0                  0x2b7282e9a0eb
PyRun_FileExFlags                 libpython3.6m.so.1.0                  0x2b7282ecf392
PyRun_SimpleFileExFlags           libpython3.6m.so.1.0                  0x2b7282ecf505
main                              PieProf/bin/main                      0x400bc7
--------------------------------------------------------------------------------
```

Figure 6: The redundant load pair reported by PIEPROF for Listing 8.

6. Case Studies

In this section we conduct three heuristic case studies. Our primary aim is to demonstrate the superior guidance provided by PIEPROF for inefficiency detection and optimization.

6.1. CNN-from-Scratch

CNN-from-Scratch is an educational project that implements a Convolutional Neural Network (it has 112 stars on GitHub). The code in Listing 8 performs tensor computation within a two-level nested loop. `d_L_d_filters` is a 8×3×3 tensor, `d_L_d_out` is a 26×26×8 tensor and `im_region` is a 3×3 tensor. The inner loop iterates `d_L_d_filters` by its first dimension, iterates `d_L_d_out` by its third dimension. In each iteration of inner loop, `d_L_d_filters[f]` performs as a 3×3 tensor, and `d_L_d_out[i, j, f]` is a number. Computation in line 5 is summarized as a 3×3 vector cumulatively adding the multiplication of a number and a 3×3 vector.

Figure 6 shows a redundant loads pair reported by PIEPROF. The redundant pair is represented as hybrid call path, and the upper call path is killed by the lower call path. For each native call path, PIEPROF reports the native function name, shared library directory and the instruction pointer.

```
1 def CEC_10(solution=None, problem_size=None, shift=0):
2     ...
3     for i in range(dim):
4         temp = 1
5         for j in range(32):
6             temp += i * (np.abs(np.power(2, j + 1) * x[i] - round(
                   np.power(2, j + 1) * x[i]))) / np.power(2, j)
7         A *= np.power(temp, 10 / np.power(dim, 1.2))
8     ...
```

Listing 10: Interaction inefficiency in Metaheuristic due to the API misuse and loop-invariant computation.

```
def CEC_10(solution=None, problem_size=None, shift=0):
    ...
    tmp_dim = 10 / np.power(dim, 1.2)
    for i in range(dim):
        temp = 1
        for j in range(32):
            frac, whole = math.modf(np.power(2, j + 1) * x[i])
            temp += i * np.abs(frac) / np.power(2, j)
        A *= np.power(temp, tmp_dim)
    ...
```

Listing 11: Optimized code of Listing 10, eliminates inefficiencies with an appropriate API and memorization technique.

For each Python call path, it reports the problematic code piece and its location in the source file. In this case, the call path pair reveals that the interaction inefficiency is introduced by line 62 of conv.py (line 5 in Listing 8). The call path also shows that the inefficiency caused by native function call `prepare_index(array_subscript)`, denotes the redundant `[]` operations. This inefficiency belongs to the category of slice under-utilization.

For optimization, we match the dimension of `d_L_d_filters`, `d_L_d_out`, and `im_region` by expanding the dimension of `im_region`, and use slice notation to replace the inner loop, as shown in Listing 9. The optimization yields a 3.9× function-level speedup and 2.5× application-level speedup.

6.2. Metaheuristics

Listing 10 shows a code snippet from Metaheuristics. It performs complex numerical computation in a two-level nested loop, where `x` is a Numpy array. PIEPROF reports a redundant loads on line 6, where the code triggers the redundant native function call `array_multiply` and `LONG_power`. Guided by this, we observe that `np.abs(np.power(2, j + 1)*x[i]` is calculated twice within every iteration, because the code aims to get the computation result's fraction part. To eliminate the redundant computation, we use `math.modf` function to calculate the fraction directly. This inefficiency belongs to the category of API misuse in native libraries. PIEPROF also reports redundant stores in line 7 with native function `LONG_power`. Upon further investigation, we find the result of `np.power(dim, 1.2)` does not change among iterations, which belong to loop-invariant computation. For optimization, we use a local variable to store the result outside the loop and reuse it among iterations.

```
1 def adx(self) -> pd.Series:
2     ...
3     adx = np.zeros(len(self._trs))
4     tmp = (self._n - 1)/float(self._n)
5     for i in range(self._n+1, len(adx)):
6         adx[i] = adx[i-1] * tmp + dx[i-1] / float(self._n)
7     ...
```

Listing 12: Interaction inefficiency in Ta due to the slice underutilization.

```
def adx(self) -> pd.Series:
    ...
    adx = np.zeros(len(self._trs))
    tmp = (self._n - 1)/float(self._n)
    for i in range(self._n+1, len(adx)):
        adx[i] = adx[i-1] * tmp
    adx[self._n+1:len(adx)] += dx[self._n:(len(adx)-1)] / float(
        self._n)
    ...
```

Listing 13: Optimized code of Listing 12, eliminates inefficiencies by performing slice notation.

The appropriate usage of API yields 1.4× application-level speedup and 1.9× function-level speedup, and eliminating loop invariant computation yields 1.04× application-level speedup and 1.4× function-level speedup, respectively.

6.3. Technical Analysis

Technical Analysis (Ta) [40] is a technical analysis Python library (it has 1.3k stars on GitHub). Listing 12 shows a problematic code region of Ta, where `adx` and `dx` are two multi-dimension Numpy arrays, and a loop iterates them and performs numerical calculations. PIEPROF reports redundant loads in line 6 with native function `array_subscript`, which denotes the code that suffers from the inefficiency of slice underutilization. However, we cannot eliminate the loop because `adx` has computing dependency among the iterations.

We optimize the access to `dx` with slice notation shown in Listing 13. Eliminating all similar patterns in Ta yields 1.1 × application-level speedup and 16.6× function-level speedup.

7. Conclusions

This paper is the *first* to study the interaction inefficiencies in complex Python applications. Initial investigation finds that the interaction inefficiencies occur due to the use of native libraries in Python code, which disjoints the high-level code semantics with low-level execution behaviors. By studying more than 100 applications, we are able to assign the interaction inefficiencies to five categories based on their root causes. We extract two common patterns, redundant loads and stores in the execution behaviors across the categories, and design PIEPROF to pinpoint interaction efficiencies by leveraging PMUs and debug registers. Besides pinpointing redundant loads and stores, PIEPROF cooperates with Python runtime to associate the inefficiencies with Python contexts. With the guidance of PIEPROF, we optimized 17 Python applications, fixed 19 interaction inefficiencies, and obtained numerous nontrivial speedups.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. Hpctoolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[3] Glenn Ammons, Thomas Ball, and James R Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *ACM Sigplan Notices*, 32(5):85–96, 1997.

[4] Matthew Arnold and Peter F Sweeney. *Approximating the calling context tree via sampling*. IBM TJ Watson Research Center Yorktown Heights, New York, US, 2000.

[5] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

[6] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, and Skye Wanderman-Milne. JAX: composable transformations of Python+NumPy programs, 2018.

[7] Milind Chabbi, Xu Liu, and John Mellor-Crummey. Call paths for pin tools. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 76–86, 2014.

[8] Milind Chabbi and John Mellor-Crummey. DeadSpy: A Tool to Pinpoint Program Inefficiencies. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 124–134, New York, NY, USA, 2012. ACM.

[9] Milind Chabbi, Shasha Wen, and Xu Liu. Featherlight on-the-fly false-sharing detection. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 152–167, 2018.

[10] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: end-to-end optimization stack for deep learning. *arXiv preprint arXiv:1802.04799*, 11:20, 2018.

[11] Ching-Yao Chuang. Ggnn: A pytorch implementation of gated graph sequence neural networks. https://github.com/chingyaoc/ggnn.pytorch.

[12] CPython Community. C-extensions for python. https://cython.org/#documentation.

[13] Ian Stapleton Cordasco. Flake8: Your tool for style guide enforcement. https://flake8.pycqa.org/en/latest/.

[14] Bjoern I. Dahlgren. Pysym documentation. https://pythonhosted.org/pysym/.

[15] Arnaldo Carvalho De Melo. The new linux perf tools. In *Slides from Linux Kongress*, volume 18, 2010.

[16] Luca Della Toffola, Michael Pradel, and Thomas R Gross. Performance problems you can fix: A dynamic analysis of memoization opportunities. *ACM SIGPLAN Notices*, 50(10):607–622, 2015.

[17] Ronald A Fisher. The use of multiple measurements in taxonomic problems. *Annals of eugenics*, 7(2):179–188, 1936.

[18] Python Software Foundation. cprofile. https://github.com/python/cpython/blob/master/Lib/cProfile.py.

[19] Python Software Foundation. Python document: Thread state and the global interpreter lock. https://docs.python.org/3/c-api/init.html#thread-state-and-the-global-interpreter-lock.

[20] Ben Frederickson. py-spy: Sampling profiler for python programs. https://github.com/benfred/py-spy.

[21] Ahmed Gad. Numpycnn: Implementing convolutional neural networks from scratch. https://github.com/ahmedfgad/NumPyCNN.

[22] gak. pycallgraph: Python call graph. https://github.com/gak/pycallgraph/.

[23] Github. Github trending. https://github.com/trending.

[24] Harvard Intelligent Probabilistic Systems Group. Autograd: Efficiently computes derivatives of numpy code. https://github.com/HIPS/autograd.

[25] Hristina Gulabovska and Zoltán Porkoláb. Survey on static analysis tools of python programs. In *SQAMIA*, 2019.

[26] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[27] M Irlbeck et al. Deconstructing dynamic symbolic execution. *Dependable Software Systems Engineering*, 40:26, 2015.

[28] Justin Johnson. Pytorch-example: the fundamental concepts of pytorch through self-contained examples. https://github.com/jcjohnson/pytorch-examples.

[29] Jyphon. Jpython homepage. https://www.jython.org/.

[30] Fotis Kapotos. Fourier-transform. https://github.com/fotisk07/Fourier-Transform.

[31] Sarvasv Kulpati. An implementation of linear regression from scratch in python. https://github.com/sarvasvkulpati/LinearRegression.

[32] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. In *The IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

[33] Shahin Mahdizadehaghdam, Ashkan Panahi, Hamid Krim, and Liyi Dai. Deep dictionary learning: A parametric network approach. *IEEE Transactions on Image Processing*, 28(10):4790–4802, 2019.

[34] Khanh Nguyen and Guoqing Xu. Cachetor: Detecting cacheable data to remove bloat. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 268–278, 2013.

[35] Thieu Nguyen, Binh Minh Nguyen, and Giang Nguyen. Building resource auto-scaler with functional-link neural network and adaptive bacterial foraging optimization. In *International Conference on Theory and Applications of Models of Computation*, pages 501–517. Springer, 2019.

[36] Thieu Nguyen, Nhuan Tran, Binh Minh Nguyen, and Giang Nguyen. A resource usage prediction system using functional-link and genetic algorithm neural network for multivariate cloud metrics. In *2018 IEEE 11th conference on service-oriented computing and applications (SOCA)*, pages 49–56. IEEE, 2018.

[37] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 562–571. IEEE, 2013.

[38] Numpy. General function for indexing numpy array. https://flake8.pycqa.org/en/latest/.

[39] Travis E Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[40] Darío López Padial. Technical analysis library in python. https://github.com/bukosabino/ta.

[41] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[42] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[43] Vincent Pelletier. Pprofile: Line-granularity, thread-aware deterministic and statistic pure-python profiler. https://github.com/vpelletier/pprofile.

[44] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[45] pydata. Numexpr: Fast numerical expression evaluator for numpy. https://github.com/pydata/numexpr.

[46] pythonprofilers. Memory profiler. https://github.com/pythonprofilers/memory_profiler/.

[47] PyTorch. Datasets, transforms and models specific to computer vision. https://github.com/pytorch/vision.

[48] James Reinders. Vtune performance analyzer essentials. *Intel Press*, 2005.

[49] Joe Rickerby. pyinstrument. https://github.com/joerick/pyinstrument.

[50] Savannah. The libunwind project homepage. https://www.nongnu.org/libunwind/.

[51] Linhai Song and Shan Lu. Performance diagnosis for inefficient loops. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 370–380. IEEE, 2017.

[52] Stasi. Numpywdl: Implement wide & deep algorithm by using numpy. https://github.com/stasi009/NumpyWDL.

[53] Lee Stott. Irisdata: Iris data example python numpy. https://github.com/leestott/IrisData.

[54] Pengfei Su, Qingsen Wang, Milind Chabbi, and Xu Liu. Pinpointing performance inefficiencies in java. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 818–829, 2019.

[55] Pengfei Su, Shasha Wen, Hailong Yang, Milind Chabbi, and Xu Liu. Redundant loads: A software inefficiency indicator. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 982–993. IEEE Press, 2019.

[56] The PyPy Team. Pypy homepage. https://www.pypy.org/index.html.

[57] XLA Team et al. Xla-tensorflow compiled, 2017.

[58] timothycrosley. Frosted documentation. https://pypi.org/project/frosted/.

[59] Gabriele N. Tornetta. Austin: A frame stack sampler for cpython. https://github.com/P403n1x87/austin.

[60] Uber. Pyflame: A ptracing profiler for python. https://github.com/uber-archive/pyflame.

[61] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22, 2011.

[62] Jeffrey S Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985.

[63] Chao-Ming Wang. Numpydl: Numpy deep learning library. https://github.com/oujago/NumpyDL.

[64] Shasha Wen, Milind Chabbi, and Xu Liu. Redspy: Exploring value locality in software. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 47–61, 2017.

[65] Shasha Wen, Xu Liu, John Byrne, and Milind Chabbi. Watching for software inefficiencies with witch. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 332–347, 2018.

[66] Victor Zhou. A convolution neural network (cnn) from scratch. https://github.com/vzhou842/cnn-from-scratch.

[67] Weijie Zhou, Yue Zhao, Guoqiang Zhang, and Xipeng Shen. Harp: Holistic analysis for refactoring python-based analytics programs. In *42nd International Conference on Software Engineering*, 2020.

[68] YiFei Zhou. Guppy 3: A python programming environment and heap analysis toolset. https://github.com/zhuyifei1999/guppy3/.