

Best-Effort Lazy Evaluation for Python Software Built On APIs

Guoqiang Zhang, Xipeng Shen
North Carolina State University
Raleigh, NC

ABSTRACT

This paper focuses on an important optimization opportunity in Python-hosted domain-specific languages (DSLs): the use of laziness for optimization, whereby multiple API calls are deferred and then optimized prior to execution (rather than executing eagerly, which would require executing each call in isolation). In existing supports of lazy evaluation, laziness is “terminated” as soon as control passes back to the host language in any way, limiting opportunities for optimization. This paper presents Cunctator, a framework that extends this laziness to more of the Python language, allowing intermediate values from DSLs like NumPy or Pandas to flow back to the host Python code without triggering evaluation. This exposes more opportunities for optimization and, more generally, allows for larger computation graphs to be built, producing 1.03-14.2X speedups on a set of programs in common libraries and frameworks.

1 INTRODUCTION

Modern software is built upon APIs. Although APIs typically encapsulate highly optimized code, suboptimal usage of APIs can cause large performance degradation. Such a problem is especially common in Python programs, as Python has become the host language of many popular libraries or domain-specific languages (DSL) targeting performance-demanding tasks, such as NumPy [25], Pandas [18], PySpark [26], Tensorflow [6], and PyTorch [23].

Suboptimal usage of APIs typically involves a sequence of API calls. For illustration purpose, we show a simple example in Figure 1(a). While the code is simple, it suffers the performance flaw of a redundant temporary value: S1 creates an object and assigns it to `x`, but after `x` points to another object in S2, Python garbage collector (GC) releases the former object as it now has zero reference count. The program can be optimized by replacing the second statement with an in-place operation: `numpy.add(x, c, out=x)`. The argument `out=x` instructs `numpy.add` to reuse `x` to store the result. The optimization not only improves data locality, but also reduces memory usage.

Existing work to tackle the problem of suboptimal API sequence relies on lazy evaluation. Several API sets, such as Spark [26], Tensorflow [6], and WeldNumpy [22], have been designed and implemented in that way. They designate some APIs as eager APIs and the rest as lazy APIs. Invocations of lazy APIs only log the APIs in a certain form rather than execute them. Once an eager API is encountered, the logged sequence of APIs will be optimized together and then executed. For instance, Figure 1b shows the WeldNumpy version of the code in Figure 1a; the two add operations are not evaluated until S4; before the evaluation happens, the WeldNumpy runtime optimizes the two add operations and avoids the unnecessary object creation for `x` in the second add operation.

A fundamental problem underlying the API-based lazy evaluation is the data dependence that arises between the invocations of the APIs and the host Python code. Figure 1c gives an illustration.

```
S1: x = numpy.add(a,b)      S1: wa = weldarray(a)
S2: x = numpy.add(x,c)      S2: x = weldnumpy.add(wa,b)
                             S3: x = weldnumpy.add(x,c)
                             S4: a[0] = 0
                             S5: x.evaluate()
(a)
S1: wa= weldarray(a)       (c)
S2: x = weldnumpy.add(wa,b)
S3: x = weldnumpy.add(x,c)
S4: x.evaluate()
```

Figure 1: NumPy example and WeldNumpy variants

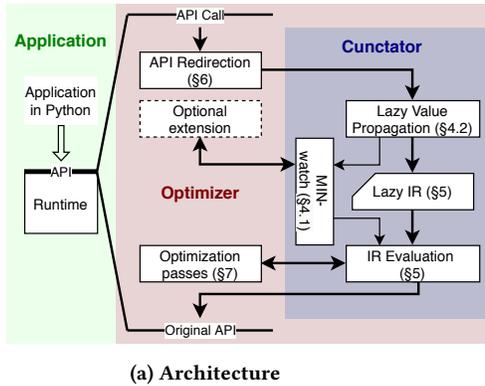
Listing 1: A Spark program with performance issues that are hard to automatically optimize away

```
1 lines = sc.textFile("foo")
2 ws = lines.flatMap(lambda l: l.split())
3 ws = ws.filter(lambda x: re.match("[^\w]+$", x))
4 word_count = ws.count()
5 total_len = ws.map(lambda w: len(w)).sum()
6 avg_len = total_len / word_count
```

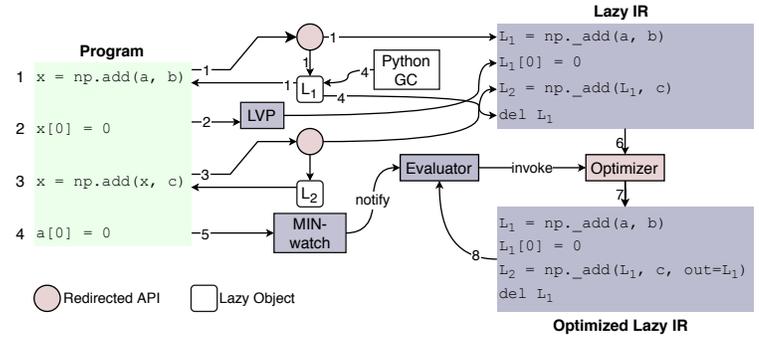
Compared to Figure 1b, the difference is that a Python statement S4 updates the input of S1 before `evaluate()`. Python statements, by default, are eagerly evaluated. But as the `weldnumpy.add` API is lazily evaluated, S2 would end up using the wrong values of `a`.

Existing frameworks either leave the issue to the programmers (e.g., in WeldNumpy [22]), relying on them to put in eager APIs at the right places, or design the library such that any API that might incur dependencies with the host code is designated as an eager API, regardless of the context (e.g., in Spark [26] or TensorFlow [6]). The former increases programmers’ burdens, while the latter often misses optimization opportunities due to its conservative design.

Listing 1 shows an example in *Spark*. It loads a text file (Line 1), splits the lines into words (Line 2), filters out illegal words (Line 3), counts the number of words (Line 4), sums the lengths of all words (Line 5), and finally outputs the average word length (Line 6). In *Spark*, the APIs `textFile`, `flatMap`, `filter`, and `map` are always lazily evaluated; both `count` and `sum` are always eagerly evaluated APIs because they return values to the host code and hence the value, in general, could potentially be operated on by the host code. When an eager API is invoked, *Spark* fuses relevant lazy APIs together into a pipeline; intermediate results are not cached. As there are two eager API calls, the lazy operations `textFile`, `filter`, and `flatMap` are evaluated twice at lines 4 and 5. The solution from *Spark* is to introduce extra APIs such that programmers can use them for caching. This “band-aid” solution further increases the burdens of programmers, who now need to be concerned of not only the usage of the many existing APIs but also the best places to use the caching APIs.



(a) Architecture



(b) Running example. Numbers on directed edges indicate the order of actions.

Figure 2: Overview of Cuncator.

Our study (§9) shows that these limitations prevent existing frameworks from tapping into the full potential of lazy evaluations for Python+API programs, leaving up to 14X performance improvement yet to harvest.

The primary goal of this work is to create a solution that overcomes the limitations of the existing methods for enabling lazy evaluation for Python+API programming. The principles for developing our solution are two fold: (1) It should be automatic such that programmers do not need to worry about manually finding the best places in their code to insert APIs to trigger evaluations; (2) it should be effective in postponing API evaluations to places as late as possible to maximize API optimization opportunities.

The key to both principles is to effectively analyze data dependencies between the host code and the APIs in a Python program. The problem is challenging. Many features of Python, such as dynamic typing and reflection, make analysis of the host code difficult. The difficulty is exacerbated by the extra need to analyze library APIs and their interactions with the host code. The lack of such automatic data dependence analysis is plausibly one of the main reasons for the unsatisfying solutions being used today.

In this work, we address the challenge by developing a *minimum interference runtime watching scheme* (MIN-watch for short). The basic idea underlying MIN-watch is simple, tracking data accesses at runtime to detect data dependencies. The novelty is in how MIN-watch makes the tracking efficient and effective for sound dependence detection in the context of Python+API programs. MIN-watch does it by taking advantage of the characteristics of Python and the special needs in lazy evaluation for Python+API. It is resilient to Python language complexities. It minimizes runtime overhead through a focused tracking scope in data and an efficient runtime checking mechanism (bit-level flagging and deferred flag resetting). It meanwhile imposes near-zero burdens on programmers. MIN-watch is based on a dependence theorem we introduce to formulate the correctness of lazy evaluation in this host+API context (§3).

Based on MIN-watch, we further develop Cuncator, a software framework for materializing the extended lazy evaluation. Cuncator consists of an intermediate representation (lazy IR) for the deferred operations, a lazy IR evaluator, a class that delegates the results of deferred operations and postpones operations applied to itself, and a set of interfaces for redirecting API calls and registering optimizers. With these components together, Cuncator provides

programmers the conveniences of enabling the automatic *Best-Effort Lazy Evaluation (BELE)* for a Python library and harvesting the optimization benefits.

To demonstrate the usefulness of Cuncator, we implement four optimizations enabled by BELE for three API packages (numpy, Spark, Pandas). Experiments on 15 programs show that the optimizations generate 1.03-14.2X speedups. Stress testing shows that the overhead of Cuncator is no greater than 2.25% (in its default setting).

In summary, this work makes the following major contributions:

- It introduces the concept of Best-Effort Lazy Evaluation, and shows that MIN-watch is effective in enabling data dependence analysis for Python+API programs to support BELE.
- It develops the first software framework to support Best-Effort Lazy Evaluation for Python+API programs.
- It demonstrates the effectiveness of the techniques in enabling optimizations of Python+API programs.

2 OVERVIEW

Figure 2a illustrates Cuncator’s architecture. When an application invokes a DSL API, the API call is redirected to a Cuncator optimizer. Instead of evaluating the API, the optimizer records the API in the form of Lazy IR (§5), and returns a *lazy object*. The lazy object supports *Lazy Value Propagation* (LVP, see §4.2), which tries to propagate a new lazy object when an operation is applied to the lazy object. Cuncator employs MIN-watch (§4.1) to monitor accesses to objects related with the deferred operations. When MIN-watch encounters host statements or APIs that prevent further delays (based on dependence theorems in §3), it triggers the evaluation of the deferred operations. During the evaluation, the Cuncator optimizer applies optimization passes (§6) onto the IR, and then invokes the original DSL APIs for evaluation. To apply Cuncator to a domain, the developers of the DSL optimizer only needs to use Cuncator interfaces to specify redirections of the domain APIs, to support MIN-watch for some common types, and to write domain-specific optimizations. The extra work an application developer needs to do is just to import one or several modules.

Figure 2b shows the execution flow of a NumPy program with Cuncator. First, the `np.add` in line 1 is redirected to Cuncator optimizer, which records the API call as a lazy IR instruction and returns a lazy object L_1 . Note, the assignment to x is not deferred

but executed, and x now points to the lazy object L_1 . The optimizer also sets up the two arguments, a and b , for watching. At line 2, because x is lazy, the Lazy Object class automatically captures and logs this operation and defers its execution. Line 3 is similar to line 1, and Cunctator defers and logs the operation. That assignment makes x point to L_2 ; L_1 's reference count reduces to zero, which triggers Python's garbage collection on L_1 . L_1 's deconstructor, however, rather than deconstructs L_1 , defers the deconstruction and inserts a `del` instruction into the IR. Line 4 tries to update a , which is captured by MIN-watch, which triggers the evaluation of all the deferred operations. The evaluator first invokes the optimizer, which reduces redundant temporary variables, and then evaluates the operations.

Before presenting the details of Cunctator, in the next section, we first define some terms and prove a dependence theorem that formulate Cunctator's correctness.

3 DEPENDENCIES BETWEEN OPERATIONS

To ensure the correctness of Cunctator, one key aspect is to properly manage the dependencies between postponed API calls and eagerly executed statements. We first introduce a set of terms that are used in the following discussions.

Terminology. Unless otherwise stated, an *object* denotes a Python object. An *operator* denotes a Python built-in operator. An *operation* denotes the process of applying an operator to its operands. For example, `foo.bar()` consists of two *operations*: The `.` operator is applied to `foo` and `"bar"` to return a function object, which becomes the operand of the `()` operator. We in addition introduce the following terms.

- *Contents of an object*: All in-memory states that could be potentially accessed or updated *directly* through the object's fields and methods. Take the `list` object `["foo", "bar"]` as an example—its contents are the references to its two elements, the string objects, rather than the characters of the strings. By this definition, two objects could share contents, namely, their methods or attributes could access or update the same in-memory state.
- *Sealed object*: An object that shares no content with other objects. This means the contents of a sealed object can be accessed or updated only by its own attributes or methods.
- *Domestic object*: An object whose attributes and methods access no external resources (e.g., files and network) but only the memory space of the current process. We are interested in sealed and domestic objects (e.g., `list` objects).
- *Dependents of an object*: The object itself and the objects referred to in the contents of the object. For example, the dependents of a `list` are itself and its elements.
- *Relatives of an object*: The union of the dependents of that object and the dependents of its relatives.
- *Regular operation*: An operation is regular if the relatives of its operands and return value are all sealed and domestic, and it only accesses or updates the contents of its operands' relatives or newly created objects during the operation. In most cases, a DSL API call is a regular operation.

Without noting otherwise, the following discussions assume regular operations and sealed and domestic objects and there is no exceptions. Section 4.4 discusses exceptions and other complexities.

Dependency types. Based on the above definitions, we classify potential dependencies between an API call O_A and a statement O_B into three types:

- *Return-access*: The return value of O_A is accessed (read or written) by O_B , as illustrated by the top left example in Figure 3.
- *Input-update*: A relative of O_A 's operands is updated in O_B , illustrated by the bottom left example in Figure 3.
- *Update-access*: O_A updates a relative of its operand I and O_B accesses that relative, illustrated on the right side in Figure 3. Cunctator uses a conservative version of this definition, which forgoes the requirement of the two relatives being the same. It simplifies runtime checking as shown later.

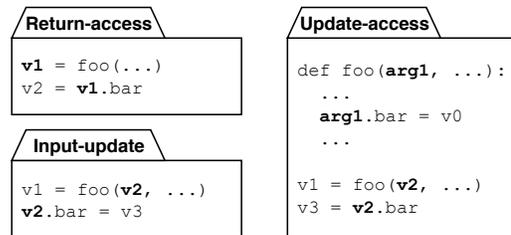


Figure 3: Three types of dependencies

Dependency Theorem. This part presents the dependence theorem governing the validity of lazy evaluation for APIs, which underpins the design of Best-Effort Lazy Evaluation.

LEMMA 1. *For an API call A followed by a statement B , deferring the execution of A to a point after B does not change the data dependencies between them if there are no return-access, input-update, or update-access dependencies between them.*

The lemma comes from the observation that for the properties of sealed and domestic objects and regular operations, the three types of dependencies cover all possible data dependencies (true dependencies, anti-dependencies, output dependencies) [7] between two statements.

THEOREM 3.1. *For an API call A followed by a sequence of statements S , deferring the execution of A to a point after S is valid if there are return-access, input-update, or update-access dependencies between A and none of the statements in S .*

This theorem is derived from the classic *fundamental theorem of dependence* [7], which states the following: Any reordering transformation that preserves every dependence in a program preserves the meaning of that program. Deferring executions is clearly a kind of reordering transformation. The deferring does not cause any dependence changes according to Lemma 1 for none of the three types of dependencies exist between A and S . The theorem hence holds.

Theorem 3.1 is essentially a variant of the *fundamental theorem of dependence* in the context of API lazy evaluation; the benefits of having it are however significant. It entails what types of dependencies are needed to consider during lazy evaluation, and what

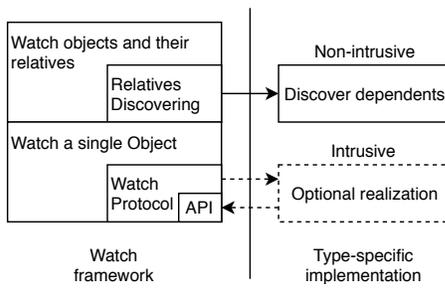


Figure 4: The watch framework

set of data objects are needed to watch, which lay the foundation for the design of MIN-watch and BELE in the next section.

4 BEST-EFFORT LAZY EVALUATION (BELE)

The purpose of BELE is to defer DSL API calls until the necessary moment. The central challenge that BELE confronts is to satisfy three mutually constrained requirements: First, BELE has to ensure correctness of the program. Second, the deferring period, or *laziness*, should be as long as possible to harvest optimization opportunities. Finally, the overhead should be low.

To address these challenges, we introduce minimum interference runtime watching (MIN-watch) in §4.1 to detect, with low overhead, *input-update* and *update-access* dependencies between deferred API calls and host code. In addition, Cuncator §4.2 employs *lazy value propagation* (LVP) to manage *return-access* dependencies while ensuring enough laziness. The overheads of Cuncator and strategies to control them are discussed in §4.3. Finally, §4.4 describes how to handle special scenarios.

4.1 Minimum interference runtime watching (MIN-Watch)

Based on Theorem 3.1, the key for BELE is in detecting data dependencies. MIN-watch takes the way of runtime object watching, which makes it not subject to the language complexities Python imposes on compilers or other static methods.

4.1.1 Overview of MIN-Watch. What makes MIN-watch distinctive over common runtime access tracking is the strategy it employs, which takes advantage of the characteristics of this problem setting and Python language properties, and uses a lightweight type-based scheme for non-intrusive implementation. Specifically, the design of MIN-watch draws on three observations: (1) In Python, most memory accesses go through object interface with multiple layers of redirection and procedure abstractions, hence a much reduced sensitivity to runtime memory access tracking overhead compared to many other languages and settings. (2) The key to BELE is the dependence between API and host. So many data accesses that are irrelevant to such dependencies can be exempted from tracking. (3) Python object assignments and parameter passing are both through references; so to check dependencies related to an actual object, it is not necessary to track references to it, if the watching scheme is put onto that object.

Built on the observations, MIN-watch has the following features: (1) By focusing on API to host dependencies and Theorem 3.1, MIN-watch concentrates runtime watching on only relevant data objects. (2) It employs an efficient runtime checking mechanism (bit-level

```
# in module numpy
# gIR: the global IR scratchpad
def add(a, b):
    setupWatch(a, True)
    setupWatch(b, True)
    id = gIR.add_call(...)
    return Lazy(gIR, id)

def setupWatch(obj, watchUpdateOnly):
    for r in findRelatives(obj):
        r.__set_watch__(watchUpdateOnly)
```

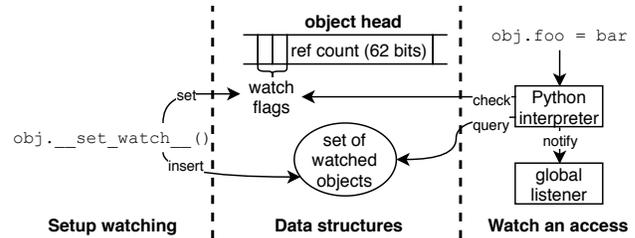
Figure 5: A high-level illustration of how MIN-watch works for API `numpy.add(a,b)`

Figure 6: Default implementation of watch protocol

flagging and deferred flag resetting) via Python interpreters to minimize interference to program executions. (3) It employs a type-based approach to enabling runtime object watching, but does it in a non-intrusive way such that application developers need to make no changes to the implementation of a data type for the approach to take effect. Moreover, the utilities in Cuncator simplify the work an optimizer developer¹ needs to do to enable MIN-watch (and BELE) for a domain DSL. The first two features make MIN-watch efficient, and the other features make it easy to use.

Figure 5 uses `numpy.add(a, b)` as an example to illustrate at a high level how MIN-watch works. The API was overloaded such that when the API is called in a program, instead of doing the computation of arrays addition, it sets up objects *a* and *b* and their relatives for runtime watching via function `setupWatch`. Function `setupWatch` calls a function `findRelatives()` to go through each relative of an object, and calls `__set_watch__` of that object to set it up for runtime watching. The setup process flags some special bits in the object header such that the extended Python interpreter, when executing a statement, can recognize such objects and invoke Cuncator lazy evaluation listener to evaluate deferred operations.

We next explain MIN-watch in detail, starting with the basic watch protocol on a single object (§4.1.2), and moving on to describe the procedure in finding and watching all relatives (§4.1.3).

4.1.2 Watch protocol. Cuncator adds the following method into the root class in Python:

```
def __set_watch__(self, watchUpdateOnly):
    # pseudo-code
    self.watch_flag = WATCH_UPDATE_ONLY
    g_watch_set.insert(self)
```

The parameter `watchUpdateOnly` determines whether the object should be watched for read and write accesses (`false`) or just writes (`true`).

¹Please note the differences between an application developer and an optimizer developer.

Figure 6 depicts the actions when the protocol takes place. In the setup time (e.g., when `numpy.add(a,b)` is called in the example in Figure 5), `__set_watch__` sets two bits in the object head to indicate whether the object is to be watched for update only (01), read/update (10), or nothing (00). These bits help the interpreter determine the corresponding action quickly. Our implementation borrows the first two bits of the reference count field of an object. That saves extra memory, and also helps ensure that most third-party binary libraries are still compatible by keeping the length of the head unchanged. The method `__set_watch__` in addition adds the object into a global set `watchSet`. It is for fast resetting at the time when the deferred operations are evaluated, which will be elaborated in §4.1.4.

We extend Python interpreter such that it notifies Cunctator lazy evaluation listener when the content of an object that is being watched is accessed by a bytecode (e.g., `LOAD_ATTR` and `STORE_SUBSCR`).

The default implementation of `__set_watch__` ignores the parameter `watchUpdateOnly` (i.e., assuming it is false). It is because when just encountering the statement, for some data types, the interpreter sometimes cannot tell whether the access would update the object. (Note that it is legitimate in Python for a seemingly read-only (e.g., `foo.bar`) operation to update the object.) This conservative implementation may reduce the laziness but won't cause correctness issues. For a given data type, the optimizer developer can choose to customize its `__set_watch__()` method and other methods to enable a more precise treatment.

It is worth noting that if an object is not sealed, accesses or updates to the object's contents through other objects are not watched with the default implementation. This is fixed in the upper relatives discovering component by not supporting the specific type, which causes the watch process to fail and thus triggers eager evaluations of involved operations.

4.1.3 Watching relatives. With the watch protocol, we can watch a single object. MIN-watch requires watching all the relatives of an object of interest as Figure 5 has shown. The watch framework holds a registry that can register user-defined procedures to discover dependents of specific types. Through recursively calling registered procedures, all relatives of an object can be found. For example, a list *A* contains objects *B*, *C*, and *D*, and *D* is another list that contains *E* and *F*. Then, the dependent-discovering procedure registered for type `list` returns *B*, *C*, and *D* for object *A*, after which a recursive call of the procedure for *D* returns *E* and *F*. Typically, a type-specific dependent-discovering procedure is easy to implement. For example, the procedure for `list` is as simple as²:

```
def list_deps(l):
    for e in l:
        yield e
```

Algorithm 1 shows the process of setting up to watch an object's relatives. The `SETWATCH` procedure first checks existing watch flags and returns in two cases: The first case is that the object is watched for access, when the procedure returns disregarding the parameter `watchUpdateOnly`. The second case is that the object is watched for updates only and the parameter `watchUpdateOnly` is `True`. In other cases, the procedure sets up to watch the object through the

²`yield` is a Python construct that returns the next value in the next call of its enclosing function.

watch protocol and then recursively calls `SETWATCH` for each of its dependents (except for the object itself). If the type of the object is not registered in the registry, the procedure raises an exception, which will be caught by Cunctator to trigger an eager evaluation of the involved operation.

Algorithm 1 Set up to watch an object's relatives.

```
1: DepReg ← the registry for discovering dependents
2: procedure SETWATCH(obj, watchUpdateOnly)
3:   if obj is watched for access then return
4:   if watchUpdateOnly ∧ obj is watched then return
5:   obj.__set_watch__(watchUpdateOnly)
6:   for all d ∈ DEPS(obj) do
7:     SETWATCH(d, watchUpdateOnly)
8: procedure DEPS(obj)
9:   if type(obj) is registered in DepReg then
10:    return DepReg.getHandler(type(obj))(obj)
11:  else
12:    raise an exception
```

When Cunctator defers an operation, it invokes the `SETWATCH` procedure for all the operands except for lazy values, whose potential dependency is handled by lazy value propagation.

4.1.4 Unwatching objects via deferred flag resetting. After the deferred operations are triggered to get evaluated (or when a watch procedure is aborted because of an unsupported type, see §4.1.3), Cunctator would need to clear the watch flags of all watched objects. Otherwise, later accesses to them would trigger unnecessary evaluations. Going through all the objects could incur substantial overhead. Cunctator circumvents it by introducing a global `watchSet`. Recall in Figure 6, `__set_watch__()` puts an object to be watched into `watchSet` at setup time. That set is emptied once the evaluation of deferred operations is triggered. Python interpreter, when it encounters an object with watching bits set, would check whether that object is within `watchSet`. If not, it cleans the watch bits; otherwise, it invokes Cunctator lazy evaluation listener.

4.2 Lazy value propagation

Return-access dependency is easy to detect for lazily evaluated operations, since all subsequent visits to the return value fall to the actually returned lazy object, which can trigger the evaluation whenever it is used, similar to how the modifier `lazy` works in some other popular languages (e.g., Scala and Swift). However, too often, a lazy object is used shortly after it is returned. For example, in the statement `(a, b) = lazy_func()`, the lazy object is used to unpack its elements right after it is returned from `lazy_func()`. In such cases, an evaluate-when-used semantics of lazy objects results in short-lived laziness, and leaves no optimization opportunities. As a solution to ensure sufficient laziness, we enhance Python with lazy value propagation (LVP), which propagates new lazy values for most operations applied to existing lazy values. The return-access dependency is thence not violated, since the operation that uses the return value is deferred as well.

When a lazy value is being operated, LVP records the operation into the lazy IR and then returns a newly created lazy object. One problem that LVP has to solve is when the propagation should stop – in other words, when the true evaluation should be triggered.

Listing 2: Lazy value propagation

```

class Lazy:
    def __init__(self, ir, v):
        self.__ir, self.__v = ir, v
    def __add__(self, r):
        if self.__ir.evaluated(self.__v):
            return self.__ir.value(self.__v) + r
        watch(r)
        newv = self.__ir.add_op2('+', self, r)
        return Lazy(self.__ir, newv)
    def __bool__(self):
        return bool(self.__ir.evaluate(self.__v))
    # More overwritten operations
    ...

```

An evident scenario is when a lazy value is used to determine the execution branch (e.g., the `if` condition). Theoretically, we could explore all possible paths and collect the lazy IR in the form of computational tree logic (CTL) [9]. Such exploration, however, would introduce large overhead, while its benefit is unclear. Another situation of stopping LVP involves overhead control (see §4.3).

Cunctator implements LVP within the class of lazy objects through operator overwriting, as shown in Listing 2. A lazy object is bound with one lazy IR variable. The `__add__` function, for instance, overwrites the operation `lazy + r`. Their operands are set up for watching before the operation is recorded in the lazy IR. Other operations are overwritten in a similar way, except for the `bool()` operation, which triggers the evaluation, as the operation is invoked when a value is used for branch selection. Although the `bool()` operation does not necessarily imply branching, it is a good heuristic.

It is worth noting that Cunctator chooses to implement LVP in pure Python for fast prototyping. We plan to re-implement it as a part of Python interpreter in the future. This built-in LVP will have lower overhead and know precisely when a value is used for branch selection.

4.3 Overhead control

If there are too many inexpensive DSL API calls or too many propagated lazy values, generating and evaluating the lazy IR could introduce too much overhead. Although such cases never appear in our experiments, we still introduce a dynamic scheme to prevent it from happening in the extreme cases. Cunctator employs a parameter \mathcal{N}_{IRPS} to control how many lazy IR instructions can be generated per second. Initially, Cunctator sets a variable \mathcal{M} to \mathcal{N}_{IRPS} . When the total number of generated instructions is equal to \mathcal{M} , Cunctator evaluates recorded IR, then it sets \mathcal{M} to $\mathcal{N}_{IRPS} * T$, in which T is the total elapsed time since the first API is deferred. If \mathcal{M} 's new value is smaller than its old value, it indicates that the program is an extreme case; Cunctator disables itself by avoiding API redirection and LVP. In our experiments, we set \mathcal{N}_{IRPS} to 1000.

4.4 Additional complexities

Exception. Theorem 3.1 assumes that neither O_A nor O_B raises exceptions. Exceptions could direct the execution to their handlers. If there is no exception handler set up, which is the case for most of the DSL programs we encountered, any raised exception would cause the program to crash. Thus, Cunctator disregards potential exceptions of an operation when there is no installed exception

handler. When the current context has exception handlers, Cunctator disables BELE, and thence, all operations are eagerly evaluated. Cunctator checks the currently installed exception handlers through an interface added to Python interpreter.

External dependency. Theorem 3.1 assumes that O_A and O_B are not dependent on each other through external resources (e.g., one writes to a file, and the other reads the file). Cunctator considers that the information of whether lazily evaluated APIs access external resources as domain knowledge and relies on the optimizer's developer to provide the knowledge. If none of the lazily evaluated operations access external resources, there is no external dependency. Otherwise, a monitor that watches the program's system calls could notify Cunctator when the program tries to access external resources; Cunctator can then avoid deferring the operations.

Unwatchable objects. Although the watch framework works in most cases, there are objects that cannot be watched because they are not sealed or domestic. For example, if an object holds a segment of shared memory, an update to the shared memory in another process will not notify the listener. In addition, it is impractical to implement MIN-watch for all potential types; thus, some uncommon types may not support MIN-watch. Any kind of unwatchable object causes an involved operation to be eagerly evaluated.

Loss of seal. A sealed object may become not sealed at runtime. For example, `numpy.ones()` creates a sealed object O ; however, `O.reshape()` may create a new object P that shares O ' data buffer (not a Python object) through pointer alias, rendering that O is not sealed any more, and updates to the data buffer by operating P cannot be monitored by watching O . Therefore, if a type supports MIN-Watch, and there is a method of the type leaks the content, the method needs to mark the involved object as unwatchable by setting watch flags' value to 11 (see §4.1.2). Subsequent attempts to set watch on O will enforce eager evaluation.

5 INTERMEDIATE REPRESENTATION

This section gives details on the design of the lazy IR in Cunctator. The lazy IR has a static single assignment (SSA) form. Each instruction is a 4-tuple:

$$\langle ID, OP, Operands, Annotation \rangle$$

ID is a globally unique name, which represents the result of current instruction. OP is the operator, such as '+', '.' (attribute access), '[' (array alike access), '()' (function calls). $Operands$ are stored as a list. $Annotation$ can be used to store any extra info that the optimizer may use. For an API call, for instance, it is logged as a call instruction (OP is '()'), the function pointer is stored in the $Operands$ field along with the function's arguments, and the API name is put into the $Annotation$ field.

An operand of an IR instruction could be either a lazy value or a non-lazy value. When an operand is a lazy value, the instruction stores its ID . For a non-lazy value, the instruction stores a reference to it. (In our discussion, L_x denotes a lazy value, N_x a non-lazy value, and V_x can be any value.)

Cunctator provides a simple interface for optimizer developers of a DSL to register optimization passes. Each optimization pass accepts a sequence of IR instructions as input, and outputs an optimized sequence. Registered optimization passes are chained in order. During an evaluation, the sequence of all recorded IR

instructions since the last evaluation is passed down through all optimization passes.

6 OPTIMIZERS

Cunctator is an enabler. By enabling BELE, it paves the way for many optimizations that are not supported by existing DSL frameworks. We have implemented proof-of-concept DSL optimizers for NumPy, Pandas, and Spark. These optimizations fall into two categories: *in-language* optimization and *cross-language* optimization. The in-language optimization tries to identify inefficient API uses and replace them with some other APIs of the DSL. The cross-language optimization tries to replace APIs of one DSL with APIs of another DSL. Two techniques for each category are illustrated in the following sections.

6.1 Reducing temporary variables in NumPy

Redundant temporary variables are a performance issue in many NumPy programs. They impair performance in two ways. First, value assignment to a new variable has worse data locality than an in-place value update. Second, depending on the array size, a temporary variable can consume a lot of memory and thus increase peak memory usage.

When the API call trace is collected as lazy IR in Cunctator, an optimizer can easily optimize away a redundant temporary variable through pattern matching and IR rewriting. At the pattern matching stage, the optimizer locates a redundant temporary variable L_a if the following conditions are all satisfied:

- L_a 's value is initialized from the result of an operation that generates a new value rather than performing in-place update.
- L_a participates in no in-place updating operations.
- L_a is passed to an operation O that generates a new value L_b , and O has a counterpart O' that performs an in-place update.
- After being used in operation O , L_a is deleted and participates in no other operations.

At the IR rewriting stage, the optimizer replaces the operation O with O' , which saves the result to L_a . Figure 2b shows an example of this optimization technique.

6.2 Adaptive caching for PySpark

PySpark is Spark's Python programming interface. Although Spark's runtime employs lazy evaluation to optimize its API call sequences, it fails to handle performance flaws similar to that in Listing 1, because an eager API does not know whether the intermediate result of a lazy API will be used by a subsequent eager API.

With Cunctator, the performance problem in Listing 1 can be optimized away by adding cache operations for intermediate results used by more than one eager operation, as shown in Figure 7. The IR shown on the left side of the figure is collected by Cunctator. Note that `del` instructions are omitted for concision. Based on the collected IR, the optimizer constructs a data flow graph for all Spark operations. If two or more eager operations share a common ancestor, the optimizer inserts a cache operation at the fork.

Another similar performance problem involves unnecessary cache operations, namely, cache operations for intermediate results used by only one eager API. Such operations introduce unnecessary memory writing and consume a lot of memory. Based on the

same graph analysis as was used for inserting cache operations, the optimizer can identify and remove unnecessary cache operations.

6.3 From NumPy to WeldNumpy

WeldNumpy [5] was developed as a replacement for NumPy with better performance, which was achieved via two main techniques. First, WeldNumpy exploits lazy evaluation instead of eager evaluation, which is used in NumPy. Second, WeldNumpy implements its APIs using Weld IR [22], an intermediate representation designed for parallel data processing. Through lazy evaluation, the IR fragments of invoked APIs are combined into an IR program. During a true evaluation, the IR program is compiled and optimized for native hardware. Some major optimization techniques are loop fusion, loop tiling, and vectorization. WeldNumpy provides `weldarray`, a subclass of NumPy's `ndarray`. Thus, after an `ndarray` is converted to a `weldarray`, the new object supports most NumPy operations and enjoys improved performance.

However, as WeldNumpy is lazily evaluated, it requires users to explicitly call `evaluate()` when necessary. The `evaluate()` method should not be invoked too often; otherwise, the WeldNumpy runtime misses optimization opportunities and introduces overheads of compiling the Weld IR. Neither should it be too late as that would cause errors. Thus, a NumPy-to-WeldNumpy translator needs to figure out the appropriate positions to insert `evaluate()`.

The evaluating positions can be located by identifying *exposed lazy variables*. A variable is exposed if it is used beyond the DSL's APIs, which means the true value of the variable may be required, or it is alive at the end of the collected lazy IR, which allows potential external usage of the variable during subsequent execution. When a variable is exposed but lazy, it should be explicitly evaluated. Such variables can be identified within an one-pass scan of the lazy IR. The translator can then insert `evaluate()` for these variables.

Figure 8 shows an example of translating NumPy to WeldNumpy. The translated IR first converts L_1 , an `ndarray`, to a `weldarray`, such that L_2 , L_3 , and L_4 enjoy WeldNumpy's optimization. However, `np.array_equal()` is not supported by WeldNumpy; thus, operand L_3 has to be evaluated before being passed. While L_4 is explicitly evaluated because of potential exposure, L_2 remains lazy, since it is deleted and has no external use.

Such a translator leverages the laziness analysis enabled by Cunctator. It might be tempting to think that the translation could be done through a compiler without Cunctator. Note that that compiler would have to face the laziness analysis problem as Cunctator tackles; if it ignores that, its replacement of an eagerly evaluated NumPy API with a lazy evaluated WeldNumpy could cause errors. Doing the laziness analysis is difficult for a compiler for the many challenges (e.g., Python complexities, API-host interplay) mentioned in the introduction section.

6.4 From Pandas to Spark

Both Pandas and Spark provide a class called `DataFrame`. They both represent logical tables, which have named and typed columns. While Pandas' operations in `DataFrame` are eagerly evaluated, most of Spark's `DataFrame` methods are lazily evaluated. During a true evaluation, Spark employs a code generation technique [20] to compile an operation sequence. Such technique renders the Spark `DataFrame` API a performant replacement of Pandas. In addition,

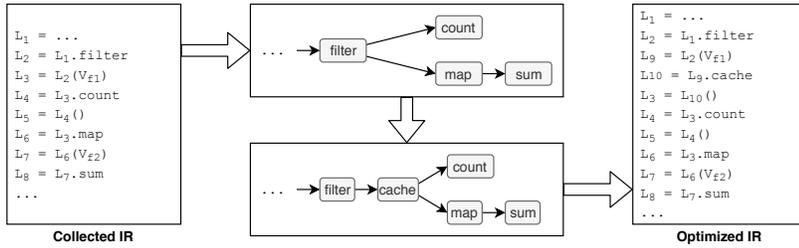


Figure 7: Adding cache operation in Spark

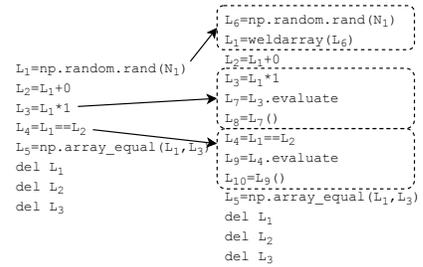


Figure 8: NumPy to WeldNumpy

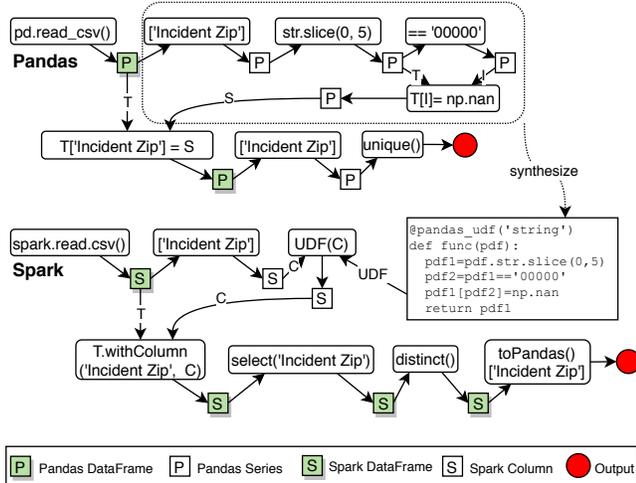


Figure 9: Pandas to Spark

Spark has native support for Pandas, including Pandas UDF [1], by which a user can apply Pandas operations to a Spark Column. Spark also contains type casting APIs that convert between Spark DataFrame and Pandas DataFrame. These features offers conveniences to translation of a Pandas program to a Spark program.

Similar to NumPy to WeldNumpy, the laziness analysis by Cunctator puts down the basis for the development of an automatic Pandas-to-Spark translator. Our prototype focuses on a common use pattern of Pandas: A program first loads a file as a DataFrame, then performs some operations on it, and finally outputs the result. In such a pattern, only one DataFrame object is involved, and no Pandas DataFrame object or Series (typically represents a column) object is exposed, so all instances of the two types only participate in Pandas operations. When such a pattern is matched, the translator tries to optimize it.

During the translation, the Pandas file loading function is replaced by a counterpart in Spark, thus creating a Spark DataFrame. Correspondingly, the Series objects selected from Pandas' DataFrame become Spark Column objects. If there is a sequence of operations on a Series that outputs another Series, the sequence is synthesized into a Pandas UDF for Spark, which is applied to the corresponding Spark Column. If a Series is assigned to the Pandas DataFrame, the corresponding Column is assigned to the Spark DataFrame as well. When an operation on a Series returns an object other than a Series, if the operation (e.g., unique()) has a counterpart in Spark, the Column is applied to the corresponding Spark operation, and then the result is converted to the expected type; otherwise (e.g.,

diff()), the Column is selected and converted to a Series before applying the operation. Figure 9 illustrates the translation for a Pandas program collected from the Pandas Cookbook [2].

7 API REDIRECTION

If a DSL's runtime needs to leverage Cunctator to perform optimization, the optimizer developer needs to redirect the APIs in the DSL through renaming and rewriting. With the Cunctator framework, the process is made simple. For example, to redirect numpy.add in NumPy's runtime, current implementation of numpy.add could be renamed to numpy._add; then, a new implementation of numpy.add will just record API calls as lazy IR instructions and returns a lazy object as shown in Figure 5.

To simplify the process, Cunctator offers some utilities. For the aforementioned example, what the optimizer developer needs to write to put the following into the module *numpy*:

```
def add(*args, **kwargs):
    return lazy_call("numpy.add", numpy._add, *args,
                    **kwargs, kwargsToUpdate={"out"})
```

Method lazy_call is the utility interface that Cunctator offers. Its first argument is for the annotation field of a call instruction (see §5). The argument kwargsToUpdate specifies that numpy._add is going to update only its argument out (if there is one). The call to lazy_call in this example will essentially materialize the method shown in Figure 5.

8 EFFORTS IN APPLYING CUNCTATOR

There is some work needed from the library developers. These work needs to be done only once for a given library; the results can benefit all programs using that library. These one-time work includes: (1) redirecting some APIs that are important for performance (other APIs can be left alone, which will be treated in the same way as host Python code is); (2) supporting MIN-watch for some common types; and (3) implementing optimization passes. Table 1 shows our prototype optimizers' summary in these work. For a common programmer that uses a library, the only change she needs to make to her code is to insert one or several lines of code to import the optimizer.

We initially considered automatic library transformations, but found that it was difficult to do for the complexities of Python. It is, for instance, often impossible for static code analysis to tell whether an argument is subject to modifications, due to dynamic types, aliases, higher-level functions, and inter-procedural complexities. The design choice made in Cunctator is a choice for practicability.

Table 1: Summary of optimizers

Optimizer	#APIs*	Supported types [†]	Opt pass LoC [‡]
NumPy	45	ndarray, dtype	50 (§6.1) 93 (§6.3)
Spark	24	RDD, StorageLevel	201 (§6.2)
Pandas	28	DataFrame, Series	436 (§6.4)

* The number of redirected APIs.

† The types that support dependent discovery.

‡ Lines of code for implementing the optimization passes described in §6.

9 EVALUATION

In this section, we conduct a series of experiments to (1) demonstrate the usefulness of the four optimizations (§6) enabled by Cunctator, and (2) measure the runtime overhead of Cunctator. Time usage is collected by the *timeit* command of jupyter [3], which adaptively chooses a number of repetitions in favor of timing accuracy. Peak memory usage is collected by *memit* command extended by *memory-profiler* [4], which profiles a program’s memory usage line by line. The test platform for NumPy and Pandas is a Linux machine with Intel Xeon Silver 4114 CPUs. Spark programs run on a cluster of eight Linux machines with AMD Opteron 6128 CPUs.

9.1 Optimizers

We collect 15 programs for the experiments that are relevant to the example optimizations described in the previous section; five for each of the three packages (NumPy, Spark, Pandas). Thirteen of them were collected from Github; the other two were the examples used in the earlier sections of this paper—we included them to show the performance benefits from the described optimizations. Table 2 shows the descriptions and inputs of all benchmarks. Figure 11 shows the speedups in different optimizer settings. Detailed results are presented in Figure 10. Each program set is discussed separately in following subsections.

9.1.1 NumPy. The temporary variable reducer (abbr. *reduceTmp*) accelerates all benchmarks, with speedups ranging from 1.19X to 1.54X. The highest speedup is achieved on P1, because its operations are easy to compute and hence the cost of temporary variable is prominent. Besides time benefits, *reduceTmp* also reduces peak memory usage. P4 highlights the reduction with a rate of 75%. The high rate is because of pipelined operations, which means each temporary variable is only used one time and then discarded.

For WeldNumpy converter, we test it with one thread (abbr. *Weld 1T*) and ten threads (abbr. *Weld 10T*) separately. *Weld 1T* shows speedups ranging from 1.01X to 1.95X. Because WeldNumpy currently supports only a limited number of NumPy APIs, for unsupported APIs, it needs to transform data from Weld format to NumPy format to perform the operations, and if necessary, the results need to be transformed back. As WeldNumpy evolves to support more APIs, *Weld 1T* is going to perform better. Moreover, with ten threads, WeldNumpy achieves significant speedups up to 12.5X. Note that Weld has built-in support for multi-threading but NumPy does not.

9.1.2 Spark. The Spark optimizer shows speedups ranging from 1.03X to 1.87X. Among the benchmarks, P1 and P2 lack `cache()`; P3 and P5 have unnecessary `cache()`; P4 has a `cache()` operation at a useless location, while the place that needs `cache()` does not have one. Our optimizer fixes them all. It adds `cache()` to P1 and

P2, removes `cache()` from P3 and P5, and corrects P4 by removing the unnecessary `cache()` and adding one at the appropriate place.

In addition, we test the benchmarks with Cunctator enabled but optimizing pass disabled (abbr. *w/o opt*). The results show no performance degradation. This confirms that MIN-watch has almost no overhead for non-watched objects, as PySpark programs typically invoke user defined functions written in Python frequently.

9.1.3 Pandas. The Pandas-to-Spark optimizer is tested with one Spark thread (abbr. *Spark 1T*) and ten Spark threads (abbr. *Spark 10T*). Note that Spark supports multi-threading but Pandas does not. *Spark 1T* shows speedups on three programs. This is impressive because, while Pandas enjoys the high performance of SIMD instructions, Spark’s query compiler emits Java bytecode. The slowdown on P1 is dominated by Spark’s CSV loader, which performs much worse than Pandas’ loader in this case. Nevertheless, *Spark 10T* enjoys speedups as high as 14.2X on all benchmarks.

9.2 Overheads

For programs cannot be optimized, a major concern is the overhead, which is highly related to the number of lazy IR instructions recorded. To investigate the overhead in different cases, we design an adversarial case for stress-testing:

```
def cps_simulator(M, N):
    for i in range(M):
        numpy.ones(N)
```

The program calls M times of `numpy.ones(N)`, which initializes a vector of size N . By tuning M and N , we can control the number of calls per second (CPS) and the total run time. We then combine some representative values of CPS and overhead control thresholds (see §4.3). For each combination, we run a ten-second experiment with Cunctator. By subtracting the results with corresponding baseline results, we obtain an overhead matrix, shown as Table 3.

The overhead increases when CPS increases. When CPS exceeds the threshold, Cunctator disables itself for the later part of the run; the overhead drops. For the default threshold (1000), the worst overhead is 2.35%, which happens in the extreme case where there are 1000 function calls per second. In practice, a program is unlikely to have a stable CPS rate close to the threshold, thus the overhead is much lower. In addition, it is worth noting that Cunctator is mainly implemented in Python, except for the MIN-watch. If we reimplement some critical components in C, such as lazy IR evaluator, a lower overhead is expected.

It is worth noting that, in the domains that we explored, the number of relatives per object is few, hence our benchmarks bear little overhead of finding relatives. For example, a NumPy array usually has no relative if its buffer belongs to itself, or only one relative if its buffer is from another object. For domains where deeply nested objects are common, the overhead control threshold can be adjusted to fit the need of the domains.

9.3 Threats to Validity

Cunctator is evaluated based on Python 3.7.3, NumPy 1.17.0, WeldNumpy 0.0.1, Pandas 0.25.0, and Spark 2.4.3. The APIs and implementation of these software packages may change after new versions are released. Thus the new releases may invalidate our optimization techniques and evaluation results. Nevertheless, new

Table 2: Descriptions of collected benchmarks

	NumPy	Spark	Pandas
P1	Program in Figure 1a	Program in Listing 1	Find names of median occurrence
Input	vectors of size 10^9	text file of 90MB	csv file of 273MB
P2	Compute vibration energy	Demultiplex a file to multiple files	Find top complaints
Input	vectors of size 5×10^8	xml file of 244MB	csv file of 526MB
P3	Find least-squares solution	Transform data format	Find ratios of noise complaints
Input	vectors of size 5×10^8	json file of 62MB	csv file of 526MB
P4	Find log-likelihood of $\mathcal{N}(\mu, \sigma^2)$	Intersect IDs in two tables	Find unique zip codes after data cleaning
Input	vectors of size 5×10^8	two csv files of 34MB	csv file of 526MB
P5	Compute Black-Scholes model	Find counts of different words	Find top occupations wrt. male ratio
Input	vectors of size 10^8	text file of 460MB	csv file of 240MB

	P1	P2	P3	P4	P5
NumPy time usage (mean \pm std. dev.)					
baseline	12.5 s \pm 2.95 ms	41.1 s \pm 75.4 ms	27.4 s \pm 5.51 ms	43.2 s \pm 57.6 ms	39.1 s \pm 23 ms
reduceTmp	8.14 s \pm 2.14 ms	34 s \pm 15.3 ms	22.9 s \pm 105 ms	33.7 s \pm 26.9 ms	32.9 s \pm 22.7 ms
Weld 1T	6.38 s \pm 34.1 ms	39.1 s \pm 87.6 ms	21.8 s \pm 45 ms	42.7 s \pm 144 ms	22.6 s \pm 28.3 ms
Weld 10T	995 ms \pm 9.24 ms	27 s \pm 142 ms	17.4 s \pm 60.3 ms	15.6 s \pm 44.4 ms	9.94 s \pm 35 ms
NumPy peak memory usage (MB)					
baseline	38181	19131	19130	15316	9213
reduceTmp	30577	11503	15317	3873	6251
Spark time usage (mean \pm std. dev.)					
baseline	31.9 s \pm 123 ms	82 s \pm 698 ms	38.5 s \pm 116 ms	20.9 s \pm 36 ms	49.1 s \pm 272 ms
w/o opt	32.1 s \pm 215 ms	81 s \pm 639 ms	38.5 s \pm 178 ms	21.1 s \pm 75.8 ms	49.2 s \pm 392 ms
optimized	17.1 s \pm 93.4 ms	48.8 s \pm 348 ms	28.8 s \pm 66.8 ms	20.3 s \pm 317 ms	47 s \pm 226 ms
Pandas time usage (mean \pm std. dev.)					
baseline	6.03 s \pm 50.3 ms	9.65 s \pm 21.8 ms	9.8 s \pm 13.4 ms	9.72 s \pm 40.7 ms	7.7 s \pm 37.4 ms
Spark 1T	17.1 s \pm 100 ms	5.51 s \pm 152 ms	3.01 s \pm 127 ms	5.76 s \pm 101 ms	7.45 s \pm 241 ms
Spark 10T	2.92 s \pm 203 ms	951 ms \pm 51 ms	690 ms \pm 26.9 ms	1.3 s \pm 145 ms	1.29 s \pm 59 ms

Figure 10: Benchmark results**Table 3: Overhead (percentage of 10s program runs)**

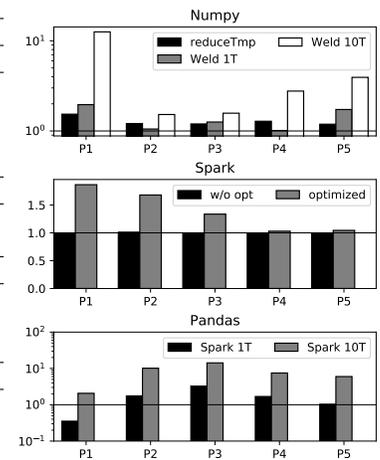
		CPS				
		50	500	1000	2000	10000
Threshold	50	0	0.85	0.85	1.7	0
	500	0.35	3.05	1.05	1.8	0
	1000	0.25	2.35	2.25	1.7	0
	2000	0.15	2.15	2.05	2.5	0.1
	10000	0.15	1.25	1.75	2.9	11.8

patterns of API misuses related to these new releases are likely to appear. Unless the new versions employ a technique similar to BELE, Cunctator can be leveraged to optimize the new patterns.

10 RELATED WORK

Lazy evaluation has been studied extensively in functional programming [8, 12–14, 17]. Scala [21] provides a lazy keyword to express the call-by-need semantics of a variable. However, Scala does not manage the potential side-effect of a thunk, the expression bound to the lazy variable; thus, the correctness of lazy evaluation relies on the programmer. Many hosted DSLs (e.g., Spark [26] and Tensorflow [6]) employ lazy evaluation; their limitations have been discussed in §1.

There are some studies on optimizing DSLs. Weld [22] and its limitations have been discussed and compared with. Delite [24] is a framework for developing Scala-hosted DSLs by leveraging generative programming [10]. Similarly to Cunctator, it lazily evaluates DSL operations and logs them as a form of IR, which will be

**Figure 11: Speedups**

optimized and executed at a certain point in time. However, Delite provides no mechanism to handle the dependencies between DSL operations and their host code.

There are several earlier studies (e.g., telescoping languages [15], Broadway [11]) that try to use manual annotations of libraries to help optimizations. They give no systematic considerations of the host-API dynamic dependencies. Numba [16] is a JIT compiler of Python that targets optimizing manipulations of ndarray in NumPy. AutoGraph [19] employs static code conversion and generative programming to transform PyTorch-style programs to Tensorflow-style programs. All these methods and tools offer a closed set of optimization techniques for specific program semantics. Cunctator does not include any optimization technique but provides a general framework to simplify the creation of a DSL optimizer.

11 CONCLUSION

This paper introduces the concept of BELE, and describes MIN-watch, the first efficient runtime monitoring method tailored to data dependence analysis between host code and APIs for BELE. The paper demonstrates the usefulness of Cunctator in enabling four optimizations that are not supported by existing frameworks, giving 1.03-14.2X speedups. While Cunctator targets Python-hosted DSLs, we believe the potentially applicability of the techniques goes much beyond Python.

REFERENCES

- [1] [n.d.]. Introducing Pandas UDF for PySpark. <https://databricks.com/blog/2017/10/30/introducing-vectorized-udfs-for-pyspark.html>.
- [2] [n.d.]. Pandas Cookbook Example. <https://nbviewer.jupyter.org/github/jvns/pandas-cookbook/tree/v0.1/cookbook/>.
- [3] [n.d.]. Project Jupyter. <https://jupyter.org/>.
- [4] [n.d.]. Python memory-profiler Library. <https://pypi.org/project/memory-profiler/>.
- [5] [n.d.]. WeldNumpy. <https://www.weld.rs/weldnumpy/>.
- [6] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283.
- [7] R. Allen and K. Kennedy. 2001. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers.
- [8] Adrienne Bloss, Paul Hudak, and Jonathan Young. 1988. Code optimizations for lazy evaluation. *Lisp and Symbolic Computation* 1, 2 (1988), 147–164.
- [9] Edmund M. Clarke and E. Allen Emerson. 1982. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs, Workshop*. Springer-Verlag, Berlin, Heidelberg, 52–71. <http://dl.acm.org/prox.lib.ncsu.edu/citation.cfm?id=648063.747438>
- [10] Krzysztof Czarnecki and Ulrich W. Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.
- [11] S. Z. Guyer and C. Lin. 2005. Broadway: A Compiler for Exploiting the Domain-Specific Semantics of Software Libraries. *Proc. IEEE* 93, 2 (Feb 2005), 342–357. <https://doi.org/10.1109/JPROC.2004.840489>
- [12] Peter Henderson and James H. Morris, Jr. 1976. A Lazy Evaluator. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages (Atlanta, Georgia) (POPL '76)*. ACM, New York, NY, USA, 95–103. <https://doi.org/10.1145/800168.811543>
- [13] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. 2007. A History of Haskell: Being Lazy with Class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages (San Diego, California) (HOPL III)*. ACM, New York, NY, USA, 12–1–12–55. <https://doi.org/10.1145/1238844.1238856>
- [14] Thomas Johnsson. 1984. Efficient Compilation of Lazy Evaluation. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction (Montreal, Canada) (SIGPLAN '84)*. ACM, New York, NY, USA, 58–69. <https://doi.org/10.1145/502874.502880>
- [15] K. Kennedy, B. Broom, A. Chauhan, R. J. Fowler, J. Garvin, C. Koelbel, C. McCosh, and J. Mellor-Crummey. 2005. Telescoping Languages: A System for Automatic Generation of Domain Languages. *Proc. IEEE* 93, 2 (Feb 2005), 387–408. <https://doi.org/10.1109/JPROC.2004.840447>
- [16] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: A LLVM-based Python JIT Compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC (Austin, Texas) (LLVM '15)*. ACM, New York, NY, USA, Article 7, 6 pages. <https://doi.org/10.1145/2833157.2833162>
- [17] John Launchbury. 1993. A Natural Semantics for Lazy Evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Charleston, South Carolina, USA) (POPL '93)*. ACM, New York, NY, USA, 144–154. <https://doi.org/10.1145/158511.158618>
- [18] Wes McKinney. 2011. pandas: a foundational Python library for data analysis and statistics. *Python for High Performance and Scientific Computing* 14 (2011).
- [19] Dan Moldovan, James Decker, Fei Wang, Andrew Johnson, Brian Lee, Zack Nado, D Sculley, Tiark Rompf, and Alexander B Wiltschko. 2019. AutoGraph: Imperative-style Coding with Graph-based Performance. In *SysML*.
- [20] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow* 4, 9 (June 2011), 539–550. <https://doi.org/10.14778/2002938.2002940>
- [21] Martin Odersky and Tiark Rompf. 2014. Unifying functional and object-oriented programming with Scala. *Commun. ACM* 57, 4 (2014), 76–86.
- [22] Shoumik Palkar, James J Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, Matei Zaharia, and Stanford InfoLab. 2017. Weld: A common runtime for high performance data analytics. In *Conference on Innovative Data Systems Research (CIDR)*.
- [23] Adam Paszke, Sam Gross, Soumith Chintala, and Gregory Chanan. 2017. Pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration. *PyTorch: Tensors and dynamic neural networks in Python with strong GPU acceleration* 6 (2017).
- [24] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM Trans. Embed. Comput. Syst.* 13, 4s, Article 134 (April 2014), 25 pages. <https://doi.org/10.1145/2584665>
- [25] Stefan van der Walt, S. Chris Colbert, and Gael Varoquaux. 2011. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science and Engg.* 13, 2 (March 2011), 22–30. <https://doi.org/10.1109/MCSE.2011.37>
- [26] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing (Boston, MA) (HotCloud'10)*. USENIX Association, Berkeley, CA, USA, 10–10. <http://dl.acm.org/prox.lib.ncsu.edu/citation.cfm?id=1863103.1863113>