Towards practical incremental recomputation for scientists: An implementation for the Python language

Philip J. Guo and Dawson Engler
Stanford University

Abstract

Computational scientists often prototype data analysis scripts using high-level languages like Python. To speed up execution times, they manually refactor their scripts into stages (separate functions) and write extra code to save intermediate results to disk in order to avoid re-computing them in subsequent runs. To eliminate this burden, we enhanced the Python interpreter to automatically memoize (save) the results of long-running function executions to disk, manage dependencies between code edits and saved results, and re-use memoized results rather than re-executing those functions when guaranteed safe to do so. There is a $\sim$20% run-time slowdown during the initial run, but subsequent runs can speed up by several orders of magnitude. Using our enhanced interpreter, scientists can write simple and maintainable code that also runs fast after minor edits, without having to learn any new programming languages or constructs.

1 Introduction

The massive increase in computing power over the past few decades has enabled scientists across many disciplines to rely more on computational techniques in their research. Many modern scientists in fields like computational biology, pharmacology, neuroscience, and astronomy are spending less time doing physical lab work and more time writing computer programs to process, transform, and extract insights from experimental data.

The style of code that these scientists write differs greatly from code written by professional programmers: *"The primary tension in the life of the scientist-programmer is to just get the science done. What we mean by this is the conflict between getting the coding done as quickly as possible so that you can move onto finishing the science, versus spending that extra week (or however long) making sure that your code is neat, tidy, well-tested and generally a glorious triumph of software engineering."*[2]

These scientists often write programs in what Brandt et al. call an *opportunistic* manner, emphasizing speed and ease of development over code robustness and run-time performance [3]. Being able to quickly implement and refine prototype code is vital to the research process, since the specifications for research code (as compared to production-quality code) are often ill-defined and constantly-changing.

Thus, scientists often choose lightweight tools that allow them to write code quickly, even if their code does not run at optimal speed. For example, several scientists we interviewed before starting this project[1] regularly write single-threaded programs in high-level, dynamically-typed languages such as Python, MATLAB, or R to process experimental data stored as plain-text files (up to several gigabytes in size). Their programs would definitely run faster if they had written, say, multi-threaded C++ code, hand-optimized hot inner loops, and used a relational database configured with advanced indexing features. However, since they are scientists and not programmers by training, they have neither the programming expertise nor the willingness to spend the time to learn these more heavyweight tools and techniques.

Even the few Computer Science researchers we interviewed (in sub-fields like machine learning and software reliability) preferred to use high-level dynamic languages like Python for their research. Although they have the know-how to write faster-running C++ code (and will do so if absolutely necessary), they felt that the convenience and ease of programming offered by languages like Python far outweighed the costs of slower run-time performance.

Problem: Scientific data processing and analysis scripts often execute for tens of minutes to several hours on a single desktop computer, thus slowing down the scientist's iteration and debugging cycle.

[1] In mid-2009, prior to starting implementation work, the first author conducted a series of informal, open-ended needfinding interviews with computational scientists to learn about their data processing workflow.

Why existing solutions are inadequate: To cope with this slowdown, scientists could write extra code to break up their computation into multiple stages and save the intermediate results of each stage to disk. This modularization can speed up subsequent script run-times (since not all stages need to be re-run after each code edit), but it increases code size/complexity and creates dependencies between source code and intermediate results.

In practice, scientists are often hesitant to spend time refactoring their scripts to improve performance, partly due to lack of programming expertise (they learn just enough programming skills to write the simplest possible script that *"just get[s] the science done"*[2]) but also because it can be hard to determine whether it's worthwhile to devote the extra up-front programming time and effort. A senior researcher we interviewed stated[2]:

> *"Many of my bug database mining scripts run overnight, and I hate how they're so slow. I sometimes think about saving intermediate results so that my scripts can run faster. However, I run lots of ad-hoc experiments when exploring my datasets, so I never know ahead of time which results are worth saving. I also don't know how many times I'll end up executing some scripts, so I'm never motivated enough to spend the extra effort on writing and maintaining the caching code."* (Senior software engineering researcher at Microsoft Research)

Another possible way to reduce iteration time is by testing the script on a tiny subset of the original dataset, to ferret out major bugs before running for hours on the full dataset. This is sometimes feasible, but oftentimes there isn't a practical way to avoid running the script on the full dataset, for some of the following reasons [4]: *1.)* The data is too complex or split across multiple files, so it is cumbersome to subset. *2.)* One can only assess correctness by inspecting the script's output from running on the entire dataset; running on a subset could never produce sensible output. *3.)* Data parsing bugs often manifest on aberrant records that appear in the middle of a large dataset (since schemas are often missing or not strictly obeyed), so extracting the first few records to form a tiny test dataset won't reveal those bugs.

Running multiple threads in parallel can also reduce execution time, but it is significantly more difficult to write and debug parallel programs, especially for scientists who are not expert programmers. Even experts prefer to start off by writing single-threaded prototype scripts and then only parallelize later when necessary [16]. When discussing related work (Section 6), we will contrast our proposed solution to parallel programming and other related approaches.

[2]All such quotes are paraphrased from notes that the first author took during needfinding interviews.

1.1 Our proposed solution: Interpreter support for incremental recomputation

We want to enable scientists to iterate quickly on data processing and analysis tasks while keeping their code simple and maintainable. To do so, we modified a programming language interpreter to automatically memoize (cache) the results of long-running function executions to disk, manage dependencies between code edits and saved results, and re-use memoized results rather than re-executing functions when guaranteed safe to do so. Our technique works as follows:

1. The scientist's script runs without modification in our custom interpreter, with a minor slowdown (usually less than 20%) due to run-time monitoring
2. The interpreter memoizes (caches) results of long-running function executions to a file
3. During subsequent runs of the same script (possibly after some edits), the interpreter loads cached results rather than re-computing when guaranteed safe to do so, which speeds up running times
4. The interpreter automatically creates and manages dependencies between executed code and cached intermediate results
5. The interpreter deletes cached results when dependent code or data changes, to ensure that results are identical to those run on an unmodified interpreter

Though our technique is language-independent, we chose to implement it as a modified Python interpreter called `IncPy` (`Inc`remental `Py`thon). In the past decade, Python has been gaining support across many fields of computational sciences because it is easy to learn, supports interactive prototyping, and has mature libraries for scientific data processing (e.g., NumPy, SciPy, Biopython). There is a growing research community around the language: Since 2002, Caltech has hosted the annual *Python in Science* (SciPy) academic conference.

1.2 Potential benefits

The potential benefits of our approach all derive from the fact that we can speed up scientists' iteration and debugging cycles *without requiring them to learn any new languages, libraries, or programming techniques*. When compared with related work (Section 6), we feel that our technique is quite practical since it can be easily deployed to the large number of scientists already using Python in their daily work: We can simply replace their existing Python interpreter with our modified `IncPy` interpreter, and all of their existing scripts should still produce the same results, albeit running significantly faster after minor edits.

The ideal workflow that we would like to support is for a scientist to be able to:

1. Write a 'quick-and-dirty' first version of the script
2. Execute the script using `IncPy`, and wait for, say, an hour to get results
3. Manually interpret results and notice a bug
4. Edit the script slightly to fix that bug
5. Re-execute the script, and wait for *a few seconds* to get new results
6. Enhance the script with some new functions
7. Re-execute the script, and wait for *a few minutes* to get new results

Note that with an ordinary Python interpreter, subsequent re-executions of the script (Steps 5 and 7) will still take an hour regardless of the size of the code edit. It is possible (although tedious and sometimes difficult) to *manually refactor* a script to support efficient recomputation, but `IncPy` does so automatically.

Anecdotes from programmers [3] and findings from psychology experiments [6] suggest that a short iteration cycle not only results in more rapid completion of tasks but also causes people to subconsciously shift to using more effective problem-solving strategies. Since the problem remains fresh in one's (short-term) working memory, one can remain intensely concentrated rather than potentially getting distracted and context-switching while waiting for each subsequent script run to complete.

Shortening execution times can also aid in the debugging process, especially for novice programmers who are not accustomed to using symbolic debuggers or systematically minimizing error-inducing inputs. For many scientist-programmers, debugging consists of cycling between making code edits, re-running the entire script, and waiting for each run to complete. A pharmacology Ph.D. student we interviewed, who is just starting to learn Python for his research, noted:

> *"I often have to re-run my scripts several times before I can get rid of the major bugs. The problem is that I often don't know there's a bug until after the script finishes, which can take several hours or even overnight. Oftentimes I discover some simple semantic error that I can fix with a tiny code edit (like using + instead of -), but then I need to re-run the entire script before I can see the new result. If subsequent runs only took a few seconds, then that could greatly help me in debugging."* (Pharmacology Ph.D. student at UCLA)

A faster iteration cycle could also qualitatively change how people work together on programming tasks. Another Ph.D. student we interviewed described how it could facilitate real-time collaboration:

> *"Whenever I show my data mining results to my advisor, he always suggests to tweak something or other. Sometimes I can make the code tweaks in a few seconds, but re-running the script might take 15 to 30 minutes, so my advisor is not going to wait around for the results. I might not find him again for another few hours or even until the next day, so my iteration cycle is far longer than the script runtime. If my scripts could re-run in seconds rather than tens of minutes, then my advisor and I could collaborate and explore alternatives in real-time."* (Computer Science Ph.D. student at UC Davis)

In the rest of this paper, we will demonstrate our technique on an example (§2), describe its algorithms (§3), our implementation for Python (§4), preliminary experimental results (§5), comparison to related techniques (§6), and conclude with plans for future work (§7).

2 Demonstration on example program

We first demonstrate the basics of our automatic memoization and dependency-tracking technique on an example program before describing its details in Section 3.

Figure 1 shows a Python program that takes a filename as command-line input (`argv[1]`), processes it by calling 3 functions, and prints a numerical result. Let's say that for a given input file `input1.txt`, this program runs for 1 hour and prints `50`. As the program is running, the interpreter memoizes (caches) the values of all arguments that each function is invoked with, and their corresponding return values. The `stageA` function has only one mapping: {`"input1.txt"` $\rightarrow$ `50`}. `stageB` and `stageC` have numerous mappings, one for each unique invoked argument value.

The interpreter also builds a dependency graph as the program executes (bottom of Figure 1), where each function's memoized results can depend on both code and data. Now, when the programmer edits chunks of code or data and runs the program again, the interpreter can consult the dependency graph to determine what functions need to be re-run. For example, if the programmer changed the contents of `masterDatabase.db` or the value of `MULTIPLIER`, then the memoized results for `stageA` and `stageB` might no longer be valid, so both functions must be re-run. If the programmer changed the code for `stageC`, then `stageA` and `stageC` must be re-run (but not `stageB`, since it doesn't call `stageC`).

Assume that the program executes for 1 hour on `input1.txt` because it makes 5 calls to `stageB` and 1 call to `stageC` (each taking 10 minutes). If we edit the end of `stageA` to return, say, the product of `transformedLst` elements rather than the sum, then re-executing the program is *nearly instantaneous* since we can re-use all the memoized results from `stageB`

```
MULTIPLIER = 2.5 # global variable

# Input:  name of file containing SQL queries
# Output: a single computed numerical value
def stageA(filename):
  lst = [] # initialize empty list
  for line in open(filename, 'r'):
    lst.append(stageB(line))
  transformedLst = stageC(lst)
  return sum(transformedLst) # returns a number

# Input:  an SQL query string
# Output: a single computed numerical value
def stageB(queryStr):
  db = sql_open_db('masterDatabase.db')
  q = db.query(queryStr)
  res = ... # run for 10 minutes processing q
  return (res * MULTIPLIER) # returns a number

# Input:  a list of numerical values
# Output: a single computed numerical value
def stageC(datLst):
  res = ...  # run for 10 minutes munging datLst
  return res # returns a number

# specify input file on command-line:
print stageA(argv[1])
```

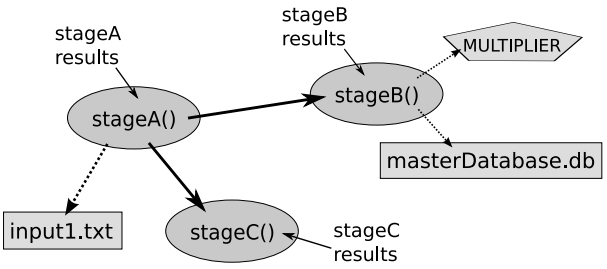

Figure 1: Example Python data processing program (top) and dependencies generated during execution (bottom). Circles represent code, squares represent file accesses, and the pentagon represents a global variable access.

and `stageC` rather than re-computing them. If we notice a bug in `stageC` and tweak its code to fix it, then re-executing the program only takes 10 minutes rather than a full hour (we must re-run `stageA` and `stageC` but can re-use memoized results from all 5 calls to `stageB`).

3 Interpreter-based technique for efficient incremental recomputation

3.1 MemoMap global data structure

The only global data structure we maintain is called `MemoMap`. Each entry in `MemoMap` is a mapping from a function to a `FuncMemoInfo` object (whose class is defined in Figure 2). The interpreter loads `MemoMap` from

```
class FuncMemoInfo {
  function myFunc
  Boolean definitelyImpure
  Map<<List<argument> , returnVal> memoizedVals
  Map<function , code> codeDeps
  Map<variable , value> globalVarDeps
  Map<file , file contents> fileDeps

  FuncMemoInfo(function _func) { // constructor
    myFunc = _func
    clearAndMarkPure()
  }
  void clearAndMarkPure() {
    definitelyImpure = False
    memoizedVals = new Map<...>
    codeDeps = new Map<...>
    globalVarDeps = new Map<...>
    fileDeps = new Map<...>
    // initialize self-dependency:
    codeDeps[myFunc] = myFunc.code
  }
  void markImpure() {
    definitelyImpure = True
    memoizedVals.clear()
    globalVarDeps.clear()
    fileDeps.clear()
    // do not clear codeDeps, still need it
  }
}
```

Figure 2: The `FuncMemoInfo` class, which keeps track of memoized values and dependencies for a function

disk at the beginning of execution (or creates an empty mapping if it doesn't yet exist), and saves it to disk at the end of execution. A `FuncMemoInfo` object keeps track of memoized values and dependencies for one function. We chose to memoize at the function level since it is a natural abstraction boundary for computations; in Section 7 (future work) we discuss some plans for memoization at a finer granularity.

The `memoizedVals` field in `FuncMemoInfo` stores memoized values, in the form of a mapping from a list of argument values to a return value (e.g., for the code in Figure 1, an entry for `stageA` might be { `"input1.txt"` → `50` }). Whenever a function is called, if its arguments are already contained in `memoizedVals`, then the interpreter can directly return the memoized return value rather than executing its code.

There are three fields that store dependencies:

- `codeDeps` (code dependencies): A mapping from a function called by this function to the current contents of its code (e.g., source code or bytecode)
- `globalVarDeps` (global variable dependencies): A mapping from a global variable read by this function to its current value
- `fileDeps` (file dependencies): A mapping from a file read by this function to its current contents

If any of these dependencies are broken, then `memoizedVals` (and all other fields) are cleared, because it might no longer be safe to re-use memoized values. For example, if a function `f` reads from a file `data.txt`, then its `memoizedVals` should be cleared if `data.txt` changes, since `f` might now compute a different result even when given the *same* input.

Only *pure functions* are eligible for memoization. Following the definition of Salcianu and Rinard, we consider a function pure if it never mutates an object that existed prior to its invocation [14]. As soon as a function mutates its argument or a global variable, it is marked impure and all data in its `FuncMemoInfo` are deleted. In addition, we always mark non-deterministic functions like `random()` and `time()` as impure.

Our technique dynamically detects purity during execution, so it avoids the usual obstacles that hinder static (compile-time) purity detection, like conservative full-program pointer analysis and reasoning about dynamically-typed and reflective language features [14]. Our purity analysis is perfectly precise modulo broken dependencies: As long as a pure function's dependencies are not broken, then subsequent executions will run identically and remain pure. When a dependency breaks, all data in `FuncMemoInfo` are deleted, so its original purity state no longer matters.

3.2 Program value accesses

Algorithm 1 shows what the interpreter does every time a value is accessed during execution.

Algorithm 1 Program value accesses

On a READ of any program value *val*:
  if *val* reachable from a global variable *GVar* **then**
    tf ← MemoMap[function at the top of stack]
    tf.globalVarDeps[*GVar*] ← deepcopy(*GVar*.value)
  end if

On a WRITE or DELETE of any value *val*:
  if *val* reachable from any global variable **then**
    for all functions *func* on stack **do**
      MemoMap[*func*].markImpure()
    end for
  else
    top ← function at the top of stack
    if *val* reachable from any argument of *top* **then**
      MemoMap[*top*].markImpure()
    end if
  end if

Whenever a function reads a value that is reachable from a global variable, that function gains a data dependency on the entire global variable. Note that we make a deep copy of the variable's current value before storing it in `globalVarDeps`, in order to protect against later mutations to it. Also, we store a dependency on the *entire* global variable and not simply on the accessed value, in order to allow for a simpler implementation. For example, if a function accesses a deeply-nested value `x.foo.bar[3].baz` where `x` is a global, it's much easier to store and to later look-up the value of the entire `x` data structure rather than having to make complex traversals within field and array dereferences.

Whenever the program mutates a globally-reachable value, *all functions* on the stack must be marked as impure (i.e., ineligible for memoization), since they were executing while a global side-effect occurred. Otherwise, if an argument of the currently-executing function is mutated, then only that function should be marked impure (its callers might still be pure).

3.3 File accesses

Algorithm 2 File accesses

On a READ of an open file *f*:
  tf ← MemoMap[function at the top of stack]
  tf.fileDeps[*f*] ← *f*.contents

On a WRITE of an open file *f*:
  for all functions *func* on stack **do**
    MemoMap[*func*].markImpure()
  end for

Algorithm 2 shows that the interpreter handles reads and writes of files in the same way that it handles global variable accesses. Files are an abstraction for any external resource that a program accesses. For instance, an implementation could model database or network accesses via this file model to maintain dependencies on, say, database table rows or web service data streams.

3.4 Function entrance and exit

Whenever the program is about to execute a function, the interpreter tries to avoid executing that function altogether and instead return a memoized value to its caller (Algorithm 3). Before doing so, the interpreter must check whether this function is impure or whether any of its own dependencies or dependencies of functions that it has transitively called have been broken (i.e., their current values differ from their previously-saved values). If so, then the function executes normally. Otherwise, if the current argument values are saved in `memoizedVals`, then the interpreter returns the matching memoized return value to the caller rather than re-computing it.

Algorithm 3 Function entrance

When program execution enters a function *func*:

```
if func not in MemoMap then
    MemoMap[func] = new FuncMemoInfo(func)
end if

// your caller has a dependency on your code:
c ← MemoMap[function right below func on stack]
c.codeDeps[func] ← func.code

f ← MemoMap[func]
if f.definitelyImpure then
    return and continue normal execution of func
end if

// check code dependencies (transitively):
for all {depFunc , depSavedCode} in f.codeDeps do
    if depFunc.code ≠ depSavedCode then
        f.clearAndMarkPure()
        return and continue normal execution of func
    end if
    traverse inside depFunc, check all its dependencies,
    and recursively traverse inside its code dependen-
    cies and check them, until fixpoint reached
end for

// check global variable dependencies:
for all {depVar , depSavedVal} in f.globalVarDeps do
    if depVar.value ≠ depSavedVal then
        f.clearAndMarkPure()
        return and continue normal execution of func
    end if
end for

// check file dependencies:
for all {depFile , depSavedContents} in f.fileDeps do
    if depFile.contents ≠ depSavedContents then
        f.clearAndMarkPure()
        return and continue normal execution of func
    end if
end for

if func.argumentValues in f.memoizedVals then
    mRetval ← f.memoizedVals[func.argumentValues]
    return mRetval to caller and don't execute func
else
    return and continue normal execution of func
end if
```

Whenever the program is about to finish executing a function, the interpreter memoizes copies of its argument and return values in `memoizedVals` if necessary (Algorithm 4). Memoization is *not* done if the function is already impure, did not execute for long enough (see next paragraph), or returns an externally-accessible *mutable* value (which could lead to subtle semantic mismatches due to deep copying of the memoized return value).

Algorithm 4 Function exit

When program execution exits a function *func*:

```
f ← MemoMap[func]
if f.definitelyImpure then
    return and exit out of func
end if
if func took less than 1 second to execute then
    return and exit out of func
end if

for all mutable values v within func.returnValue do
    if v reachable from any global variable or from any
    current argument of func then
        return and exit out of func
    end if
end for

// memoize copies of arguments and return value:
argCopy ← deepcopy(func.argumentValues)
retvalCopy ← deepcopy(func.returnValue)
f.memoizedVals[argCopy] = retvalCopy
return and exit out of func
```

We think it's practical to use a heuristic of only memoizing function executions that last for more than 1 second. The vast majority of functions run for less than 1 second, so it's not worth the extra work of memoizing their results when they could be instantly re-computed. The interpreter should only spend effort memoizing functions that run for a non-negligible amount of time.

4 Python interpreter implementation

We implemented our technique as a custom Python interpreter called `IncPy`, which we created by adding ∼3000 lines of C code to the official Python 2.6 interpreter. `IncPy` passes the entire Python regression test suite (except for a test of multithreading, which it doesn't support). It works as both an interactive shell and for running script files, memoizing and tracking dependencies in all imported Python modules. We now describe some implementation decisions, with a focus on optimizations:

Data structures: Since our code is part of the Python interpreter, we have full access to the implementations of Python's built-in data structures. Thus, it was easy and efficient to use Python list objects to maintain lists of argument values and Python dictionary objects to maintain mappings for `FuncMemoInfo` fields.

On-disk persistence: We use the `cPickle` module in the Python standard library to serialize Python objects

to disk, in order to allow memoized values and dependencies to persist across executions. (Some Python data types are not serializable, so we cannot memoize functions that access such data.)

We store each `FuncMemoInfo` entry in a separate file, load it from disk the first time it is needed, record whether it has been modified throughout execution, and only save it back to disk at the end of execution if it has been modified (lazy write-back). Serializing at this level of granularity seems sufficient for our initial implementation; in the future, we might use an object database or persistent in-memory cache (e.g., `memcached`) if we need finer control over performance.

File dependencies: Rather than saving and comparing file contents (or md5sums), we simply save and compare their last modification times, which is far more efficient. This is akin to how Makefile dependencies work.

Code dependencies: We save and compare Python bytecode rather than source code. We do so because making source edits that change spacing, comments, and other minor cosmetic tweaks do not alter a function's behavior, so should not break any dependencies on it.

Lazy reachability detection: Several of our algorithms require the interpreter to determine whether a value is reachable from a global variable. In Python, all accesses to globally-reachable values must originate from a read of some global variable, followed by zero or more reads of its attributes (e.g., object fields, list members). On a read of a global variable, the address of its current value is added to the globally-reachable set; on the read of an attribute like `obj.attr`, if the address of the current value of `obj` is in the globally-reachable set, then the address of the value of `attr` is also added to that set. Therefore, given an arbitrary value, a simple address lookup in that set tells us whether it's globally-reachable (we also maintain some extra data to determine *from which* global variables each value is reachable).

Copy-on-write optimization: Recall that we need to make a deep copy of the values of global variables, function arguments, and return values before saving them in `FuncMemoInfo`, since we want to capture a snapshot at a certain point in execution. However, performing a deep copy can be slow, especially for large collections (e.g., a list of 100,000 dictionaries, each containing a few mappings of strings to integers). We noticed that these large objects were rarely modified later in execution, so the deep copy wasn't even necessary.

Thus, we implemented copy-on-write, which defers the deep copy until the moment an object is mutated. To evaluate `y=deepcopy(x)`, we first set `y=x` and map the address of `y` to a 'watch set' containing the address of `x`. Then we traverse inside of `x` and add the addresses of all enclosed *mutable* objects to the watch set for `y` (often much faster than copying since most collections contain only immutable objects, which don't need to be tracked). Whenever a value is mutated, if its address is in the watch set for `y`, then we deep copy `x`, assign it to `y`, update references to it, and delete its watch set.

5 Preliminary experimental results

We performed an informal preliminary evaluation by using `IncPy` to run Python scripts that we wrote in 2007–2008 for a research project to analyze revision control history and static analysis bug reports for the Linux kernel [7]. We started developing `IncPy` about a year after writing these scripts, partly motivated by our experiences of waiting for them to run for minutes after minor edits and having to manually write extra caching code. For this evaluation, we examined 3 sets of scripts that show ways in which `IncPy` could have sped-up our workflow. Soon we plan to do a more comprehensive evaluation by running other researchers' scripts through `IncPy`.

Since the current version of `IncPy` only does function-level memoization, we manually tweaked some of our target scripts by hoisting loop bodies into nested functions and having each loop iteration be a function call. This transformation is purely mechanical and easy to automate; we plan to add memoization of loop bodies in the near future (see Section 7).

Experiment 1: We first examined a 200-line Python script that calculates, for all Linux source code files modified in each developer's i^{th} committed patch, how many other developers subsequently modified that file and when did they make those edits. Each loop iteration extracts data from an `sqlite` database and performs the requisite calculations for a particular i (via a helper function call), then prints out aggregate information like the number of developers and mean time between patches.

When running for 40 iterations (the script's default setting), the regular Python interpreter took 147 seconds[3], while `IncPy` took 160 seconds (a **9% slowdown**). When we re-ran the script in `IncPy` (without making any code edits), it only took 8 seconds, an **18X speed-up** over the regular Python interpreter. After making edits like changing the aggregate information printed at each iteration (e.g., using *median* instead of *mean*) or tweaking ways to combine extracted fields to form derived fields, the script still only took 8 seconds to re-run. These edits, while small, are indicative of the sort that a scientist typically makes while tweaking his/her scripts: Each iteration of a data processing script often consists of a long-running extraction portion (that rarely changes) followed by short-running post-processing code. Being

[3] Running on a 3GHz Mac Pro with 4GB RAM and Mac OS X 10.4

able to re-run and see new results in 8 seconds rather than waiting for several minutes allows the programmer to keep the task in working memory [6] while iterating and debugging.

However, a more intrusive edit like changing the long-running SQL query and data extraction code would require the entire script to re-run. Memoizing at a finer granularity (e.g., per code statement) could prevent the entire script from re-running, but it would incur a greater run-time overhead during the initial run. We plan to explore this trade-off between tracing precision and run-time overhead in future work.

Experiment 2: Next we examined a longer-running Python script that calculates the probability that a Linux source code file will be modified in a given time period, conditioned upon whether it was modified in the past. It performs a sliding window calculation by setting an initial period length (e.g., 1 week), a subsequent period length (e.g., 1 year), and the number of days to slide after each iteration (e.g., 1 day). Each iteration of the main loop takes a starting date, queries the database for files modified in the 1 week prior to that date and in the 1 year following that date, then computes probabilities.

When we ran the sliding window over the first year of our 7-year dataset (365 iterations, since each iteration slides by 1 day), the regular Python interpreter took 34 minutes and 33 seconds, while `IncPy` ran for only 38 seconds longer (a **2% slowdown**). For this run, `IncPy` only memoized 83 kB of data, so it had minimal object serialization overhead; `IncPy`'s slowdown is usually proportional to the amount of memoized data, not to the total code size or running time of the script.

When we re-ran the script in `IncPy`, it only took 0.2 seconds (a **10,000X speed-up** over the regular interpreter). We can make edits to adjust how the probabilities are calculated within each iteration, and instantly re-run to see new results. Also, we can adjust the iteration slide amount to a larger value (e.g., 1 week) and instantly re-run to produce a subset of our output data; this operation is useful for rendering graphs with different parameters. Currently, scientists must save the output data to a `.csv` (text) file and then write another script to parse that `.csv` file and render graphs. By having `IncPy` automatically memoize that data, there is no need to create and maintain dependencies for intermediate `.csv` files; the data analysis and graphing code can be together in one script, and `IncPy` ensures that when graphing code is tweaked, the analysis code does not need to re-run.

After processing only the first year of our dataset, we made the script process the first 2 years. `IncPy` re-used the results for the first year and took 36 minutes to process the second year. The regular Python interpreter had to start from scratch, so it took twice as long (71 minutes). This scenario simulates how `IncPy` allows scripts to re-start processing after a bug fix. Data processing bugs often manifest on aberrant records that appear in the middle of a large dataset [4] (say, at the end of the first year in this example); if the bug fix is sufficiently small, then it might be possible to re-start at the buggy record rather than re-computing all the prior (correct) records.

Experiment 3: We examined a directory of scripts that compute statistics about the concentration of bug reports per Linux source file. We initially wrote these scripts to extract data from several tables in one `sqlite` database. However, they ran too slowly since they were doing complex joins, so we derived 4 intermediate datasets from that database and refactored the scripts to extract data from those datasets, which were stored on disk as persistent Python `pickle` objects (filenames abbreviated):

- `A.pickle`: 371 kB, took 2 seconds to create
- `B.pickle`: 3.6 MB, 21 sec to create
- `C.pickle`: 26 MB, 43 sec to create
- `D.pickle`: 16 MB, 69 sec, depends on `C.pickle`

By manually creating and maintaining these intermediate datasets, we sped up run-times at the expense of making our code more complex and harder to maintain: We had to write a script to create each dataset, add extra code to load and save intermediate data, and remember to manually re-run those scripts to re-create the proper datasets whenever their dependent data changed. We updated our master database several times to do clean-ups (e.g., canonicalizing alternate forms of a person's name) and had to re-create the 4 derived datasets each time, making sure to re-create `C.pickle` *before* `D.pickle` (since the latter is derived from the former).

If we had used `IncPy` when writing these scripts back in 2008, then we could have simply put all the code together in one script and let `IncPy` create the derived datasets and maintain dependencies for us. That way, whenever we changed the master database, all 4 datasets would be re-created, and whenever we tweaked the code for creating one of the datasets, only it would be re-created. We combined all 4 dataset-creation scripts into one, removed the manual serialization code, and ran it through `IncPy`. It produced the same-sized `pickle` files, but ran for a minute longer (**$\sim$44% slower**).

6 Related work

Our technique relates to provenance in that it uses a programming language interpreter to collect a limited form of provenance for data resulting from function executions. The provenance we collect for each piece of data includes how long it took to compute and from which function executions, global variables, and input files did

it derive. We use this provenance to facilitate efficient incremental recomputation of imperative programs.

Several related systems collect provenance via runtime execution tracing. Muniswamy-Reddy et al. designed a storage system that tracks coarse-grained provenance about individual files, which could be the intermediate results of command-line script invocations [10]. Zhang et al. designed a system to collect very fine-grained provenance at the byte level by tracing the execution of x86-Linux binary executables (usually from C or C++ programs) [17]. Neither system aims to use provenance to facilitate efficient incremental recomputation; instead, they focus more on using it to understand, debug, validate, and reproduce experimental results.

Scientific workflow systems like Kepler [9], Taverna [11], and VisTrails [15] provide graphical integrated development environments for designing and executing scientific computations. Scientists create workflows by using a GUI to visually connect together blocks of premade functionality in a data-flow graph.

These systems collect provenance to facilitate debugging, validation, reproduction, and — similar to `IncPy` — efficient incremental recomputation of results. It is fairly easy for them to do incremental recomputation since dependencies must be explicitly drawn in the data-flow graph and the composed computational blocks are mostly pure; in contrast, `IncPy` must infer dependencies and function purity from arbitrary Python programs.

`IncPy` differs from scientific workflow systems in that it is a lightweight solution to facilitate prototyping in a popular, easy-to-learn, and general-purpose programming language. In contrast, while workflow systems have more specialized and sophisticated features, the designers of VisTrails admit: *"While significant progress has been made in unifying computations under the workflow umbrella, workflow systems are notoriously hard to use. They require a steep learning curve: users need to learn programming languages, programming environments, specialized libraries, and best practices for constructing workflows."* [15]

Self-adjusting computation enables efficient recomputation in response to changes in input data. The technique focuses solely on fine-grained changes to input data (e.g., linked list elements), whereas `IncPy` tracks code, data, and external environment changes at a coarser level. Hammer, Acar, et al. have implemented self-adjusting computation for statically-typed languages (ML and C) [8]. These systems have potential for massive speed-ups but require programmers to write algorithms using new language constructs and forbids the use of existing constructs like returning values from functions (must instead use *destination-passing style* [8]). We feel `IncPy` is easier for non-expert programmers to adopt since it targets a dynamically-typed language widely-used for scripting and does not require the learning of any new constructs or programming styles.

Just-in-time compilers for dynamic languages (e.g., Psyco [13] for Python, Rubinius for Ruby, TraceMonkey for JavaScript) can speed up script execution times without requiring the programmer to learn any new constructs, which is similar in spirit to the goals of `IncPy`. However, JIT compilers focus on micro-optimizations of CPU-bound code like hot inner loops, whereas `IncPy` deals more end-to-end, using memoization to avoid long-running computations regardless of their source. For example, no JIT compiler optimizations could speed up I/O or network-bound scripts, which is what often consumes lots of time in scientific data processing applications. Features from `IncPy` could easily be integrated into a JIT compiler, to get the benefits of both approaches.

Parallel execution of code can vastly speed up scientific data processing scripts, at the cost of increased difficulty in programming and debugging such scripts. In recent years, the emergence of libraries like Parallel Python [1], frameworks like MapReduce [5], and new high-level languages like Pig [12] have made it easier to write parallel code. However, we (and many of the researchers we interviewed) believe that the learning curve for writing parallel code is still high, especially for scientists without much programming experience. It is much easier for most people to think about algorithms sequentially, and even experts prefer to write single-threaded prototypes and then only parallelize later when necessary [16].

Also, the goals of `IncPy` are complementary to those of parallel computation: Eliminating redundant computations can still be useful for speeding up parallel code running on a cluster. For instance, a bioinformatics graduate student we interviewed lamented how a postdoc in his research group was constantly using up all the compute-power on the group's 124-CPU cluster. This postdoc's data-parallel Perl script consisted of several processing stages that took dozens of hours to run on each cluster node (on a slice of the input genomic data). He would often make tweaks to the second stage and re-run the entire script, thus needlessly re-computing all the results of the first stage. After a few weeks of grumbling, the graduate student finally inspected the postdoc's code and saw that he could refactor it to memoize intermediate results of the first stage, thus dramatically reducing run-time and freeing up the cluster for labmates. The postdoc, who was not an adept programmer, perhaps did not have the expertise or willingness to refactor his code; he was simply willing to wait overnight for results (to the chagrin of his labmates). With an interpreter like `IncPy` installed on their cluster machines, there would be no need to perform this sort of manual code refactoring.

7 Ongoing and future work

From chatting with researchers interested in using `IncPy`, we have heard the following feature requests:

- **Intraprocedural analysis**: Currently, we only memoize at the function level, but many scripts contain memoization opportunities within individual functions (or are simply written as one monolithic top-level `main` function). For instance, a long-running loop that populates a list with values can be automatically refactored to its own function, so that its results can be memoized and re-used.
- **Database-aware caching**: Several researchers we interviewed stored their datasets in a relational database, used Python scripts to make simple SQL queries and then performed sophisticated data transformations using Python code. They found it easier and more natural to express their algorithms in an imperative manner rather than in SQL. If `IncPy` were augmented to intercept Python's database API calls, then it could track finer-grained data dependencies between database entries and Python code.
- **Network-aware caching**: Several CS researchers who extract large amounts of data from the Internet for their research told us that they found it annoying to have to re-run their extractor scripts after fixing bugs or making tweaks. If `IncPy` were augmented to intercept Python's networking API calls, then it could transparently cache data retrieved from the Internet on the researcher's machine, maintain the proper dependencies, and speed up subsequent runs by eliminating unnecessary network activity.
- **Lightweight annotations**: Some colleagues suggest allowing the programmer to annotate their scripts to gain more control over memoization (e.g., explicitly marking a block of code as a unit to memoize rather than only relying on the interpreter to infer what to memoize). We currently don't have annotations since we don't want to require scientists to learn any new constructs in order to use `IncPy`, but we will add simple annotations if there is demand.
- **Simple provenance browsing**: Some researchers would like ways to interactively browse (or even manually edit) the saved intermediate results of their scripts. `IncPy` already records a limited form of provenance, so allowing a user to directly view the data and its provenance could help them manually verify the correctness of intermediate results, debug more effectively, and predict how long it might take to re-execute portions of their workflow.

Acknowledgments: We thank Joel Brandt and Robert Ikeda for editorial help, all of our interview subjects, and the NSF fellowship for funding Philip's graduate studies.

References

[1] Parallel Python `http://www.parallelpython.com/`.

[2] Programming for Scientists (blog) `http://www.programming4scientists.com/`.

[3] BRANDT, J., GUO, P. J., LEWENSTEIN, J., DONTCHEVA, M., AND KLEMMER, S. R. Opportunistic programming: Writing code to prototype, ideate, and discover. *IEEE Software 26*, 5 (2009), 18–24.

[4] BRONSON, N. personal email communication, discussing hindrances to data subsetting, 2009.

[5] DEAN, J., AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation* (2004), USENIX Association, pp. 10–10.

[6] GRAY, W., AND BOEHM-DAVIS, D. Milliseconds matter: An introduction to microstrategies and to their use in describing and predicting interactive behavior. *Journal of Experimental Psychology: Applied 6*, 4 (2000), 322–335.

[7] GUO, P. J., AND ENGLER, D. Linux kernel developer responses to static analysis bug reports. In *Proceedings of the USENIX Annual Technical Conference* (2009), USENIX Association, pp. 285–292.

[8] HAMMER, M. A., ACAR, U. A., AND CHEN, Y. CEAL: a C-based language for self-adjusting computation. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation* (2009), ACM, pp. 25–37.

[9] LUDÄSCHER, B., ALTINTAS, I., BERKLEY, C., HIGGINS, D., JAEGER, E., JONES, M., LEE, E. A., TAO, J., AND ZHAO, Y. Scientific workflow management and the Kepler system: Research articles. *Concurr. Comput. : Pract. Exper. 18*, 10 (2006), 1039–1065.

[10] MUNISWAMY-REDDY, K.-K., HOLLAND, D. A., BRAUN, U., AND SELTZER, M. Provenance-aware storage systems. In *Proceedings of the USENIX Annual Technical Conference* (2006), USENIX Association, pp. 4–4.

[11] OINN, T., GREENWOOD, M., ADDIS, M., ALPDEMIR, M. N., FERRIS, J., GLOVER, K., GOBLE, C., GODERIS, A., HULL, D., MARVIN, D., LI, P., LORD, P., POCOCK, M. R., SENGER, M., STEVENS, R., WIPAT, A., AND WROE, C. Taverna: lessons in creating a workflow environment for the life sciences: Research articles. *Concurr. Comput. : Pract. Exper. 18*, 10 (2006), 1067–1100.

[12] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the ACM SIGMOD international conference on Management of data* (2008), ACM, pp. 1099–1110.

[13] RIGO, A. Representation-based just-in-time specialization and the Psyco prototype for Python. In *Proceedings of the ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation* (2004), ACM, pp. 15–26.

[14] SALCIANU, A., AND RINARD, M. C. Purity and side effect analysis for Java programs. In *VMCAI* (2005), pp. 199–215.

[15] SCHEIDEGGER, C. E., VO, H. T., KOOP, D., FREIRE, J., AND SILVA, C. T. Querying and re-using workflows with VisTrails. In *Proceedings of the ACM SIGMOD international conference on Management of data* (2008), ACM, pp. 1251–1254.

[16] SEO, J. personal email communication, discussing sequential vs. parallel scripts, 2009.

[17] ZHANG, M., ZHANG, X., ZHANG, X., AND PRABHAKAR, S. Tracing lineage beyond relational operators. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases* (2007), VLDB Endowment, pp. 1116–1127.