Sloppy Python: Using Dynamic Analysis to Automatically Add Error Tolerance to Ad-Hoc Data Processing Scripts

Philip J. Guo
Computer Systems Laboratory
Stanford University
pg@cs.stanford.edu

ABSTRACT

Programmers and data analysts get frustrated when their long-running data processing scripts crash without producing results, due to either bugs in their code or inconsistencies in data sources. To alleviate this frustration, we developed a dynamic analysis technique that guarantees scripts will never crash: It converts all uncaught exceptions into special NA (**N**ot **A**vailable) objects and continues executing rather than crashing. Thus, imperfect scripts will run to completion and produce partial results and an error log, which is more informative than simply crashing with no results. We implemented our technique as a "Sloppy" Python interpreter that automatically adds error tolerance to existing scripts without any programmer effort or run-time slowdown.

Categories and Subject Descriptors:
D.3.4 [*Processors*]: Interpreters, Run-time environments
H.5.2 [*User Interfaces*]: Prototyping

General Terms: Languages, Reliability

Keywords: Data processing, fault tolerance, scripting

1. INTRODUCTION

Programmers across a wide range of disciplines (e.g., bioinformatics, neuroscience, econometrics, finance, data mining, information retrieval) often write scripts to parse, transform, process, analyze, and extract insights from data. We refer to these as *ad-hoc data processing scripts* because both the scripts and the data they operate on are ad-hoc in nature:

- **Ad-hoc scripts**: Programmers write these scripts in a "quick-and-dirty" manner to explore their datasets and to formulate, test, and refine hypotheses. They do not spend much time making these scripts robust, error-tolerant, or modular, since their primary goal is to discover insights from their data, not to produce a well-engineered piece of production software.
- **Ad-hoc data**: Vast amounts of real-world data are stored in ad-hoc formats [10]. Examples of such semi-structured and unstructured data are those scraped from web pages, mined from emails, and logs produced by computer systems and scientific lab equipment. Ad-hoc data often contains inconsistencies and errors due to lack of well-defined schemas, malfunctioning equipment corrupting logs, non-standard values to indicate missing data, or human errors in data entry [10].

WODA '11, July 18, 2011, Toronto, ON, Canada

Although domain-specific data processing languages are being developed in research labs [10, 13, 14], most modern programmers write ad-hoc data processing scripts in general-purpose languages such as Python, Perl, and Ruby due to their flexibility, support for rapid prototyping, and actively-maintained ecosystems of open-source libraries.

Problem: We have heard our colleagues gripe about the following problem regarding the brittleness of their scripts:

1. They start executing a script that is expected to take a long time to run (e.g., tens of minutes to a few hours).
2. They work on another task or go home for the evening.
3. When they return to their computer, they see that their script crashed upon encountering the first uncaught exception and thus produced no useful output.

The crash could have occurred due to a bug in the script or the data: Since programmers write scripts in a "quick-and-dirty" manner without carefully handling edge cases, many bugs manifest as uncaught run-time exceptions. Also, ad-hoc data sources often contain badly-formatted "unclean" records that cause an otherwise-correct script to fail [10].

Regardless of cause, the programmer gets frustrated because he/she has waited for a long time and still cannot see any results. The best he/she can do now is to try to fix that one single visible error and re-execute the script. It might take another few minutes or hours of waiting before the script gets past the point where it originally crashed, and then it will likely crash again with another uncaught exception. The programmer might have to repeat this debugging and re-executing process several times before the script successfully finishes running and actually produces results.

For another scenario that exhibits this problem, consider an online real-time data analytics script (e.g., deployed on a server) that continuously listens for input and incrementally processes incoming data. Unless the programmer meticulously handled all edge cases, that script will inevitably encounter some data it cannot process and crash with an uncaught exception. Discovering that the script crashed, debugging and fixing the error, and re-deploying it might lead

to missing a few hours' or days' worth of incoming data, all just because of one unexpected crash-causing input.

Our proposed solution: To address this problem of script brittleness, we developed a dynamic analysis technique that automatically adds error tolerance to existing scripts without requiring any programmer effort. It works as follows:

1. The script executes in a custom interpreter.
2. When an expression throws an uncaught exception, the interpreter creates an NA (**N**ot **A**vailable) object representing the exception, assigns it to the expression's target, saves the NA object and local variable values to a log file, and continues executing normally.
3. Whenever an NA object appears in an expression, the interpreter handles it according to special rules and logs a warning. For example, all unary and binary operations involving an NA object will return NA.

Our technique guarantees that scripts will never crash. As soon as a script throws an exception when processing a record in the dataset, the interpreter marks values derived from that record as NA and continues processing subsequent records. Since scripts usually process each record independently, NA objects do not propagate to taint values derived from neighboring records (i.e., they are confined to the error-inducing records). When the script finishes running, it outputs results for the successfully-processed records, which often provides more information than prematurely crashing.

The interpreter also maintains a log of what went wrong when processing dataset records that caused the script to throw exceptions. This log can be used to debug errors and can even be processed to incrementally recover from errors without re-executing the script on the entire dataset. We demonstrate an incremental error recovery example in §4.3.

Main benefits: Our technique allows programmers to iterate faster when developing ad-hoc scripts in three ways:

1. **Faster coding**: Programmers can write "quick-and-dirty" scripts, execute them, and see partial results (and all errors) without being slowed down by the burden of worrying about full correctness. They also do not need to write extra code to handle exceptions that could arise in *almost every line of code* (e.g., every array operation could throw an out-of-bounds exception). A study found that 50–70% of the code in reliable systems software was for handling edge cases [11]; although that study was not on data processing scripts, it shows that programmers spend lots of effort on error handling in general. Our technique acts as a safety net to guard against unexpected edge cases without any programmer effort. As their scripts mature, programmers can write specialized error handling (or even recovery) code to wean themselves off of this safety net.
2. **Fewer edit/run/debug iterations**: After every script execution, programmers can see a log of all run-time errors and detailed information about their context. Thus, they can fix multiple bugs in one round of code edits rather than fixing one at a time and waiting for minutes or hours before seeing whether there are additional errors. As an analogy, if a compiler only displayed the first syntax error when compiling code, then the programmer can only fix one error at a time and must re-compile to see the next error. Fortunately, modern compilers display as many useful errors and warnings as possible per invocation.
3. **More insights per iteration**: When a script has bugs, partial results always provide more insights than no results, thus helping to inform code edits for the next iteration. Partial results can actually be accurate when computing aggregate statistics: e.g., if the true mean housing price in a large dataset is $200k, a script might return a mean of $198k if it can only process 95% of the data (the other 5% might require more detailed parsing logic). They can even provide exact answers: e.g., Condie et al. found the top 10 most frequently occurring words in a 5.5GB corpus of Wikipedia article text after only processing about half of the dataset [8].

2. TECHNIQUE

Our technique involves altering run-time exception handling by creating special NA objects rather than crashing. We find it easiest to explain and implement in the context of a programming language interpreter, but it can also be implemented in a compiler or bytecode transformation tool.

2.1 Creating NA from Uncaught Exceptions

When a regular interpreter executes a script, as soon as an expression throws an exception, the interpreter inspects all functions on the stack to try to find an exception handler (i.e., code in a try-catch block). If it can find a handler, then it transfers control to that handler and keeps executing. If it cannot find a handler, then it will crash the script and print an error message. For example, executing `x = 1 / 0` will crash the script with a divide-by-zero exception.

Our technique alters the interpreter so that, instead of crashing the script on an uncaught exception, it creates a special NA (Not Available) object that becomes the result of the expression that threw the exception. (If the script provides an exception handler, then it executes as usual.) An NA object has three fields:

- **Exception type** (e.g., array out-of-bounds)
- **Exception details** (e.g., *"index 25 is out of bounds of the array named x"*)
- **Stack backtrace** indicating all currently-executing function names and line numbers

When our modified interpreter executes `x = 1 / 0`, it creates a new NA object to represent the exception, assigns it to the variable `x`, and continues normal execution.

2.2 Logging Exception Context for Debugging and Incremental Error Recovery

When the interpreter creates a new NA object, it saves a snapshot of that object and the current values of all local variables to a log file. For example, consider this Python script, which iterates through each line in `big_data.txt`, splits it into tokens based on whitespace, converts the first two tokens into numbers, and prints their quotient (ratio):

```
line_number = 1
for line in open('big_data.txt'):
  tokens = line.split(' ')
  ratio = float(tokens[0]) / float(tokens[1])
  print ratio
  line_number += 1
```

Although this script is tiny, it can throw three possible exceptions when processing `big_data.txt`:

1. If a line contains fewer than two tokens, then accessing `tokens[1]` will throw an out-of-bounds exception.
2. If a token does not represent a floating-point number, then converting it to a `float` will throw an exception.
3. If `float(tokens[1])` is 0, then there is a divide-by-zero exception.

If `big_data.txt` has 1 million lines, then when the script finishes processing it, the warning log file might look like:

```
Divide-by-zero exception: 5 / 0
    line_number=291818, line='5 0', tokens=['5', '0']
    < ... full stack backtrace ... >
Invalid float exception: 'six' is not a valid float
    line_number=320183, line='5 six', tokens=['5', 'six']
    < ... full stack backtrace ... >
Out-of-bounds exception: tokens[1] is out of bounds
    line_number=983012, line='2', tokens=['2']
    < ... full stack backtrace ... >
```

This example log file indicates that the script failed to process three records in `big_data.txt`. The exception object, stack backtrace, and values of local variables pinpoint the exact cause of each failure. The programmer now knows exactly where to insert error handling or recovery code to avoid triggering those exceptions in future runs. Alternatively, he/she could edit the dataset to alter or remove those problematic records (located at `line_number`).

Finally, in some cases, he/she can write an amended version of the script to *process the log file itself* and merge the results with those from the original computation. Since the log file contains partially-processed records stored in local variables, an amended script can directly process that file, which can be much faster than re-executing on the entire dataset (a technique colloquially known as "sloppy seconds"). In the above example, an amended script would only need to process the three error-inducing records in the log file rather than the 1 million in the original dataset. In Section 4.3, we demonstrate this form of incremental error recovery.

2.3 Treatment of NA Objects

The interpreter treats NA objects in the following special ways throughout execution:

- A unary operation on an NA object (e.g., negation) returns itself (the original NA object).
- A binary operation involving an NA object (e.g., addition, subtraction) returns itself [1].
- A comparison involving an NA object (e.g., less than, equal to, greater than) returns itself [1], since the result is unknown (neither TRUE nor FALSE).
- Calling an NA object as a function returns itself.
- Accessing an element of an NA object (e.g., via array indexing) returns itself.
- Accessing a named field of an NA object returns itself.
- NA `and` FALSE (conjunction) returns FALSE
- NA `or` TRUE (disjunction) returns TRUE
- Mutating an NA object does nothing.

[1]If it involves two NA objects, then it returns the first one.

To inform the user about how NA objects propagate during execution, the interpreter logs all NA-related operations (e.g., a comparison involving an NA object) to the log file.

When an NA object appears in most types of expressions, the interpreter simply propagates it to the expression's result. The intuition here is that if an operand has an unknown (NA) value, then the result should also be unknown. For example, if `x` is NA, then after executing `z = x + 5`, the value of `z` is also NA. Note that `x` and `z` refer to the same NA object since the interpreter re-uses (aliases) the NA operand of the addition as the result. This is not only more efficient than creating a new object, but it also propagates information about the original error to aid in debugging.

Note that this technique is most effective on scripts with short error propagation distances, like those that process each record independently of one another [15]. If script execution causes NA objects to taint all subsequent values, then the interpreter cannot produce much useful output.

Iterators: Scripts often compute aggregate statistics by iterating over a collection (e.g., a list) and performing computations such as summing up the elements. If a collection contains even one NA object, then that will taint the computation's result as NA. Thus, when iterating over a collection, the interpreter should skip over NA objects rather than emitting them. This provides the illusion that a collection contains no NA elements, so iterator-based aggregate computations can produce partial results rather than NA. At the same time, though, NA objects remain in the collection, so that direct indexing still returns the correct element.

Branch conditions: When a script must branch based on an NA condition value (e.g., in an `if` statement), which side should the interpreter take? Technically, neither is correct, since an NA value is neither TRUE nor FALSE. Thus, the interpreter can pick either side (our implementation always takes the `if` side rather than the `else` side). At the end of execution, the user can consult the log file to see which branches had NA conditions, so that he/she can be more suspicious about results dependent on those branches. We might consider adding a mode where the interpreter forks to take *both* sides of NA-based branches, but then we need to address the obvious challenges of exponential path explosion.

2.4 Special Handling for Assertion Failures

Programmers write assertions to specify constraints on their scripts' data. A regular interpreter crashes with an assertion failure as soon as one of these constraints is violated, since it is incorrect to continue executing with invalid data.

Our technique alters the interpreter so that when it encounters an assertion failure, rather than crashing, it creates an NA object and assigns it to all variables involved in the assertion. The intuition here is that when an assertion fails, only the data involved in the assertion is invalid, so it is safe to continue executing as long as they are marked as NA.

Here is the example from Section 2.2 augmented with an assertion that all computed ratios must be less than 1:

```
line_number = 1
for line in open('big_data.txt'):
  tokens = line.split(' ')
  ratio = float(tokens[0]) / float(tokens[1])
  assert(ratio < 1.0)
  print ratio
  line_number += 1
```

This assertion indicates that the programmer assumes all ratios in `big_data.txt` are less than 1. If some records violate this assumption, then the interpreter assigns `ratio` to a new NA object and prints "`<NA>`" when processing those records. At the end of execution, the log file shows which records led to assertion failures, so that the programmer can either fix the dataset or reconsider his/her assumptions.

2.5 Discussion: Benefits of Precision

The simple example we have shown so far might give the impression that our technique merely skips over bad records. A programmer could accomplish this same goal by simply wrapping the main loop body in a try-catch block and using a `continue` statement as the exception handler (to continue onto the next record). Also, data processing frameworks like MapReduce [9] can automatically skip over bad records.

Our technique is actually more precise, since it catches exceptions at the expression level. This means that if a record contains, say, 30 fields and the script throws an exception when processing one particular field, then only data from that field becomes NA, but the remaining 29 fields are successfully processed. Thus, our technique can automatically skip over portions of bad records rather than entire records. To mimic this level of precision, a programmer would need to wrap *every single program expression* in a try-catch block.

3. PYTHON IMPLEMENTATION

We implemented our technique for Python, since it is a popular language for writing data processing scripts. We created a prototype open-source interpreter named SlopPy (**Slop**py **Py**thon) [3] by adding 500 lines of C code to the Python 2.6 interpreter. SlopPy passes all of the Python regression tests and works on existing scripts and 3rd-party libraries without any code modifications. The behavior of SlopPy is identical to that of regular Python during normal (exception-free) execution, so scripts run at the same speed.

We implemented all features in Section 2 in a straightforward way by modifying how the Python interpreter handles uncaught exceptions and by defining a new NA type. We also hacked the interpreter's handling of iterators and Python generators [2] (generalized form of iterators) to skip over NA objects rather than emitting them during iteration.

When SlopPy encounters an uncaught exception, it unwinds the stack until it finds the first function *not* in the Python standard library and creates the NA object in that function's frame. In our experiments, this provided a better user experience since the user can more easily reason about exceptions in his/her own code rather than in library code.

SlopPy produces two warning log files: a human-readable text log, and a binary log of serialized Python objects that an incremental recovery script can directly process without needing to parse the text log (see Section 4.3).

4. PRELIMINARY EVALUATION

To demonstrate some of SlopPy's capabilities, we ran three informal experiments on a 3 GHz Mac Pro with 4 GB of RAM, with regular Python 2.6 and SlopPy both compiled as 32-bit binaries for Mac OS X 10.6.

4.1 Supercomputer Event Log Analysis

To show how SlopPy allows programmers to write simple scripts without worrying about error handling, we wrote a script to analyze an event log from the Spirit supercomputer installed in Sandia National Laboratories [4]. In 2007, Oliner and Stearley released event logs from 5 supercomputers [12]; for this experiment, we used the log for Spirit since it is the largest in size (37 GB). This log file contains 272,298,969 lines, where each line documents an event like an incoming network service request, kernel panic, or hardware failure.

System administrators routinely write ad-hoc scripts to query supercomputer event logs to monitor system health, discover aberrations, and diagnose failures. Oliner and Stearley describe how log files contain inconsistent or ill-defined structure, corrupted records, and duplicated records, all of which make it harder to write robust log analysis scripts [12].

For this experiment, we emulated a system administrator and wrote a script to print out the IP address of each machine that requested services from the Spirit supercomputer via the UNIX `xinetd` daemon. A sysadmin might use this data to plot a histogram and inspect the distribution of requests by IP addresses or corresponding geographic region; addresses with unusually high activity might indicate either an internal system malfunction or an intrusion attempt.

From a cursory glance at the log file, we saw that `xinetd` events seemed to obey a straightforward format, as shown in Figure 1. We wrote the simplest possible script to extract and print the IP addresses: It iterates over all lines of the log file, splits each into whitespace-separated tokens, finds lines whose 8th token (0-indexed) starts with "`xinetd`", then extracts the IP address from the 12th token, which should be formatted like "`from=172.30.80.251`". To avoid biases due to duplicated records, our script coalesces all events within a 5-second interval into one, which Oliner and Stearley also do in their analyses [12]. Here is our entire script:

```
cur_time = -99999

for line in open('spirit.log'):
  tokens = line.split(' ')
  utime = int(tokens[1])
  component = tokens[8]

  if component.startswith('xinetd'):
    # coalesce events within 5-second interval
    if (utime - cur_time) <= 5: continue
    else: cur_time = utime

    ip_addr = tokens[12].split('=')[1]
    ip_lst = ip_addr.split('.')
    ip_byte0 = int(ip_lst[0])
    ip_byte1 = int(ip_lst[1])
    ip_byte2 = int(ip_lst[2])
    ip_byte3 = int(ip_lst[3])
    print ip_byte0,ip_byte1,ip_byte2,ip_byte3
```

Running our script using SlopPy took 20 minutes, 4 seconds and printed 39,225 IP addresses. The SlopPy warning log showed that it caught 11,076 exceptions, which indicates that our script could not process 11,076 lines (out of 272,298,969 total lines, which is only 0.004%).

Since the SlopPy warning log contains the context of each exception (see Section 2.2), the values of the `line` local variable at each exception show the contents of all lines our script could not process. We searched through the `line` values and could not find any IP addresses, which means that our script extracted all of them (100% precision and recall). The 11,076 exceptions were all due to lines that looked similar to a syntactically-correct `xinetd` request record but did not contain an IP address. Example lines contained:

```
- 1104594301 2005.01.01 sadmin1 Jan 1 07:45:01 sadmin1/sadmin1 xinetd[2228]: START: rsync pid=616 from=172.30.80.251
- 1106166706 2005.01.19 sadmin1 Jan 19 12:31:46 sadmin1/sadmin1 xinetd[7746]: START: rsync pid=439 from=172.30.73.8
- 1107350922 2005.02.02 sadmin1 Feb 2 05:28:42 sadmin1/sadmin1 xinetd[7746]: START: tftp pid=8381 from=172.30.72.163
```

Figure 1: Example records for xinetd sessions in the Spirit supercomputer event log file [4].

- Bizarre numbers that are not valid IP addresses: e.g., `from=#564#`
- Null sentinel value: e.g., `from=<no address>`
- Various error messages: e.g., `xinetd[14743]: warning: can't get client address: Connection reset by peer`

Since the Spirit event log file was huge, we did not see these aberrant records when we manually looked through it to learn the schema in preparation for writing our script. All the `xinetd` records we saw followed the format in Figure 1, so we wrote a simple script that was only sufficient to process records in that exact format. SLOPPY freed us from having to think about error handling. In this case, our script achieved 100% precision and recall when extracting IP addresses, without requiring any error-handling code.

In contrast, when we run our script with regular Python, every time it throws an exception, the interpreter crashes and we need to edit the script to handle that exception and then re-execute. Depending on when the next exception occurs, each run can take from seconds to tens of minutes.

4.2 Computational Biology Case Study

To show how SLOPPY can add error tolerance to an existing script, we used it to run a computational biology script written by Peter, a Ph.D. student in our department.

When we first told Peter about our project, he immediately showed us a script where SLOPPY would have saved him a day of labor. Peter's 200-line Python script runs the Viterbi dynamic programming algorithm [6] on human genomic data and prints out a textual table of results. When he first ran the script, it computed for 7 hours (overnight); when he returned to his computer the next morning, he saw that it had crashed with an exception caused by taking the logarithm of zero[2]. He was especially frustrated since the exception occurred *at the very end of the run* when the script was post-processing and printing out the results table. In other words, all the real 7-hour computation was already done, but the script crashed while formatting the output.

After seeing that exception, Peter made a simple fix: He defined a custom `log` function that returns a sentinel "negative infinity" value for `log(0)`. When he edited and re-ran his script, everything worked fine, but he lost a day of productivity just because of an unexpected exception.

When we ran the original version of Peter's script using SLOPPY (before he patched the `log` function), it finished in 7.5 hours, printing a results table with 328,879 rows and 16 numeric columns (5.2 million total numbers). The SLOPPY warning log showed that it caught 35 exceptions, all of which arose from taking the logarithm of zero. The 35 corresponding NA values appeared in the results table as entries that printed as an "`<NA>`" string rather than an actual number.

Since the exceptions occurred at the end of execution after the input data had been transformed and mixed across several matrices, there were no individual "bad records" to blame for the crashes. It just happened that in 35 rare cases, running the Viterbi algorithm and post-processing on some combination of fields within input records led to `log(0)` being performed. Peter developed and tested his script on a small dataset so that each run only took a few seconds. He never saw the `log(0)` exception during testing; it only occurred after running for 7 hours on the full dataset.

If Peter had run his script using SLOPPY rather than regular Python, he would be able to see full results after the first run. Even though the results table contains 35 NA values (out of 5.2 million total values), he knows that all NA values represent `log(0)`, so he does not lose any information.

4.3 Incremental Recovery for HTML Parsing

To show how SLOPPY enables incremental error recovery, we wrote a simple script to compute a reverse webpage link graph from a corpus of HTML files. When we ran the script, it failed to parse 1% of the files, but we were able to process the SLOPPY warning log to recover some data from those unparsable files and merge them into the original graph.

A reverse web-link graph associates each HTML webpage with a set of webpages that link to it. A search engine can use this graph to compute reputation metrics like PageRank. We took this example from the MapReduce paper [9].

For simplicity, we wrote a sequential script to compute the reverse web-link graph, but SLOPPY should provide the same benefits for a MapReduce-style Python script. Our script iterates over 71,768 HTML files we downloaded from a 1.3GB public corpus [5], parses each using the HTML parser from the Python standard library, finds all outgoing links, and builds a graph mapping target URLs to source URLs.

Our script takes 10 minutes, 19 seconds to run to completion with SLOPPY and generates a graph with 246,932 nodes and 1,256,808 edges. The warning log showed that the standard Python HTML parser threw an exception on 736 out of the 71,768 HTML files in our corpus (1%). HTML files often do not conform to W3C standards, so they crash parsers with messages like these that appeared in our log:

```
Error: unexpected '\\xa9' char in declaration
Error: expected name token at \'<!:iB;@ i39#.6o;H8g" \'
```

SLOPPY allows our script to tolerate these parse errors and continue processing rather than crashing; in the end, 99% of the files were properly processed in the initial run.

The warning log contains the values of local variables at the time each exception was thrown: One of the variables contains the string contents of the unparsable HTML file, and another contains its filename. Thus, a script could directly process that log to access the contents of the 736 HTML files (1%) that our original script could not parse.

We modified our script to not use an HTML parser (since that would just crash again) but instead to use a regular expression to extract outgoing links. The reason why we did not originally use a regular expression was because it is less accurate than a real parser, so it might miss some links. However, a regular expression is more robust, since it does not care whether the HTML conforms to W3C standards.

We ran our modified script on the warning log and merged the newly-found links into the existing reverse web-link graph.

[2]The Python math library throws an exception for `log(0)`

This recovery run completed in 6 seconds and added 1116 new nodes and 4987 new edges to the graph, which only made it 0.4% larger. We call this run *incremental* because it did not re-process the entire corpus, which would have taken the full 10 minutes. Although 10 minutes is not too long to wait, if we had access to a larger HTML corpus, then a full run could have taken hours, even when parallelized.

5. RELATED WORK

Failure-oblivious computing: Our technique was inspired by failure-oblivious computing, a technique that makes C programs immune to memory errors by ignoring out-of-bounds writes and returning fake values for out-of-bounds reads [15]. Failure-oblivious computing returns ordinary small integer values for erroneous memory reads, whereas our technique creates and propagates a special NA object and snapshots execution context, to aid in debugging and incremental error recovery. Also, since failure-oblivious computing works on C code, it requires re-compiling the target program and incurs a slowdown due to memory bounds checking, whereas we can transparently deploy our technique by replacing the Python interpreter with SlopPy and incurring no slowdown.

Error tolerance in data processing systems: Google's MapReduce [9] and the open-source Hadoop [1] both have a mode that skips over bad records (i.e., those that throw exceptions when processed). Our technique is similar in spirit but is finer-grained, enabling it to skip over portions of bad records (see Section 2.5). The Sawzall domain-specific data processing language (built on top of MapReduce) can also skip over portions of bad records [14], but it lacks the generality of a language like Python. Also, unlike the above systems, SlopPy stores exception contexts (i.e., values of local variables), which aids debugging and allows the programmer to incrementally re-process only the bad records and then merge with the original results (see Section 2.2).

Silent errors in programming languages: Some programming languages, most notably Perl, are designed to silence errors as much as possible to avoid crashing scripts [7]. For example, Perl and PHP automatically convert between strings and integers rather than throwing a run-time type error, which seems convenient but can produce unexpected results. Although SlopPy shares the same goal of making scripts robust to crashes, it *does not* silently hide errors. Instead, SlopPy taints erroneous values as NA, logs warning messages, and propagates NA so that the programmer knows which portions of results are derived from NA values.

6. DISCUSSION AND FUTURE WORK

Our design philosophy underlying SlopPy is that programmers doing ad-hoc data processing tasks can be more productive if the run-time system allows their imperfect, buggy programs to run to completion and produce partial results rather than mercilessly crashing them. We want programmers to be able to rapidly hack on ad-hoc scripts and discover insights about their data without worrying about the mundane details of error handling. This project is still in its early stages, so here are some directions for future work:

Since SlopPy allows incorrect programs to keep running, it is vital to have their partial results be accompanied by a precise explanation of what went wrong. The warning log file that SlopPy produces is a first step towards this goal, but there is still plenty of room for improvement. We would like to accurately track both control- and data-flow dependencies of NA values and report this provenance data in such a way that the programmer can easily find out what went wrong, why, and how it can be fixed. We would also like the programmer to get a realistic sense of how much he/she can "trust" particular components of the output, and perhaps even get probabilistic bounds on how badly a particular NA value might corrupt indirectly-affected numerical results.

Also, we might want to explore more principled ways of isolating the buggy parts of execution from the correct parts, in order to preserve the validity of partial results. For example, if NA-tainted execution were about to delete previously-computed results, then it might be better to actually terminate execution rather than continuing to execute.

Finally, since more and more data processing is being done in the cloud (e.g., via Amazon EC2 and MapReduce), could ideas from SlopPy be generalized to work on parallel and distributed data processing systems? One unique challenge in this space is that these systems are often comprised of multiple layers implemented in different languages (e.g., SQL, Hive, Pig, Java, Python), so error handling, propagation, and reporting must cross language boundaries.

Acknowledgments: Thanks to Dawson Engler, Kathleen Fisher, Robert Ikeda, and Jean Yang for feedback on drafts, to Martin Rinard and an anonymous reviewer for some of the future work ideas, and to the NSF fellowship for funding.

7. REFERENCES

[1] Apache Hadoop home page `http://hadoop.apache.org/`.

[2] Python home page: PEP 255 — Simple Generators `http://www.python.org/dev/peps/pep-0255/`.

[3] SlopPy home page: source code and documentation `http://www.stanford.edu/~pgbovine/SlopPy.html`.

[4] Supercomputer Event Logs `http://www.cs.sandia.gov/~jrstear/logs/`.

[5] The Phoenix System for MapReduce Programming `http://mapreduce.stanford.edu/`.

[6] Viterbi algorithm `http://en.wikipedia.org/wiki/Viterbi_algorithm`.

[7] Stopping silent errors with exceptions `http://perltraining.com.au/tips/2005-04-12.html`. *Perl training Australia*, 2005.

[8] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce online. In *NSDI '10*. USENIX Association, 2010.

[9] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI '04*. USENIX Association, 2004.

[10] K. Fisher and R. Gruber. PADS: a domain-specific language for processing ad hoc data. In *PLDI '05*, pages 295–304. ACM, 2005.

[11] N. Gehani. Exceptional C or C with exceptions. *Software - Practice and Experience*, 22(10):827–848, 1992.

[12] A. Oliner and J. Stearley. What supercomputers say: A study of five system logs. In *DSN '07*, pages 575–584, Washington, DC, USA, 2007. IEEE Computer Society.

[13] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08*, pages 1099–1110. ACM, 2008.

[14] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4):277–298, 2005.

[15] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee, Jr. Enhancing server availability and security through failure-oblivious computing. In *OSDI '04*. USENIX Association, 2004.