Detecting Unread Memory using Dynamic Binary Translation

Jon Eyolfson and Patrick Lam

University of Waterloo

Abstract. Reading from uninitialized memory—that is, reading from memory before it has been written to—is a well-known memory usage error, and many static and dynamic tools verify that programs always write to memory before reading it. This work investigates the converse behaviour—writes that never get read, which we call "unread writes". Such writes are redundant—at best, they do not perform any useful work; furthermore, work done to compute the values to be written could corrupt the program state or cause a crash. We present a novel dynamic analysis, implemented on top of the Pin dynamic binary translation framework, which detects instances of unread writes at runtime. We have implemented our analysis and present experimental data about the prevalence of unread writes in a set of benchmark applications.

1 Introduction

Modern languages and compilers detect memory usage errors caused by reads from uninitialized memory: in Java, it is an error to read variable `x` before writing a value to it, and `gcc` warns about uses of uninitialized variables. Programs also contain the converse phenomenon: writes to memory which are never read. Such writes are redundant; at best, they don't perform any useful work. Computations that produce values used only in unread writes do not contribute to the goal of the program, gratuitously consume computational and memory resources, and may, in the worst case, crash the program—for example, the Ariane 5 crash was caused by an exception while computing an unused value[1].

Because compilers detect memory problems ahead of time, most compilers only report errors and warnings at an intraprocedural level, and only for local variables and private fields of classes. (`gcc` 4.6, for instance, reports warnings for unused but set variables.) Static approaches to memory error detection require detailed pointer information to detect memory errors on heap accesses: the compiler needs to know which heap references may and must alias, so that it can determine the access history of individual abstract memory locations. Must-alias analysis is critical for reducing the rate of false positives. However, implementations of whole-program must-alias analyses are rare.

Recently, Valgrind's Memcheck tool [1] has used dynamic binary translation to detect memory errors, including reads from uninitialized memory, at runtime.

[1] Section 2.1, http://www.di.unito.it/~damiani/ariane5rep.html

Purify [2] detects a similar class of errors by inserting instrumentation code at compile time. In either case, runtime verification can ensure the absence of memory errors on an observed execution. Dynamic analyses need not reason about the heap, as a pointer comparison suffices to disambiguate heap addresses.

Our Tracerory tool implements a dynamic analysis to detect unread memory in realistic C and C++ applications. It supports multithreaded programs. We detect 1) *unread memory allocations* and 2) *unread writes* to the heap—writes with no corresponding read. When a developer runs their code under Tracerory, it reports instances of unread memory. Developers can use the report to manually inspect flagged program points and fix their code.

Figure 1 shows a high-level overview of our tool's operation. Tracerory takes two inputs: an executable to be monitored, and specifications about which parts of the program to monitor. While Tracerory executes the program, its runtime monitor processes the stream of memory allocations, reads, and writes, reporting unread writes and memory allocations.

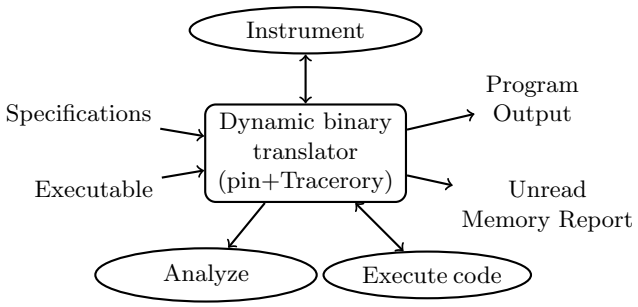

Fig. 1. Tracerory operation.

The contributions of this paper are:

- the identification of unread memory as a source-level phenomenon of interest;
- a novel dynamic analysis to detect unread memory in programs at runtime;
- an implementation of our dynamic analysis in the Pin dynamic binary translation framework; and
- qualitative and quantitative results outlining the prevalence of unread memory in a collection of open-source benchmarks.

2 Overview

This section presents, using an example, the two suspicious memory usage patterns that our Tracerory dynamic monitoring tool detects. Section 5 presents additional instances of unread writes drawn from real-world programs.

```
1  int main(int argc, char *argv[])
2  {
3      X* x = new X();
4      Y* y = new Y();
5      for (int i = 0; i < 4; ++i) {
6          x->data = i;
7          y->data = i;
8      }
9      cout << x->data << endl;
10     delete x;
11     return 0;
12 }
```

Fig. 2. Example program with unread allocations and writes.

```
Allocations unread and/or with unread writes: 2
Unique allocation sites unread and/or with unread writes: 2

Created: example.cpp:3 (1)
Destroyed: example.cpp:10 (1)
Unread: false (1)
Unread Writes (3):
    example.cpp:6 (3)

Created: example.cpp:4 (1)
Destroyed: [not destroyed] (1)
Unread: true (1)
Unread Writes (4):
    example.cpp:7 (4)
```

Fig. 3. Tracerory output for motivating example.

The example program in Figure 2 allocates two objects, `x` and `y`. It then performs 4 writes to each object, reads from `x`, and finally deletes `x`.

Lines 3 and 4 allocate the memory objects, which we initially mark unread. Next, line 6 writes to `x` and line 7 writes to `y`. Our tool records the first two writes to `x` and `y` in the first iteration of the loop and marks the memory locations as having an active unread write. In the second iteration, the new writes overwrite the previously active unread writes. The tool marks the writes from the previous iteration as unread. At the end of the loop, there are 3 unread writes and 1 active unread write to each object.

Finally, the program reads from `x` on line 9 and deletes it on line 10. If an object is not deleted, we implicitly delete it when the program terminates (e.g. `y`). After an object is deleted, no further reads can be made to it. Therefore, we report the active unread write to `y` (in the last iteration of the loop). We also report the object itself as completely unread. Upon exit, our tool reports the unread object `y`, plus all unread writes to heap objects.

Tracerory outputs, for each "bad" object (with unread writes or itself unread): the location that created and destroyed the object; whether or not it is an unread object; and the number of unread writes to the object, along with the locations which performed the writes. If the tool observed a constructor call for the object, it outputs the object's type. To minimize false positives, the tool

only reports statement s as an unread write if all previous dynamic writes at s are unread. For instance, if s occurs in a loop, then our tool only reports s if all of its executions perform unread writes.

Figure 3 shows Tracerory's output for our example program. First, Tracerory reports `x` as an object of type `X` with 3 unread writes. Next, it reports `y` as an object of type `Y`, with 4 unread writes, which is completely unread. The unread writes on `x` indicate potentially-important information being ignored; writes to `y` may correspond to wasted memory and redundant, potentially harmful, work.

As is standard for dynamic analyses, we only report unread writes from a single program execution at a time. It is the responsibility of the developer to execute the program with enough test coverage to adequately explore its behaviour. A particular write may be unread for some, but not all, inputs. While such a write is most likely not problematic, we believe that the developer is best-placed to decide whether code changes are appropriate in such cases; perhaps the write could have been avoided on that input.

To help developers prioritize the generated reports, our tool coalesces and sorts its output. That is, it combines all objects allocated at the same static site, and displays a count of "bad" objects, unread objects, and unread writes for all objects allocated at that site. It lists the allocation sites which account for the most unread objects first. In the future, we hope to combine unread write reports from multiple executions, thus increasing the relevance of the reports.

3 Dynamic Analysis

To validate our design, we implemented the Tracerory tool atop the Pin dynamic recompilation toolkit [3]. Our tool works on x86-64 Linux binaries. Pin supports multithreaded programs and we have used appropriate data structures to ensure that Tracerory also supports multithreading. Because Pin's API provides an abstraction layer, Tracerory should also work on x86 binaries.

Generally, Pin tools run in two phases: a (slightly) ahead-of-time instrumentation phase, and an monitoring phase. Section 3.1 describes the instrumentation phase while Section 3.2 describes the analysis phase, which implements a runtime monitor to detect unread memory.

Our unread memory detection only monitors images (binaries and libraries) explicitly specified by the user. We call such images "watched images;" watching only specific images enables developers to focus their attention on memory usage which they are responsible for and can fix.

3.1 Instrumentation Phase

The instrumentation phase transforms the input executable to invoke our runtime monitor, which will be described in Section 3.2. Here, we describe our instrumentation points and the information they pass to the monitor.

Allocations and Deallocations Our tool records all memory management calls by instrumenting standard C/C++ allocation and deallocation functions. For allocation functions (`malloc`, `calloc`, `realloc`), we insert a call to our monitor at the function entry and exit points. At the entry point, we pass the `size` argument to the monitor. At the exit point, we pass the returned pointer to the monitor. For deallocation functions (`free`), we instrument the function entry point and pass the pointer argument to the monitor.

Memory Accesses Our tool instruments every memory read, plus memory writes from watched images. For both reads and writes, we pass all accessed memory addresses to the monitor. For writes, we also pass the instruction pointer.

Debugging Information To help developers localize memory problems, our tool uses debug information to identify all program events (allocations, deallocations, reads, writes) by source code line number.

At each `call` instruction, our tool records, in thread-local storage, the debugging location and image name immediately before the call; this information is then available upon entry to the callee. A debugging location consists of a source line number, when available, or the procedure name and memory offset otherwise. It enables the tool to attribute memory allocations and deallocations to the code that requested or released the memory. Our tool uses the image name to ignore memory allocations from unwatched images.

Dynamic loaders introduce indirect calls, which may overwrite the debugging information as they call helper functions to load the function and resolve the function address. To prevent this, we ignore calls to the dynamic loader library. For 64-bit Linux, this library is `/lib64/ld-linux-x86-64.so.2`. We found that ignoring the dynamic loader does not negatively affect our tool, as most applications do not interact directly with it.

Virtual Functions Our initial results included many unread writes to objects with virtual functions. These were writes to the virtual table, which developers do not control. Such writes should be ignored. We found that source locations for virtual table writes corresponded to definition points for member functions. We collect these source locations by inspecting every member function and recording its definition point; the monitor then ignores these locations.

3.2 Monitoring Phase

To enable the classification of memory accesses, our tool records immutable facts about each block of allocated memory (memory block), along with facts which change during the execution. Figure 4 illustrates the structure of a memory block. Our monitor stores memory blocks in an append-only list of unread memory blocks and a map of currently-allocated blocks keyed by base address.

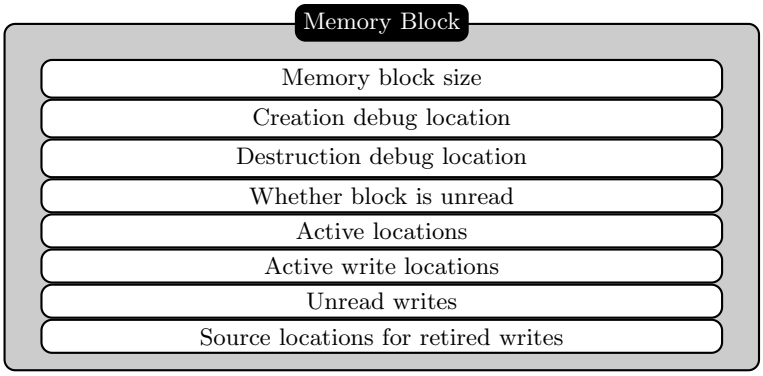

Fig. 4. Structure of a Memory Block.

Allocations At a call to an allocation function (e.g. `malloc`) originating from a watched image, the monitor records the requested memory size and calling source location in thread-local storage. Upon exit from the allocation, the monitor receives the returned pointer and creates a new currently-allocated memory block using that pointer as the base address. It also initializes the block's size and created location with the values recorded on entry and marks the block unread.

We take care to collect reliable source locations for custom allocator wrappers. Consider C++ object allocation; the `new` call is essentially a wrapper for `malloc`. Ordinarily, our tool would report a debugging location for `malloc` from within the `new` implementation. However, we would prefer to know `new`'s caller rather than `malloc`'s. To do this, we instrument the entry point to `new`, ensuring that the caller belongs to a watched image. If it does, we record the calling source location for the `new` and ignore the watched image check for the `malloc`. We support arbitrary user-specified custom wrappers in the same style.

Reads and Writes To analyze a memory access, the tool looks up the requested memory address in the allocated-memory structure. It uses the `lower_bound` operation of the C++ STL `map`. If the address falls in the range `[address, address + size)` for any blocks, the tool then carries out the appropriate update on those blocks. The tool assumes that memory accesses occur at machine word granularity. The mutable part of the per-block structure includes a flag indicating whether the block has ever been read, as well as the following sets: active writes to the block; active locations (which have been accessed at least once after initialization); unread writes (a list); and source locations for retired writes (those which have been read). We ensure that concurrent threads do not simultaneously update the block structures.

The per-instruction updates are as follows:

– At a *read*: the monitor marks the containing memory block as read and removes any writes to the requested location from the active writes. It adds any removed writes to the set of retired writes.

To reduce the false positive rate, we only report unread writes from static program points from which no writes are ever read—equivalent to applying intersection to unread writes. Hence, if we ever observe a retired write from a given static program point, we filter out writes from that program point.
- At a *write*: if the write's destination already has an active write, then that value will be overwritten, hence unread. We thus add the previous write to the list of unread writes, if its source location belongs to a watched image. The tool also adds the current write to the list of active writes, if the source location is not retired.

At every memory access beyond the initial write, we mark the destination location as active. Our monitor uses this to omit initial values which are never subsequently accessed, when processing the block's deallocation.

Deallocations Deallocations remove memory blocks from the set of currently-allocated blocks, moving them to the unread writes structure if appropriate. We only add active writes to unread writes if the memory location was active, i.e. accessed at least once after initialization. We move a block to the unread writes structure if the block has at least one unread write or is itself unread. We also set the block's deallocation location.

We do not need to instrument the `delete` function: an instrumented destructor call will always happen first. At a destructor, if `this` is an allocated memory block, our monitor follows the same process as for `free`.

Program Termination Upon program exit, the tool simulates `free`s for all currently-allocated blocks, with a deallocation location of "[not destroyed]". It then traverses the heap structure abstraction and outputs summaries for all of the unread memory blocks and unread writes. To help the developer prioritize the most important program points, the tool sorts its output, putting blocks which account for more writes first. Within each block, the tool also sorts unread writes by descending order of unread write count. For each entry, the tool outputs the location where the memory block was created, the location where it was destroyed, whether or not it was unread, and its unread write locations.

False Positives We summarize some false positives that Tracerory will report. Some false positives are due to analysis imprecision, and could be filtered out.

- Our analysis identifies many field initializers in C++ as unread writes; classes will often explicitly initialize all of their fields in the constructor and then re-initialize the fields later. The first initialization is unread.
- Data structure implementations in watched images often lead to unread writes. For instance, we found a implementation of linked list insertion:

```
*new_edge = edge; new_edge->next = stl->tail;
```

`next` is copied from an input and immediately overwritten, hence unread.

- Other false positives include idioms like resetting pointers to NULL after freeing them. Although this is good programming practice, it causes unread writes—freed pointers are never read.
- Our tool does not capture block memory accesses. Although we did not observe any instances of spurious unread writes caused by block accesses in our benchmarks, system calls like `read()` may potentially use DMA, which would cause our analysis to miss some reads.

Remark on concurrency In multithreaded programs, the execution depends on both the input and the scheduler. Our monitor only reports what happens on one execution; a write may be unread on an execution and read on another execution, even if the accesses are properly protected by locks. This is because mutual exclusion locks (without condition variables) do not impose an ordering. We believe that a write that is unread on any execution ought to be investigated as suspicious code; it is even more suspicious if it is only unread on some (but not all) executions. Such a program's results depend on the scheduler.

4 Formal Definitions

We continue by giving a precise definition of an unread write and formally stating the property that our runtime monitor enforces. The runtime monitor watches an execution trace on-line.

Definition 1 *An* execution trace t *is a sequence* $t = t_1, \ldots, t_n$ *of executed instructions* $t_i = \langle pc_i, op_i \rangle$*, where* pc_i *is a program counter value and* op_i *is an operation. Operations include allocations, reads (*$\mathsf{READ}\ addr_i$*) and writes (*$\mathsf{WRITE}\ addr_i$*), where* $addr_i$ *is of the form* $base_i + off_i$*.* $base_i$ *is a block's base address, returned from a previous allocation call.* off_i *is an offset into a block.*

Our definition of execution traces uses the scheduler's interleaving of instructions from different threads. A single input may give rise to multiple execution traces.

We can now define the notion of an unread write.

Definition 2 *An* unread write *is an executed instruction* $t_u = \langle pc, \mathsf{WRITE}\ b+o \rangle$ *with no subsequent* READ *from* $b+o$ *in that execution trace, such that 1) there is no preceding pair in the trace* ($t_p = \langle pc, \mathsf{WRITE}\ b+o' \rangle$, $t_{p'} = \langle pc', \mathsf{READ}\ b+o' \rangle$)*, where* $p < p' < u$*, and 2) there exists some other access to the same location,* $t_i = \langle pc'', op\ b+o \rangle$*, where* $i \neq u$*.*

The definition primarily states that the memory location of the write must not subsequently be read. However, an otherwise-unread write should not be considered unread if any previous instruction at the same program counter value wrote to the same block and that value subsequently got read. Also, we do not report an unread write if it is the only access to a memory location; such writes are often one-time memory initializations.

Using Definition 2, we can state what our runtime monitor is looking for.

Proposition 1 *The runtime monitor described in Section 3 detects all unread writes in an execution trace.*

The proposition follows immediately from the design of our runtime monitor.

5 Experimental Results

In this section, we present the results of our unread memory analysis on a series of benchmark programs. We found that our tool successfully identified a number of instances of suspicious code as well as writes that were useless for a given execution. On our benchmarks, Tracerory caused a slowdown of 87× (geometric mean) over the original execution time, demonstrating its feasibility for occasional use on real codebases.

5.1 Qualitative Results

The main experimental results in this paper demonstrate the efficacy of our tool on five benchmarks: abiword, sqlite, crafty, ImageMagick, and Python. In all cases, our unread memory tool identified interesting code within the benchmarks; two of the benchmarks could be improved using the tool results, while the results illustrate some perplexing behaviour by ImageMagick.

abiword AbiWord is a word processor written in C++. We used version 2.6.8 of AbiWord, which contains over 559,000 lines of code. Although we explored a number of workloads, we will present results from a run of AbiWord's command-line file-conversion mode which converts a 1.28M AbiWord file into plain text. Since AbiWord has not been tuned for performance, we expected to find a number of unread writes in its codebase. In addition to the base executable, we added `libabiword-2.8.so` and plugin libraries to our watched images.

The top sources of unread memory were utility routines, particularly string and vector implementations. For instance, AbiWord allocates 116,436 strings which it never reads. AbiWord also allocates 6,081 completely unread vectors. Note the role of watched images here: had AbiWord used the standard STL implementation, our tool would assume that the developers weren't interested in modifying the STL, and would therefore not report these writes. On the other hand, because the offending allocations and writes lie in AbiWord code, Tracerory reports these routines.

The remainder of the discussion presents domain-specific unread writes. We will ignore library-like unread writes.

- We found 11,336 unread writes to the private `m_leader` field in the `fp_TabRun` class. This field is never read on the file-conversion executions; it is only accessed by the `_draw()` method of `fp_TabRun`, which is never called on a file-conversion workload. There are no calls to the `getLeader()` method anywhere in the code.
  There are also 11,336 writes to the private `m_tabType` field, which is never read on this workload, or outside its defining class on any workload.

- We also found about 28,000 unread writes at each of the `fp_Run::setTmpLine`, `::setTmpX`, `::setTmpY`, and `::setTmpWidth` methods. These methods are only called by the `format()` method, and there are no other writes of the fields. The only reads of these fields are in the `clearIfNeeded()` method, which is only called by `format()`. It appears that `clearIfNeeded()` only executes when the document is reflowed, which never occurs on the file-conversion workload. We investigated the underlying fields and found that they were used to store the previous metrics of the run, allowing AbiWord to decide whether it actually needs to reflow the text. The text-conversion workflows only reflow the text once.
- Finally, we found 7,085 unread writes of a private field `m_iDrawWidth`. The write follows a discussion, in the comments, about the proper value for this field. It appears that the value does not matter on this workload, at least. It is, however, read in the `_clearScreen()` and `_draw()` methods, which are invoked in other workloads.

These examples illustrate how our tool correctly identifies writes which are redundant on a given execution.

sqlite SQLite is a ubiquitous SQL database engine. We examined version 3.7.13 of SQLite (138,797 lines of C code) under its provided "zerodamage" workload and found a number of unread writes. We will describe the first three unread writes that our tool found.

- The first two unread writes are both to the `CellInfo` structure and enable SQLite to handle cases where an SQLite cell overflows a page. Tracerory reported 1,999 unread writes to the `nPayload` and `iOverflow` fields of `CellInfo`. The `nPayload` field is seldom read; two of the reads occur in assertions and a third is compiled in conditionally. The only regular read of this field is in the `clearCell()` function, which must not have been called on this execution. Our tool could help in ensuring good test coverage—it seems that it would be worthwhile to specifically craft a test to verify that the field value is correct. The `iOverflow` field is read more often than `nPayload`, with 6 static instances of reads in the code. (One of these reads is never compiled and belongs to a function annotated with the comment "This function does not contribute anything to the operation of SQLite.") The other reads occur in functions like `clearCell()` and `fillInCell()`.
- The third unread write is to the `validNKey` field of the `BtCursor` structure, a cursor over sqlite's central b-tree data structure. This field only has one read, which occurs in the static function `sqlite3BtreeMovetoUnpacked()`. It is written to 11 times across different parts of the SQLite code. Our tool suggests that it might be worthwhile to closely inspect these writes to ensure that they are correct, as they are not often used on this workload.

crafty Crafty is a chess program and one of the SPECCPU benchmarks; version 23.4, which we examined, contains 34,792 lines of C code. We ran crafty's

included "bench" command after editing the code to evaluate only the first position (for performance reasons). Because Crafty is tuned for chess competitions, we did not expect to find inefficiencies in its main loop; however, all three unread memory reports from Tracerory were instances of suspicious or buggy code.

We manually investigated each of the unread memory blocks and found a number of code idioms which could be improved:

- We learned that crafty contains code to parse its command-line options—it does not use a library. The `main()` function allocates space for 512 potential arguments, each of maximum length 128, and calls `ReadParse()` to copy the arguments into its buffer. Since our test run does not use any command-line arguments, the allocation for the arguments is unread memory, and Tracerory lists it in its output, as we would expect.
  Furthermore, inspecting the code, we found a buffer overflow: it does not check that the command-line arguments are shorter than the buffer.
- We found 973,169 unread writes in one of the memory blocks allocated in the `InitializeHashTables()` function. The accompanying comment indicates that this function is supposed to completely clear the `pawn_hash_table` between test positions. The code itself iterates through an array and sets all but one field to 0; the remaining field gets -1.
  Calling `memset()` would be a more efficient way to clear the memory, and would be less likely to leave forgotten state around (especially in the context of program maintenance, where a developer might add a new field to the struct stored in the hash table.)
  We were surprised that our tool reported this code, since it appears to be initialization code. However, on our test run, `InitializeHashTables()` executes twice; our tool reports the second set of writes as unread writes.
- The final unread memory block points out code marked as a kludge in the comments. When crafty is asked to log its commands, it searches for the first nonexistent or small file named `log.NNN`. It uses `fstat` to identify small files if they already exist, but unconditionally allocates (and does not deallocate) the memory block for the `stat *` return information from `fstat`.

Although the problems in this benchmark were not directly caused by unread writes, we believe that it was useful to run Tracerory on crafty—inspecting unread memory in crafty pointed us to bugs and inefficiencies in the code.

ImageMagick ImageMagick is a collection of tools for manipulating images, which consists of over 400,000 lines of code as of version 6.7.4-9. This benchmark uses many external libraries to open and process images, such as `libjpeg`; in this section, we report only the behaviour of the `convert` binary while watching ImageMagick-6.7.4-9 with its libraries `libMagickCore` and `libMagickWand`. ImageMagick is highly tuned for performance, and we did not expect to find many inefficiencies in its code. In addition, the README reports that the maintainers perform a "comprehensive security assessment that includes memory and thread error detection to prevent security vulnerabilities" before each release.

We watched ImageMagick resize a picture from its original size of 1404×625 to a new size of 1280×720. We manually investigated some of the reported results from the unread memory tool, and present our findings below.

- Our tool reported 729,600 calls (somewhat, but not exactly, related to the number of pixels in the output) to `SetPixelOpacity` originating from the source file `resize.c`. The offending line is

  ```
  SetPixelOpacity(q, ClampToQuantum(pixel.opacity));
  ```

  We found that JPEG does not encode opacity—ImageMagick manufactures `OpaqueOpacity` for each pixel from JPEG input and discards it upon write. The writes to opacity are therefore redundant work on this workload.
- The largest offender, accounting for 2,632,500 writes, was `jpeg.c`. Unfortunately, this appears to be a false positive; ImageMagick converters write data to a temporary buffer, `QueueAuthenticPixels`, with the values of the pixels' red, blue, green and opacity channels. `ImageMagick` seems to write the data 1 byte at a time, and read 4 bytes at a time (which accounts for the number of unread writes $1404 \times 625 \times 3$). There are no reported unread writes for the blue channel—the blue channel is the offset that gets read.
- The third-largest source of unread writes, accounting for 2,944 writes, is apparently also a spurious report. These writes copy the ICC colour profile. The code is rather opaque and worth examining in detail for possible bugs, since the effect of the code is not obvious at all (Figure 5).

```
p=GetStringInfoDatum(profile);
for (i=(ssize_t)GetStringInfoLength(profile)-1;
     i >= 0; i--)
  *p++=(unsigned char) GetCharacter(jpeg_info);
```

Fig. 5. ImageMagick code showing an unread write.

We verified that commenting out the writes does change the program output (although not visibly, as colour profiles are only used in internal calculations).

Tracerory pointed out a number of interesting idioms in the ImageMagick code. Our false positives tell us that ImageMagick produces output by batching up reads to its buffer, inconsistent with the original per-byte buffer write mode.

CPython CPython is the default bytecode interpreter for the Python programming language. This application is multithreaded. It consists of over 350,000 lines of C code as of version 3.3.0a3. We ran the 366 individual tests shipped with CPython and watched the CPython executable, the `libpython` library, and all other Python libraries built in the standard configuration.

- Our tool reported 55,343 unread writes to the `overflowed` field of the `PyThreadState` object. Deeper inspection revealed that the implementation for protecting the stack from overflowing is in a haphazard state. The code also referenced a mailing list discussion which pointed out potential problems with the implementation. Our tool adds to the discussion by pointing out that the write which clears the overflowed flag is often unread.
- Our tool also reported unread writes to other parts of the interpreter state, including the line number `f_lineno`; and the previous instruction `f_lasti`, on `LOAD_FAST` and `STORE_FAST` instructions. The code revealed that the interpreter reads `f_lineno` only in tracing mode, which we were not using; those writes are therefore unnecessary in the interpreter's normal operation. The writes to `f_lasti` are generated by a macro. That field is used to report the current line number (e.g. when generating a stack trace) and during tracing.

Summary Our tool illustrates the additional complexity added by unused modes of operation, as with Python and its tracing function. The extra state for unused modes are unused in normal operation. Such rarely-accessed state is likely to be less reliable than state which is regularly used. Programs with fewer modes will certainly be simpler than programs with more modes.

Internal library (vectors and strings) usage accounted for many of the unread memory reports. Some string implementations, e.g. AbiWord and python, track the string length and zero-terminate the string; we observed 152952 unread writes of the final 0 on one of our test cases. AbiWord's string implementation also includes the buffer length, which is unread for any string which is never grown. We observed thousands of unread writes of AbiWord vector elements.

5.2 Performance

To establish that our tool's performance is adequate, we timed it on a number of benchmarks, including the qualitative benchmarks above. Our test system is an Intel Core i7-3930K at 3.20GHz running ArchLinux GNU/Linux, and our tool runs on top of Pin 2.11 (release 49306). We compared the base runtime (without Pin) to the runtime with Pin alone and with our tool. Reported times are an average over three runs. Figure 6 presents our analysis times. The geometric mean of our slowdown compared to the raw execution time is 87×, while we add a geometric mean of 14× slowdown over pin with no instrumentation.

6 Related Work

We discuss two areas of related work. First, we summarize past work on investigating writes to memory; the related work in that area attempts to reduce memory bandwidth, while we are advocating the use of unread writes to improve code quality. Next, we discuss alternatives in the dynamic binary translation space, including other memory checkers which also use dynamic translation, and other applications of dynamic binary translation tools.

	raw (s)	pin (s)	pin slowdown	tracerory (s)	tracerory slowdown/raw	tracerory slowdown/pin
imagemagick	0.22	2.78	12.6	37.76	172	13.6
python	170.63	208.24	1.22	224.63	1.32	1.08
abiword	28.92	63.45	2.19	7116.43	239	112
ffmpeg	2.31	6.76	2.93	446.46	193	66.0
crafty	15.57	22.36	1.44	906.49	58.2	40.5
sqlite	0.01	3.47	347	7.21	721	2.08

Fig. 6. Unread Memory analysis times.

6.1 Optimizing Memory Writes

The program transformation most closely related to the present research is the store elimination transformation proposed by Ding and Kennedy [4]. Their work generally attempts to reduce applications' memory bandwidth usage. They propose a loop-based transformation, store elimination, which eliminates redundant writes inside loop bodies. In store elimination, some loops write values back to an array while performing a computation of some summary information (e.g. a sum) over the array. If the code never reads the final values written to the array, then store elimination will eliminate the unread writes. Because our goal is to examine all of the program code for potential bugs, our dynamic analysis does not focus on loops and arrays, but rather considers all writes to the heap.

Trace optimizers like Dynamo [5] and rePLay [6] eliminate unread writes at runtime. A pass through a trace that is about to execute suffices for removing some unread writes, and such an optimization is therefore standard in the trace optimizer context. A more-powerful dynamic analysis could eliminate most unread writes. Our work, however, aims to help developers improve program quality by enabling them to remove unread writes, rather than to improve performance.

Arnold et al [7] have implemented Virtual-Machine level runtime monitors which detect a subset of our memory properties—their QVM can detect *idle objects*, i.e. objects on which only the constructor is called. We would report such objects as unread as long as the constructor only initializes the object.

6.2 Dynamic Binary Translators

We chose to build our monitor on top of the Pin engine [3]. Other dynamic binary translation engines would have been as effective for monitoring. DynamoRIO [8] and Valgrind [1] would also support unread memory detection.

Valgrind's Memcheck tool performs runtime verification for the following memory errors: accesses to unallocated memory, uninitialized memory, memory leaks, double frees and overlapping memory. Our tool analyzes unread writes, which are not detected by Valgrind. While not as serious as memory errors (they don't cause crashes), unread writes may lead to bugs or at least wasted resources—they should qualify as a novel "code smell" [9].

Another approach is to statically rewrite the program source by inserting monitoring calls. Purify [2] follows this approach; it transforms the program at link time and inserts instrumentation code to detect memory errors. A program rewriting approach could potentially be equally effective for detecting unread memory; however, this approach requires recompilation of the program and libraries, which our current scheme does not need.

7 Conclusion

We have presented a novel dynamic analysis, unread memory, that investigates the converse of the standard memory safety property "all reads to memory must have previously been written". Our analysis instead identifies writes to memory that never get read. We explained the design and implementation of our analysis, using dynamic binary translation, and presented experimental results from a set of benchmarks. We found that unread writes often indicate something interesting in the code; a number of the writes that we found could be eliminated or improved without affecting the program semantics.

Acknowledgments This research was supported in part by Canada's Natural Science and Engineering Research Council and an Ontario Graduate Scholarship. We'd like to thank Emina Torlak for helpful comments on a draft of this paper.

References

1. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007), San Diego, California, USA, ACM Press (June 2007) 89–100
2. Hastings, R., Joyce, B.: Purify: Fast detection of memory leaks and access errors. In: Proc. of the Winter 1992 USENIX Conference. (1991) 125–138
3. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building customized program analysis tools with dynamic instrumentation. In: PLDI 2005, Chicago, IL, USA (June 2005) 190–200
4. Ding, C., Kennedy, K.: The memory bandwidth bottleneck and its amelioration by a compiler. In: IPDPS, IEEE Computer Society (2000) 181–190
5. Bala, V., Duesterwald, E., Banerjia, S.: Dynamo: a transparent dynamic optimization system. In: PLDI '00, New York, NY, USA, ACM (2000) 1–12
6. Fahs, B., Bose, S., Crum, M., Slechta, B., Spadini, F., Tung, T., Patel, S.J., Lumetta, S.S.: Performance characterization of a hardware mechanism for dynamic optimization. In: MICRO 34, Washington, DC, IEEE Computer Society (2001) 16–27
7. Arnold, M., Vechev, M.T., Yahav, E.: Qvm: An efficient runtime for detecting defects in deployed systems. ACM Trans. Softw. Eng. Methodol. **21**(1) (2011) 2
8. Bruening, D., Garnett, T., Amarasinghe, S.P.: An infrastructure for adaptive dynamic optimization. In: CGO 2003, San Francisco, CA (March 2003) 265–275
9. Fowler, M., Beck, K.: Refactoring: improving the design of existing code. Addison-Wesley Professional (1999)