ColorSafe: Architectural Support for Debugging and Dynamically Avoiding Multi-variable Atomicity Violations

Brandon Lucia† Luis Ceze† Karin Strauss†‡

†University of Washington
http://sampa.cs.washington.edu

‡Microsoft Research
http://research.microsoft.com

{blucia0a,luisceze}@cs.washington.edu kstrauss@microsoft.com

ABSTRACT

In this paper, we propose ColorSafe, an architecture that detects and dynamically avoids single- and multi-variable atomicity violation bugs. The key idea is to group related data into colors and then monitor access interleavings in the "color space". This enables detection of atomicity violations involving any data of the same color. We leverage support for meta-data to maintain color information, and signatures to efficiently keep recent color access histories. ColorSafe dynamically avoids atomicity violations by inserting ephemeral transactions that prevent erroneous interleavings. ColorSafe has two modes of operation: (1) *debugging mode* makes detection more precise, producing fewer false positives and collecting more information; and, (2) *deployment mode* provides robust, efficient dynamic bug avoidance with less precise detection. This makes ColorSafe useful throughout the lifetime of programs, not just during development. Our results show that, in deployment mode, ColorSafe is able to successfully avoid the majority of multi-variable atomicity violations in bug kernels, as well as in large applications (Apache and MySQL). In debugging mode, ColorSafe detects bugs with few false positives.

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors); D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Design, Reliability, Languages

Keywords

Debugging, Bug Avoidance, Atomicity Violations, Concurrency Errors, Multi-variable, Data Coloring

1. INTRODUCTION

Concurrency bugs are easy to create but very difficult to fix. In addition, nondeterminism in multiprocessor systems makes bugs difficult to reproduce, complicating testing and debugging. Nevertheless, the ubiquity of multicores is adding pressure to write multithreaded code that takes advantage of the increased parallelism. This trend has spurred significant interest in concurrency bug detection tools.

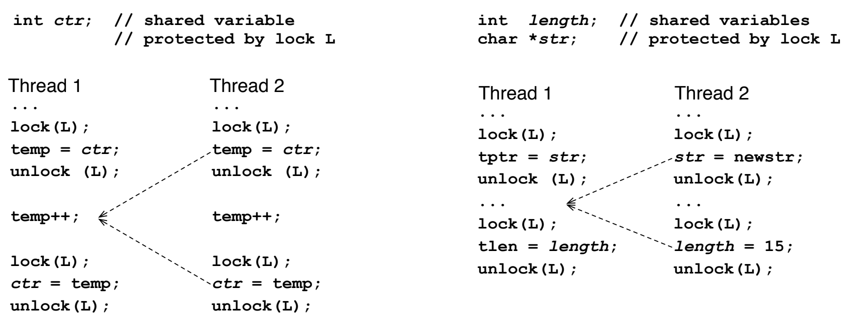
Many software-only tools were proposed to detect bugs such as data-races [22], locking discipline violations [24], and atomicity violations [11]. However, the overhead these techniques incur is very high. This motivated the proposal of architectural support for debugging [11, 14, 15, 19, 30, 33]. Unfortunately, even with sophisticated tools and thorough testing [18], hard bugs still make it to deployment. For this reason, recent research has also proposed improving the reliability of multithreaded execution with dynamic concurrency bug avoidance [14, 20, 21] by decreasing the probability that concurrency bugs will actually manifest in the field.

Most past work on architectural support for concurrency debugging has dealt with bugs involving a single variable, making them fundamentally limited in helping with a wide variety of defects. The likely reason is that the complexity (*i.e.*, state required and number of cases to be considered) of concurrency bug detection grows very quickly with the number of variables involved. Also, until not long ago, the belief was that multi-variable bugs were rare. However, recent work [12, 13] shows that multi-variable concurrency bugs are much more common than expected.

What we need, then, are general solutions for both single- and multi-variable concurrency bugs that use a single set of simple architectural mechanisms. Moreover, to justify architecture support, these mechanisms must be useful for the system's lifetime, from development to deployment. This is the gap this paper fills.

Our work focuses on atomicity violations, which are an especially challenging and pervasive category of concurrency errors. They occur when programmers overlook the need for atomicity of a set of memory accesses and fail to enclose them inside the same critical section. This allows interleavings of remote operations that lead to wrong behavior. What makes them tricky is that they can occur even in programs free of data-races. According to a recent study by Lu *et al.* [13], atomicity violations account for about 66% of non-deadlocking concurrency errors. This study also shows that a significant fraction of atomicity violation errors (33%) involve multiple variables. However, prior work such as AVIO [11] and Atom-Aid [14] address only single-variable atomicity violations or a subset of multi-variable atomicity violations (SVD [30]), and therefore miss a significant fraction of important concurrency bugs.

In this paper, we make the observation that combining variables



```
int ctr;  // shared variable
          // protected by lock L

Thread 1           Thread 2
...                ...
lock(L);           lock(L);
temp = ctr;        temp = ctr;
unlock (L);        unlock (L);

temp++;            temp++;

lock(L);           lock(L);
ctr = temp;        ctr = temp;
unlock(L);         unlock(L);
```

(a)
Single-variable atomicity violation. Concurrent increments to `ctr` might lose an update.

```
int  length;  // shared variables
char *str;    // protected by lock L

Thread 1           Thread 2
...                ...
lock(L);           lock(L);
tptr = str;        str = newstr;
unlock (L);        unlock(L);
...                ...
lock(L);           lock(L);
tlen = length;     length = 15;
unlock(L);         unlock(L);
```

(b)
Multivariable atomicity violation. Thread 1 reads inconsistent `str/length`.

Figure 1: Examples of a single-variable (a) and a multi-variable (b) atomicity violation. The example in (b) was distilled from https://bugzilla.mozilla.org/show_bug.cgi?id=73291.

into sets (or colors) and then performing serializability checks on colors rather than on individual variables allows us to detect and avoid single- and multi-variable atomicity violations. We leverage the insight of assigning colors to variables from data-centric synchronization [4, 26] and combine it with serializability analysis for bug detection [11, 30] and avoidance techniques [14]. We propose to combine these techniques in a novel way. By grouping variables into colors, our detection mechanism can treat a set of variables as a single unit. Detecting erroneous interleavings of accesses to same-colored data fundamentally enables us to detect both single- and multi-variable atomicity violations. The result is ColorSafe. ColorSafe provides dynamic bug avoidance using ephemeral transactions to prevent unintended interleavings. This is different from prior work on avoidance [14] because it doesn't require the architectural complexity of transactions-all-the-time and avoids a more general class of bugs.

ColorSafe has two modes of operation: (1) *debugging mode* collects more information and makes detection more strict to reduce false positives; and, (2) *deployment mode* provides robust, efficient dynamic bug avoidance by making detection less strict but more proactive in detecting potential bugs. This makes ColorSafe useful throughout the lifetime of programs, not just for development, and therefore more compelling to processor manufacturers. Moreover, the mechanisms used in both modes are almost exactly the same.

In the rest of this paper, Section 2 provides background information on atomicity violations and contrasts single- and multi-variable bug categories. Section 3 describes ColorSafe and provides a high-level description of how it works. In Section 4, we provide a more detailed description of ColorSafe and its architectural components. Section 5 discusses our debugging framework based on ColorSafe. In Section 6, we present an evaluation of ColorSafe in debugging and deployment mode. Section 7 discusses related work and Section 8 concludes.

2. BACKGROUND: ATOMICITY VIOLATIONS AND SERIALIZABILITY

Consider the example shown in Figure 1(a). The shared variable `ctr` is being incremented by two threads simultaneously. The read and the write of `ctr` are inside critical sections, so the program is data-race free. However, it is still incorrect: the write from Thread 2 can interleave with the read and write from Thread 1 and cause a counter increment to be lost. In this example, what is missing is *atomicity* [7], since both the read and write of `ctr` should have been atomic and isolated[1] to avoid unwanted interleaving of accesses to `ctr` from other threads. This is an example of a single-variable *atomicity violation*, since it only involves accesses to `ctr`. Additionally, such an interleaving is said to be *unserializable*, as there is no un-interleaved execution of the accesses that results in the same final state as the interleaved execution. Lu *et al.* [11] provide a list of unserializable memory operations to a single variable.

In contrast, consider the example in Figure 1(b), which is a classic example of a *multi-variable* atomicity violation. Two shared variables, `str` and `length`, are used to express related properties of a string. Just after Thread 1 reads the `str` pointer, Thread 2 updates both variables, causing Thread 1 to read a value of `length` inconsistent with the value of `str` that it read. This could later result in a crash or, even worse, silent data corruption. Note that, if accesses to each individual variable are considered separately (*e.g.*, read and write of `str`), they do *not* characterize an atomicity violation. Conversely, we make the observation that, if `str` and `length` are considered together as a unit, the atomicity violation is clear: two reads of the unit by Thread 1 are interleaved by writes to the same unit by Thread 2. Moreover, these accesses are unserializable with respect to the unit, since there are no un-interleaved executions that would produce the same result as the interleaved one. To ensure the examples in Figure 1 behave as intended, the programmer needs to enclose both memory operations inside the same critical section, instead of each in its own critical section or in none whatsoever. Vaziri *et al.* [26] provide a serializability analysis of accesses involving two variables.

In this work, we employ the idea of detecting unserializable interleavings not only to detect single-variable atomicity violations, but also to detect and avoid multi-variable atomicity violations, which are fundamentally harder and more elusive. The next section explains how.

3. COLORSAFE: DETECTING AND AVOIDING MULTI-VARIABLE ATOMICITY VIOLATIONS

We now explain how grouping variables into colors enables detection of multi-variable atomicity violations. We focus on describing the conceptual mechanisms ColorSafe uses to detect and dynamically avoid atomicity violations. Section 4 describes the actual architecture extensions and their implementation.

[1]In this paper, when we refer to atomicity, we also imply isolation.

3.1 Leveraging Data Coloring

Detecting multi-variable atomicity violations is uniquely challenging because code may contain no violations of atomicity with respect to a single variable. Recall the example in Figure 1(b). In this example, we only see the atomicity violation when we consider `str` and `length` as a single unit of data. This concept of considering groups of variables together, instead of single variables alone, is the cornerstone of ColorSafe.

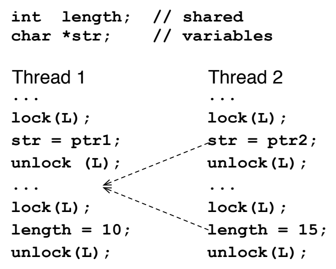
We propose to associate *colors* with shared variables, giving related variables the same color. ColorSafe then monitors interleavings of accesses to colors to determine whether they are serializable. This contrasts with past proposals such as AVIO [11] and Atom-Aid [14], which monitor interleavings of accesses to individual addresses. For example, in Figure 1(b), `str` and `length` would be given the same color since they are semantically correlated. In the color space, this example consists of two reads interleaved by at least one remote write to the same color, which is unserializable.

In Table 1, we discuss each type of multi-variable unserializable interleaving. Note that Cases 1-4 are unserializable from a single-variable point of view as well. Case 5, however, is only unserializable if multiple variables are involved. To understand Case 5, consider the example in Figure 2. The writes from Thread 2 interleaved with the writes from Thread 1, leaving the consistency between `length` and `str` compromised: `str` may point to `ptr2` but `length` will have value 10. Note that, if `str` and `length` are given the same color, the color access interleaving just described is not serializable.

Case	Interleaving	Description
1	R ←W R	The interleaving write makes the second read see inconsistent data.
2	R ←W W	The second write writes data based on stale or inconsistent data.
3	W ←W R	The read gets data from the interleaving write which is inconsistent.
4	W ←R W	The interleaving read gets inconsistent data.
5	W ←W W	The interleaving write leaves the data inconsistent (Figure 2).

Table 1: Unserializable access interleavings to colors.

Grouping correlated variables into colors and performing atomicity violation detection based on accesses to colors, as opposed to memory addresses, greatly simplifies the detection of multi-variable atomicity violations. One can think of it as reducing the high complexity of detecting multi-variable atomicity violations [12, 26] to the complexity of detecting single-color (or single-variable) atomicity violations.



```
int  length;  // shared
char *str;    // variables

Thread 1            Thread 2
...                 ...
lock(L);            lock(L);
str = ptr1;         str = ptr2;
unlock (L);         unlock(L);
...                 ...
lock(L);            lock(L);
length = 10;        length = 15;
unlock(L);          unlock(L);
```

Figure 2: Example of Case 5 in Table 1. `str` and `length` are left mutually inconsistent.

Coloring Data. Program data structures can be colored either manually or automatically. On one hand, if programmers manually color data, they encode precise information about the relationships between the data. Automatic coloring, on the other hand, does not require programmer effort, but may yield less accurate correlation information.

The main use of manual coloring is in debugging. Manual coloring requires source code annotations, by which the programmer expresses the semantic relationship between variables. It is reasonable to expect a programmer to manually color data correctly. During debugging, the programmer is likely starting from a bug report, and can identify which data are involved in a bug from the symptoms described. Furthermore, grouping related data manually has been used in past work to simplify the specification of synchronization constraints of data (*e.g.*, atomic sets [26], coloring [4], and locking discipline [17]). This past work showed that expressing which data should be grouped is not excessively difficult, since it requires only reasoning about the data at the point of its declaration. In contrast, correctly implementing synchronization (writing bug-free code) requires complex global reasoning at every access.

Automatic coloring is primarily useful for bug avoidance. It is better suited to this task than manual coloring because it doesn't require the programmer to anticipate all potential bugs, and color the involved data. Instead, data are colored according to a heuristic. There are many ways to automatically color data. For example, we could assign the same color to all fields of a `struct`, to all blocks of memory allocated by the same `malloc()` call, or even entire object instances if using object-oriented languages. Exhaustively exploring different coloring techniques is beyond the scope of the paper. We do, however, explore both manual coloring and automatic coloring to give the reader a flavor of the differences. We evaluate one automatic coloring technique by giving the same color to data allocated together (malloc-coloring). Past work has addressed ways of finding correlations between variables [12], but it did not thoroughly address atomicity violations. Such work is complementary to ColorSafe, as we can use its correlations to color data.

3.2 Detecting Unserializable Color-Access Interleavings

In this section we explain how ColorSafe detects actual unserializable interleavings in debugging mode. The next section explains its differences from deployment mode. The atomicity violation detection mechanism used by ColorSafe consists of three conceptual components: (1) a history of accesses performed by the local processor (*local history*); (2) a history of accesses performed by remote processors (*remote history*); and, (3) a set of rules that determine whether the interleaving of accesses in the history is serializable.

It is important to note that unserializable interleavings are not necessarily manifestations of atomicity violations. However, for an atomicity violation to manifest itself, it is necessary that an unserializable interleaving happens. Moreover, unserializable interleavings, especially in a short window, are strong indicators of atomicity violation bugs [11, 14]. Hence, we identify likely bugs by detecting unserializable interleavings.

Since our goal is to detect whether accesses to colors are serializable, ColorSafe's access histories are kept in terms of colors. This means the data addresses of memory accesses need to be translated to colors before being inserted in the history. Local accesses are inserted into the local history as the local processor performs them. Remote access histories are built from coherence protocol events; read and invalidate requests are inserted into the remote history as

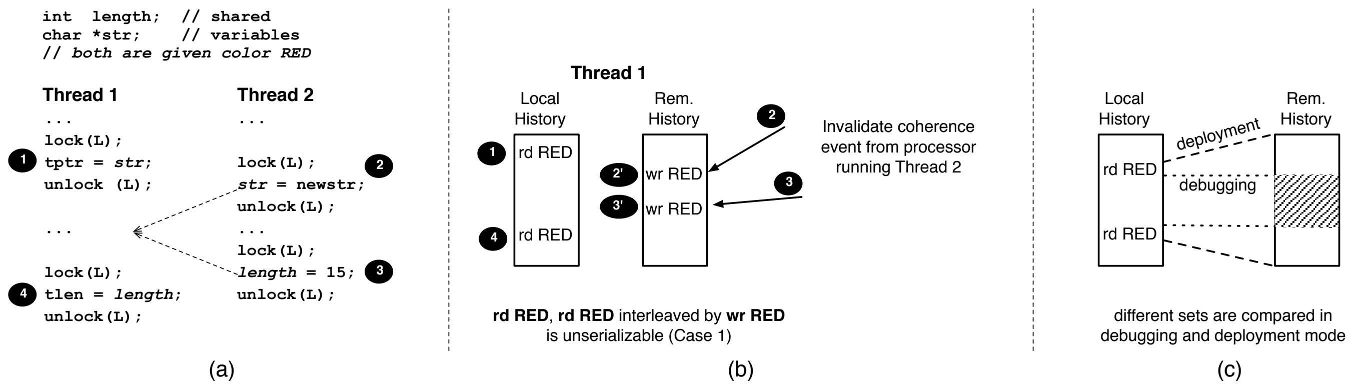


Figure 3: Overview of how ColorSafe detects multi-variable atomicity violations. The numbers in the dark circles denote the order of events happening in (a) and (b).

the processor receives those requests. Local and remote histories are kept separately and ColorSafe retains some information about the relative order of groups of local and remote accesses.

To detect atomicity violations, ColorSafe determines whether any unserializable interleaving from Table 1 exists between the local and remote histories. Figure 3 shows an example of this process using the same code from Figure 1(b), which is reproduced in Figure 3(a), and contains a multi-variable atomicity violation involving two variables. Variables `str` and `length` are both colored *RED*. The numbers in the dark circles denote the order of accesses. The accesses are inserted into their corresponding history as they happen (Figure 3(b)). Accesses 2 and 3 performed by thread 2 generate invalidate coherence messages and are inserted as writes into thread 1's remote history (2' and 3'). As soon as access 4 is performed, ColorSafe detects that the accesses in the history are unserializable, matching Case 1 in Table 1 (two reads interleaved by a write).

One might wonder how long a history we need to capture the majority of unserializable interleavings that indicate bugs. Recent work [13, 14] shows that, for many known bugs, this window is fairly small, on the order of tens of thousands of instructions at most. This is intuitive because the longer the distance between operations that should have been atomic, the higher the chance that the bug manifests itself during testing. Therefore, hard bugs tend to be the ones that occur in a short window.

3.3 Debugging vs. Deployment Mode

The previous section described what we call *debugging mode*, as it detects only interleavings that are actually unserializable. In *deployment mode*, ColorSafe attempts to dynamically avoid atomicity violations by (1) detecting when an atomicity violation is likely to happen, and (2) dynamically starting a special form of transaction — an ephemeral transaction — to prevent an unserializable interleaving from happening.

In deployment mode, we aim to detect *potentially* unserializable interleavings that are related to atomicity violations. The avoidance mechanism can then be triggered before an unserializable interleaving actually happens, to prevent it from happening. We relax the criterion used to detect unserializable interleavings in debugging mode such that it covers interleavings that could potentially happen. More precisely, we define a *potentially unserializable interleaving* as a pair of memory accesses in the local history that, if interleaved by any access in the remote history, would have been unserializable according to Table 1. For example, consider the scenario in Figure 3: if the remote writes to *RED* had happened anywhere in the remote history window, even if they did not actually interleave with the reads from *RED* in the local history, this would be detected as a potentially unserializable interleaving. Figure 3(c) illustrates the difference between debugging and deployment mode.

The intuition behind this definition of a potentially unserializable interleaving is that, because ColorSafe observed an unserializable interleaving that almost happened, it could observe the actual interleaving at a future point in the execution. This could result in an atomicity violation manifesting itself, which is what ColorSafe is trying to avoid. It is possible that potentially unserializable interleavings are, in fact, just benign accesses. In this case, the only effect is that ColorSafe initiates unnecessary bug avoidance actions[2].

Once a potentially unserializable interleaving is detected, the color of the data accessed is inserted into a set called the *HazardColorSet*. From then on, all accesses to data whose color is in the HazardColorSet trigger an ephemeral transaction of a finite size. The ephemeral transaction will make the short period of the execution beginning with these accesses appear to execute atomically and in isolation, effectively preventing any unwanted interleaving with remote accesses from happening in the meantime. The goal is that this ephemeral transaction will begin with the first instruction of an atomicity violation and be long enough to cover all local memory accesses involved in the violation, consequently preventing its manifestation.

Ephemeral transactions are transactions dynamically inferred by ColorSafe and do not correspond to any program annotation. It is important to point out that the ephemeral transactions inserted by ColorSafe cannot, in any way, break the semantics of the program, since the resulting interleaving of accesses with ephemeral transactions is still a valid interleaving with respect to the program semantics. Section 4.4 provides more details.

4. ARCHITECTURAL SUPPORT

ColorSafe needs four basic architectural mechanisms: support for data coloring (Section 4.1); histories of recent memory accesses, in terms of colors (Section 4.2); a means of detecting unserializable interleavings based on the access histories (Section 4.3); and, for bug avoidance, a way of maintaining the set of colors involved in unserializable interleavings together with support for ephemeral transactions (Section 4.4).

[2]While this is not a correctness problem and typically not a performance problem either, the system provides hooks to the programmer to disable avoidance actions in performance sensitive parts of the code.

4.1 Support for Data Coloring

ColorSafe represents the color of data items as meta-data (memory tags). There have been several proposals to support memory tagging for various purposes, such as security and information flow tracking [6, 32] and as support for new programming models [4]. To support ColorSafe, we chose a design similar to Colorama [4], which is based on the Mondrian Memory Protection scheme [29]. Mondrian provides an efficient way to associate protection information with arbitrary regions of memory by using a hierarchical multilevel permissions table. ColorSafe uses the same structure but stores ColorIDs instead of permission information. We call this table the Multilevel Color Table. Based on the number of colors required in the applications used in our experiments, we used a 12-bit ColorID field.

The Multilevel Color Table resides in memory and is accessible by all processors. Its ranges of addresses are expanded to keep the ColorID information at the desired granularity (word, line, page, etc.). The Color Lookaside Buffer (CLB) directly caches coloring information from the Color Table to provide fast lookup. To look up an address, the processor checks the CLB. In case of a miss, the processor fetches the entry from the Multilevel Color Table in memory. Software can update color information in user-mode by writing to the Multilevel Color Table. When the color table is written, the CLB needs to eventually be updated, but not immediately. ColorSafe can tolerate this transient color information incoherence, since this will not affect program semantics in any way.

Note that there are other alternatives to providing support for data coloring. For example, Loki [32] proposes a multi-granular tagging mechanism, in which tags can be associated with whole pages and, only when necessary, expanded to individual words to provide fine-grain tagging. Such a scheme would also be adequate for our purposes. Yet another alternative would be to add a ColorID field on a per cache line basis. We opted not to do this for three reasons: (1) we want to allow arbitrary coloring without forcing the user to adjust data layout; (2) the Multilevel Color Table is more space efficient; and, (3) we did not want to touch sensitive structures in the memory hierarchy.

4.2 Color Access Histories

In ColorSafe, each processor stores information about the recent history of color accesses in a *history buffer*. A history buffer holds four types of histories: (1) local read, (2) local write, (3) remote read, and (4) remote write.

ColorSafe keeps tens of thousands of instructions worth of history. Therefore it needs a resource-efficient way of keeping them, as we cannot use a searchable FIFO with tens of thousands of items. We chose to encode the color of accesses in bloom-filter-based signatures [2]. This way, each of the four history types is a *signature file* organized as a FIFO queue, in which each signature is a superset hash-encoding of ColorIDs of memory accesses for an arbitrary number of dynamic instructions. Since only colors, instead of individual addresses, are recorded into signatures, the amount of imprecision (aliasing) in the signatures is low. Figure 4(a) shows a color access signature file.

ColorSafe divides the execution of a program into *epochs*, arbitrary length sequences of consecutive dynamic instructions (*e.g.*, 400). A *history item* is a set of four signatures (one of each history type) that contains color accesses collected during an epoch. Figure 4(b) shows a complete history buffer, which is a set of history items that covers the last n epochs of execution. When an epoch ends, both local and remote accesses start being encoded in the next history item. In summary, a history item H_i consists of a Local Read signature $LocR_i$, a Local Write signature $LocW_i$,

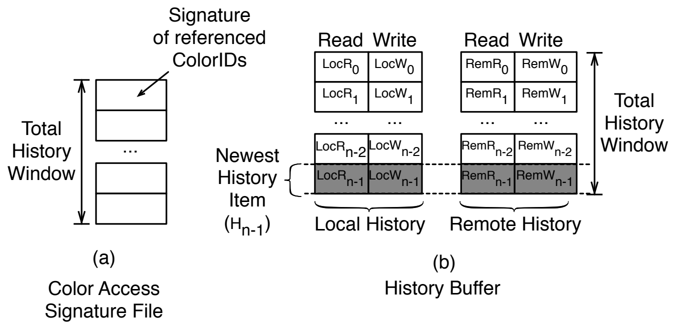

Signature of referenced ColorIDs
Total History Window
...
(a)
Color Access Signature File
Read Write
$LocR_0$ $LocW_0$
$LocR_1$ $LocW_1$
$LocR_{n-2}$ $LocW_{n-2}$
$LocR_{n-1}$ $LocW_{n-1}$
Read Write
$RemR_0$ $RemW_0$
$RemR_1$ $RemW_1$
$RemR_{n-2}$ $RemW_{n-2}$
$RemR_{n-1}$ $RemW_{n-1}$
Newest History Item (H_{n-1})
Local History
Remote History
Total History Window
(b)
History Buffer

Figure 4: Keeping color access history.

a Remote Read signature $RemR_i$, and a Remote Write signature $RemW_i$.

Note that ColorSafe sacrifices information about the relative order of operations within a history item. Information about the relative order across history items is preserved, though. The trade-off in determining the history item granularity (assuming fixed total history window) is one of precision versus cost. Smaller history items (finer-grain) improve precision by preserving more relative order information and suffer from less signature aliasing. Larger (coarser-grain) history items use less storage and comparison logic, since fewer history items are necessary and consequently fewer intersection operations need to be performed.

Collecting Local Access Information. Local access information can be easily obtained. ColorSafe uses the mechanism described in Section 4.1 to look up the ColorID for each load and store issued locally. The resulting ColorID is then encoded in the appropriate local read and write signatures for the current history item. When an epoch completes, accesses start being inserted in the next history item.

Collecting Remote Access Information. Recording remote color accesses requires minimal additional cache coherence protocol support. To collect color information for remote accesses, we augment coherence requests without affecting coherence protocol functionality in any way. On a read miss, the processors retrieve the color information of the data being accessed (actual referenced address, as opposed to block address) and append it to the coherence request sent to potential sharers. When a processor receives a read request from a remote processor, it adds the ColorID in the request to its current remote read signature. Likewise, an invalidate request generated by a write miss or a write on shared miss is augmented with a ColorID. Receiving processors add the ColorID to their current remote write signature. ColorSafe only needs color information for accesses that cause inter-processor communication, and so it is sufficient to piggyback on coherence protocol messages.

4.3 Detecting Unserializable Interleavings

ColorSafe detects unserializable interleavings by intersecting signatures in the history buffer. A signature intersection is a simple bit-wise *AND*. For example, suppose we want to detect whether Case 1 in Table 1 happened in the history buffer. Using the symbols in Figure 4(b), ColorSafe computes $LocR_i \cap LocR_j \cap RemW_k$, for all i and j, where $i \neq j$, and for values of k that depend on whether ColorSafe is in debugging mode or deployment mode (Figure 5), which we discuss shortly. If the resulting signature is not empty, then it is likely that the execution contains an unserializable interleaving involving the color(s) in the resulting set. Testing for the

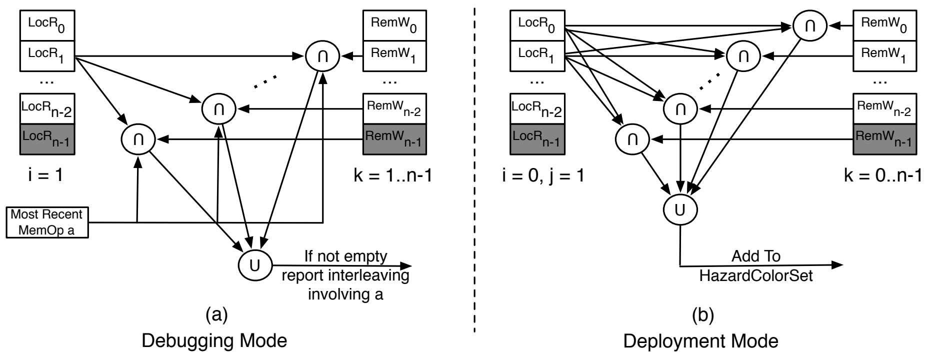


Figure 5: Detecting unserializable interleavings in (a) debugging mode and (b) deployment mode. In (a), only actual interleavings are being considered for the serializability test: the current access to a, the local history item i and the remote history items with $k \geq i$. In (b), all items in the remote history are being considered for the serializability test: local history item i, followed by local history item j, and all possible remote history items ($k = 0, ..., n-1$).

remaining cases in Table 1 is analogous, except signatures of the applicable type are intersected.

Two issues remain: (1) determining when to evaluate the detection expression and (2) choosing values of k. These choices depend on whether ColorSafe is in debugging mode or in deployment mode.

Debugging mode. In debugging mode, we want ColorSafe to detect only unserializable interleavings that actually occur. We also want to know the instruction address of the memory operation that led to the interleaving. Therefore we choose k such that $i \leq k$ and we use a set containing only the most recent memory operation issued locally as the second local set in the intersection (instead of all possible H_j). This implies that only history items that actually interleave history item H_i and the most recent memory operation are considered. Figure 5(a) illustrates this process. If the resulting set is not empty, then an unserializable interleaving involving the just-issued memory instruction is reported. This includes both the instruction address and the type of interleaving (cases in Table 1). Note that for efficiency, the set of all intersections need not be evaluated from scratch at every memory operation, because the partial intersection between local and remote history items can be reused until the corresponding history items are pushed out of the history buffer.

Deployment mode. In deployment mode, we want ColorSafe to detect *potentially* unserializable interleavings. Hence, ColorSafe performs 3-way intersections consisting of all pairs of distinct local signatures and each remote signature, regardless of whether the history items actually interleave (*i.e.,* $k = 0...n-1$), shown in Figure 5(b). When the result set is not empty, it is added to the HazardColorSet. Also, we do not need to know the instruction that led to a potential atomicity violation — we need only the colors to enable dynamic avoidance. This allows us to perform detection only at the end of an epoch instead of at each memory access, which significantly decreases the frequency of intersection operations.

4.4 Support for Dynamic Avoidance of Multi-variable Atomicity Violations

Whenever a potentially unserializable interleaving happens in deployment mode, ColorSafe adds the color involved to the HazardColorSet. Each ColorSafe processor has its own HazardColorSet. This set is also encoded as a signature. Insertions are a bitwise *OR* operation between the detection intersections' results and the HazardColorSet itself. Upon a memory access, the processor checks whether the access's ColorID is in the HazardColorSet. If so, it starts an ephemeral transaction to prevent a potentially unserializable interleaving. If there is an ephemeral transaction already in progress, the event is ignored, *i.e.*, transactions do not nest.

Implementing Ephemeral Transactions. From a mechanism perspective, ephemeral transactions are just like typical memory transactions [9]. Unlike regular transactions, though, ephemeral transactions are implicit, as they do not rely on code markers. As such, they do not guarantee that a set of dynamic instructions will always execute atomically and in isolation. Ephemeral transactions provide strong atomicity, since they must roll back in the event of conflicts with any remote accesses and attempt to execute again. To guarantee forward progress, ColorSafe provides a mechanism for recognizing repeated rollbacks. ColorSafe then reduces the size of the ephemeral transaction until it is able to commit or falls back to non-transactional execution.

4.5 Discussion on Hardware Complexity

Although ColorSafe requires additional hardware support, we argue that its cost is reasonable and leverages well understood technology. Mechanisms to keep track of sets of addresses and memory tagging have been proposed before (*e.g.,* Mondrian Memory Protection [29] and Loki [32], IBM 801 [5]). Mondrian and Loki use hierarchical data structures to map memory regions to tags, keeping storage overheads manageable. The buffers and logic required to handle history items are very simple, since they are based on address signatures [2, 23]. The additional support in the coherence protocol involves only an extra field in request messages and does not change any protocol state machine. Finally, support for transactional memory is being considered for actual off-the-shelf processors [1].

5. DEBUGGING WITH COLORSAFE

We have developed a debugging methodology for ColorSafe's debugging mode. The key idea is to reduce the rate of false-positives by focusing on detections most likely to indicate bugs.

Invariant-based reduction of false positives. We assume a mode in which developers run the program multiple times and classify

Type	Name	Description	% Bug Ex.	Intlv. Type
Kernel	nsText	Mozilla-0.9: During update of string buffer offset and length, inconsistent data can be read.	0.19%	WRW
	NetIO	Mozilla-0.9: Read of flag and conditional write can be interleaved, invalidating data.	0.14%	RWW
	jsStr	Mozilla-0.9: Between update to string buffer and length, inconsistent data can be read by remote read.	0.22%	WRW
	interp	Mozilla-0.8: Between table update and flag update, interleaving can make table inconsistent.	2.8%	WWW
	msgPane	Mozilla-0.8: Interleaving read of flag indicates content loaded in msg. pane before content is loaded.	0.22%	WRW
Full	Ap2.0	Apache-2.0.48: Character buffer and string length made inconsistent by concurrent accesses.	0.91%	WRW
	AGet	AGet-0.4: During update of log contents/length, inconsistent data can be read by signal handler.	0.47%	WRW
	MySQL	MySQL-3.23.57: Accesses can be logged out of order by highly concurrent access to replay log.	33.21%	WWW

Table 2: Bugs used to evaluate ColorSafe.

each execution as buggy or non-buggy by observing its outcome (crashes, data corruptions, etc). For each execution, we collect ColorSafe's output and produce a set of *detection identifiers*, which are composed of the instruction address where an unserializable interleaving was detected, and the type of interleaving (Table 1). The detection identifiers from buggy runs are added to the *buggyDetSet*, and those from non-buggy execution are added to *nonBuggyDetSet*. We then set-subtract *nonBuggyDetSet* from *buggyDetSet*, producing the *vioSet*. This set contains the detection identifiers for interleavings that occurred only during the buggy runs. These are the points in the code on which a developer should focus to locate the bug. In Section 6.4, we show that this simple technique actually prunes most false positives.

6. EVALUATION

Our goals in evaluating ColorSafe are to assess how well deployment mode dynamically avoids atomicity violations (Section 6.2) and at what performance cost (Section 6.2.1), to understand design trade-offs (Section 6.2.2 and Section 6.2.3), to understand metadata usage (Section 6.3), and to assess how accurately debugging mode locates bugs in the code (Section 6.4).

6.1 Experimental Setup

We developed a ColorSafe simulator using the PIN binary instrumentation framework [16]. The simulator models all ColorSafe structures, including the Multilevel Color Table, translation of data addresses to colors, the history buffer, unserializable interleaving detection using signature operations on history items, the HazardColorSet and ET support. The simulator models both debugging and deployment mode. In debugging mode, it produces the unserializable interleaving detection output that is used by our invariant-based debugging framework. In deployment mode, the simulator determines how often atomicity violations were avoided by determining whether the violation executed entirely within an ET. To assess performance impact, we model ET conflicts.

We use a variety of benchmarks consisting of "bug kernels" and full applications. Table 2 provides a description of each kernel and application, along with the portion of the dynamic execution spent in buggy code (Column 4) and the interleaving pattern that causes the bug (Column 5). The bug kernels are segments of buggy code extracted from full applications. We extracted five kernels from various versions of the Mozilla Project, all previously discussed in the literature [12, 13]. We paid special attention to maintaining the original data structure hierarchies and the layout of the code surrounding the bug. As our full applications workloads, we use the AGet parallel download accelerator, the Apache httpd webserver and the MySQL database server. To exercise the buggy regions of Apache, we used scripts to repeatedly launch 100 concurrent requests. We exercised the MySQL bug using a version of the sql-bench benchmark modified to execute many concurrent requests. The bug in AGet involves a signal handler, so to exercise the buggy code, we fetched a file from a network resource, and interrupted the transmission with a Unix signal.

We experiment with both manual coloring and malloc-coloring, as described in Section 3.1. To perform manual coloring, we added explicit annotations to the code to associate colors with data. For malloc-coloring, our simulator monitors calls to memory allocation functions and assigns a new color to the allocated region.

6.2 Deployment Mode: Bug Avoidance

We start by showing that ColorSafe is able to avoid most instances of atomicity violations in bug kernels and applications. All experiments had epochs of 400 instructions, a total history window of 12,000 instructions (*i.e.,* 30 history items), and 3,000-instruction ETs. Table 3 shows the number of violation instances avoided.

App.	% Avoided	
	Manual	Malloc
nsText	99.95	99.95
NetIO	99.95	99.95
jsStr	100	100
interp	99.95	0
MsgPane	99.95	0
Ap2.0	98.72	94.18
AGet	99.28	0
MySQL †	77.0	71.4

Table 3: Violations avoided in bug kernels and full applications using manual and malloc data coloring. †We used a different system configuration for MySQL. We explain the details below (Difficulties with MySQL).

For bug kernels, ColorSafe avoids nearly 100% of the violation instances using manual coloring. Malloc-coloring is capable of avoiding almost all atomicity violations in most kernels, but it is not effective for all kernels. The bugs in *interp* and *msgPane* each involve accesses to one global variable, and one dynamically allocated variable. They are not allocated together, so using malloc-coloring does not capture their correlation. As a result ColorSafe is unable to avoid these bugs.

Table 3 shows that ColorSafe avoids nearly all violation instances in our full application benchmarks. In runs of Ap2.0, ColorSafe avoids virtually all instances of the violation, using both manual and malloc-coloring. Malloc-coloring has a slightly lower rate of avoidance. This is because ETs triggered by accesses to data unrelated to the violation end up preventing useful ETs from proceeding. In runs of AGet, ColorSafe avoids more than 99% of instances of the violation using manual coloring. The bug in AGet involves a dynamically allocated variable and a global variable, so unfortunately malloc-coloring is unable to identify their correlation.

Difficulties with MySQL. ColorSafe was unable to avoid violation instances in MySQL using our standard configuration. This is because the violation is nearly 20,000 instructions long, and cannot execute entirely within an ET of 3,000 instructions. We re-ran MySQL using 64,000-instruction ETs, 1,000-instruction epochs, and a total history window of 30,000 instructions. With this configuration, ColorSafe avoids 77% of the violations using manual coloring, and 71.4% of violations using malloc-coloring. ColorSafe's avoidance is lower using this configuration because longer ETs triggered in response to false positive detections prevent useful ETs from beginning over a much longer window.

Could Avoidance Happen by Chance? A Comparison with Random Ephemeral Transactions. One may wonder whether the bug avoidance achieved by ColorSafe would be possible simply by starting ETs at random points. Here we show empirically that this is not the case. Consider an experiment using AGet. Using 3,000-instruction ETs and at random starting about 5 ETs per 100,000 dynamic instructions (the rate of transaction starts for AGet using standard ColorSafe) avoids 1.8% of all violations. ColorSafe is able to avoid 99.28% of all violations using the same configuration. Performing the same experiment with Ap2.0, we see that random ETs avoid only 6.97% of violations. ColorSafe avoids 98.72% of violations. Results were similar for other benchmarks. This stark contrast shows that ColorSafe's avoidance performs significantly better than chance.

6.2.1 Overheads

ColorSafe in deployment mode imposes modest impact on performance. We now discuss and quantify the key sources of overheads, which are coloring support and ETs. ColorSafe leverages existing cache coherence support to handle the exchange of color information between processors. As a result, communicating color information imposes negligible runtime overhead. The interconnect traffic overhead associated with communicating colorIDs is not likely to be problematic because the meta-data is small, and is only communicated between processors on a subset of cache misses. Color information lookup depends on the meta-data scheme underlying ColorSafe. While a lookup is not free, the cost is minimized using caches for meta-data information. Moreover, color information is mostly read-only (*i.e.,* written only at allocation time). This means that any additional overhead associated with meta-data writes is unlikely to affect performance. Finally, regarding energy, the structures used to store color information are similar to TLBs, and like TLBs, amount to a small fraction of total power consumption.

The main sources of performance degradation are ETs. Namely, they are bookkeeping overhead and re-execution due to conflicts. In Table 4, we report the percentage of dynamic instructions that triggered an ET (% ET Start), the number of ETs that were useful in preventing an atomicity violation (% Useful ETs), and the percentage of useless ETs that experienced a conflict (% Useless Conflicts). We report the number of ET starts as a fraction of the total number of dynamic instructions to quantify how often the overhead of starting an ET is incurred. The fraction of useful ETs is a measure of how often the cost of an ET was worthwhile, because it prevented a violation. The fraction of conflicting useless ETs is an approximation of the amount of work wasted in ETs that served no purpose and still had to be re-executed. In Table 4 we also show the fraction of total dynamic instructions that executed inside ETs (% in ETs), that executed in useful ETs (% in Useful ETs), and that executed in useless ETs that had conflicts (% in Useless ETs w/ Conflicts). We report data only for full applications, as kernels execute in tight loops around buggy code, making them unsuitable for this analysis.

There are two important results in these data. First, for all applications, the rate at which ETs are triggered is very low: 3 ETs per 100,000 instructions for MySQL, 5 per 100,000 for AGet, and 20 per 100,000 for Ap2.0. The low frequency of ET starts indicates that the cost of starting, ending and verifying ETs will have little effect on performance. We also see a relatively small fraction (12–38%) of the execution is executed transactionally.

App.	% ET Start	% in ETs	% Useful ETs	% in Useful ETs	% Useless Conflicts	% in Useless ETs w/ Conflicts
Ap2.0	0.02	38.4	7.4	5.8	4.1	3.2
AGet	0.005	12.8	63.8	10.7	6.4	1.1
MySQL	0.003	24.7	9.0	20.2	0.5	1.2

Table 4: The rate of ET starts, % of useful ETs, and % of conflicting useless ETs for full applications in deployment mode. Ap2.0 and MySQL were run using malloc coloring, and AGet, manual coloring. MySQL was run with the same modified configuration as above (Difficulties with MySQL).

Second, very little computation is wasted by re-executing useless ETs. The data show that the fraction of useful ETs ranges from 7.4% (Ap2.0) to 63.8% (AGet). At first glance this may suggest that useless ETs are frequent, and hence problematic. However, the rate of aborts for useless ETs is very low — just 0.5% for MySQL, and at most 6.4% in AGet. The work wasted in these useless, aborted ETs amounts to just a small fraction of dynamic instructions, from 1.1% to 3.2%. Thus, if an ET is useless, it rarely experiences a conflict, so very little work is wasted. If an ET is useful, it is more likely to abort, but we consider it profitable to sacrifice this small amount of performance in exchange for prevention of buggy behavior. Additionally, only a small fraction of the execution (5.8%–20.2%) executes in useful ETs, and incurs the higher likelihood of abort.

6.2.2 Sensitivity to Ephemeral Transaction Length

Table 5 shows avoidance for each application as the size of ETs is varied between 3,000 and 15,000. For all the bugs we considered — except the very long MySQL bug — ColorSafe's avoidance is stable, for the sizes shown, and all sizes within this range. This insensitivity to ET size shows two things: (1) large ETs do not inhibit the avoidance capability of ColorSafe; and (2) there is flexibility in the selection of this design parameter. We chose a default ET size of 3,000 instructions; any smaller, and we risk being unable to avoid modestly large violations; any larger and we increase the chances of unnecessary abort.

App.	% Violations Avoided			
	3,000 Inst ET	5,000 Inst ET	10,000 Inst ET	15,000 Inst ET
nsText	99.95	99.95	99.95	99.95
NetIO	99.95	99.95	99.95	99.95
jsStr	100.0	100.0	100.0	100.0
interp^m	99.95	99.95	99.95	99.95
msgPane^m	99.95	99.90	99.90	99.95
AGet^m	99.28	97.93	99.10	99.18
Ap2.0	94.18	90.55	98.64	94.16

Table 5: Stable bug avoidance for a variety of ET sizes. Applications marked with a ^{m} were run using manual coloring, because their bugs involve global and heap variables; All others were run with malloc-coloring.

6.2.3 Sensitivity to History Buffer Configuration

The history buffer configuration determines which interleavings are observable by ColorSafe and affects which unserializable interleavings can be detected. We now evaluate the effect of varying the granularity of the history items in the history buffer. We do this by injecting "noise" into a bug kernel to simulate high-frequency concurrent access to shared data. We add noise by allocating an array of random integers unrelated to the bug in the kernel. Randomly, 1% of the elements in the array are given the same color as the data in the bug. We add five extra threads to the program that spin in a loop, repeatedly accessing noise data. Each iteration, they make a random number of accesses between 1 and 10 and determine whether each is a read or a write by "flipping a coin." The noise level parameter is the inverse of the size of the array of noise data: The higher the noise level, the smaller the array. At higher noise levels, there is a higher probability that a random access into the array will access an element colored the same as the data involved in the bug.

Figure 6 shows avoidance in the presence of noise, with a fixed total instruction history (12,000 instr.) and using both coarse-grain (1,200 instr.) and fine-grain (400 instr.) history items. These data show that as the noise level decreases, avoidance improves. Note though, that avoidance is still effective even at the highest noise level. We see the improvement because as noise decreases, the number of ETs triggered by accesses unrelated to the bug decreases, permitting useful ETs to proceed (recall ETs don't nest).

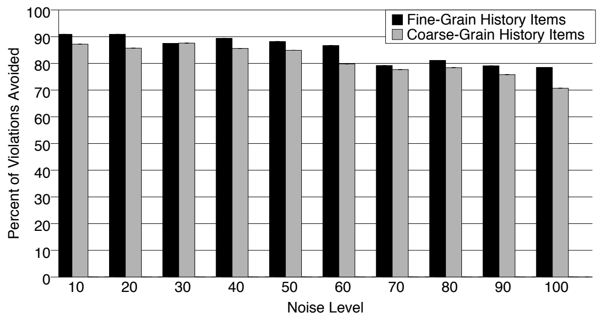


Figure 6: Atomicity violations avoided in kernel NetIO under synthetic noise for fine- and coarse-grain history items with a constant history window.

This figure also shows that using fine- rather than coarse-grain history items improves avoidance. The reason is that fine-grain history items encode the relative order of memory accesses more precisely. Using coarse history items, more accesses occur within the same history item and are considered simultaneous. This prevents ColorSafe from seeing an interleaving of these accesses. When fine-grain history items are used, ColorSafe is able to observe these interleavings, enabling earlier detection of buggy interleavings.

Figure 7 shows the proportion of useful ETs in these experiments. There is an inverse relationship between the noise level and the fraction of useful ETs. This relationship corroborates our above conclusion that avoidance is slightly impeded by the presence of noise because more useless ETs occur. The data also show that using fine-grain history items results in a larger fraction of useful ETs. This is in agreement with our findings from Figure 6, that fine-grain history items lead to more precise detection.

6.3 Characterizing Meta-Data Requirements

The size of the color meta-data in ColorSafe is derived from the total number of necessary colors. In our experiments using malloc-coloring, we saw as few as 265 colors in Ap2.0, around 900 colors in MySQL and as many as 4,015 colors (in jsStr). Using manual coloring, we saw just a single color in most cases (kernels, AGet, MySQL), and at most 5 colors in Ap2.0. These numbers represent

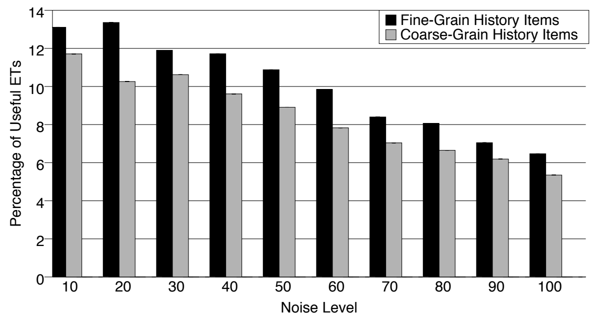


Figure 7: Percentage of useful ETs in NetIO with synthetic noise, 12,000-instruction total history length, and varied history item granularities.

the total number of coloring events (*e.g.*, allocations) in the execution.

We chose to use 12 bits of meta-data to represent colors in ColorSafe. For manual coloring, 12 bits provides ample space, according to our experimental results. Using malloc-coloring, capacity will be an issue for applications that frequently allocate memory. One way of handling this is to recycle colors when their associated memory is deallocated. Also, at additional cost, ColorSafe could be implemented with wider meta-data fields.

6.4 Debugging Mode: Locating Bugs in the Code

We focus on full applications for our debugging experiments. We collect the report of unserializable interleavings generated by ColorSafe using both manual coloring and malloc-coloring (when possible). We present a comparison of the detection capability in deployment mode versus debugging mode to show the effect of a stricter detection policy. We quantify the report in terms of code points — *i.e.*, lines of code to inspect.

Using malloc-coloring in debugging mode, ColorSafe reported a large number of detection code points — 1,493 for Ap2.0. By applying invariant-based processing, we saw a marked reduction in detections to just 58. This much smaller set of detections would lead a programmer directly to a bug, or short of that, would help a programmer decide how to manually color data.

We foresee ColorSafe being more useful for debugging if data structures that are suspected to be related to a bug are manually colored by a developer. We consider it reasonable to assume a programmer would be able to do this, given a standard bug report, output from ColorSafe using malloc-coloring, and knowledge about the data structures in the program. It is realistic to assume that the programmer can manually color data because it requires only local reasoning, when the data is declared (or allocated). Reasoning of this sort is the basis of prior proposals focusing on data-centric synchronization models [4, 10, 26].

Table 6 shows the number of code points produced by ColorSafe using manual coloring. We show results for deployment mode (Column 2), debugging mode (Column 3) and using our invariant approach described in Section 5 (Column 4). Most notably, ColorSafe is able to detect the bug in Ap2.0, with only 2 false positives (3 code points) in deployment mode, and just 1 false positive (2 code points) in debugging mode. ColorSafe detects the bug in AGet with just a handful of false positives as well, in both deployment mode, and debugging mode.

Two other facts stand out amongst these data. First, there is a decrease in detections from deployment mode to debugging mode. This is because debugging mode has a stricter policy for determining that an interleaving is unserializable (Section 3.2). There is a reduction of approximately 21% (256 code points) in the number of code points reported for MySQL, and 17% (4 code points) for AGet. However, further improvement is still desirable — even necessary — with hundreds of code points left to sift through for MySQL. This brings us to our second result: The reduction in reports resulting from applying invariant-based processing is dramatic. Invariant-based pruning eliminates hundreds of false positives for MySQL, leaving 40 code points to be analyzed in a software package of over a million lines. For AGet, we reduce the number of code points reported to just 8.

This shows that ColorSafe debugging, coupled with manual coloring and our invariant-based approach, leads to very few false positives, or virtually none. The reason that the invariant based approach works so well is that there are many similarities between non-buggy runs and buggy runs in terms of the unserializable interleavings. These similarities are filtered by our invariant-based approach, leaving only relevant detections.

App.	# Detections		
	Deployment	Debugging	Post-Processed
Ap2.0	3	2	2
AGet	24	20	8
MySQL	821	677	40

Table 6: Number of code points reported by ColorSafe using deployment mode, debugging mode, and debugging mode with invariant post-processing.

7. RELATED WORK

There are several major differences between ColorSafe and prior work on architectural support for concurrency bug detection. First, ColorSafe provides a general solution to support for atomicity violation detection (single-variable and multi-variable). Second, ColorSafe not only detects but also dynamically avoids multi-variable atomicity violations in the field. Finally, the same set of mechanisms are used for debugging and deployment, making ColorSafe useful throughout the lifetime of a system.

Atom-Aid [14] leverages the observation that architectures that execute programs as a continuum of implicit transactions [3, 9, 25, 28] can decrease the amount of interleaving of memory accesses between threads and potentially avoid atomicity violations. Atom-Aid detects single-variable atomicity violations and manipulates the boundaries of implicit transactions to avoid them. ColorSafe addresses the more general problem of multi-variable atomicity violations (not detected by Atom-Aid) and does *not* rely on continuous implicit transactions, but rather on ephemeral transactions, which are simpler and less costly. In addition, ColorSafe has drastically fewer false positives.

AVIO [11] was one of the first proposals of architectural support for atomicity violation detection. AVIO is only useful for debugging. AVIO monitors interleaving by extending the caches, so it is inherently limited to single-variable atomicity violations. In contrast, ColorSafe uses a history buffer of color accesses decoupled from caches. AVIO uses training runs to extract interleaving invariants and then checks if these invariants hold in future runs. This was the inspiration for our invariant-based debugging framework. MUVI [12] proposes static analysis to detect data correlations, but does not address atomicity violations.

ToleRace [21] and ISOLATOR [20] avoid concurrency errors that happen when one thread correctly follows a locking discipline but other threads do not. ToleRace detects such conditions by determining if the atomicity of a critical section was violated via single-variable serializability analysis, similarly to AVIO and Atom-Aid. ISOLATOR detects such scenarios by having the programmer annotate the code to declare the locking discipline. The avoidance mechanism used by these systems is based on keeping shadow copies of data that are speculatively updated and committed at the end of critical sections.

Serializability Violation Detector (SVD) [30] heuristically infers atomic sections based on control and data dependencies. SVD then detects single-variable atomicity violations and some multi-variable atomicity violations in these inferred sections. The authors briefly mention that their technique could use hardware support and avoid bugs via global checkpoint and restart. ColorSafe detects and avoids a broader class of bugs because of its more inclusive serializability analysis and its data correlation information. ColorSafe also differs from SVD in that it does *proactive* avoidance, preventing potential bugs, and without costly global checkpointing.

The Interleaving Constrained Shared Memory MultiProcessor [31] is a bug avoidance technique based on building invariants during testing and then using architecture support to enforce these invariants at run time. The invariants are encoded in sets of happens-before relationships between static memory instructions (PSets). PSets cannot be used to avoid multi-variable bugs because PSets do not encode correlation between memory locations (which is the cornerstone of ColorSafe). PSets requires a training phase to enable avoidance, whereas ColorSafe does not. Moreover, it requires checking whether memory accesses are allowed to proceed, which the authors claim that can be done with online binary rewriting and hardware support. Finally, PSets focuses on avoidance, so it is unclear whether the false positives would be tolerable for debugging.

Object Race Detection [27] is a software-only technique for dynamic race detection that tracks accesses at the level of objects in Java. While Object Race Detection does not detect violations of atomicity, it is related to ColorSafe because it tracks whole objects as units, similarly to ColorSafe's use of colors. Data coloring in ColorSafe, however, is more general, as it does not need to follow object boundaries and is not tied to a specific category of languages.

Colorama [4] is an architecture that supports a programming model based on data coloring, in which the programmer groups related data-structures into colors and the system automatically infers critical sections dynamically. Vaziri [26] *et al.* proposed a software implementation of such a model for object oriented languages, in which programmers group data into atomic sets and a compiler infers critical sections. These two pieces of work are related to ColorSafe in that they use data grouping to convey information about data-consistency, but none of them uses this information for debugging or bug avoidance. More recently and concurrently with our work, Hammer *et al.* [8] used atomic sets for serializability violation detection in Java programs. The required annotations go beyond grouping objects in atomic sets: it also requires programmers to annotate the expected atomicity of some methods ("units of work"). ColorSafe detects this automatically with high accuracy.

8. CONCLUSIONS

Shared memory parallel programming is a challenge for software developers. We believe it is extremely important to provide support for debugging such programs. It is still very likely that bugs will elude developers, making bug avoidance in the field appealing. We advocate that mechanisms to support bug detection should also enable dynamic bug avoidance post-deployment, making them useful throughout the lifetime of systems.

We propose ColorSafe, an architecture that provides both precise detection and dynamic bug avoidance. It is general, covering both single- and multi-variable atomicity violations and could be extended to support other forms of bugs. The key idea is to group correlated variables into colors and then to monitor access interleavings in the color space. This enables detection of bugs involving a set of variables, not just a single variable.

ColorSafe has two modes of operation: debugging mode produces detailed information about how and where atomicity violations may have happened; deployment mode performs less strict detection, but automatically starts ephemeral transactions to avoid erroneous interleavings, without affecting the semantics of the program. Our results show that ColorSafe in deployment mode is able to avoid virtually all violations in bug kernels and the vast majority of violations in full applications, such as Apache and MySQL, and imposes little performance overhead. In addition, ColorSafe in debugging mode yields very few false positives using a simple post-processing technique to prune spurious reports.

Acknowledgements

We thank the anonymous reviewers for their helpful feedback. We thank the SAMPA group at the University of Washington for their invaluable feedback on the manuscript and insightful discussions. This work was supported in part by NSF under grants CNS-0720593, a Microsoft Research Faculty Fellowship, and gifts from Microsoft Research and Intel.

9. REFERENCES

[1] Advanced Synchronization Facility: Proposed Architectural Specification. `http://developer.amd.com/assets/45432-ASF_Spec_2.1.pdf`, March 2009.

[2] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *International Symposium on Computer Architecture*, 2006.

[3] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk Enforcement of Sequential Consistency. In *International Symposium on Computer Architecture*, 2007.

[4] L. Ceze, P. Montesinos, C. von Praun, and J. Torrellas. Colorama: Architectural Support for Data-Centric Synchronization. In *International Symposium on High-Performance Computer Architecture*, 2007.

[5] A. Chang and M. Mergen. 801 Storage: Architecture and Programming. *ACM Transactions Computer Systems*, February 1988.

[6] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A Flexible Information Flow Architecture for Software Security. In *International Symposium on Computer Architecture*, 2007.

[7] C. Flanagan and S. Qadeer. A Type and Effect System for Atomicity. In *Conference on Programming Language Design and Implementation*, 2003.

[8] C. Hammer, J. Dolby, M. Vaziri, and F. Tip. Dynamic Detection of Atomic-Set-Serializability Violations. In *International Conference on Software Engineering*, 2008.

[9] L. Hammond, V. Wong, M. Chen, B. Carlstrom, J. Davis, B. Hertzberg, M. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. In *International Symposium on Computer Architecture*, 2004.

[10] C. Hoare. Monitors - An Operating System Structuring Concept. *Communications of the ACM*, 1974.

[11] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[12] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. Popa, and Y. Zhou. MUVI: Automatically Inferring Multi-variable Access Correlations and Detecting Related Semantic and Concurrency Bugs. In *Symposium on Operating System Principles*, 2007.

[13] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from Mistakes-A Comprehensive Study on Real World Concurrency Bug Characteristics. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[14] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-Aid: Detecting and Surviving Atomicity Violations. In *International Symposium on Computer Architecture*, 2008.

[15] B. Lucia and L. Ceze. Finding Concurrency Bugs With Context-Aware Communication Graphs. In *International Symposium on Microarchitecture*, 2009.

[16] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. PIN: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Conference on Programming Language Design and Implementation*, 2005.

[17] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: Synchronization Inference for Atomic Sections. In *Symposium on Principles of Programming Languages*, 2006.

[18] M. Musuvathi and S. Qadeer. CHESS: Systematic Stress Testing of Concurrent Software. In *International Symposium on Logic-based Program Synthesis and Transformation*, 2006.

[19] M. Prvulovic and J. Torrellas. ReEnact: Using Thread-Level Speculation Mechanisms to Debug Data Races in Multithreaded Codes. In *International Symposium on Computer Architecture*, 2003.

[20] S. Rajamani, G. Ramalingam, V. Ranganath, and K. Vaswani. ISOLATOR: Dynamically Ensuring Isolation in Concurrent Programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[21] P. Ratanaworabhan, M. Burtscher, D. Kirovski, B. Zorn, R. Nagpal, and K. Pattabiraman. Detecting and Tolerating Asymmetric Races. In *Annual Symposium on Principles and Practice of Parallel Programming*, 2009.

[22] M. Ronsee and K. De Bosschere. RecPlay: A Fully Integrated Practical Record/Replay System. *Transactions on Computer Systems*, 1999.

[23] D. Sanchez, L. Yen, M. Hill, and K. Sankaralingam. Implementing Signatures for Transactional Memory. In *International Symposium on Microarchitecture*, 2007.

[24] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. *Transactions on Computer Systems*, 1997.

[25] E. Vallejo, M. Galluzzi, A. Cristal, F. Vallejo, R. Beivide, P. Stenstrom, J. Smith, and M. Valero. Implementing Kilo-Instruction Multiprocessors. In *International Conference on Pervasive Services*, 2005.

[26] M. Vaziri, F. Tip, and J. Dolby. Associating Synchronization Constraints with Data in an Object-Oriented Language. In *Symposium on Principles of Programming Languages*, 2006.

[27] C. von Praun and T. Gross. Object Race Detection. In

Conference on Object-Oriented Programming Systems, Languages and Applications, 2001.

[28] T. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Mechanisms for Store-wait-free Multiprocessors. In *International Symposium on Computer Architecture*, 2007.

[29] E. Witchel, J. Cates, and K. Asanovic. Mondrian Memory Protection. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[30] M. Xu, R. Bodik, and M. Hill. A Serializability Violation Detector for Shared-Memory Server Programs. In *Conference on Programming Language Design and Implementation*, 2005.

[31] J. Yu and S. Narayanasamy. A Case for an Interleaving Constrained Shared-Memory Multi-Processor. In *International Symposium on Computer Architecture*, 2009.

[32] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis. Hardware Enforcement of Application Security Policies Using Tagged Memory. In *Symposium on Operating Systems Design and Implementation*, 2008.

[33] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: Hardware-Assisted Lockset-based Race Detection. In *International Symposium on High-Performance Computer Architecture*, 2007.