Bypassing Races in Live Applications with Execution Filters

Jingyue Wu, Heming Cui, Junfeng Yang
{*jingyue, heming, junfeng*}*@cs.columbia.edu*
Computer Science Department
Columbia University
New York, NY 10027

Abstract

Deployed multithreaded applications contain many races because these applications are difficult to write, test, and debug. Worse, the number of races in deployed applications may drastically increase due to the rise of multicore hardware and the immaturity of current race detectors.

LOOM is a "live-workaround" system designed to quickly and safely bypass application races at runtime. LOOM provides a flexible and safe language for developers to write *execution filters* that explicitly synchronize code. It then uses an *evacuation* algorithm to safely install the filters to live applications to avoid races. It reduces its performance overhead using *hybrid instrumentation* that combines static and dynamic instrumentation.

We evaluated LOOM on nine real races from a diverse set of six applications, including MySQL and Apache. Our results show that (1) LOOM can safely fix all evaluated races in a timely manner, thereby increasing application availability; (2) LOOM incurs little performance overhead; (3) LOOM scales well with the number of application threads; and (4) LOOM is easy to use.

1 Introduction

Deployed multithreaded applications contain many races because these applications are difficult to write, test, and debug. These races include data races, atomicity violations, and order violations [33]. They can cause application crashes and data corruptions. Worse, the number of "deployed races" may drastically increase due to the rise of multicore and the immaturity of race detectors.

Many previous systems can aid race detection (*e.g.,* [31, 32, 37, 47, 53]), replay [9, 18, 28, 36, 43], and diagnosis [42, 49]. However, they do not directly address deployed races. A conventional solution to fixing deployed races is software update, but this method requires application restarts, and is at odds with high availability demand. Live update systems [10, 12, 15, 35, 38, 39, 51] can avoid restarts by adapting conventional patches into *hot patches* and applying them to live systems, but the reliance on conventional patches has two problems.

First, due to the complexity of multithreaded applications, race-fix patches can be *unsafe* and introduce new errors [33]. Safety is crucial to encourage user adoption, yet automatically ensuring safety is difficult because conventional patches are created from general, difficult-to-analyze languages. Thus, previous work [38, 39] had to resort to extensive programmer annotations.

Second, creating a releasable patch from a correct diagnosis can still take time. This delay leaves buggy applications unprotected, compromising reliability and potentially security. This delay can be quite large: we analyzed the Bugzilla records of nine real races and found that this delay can be days, months, or even years. Table 1 shows the detailed results.

Many factors contribute to this delay. At a minimum level, a conventional patch has to go through code review, testing, and other mandatory software development steps before being released, and these steps are all time-consuming. Moreover, though a race may be fixed in many ways (*e.g.,* lock-free flags, fine-grained locks, and coarse-grained locks), developers are often forced to strive for an efficient option. For instance, two of the bugs we analyzed caused long discussions of more than 30 messages, yet both can be fixed by adding a single critical section. Performance pressure is perhaps why many races were *not* fixed by adding locks [33].

This paper presents LOOM, a "live-workaround" system designed to quickly protect applications against races until correct conventional patches are available and the applications can be restarted. It reflects our belief that the true power of live update is its ability to provide immediate workarounds. To use LOOM, developers first compile their application with LOOM. At runtime, to workaround a race, an application developer writes an *execution filter* that synchronizes the application source to filter out racy thread interleavings. This filter is kept separate from the source. Application users can then download the filter and, for immediate protection, install

Race ID	Report	Diagnosis	Fix	Release
Apache-25520	12/15/03	12/18/03	01/17/04	03/19/04
Apache-21287	07/02/03	N/A	12/18/03	03/19/04
Apache-46215	11/14/08	N/A	N/A	N/A
MySQL-169	03/19/03	N/A	03/24/03	06/20/03
MySQL-644	06/12/03	N/A	N/A	05/30/04
MySQL-791	07/04/03	07/04/03	07/14/03	07/22/03
Mozilla-73761	03/28/01	03/28/01	04/09/01	05/07/01
Mozilla-201134	04/07/03	04/07/03	04/16/03	01/08/04
Mozilla-133773	03/27/02	03/27/02	12/01/09	01/21/10

Table 1: *Long delays in race fixing.* We studied the delays in the fix process of nine real races; some of the races were extensively studied [9, 31, 33, 42, 43]. We identify each race by "$Application - \langle Bugzilla\ \# \rangle$." Column **Report** indicates when the race was reported, **Diagnosis** when a developer confirmed the root cause of the race, **Fix** when the final fix was posted, and **Release** when the version of application containing the fix was publicly released. We collected all dates by examining the Bugzilla record of each race. An N/A means that we could not derive the date. The days between diagnosis and fix range from a few days to a month to a few years. For all but two races, the bug reports from the application users contained correct and precise diagnoses. Mozilla-201134 and Mozilla-133773 caused long discussions of more than 30 messages, though both can be fixed by adding a critical region.

it to their application without a restart.

LOOM decouples execution filters from application source to achieve safety and flexibility. Execution filters are safe because LOOM's execution filter language allows only well formed synchronization constraints. For instance, "code region r_1 and r_2 are mutually exclusive." This declarative language is simpler to analyze than a general programing language such as C because LOOM need not reverse-engineer developer intents (*e.g.,* what goes into a critical region) from scattered operations (*e.g.,* `lock()` and `unlock()`).

As temporary workarounds, execution filters are more flexible than conventional patches. One main benefit is that developers can make better performance and reliability tradeoffs during race fixing. For instance, to make two code regions r_1 and r_2 mutually exclusive when they access the same memory object, developers can use critical regions larger than necessary; they can make r_1 and r_2 always mutually exclusive even when accessing different objects; or in extreme cases, they can run r_1 and r_2 in single-threaded mode. This flexibility enables quick workarounds; it can benefit even the applications that do not need live update.

We believe the execution filter idea and the LOOM system as described are worthwhile contributions. To the best of our knowledge, LOOM is the first live-workaround system designed for races. Our additional technical contributions include the techniques we created to address the following two challenges.

A key safety challenge LOOM faces is that even if an execution filter is safe by construction, installing it to a live application can still introduce errors because the application state may be inconsistent with the filter. For instance, if a thread is running inside a code region that an execution filter is trying to protect, a "double-unlock" error could occur. Thus, LOOM must (1) check for inconsistent states and (2) install the filter only in consistent ones. Moreover, LOOM must make the two steps atomic, despite the concurrently running application threads and multiple points of updates. This problem cannot be solved by a common safety heuristic called *function quiescence* [2, 13, 21, 39]. We thus create a new algorithm termed *evacuation* to solve this problem by proactively quiescing an arbitrary set of code regions given at runtime. We believe this algorithm can also benefit other live update systems.

A key performance challenge LOOM faces is to maintain negligible performance overhead during an application's normal operations to encourage adoption. The main runtime overhead comes from the engine used to live-update an application binary. Although LOOM can use general-purpose binary instrumentation tools such as Pin, the overhead of these tools (up to 199% [34] and 1065.39% in our experiments) makes them less suitable as options for LOOM. We thus create a *hybrid instrumentation* engine to reduce overhead. It statically transforms an application to include a "hot backup", which can then be updated arbitrarily by execution filters at runtime.

We implemented LOOM on Linux. It runs in user space and requires no modifications to the applications or the OS, simplifying deployment. It does not rely on non-portable OS features (*e.g.,* `SIGSTOP` to pause applications, which is not supported properly on Windows). LOOM's static transformation is a plugin to the LLVM compiler [3], requiring no changes to the compiler either.

We evaluated LOOM on nine real races from a diverse set of six applications: two server applications, MySQL and Apache; one desktop application PBZip2 (a parallel compression tool); and implementations of three scientific algorithms in SPLASH2 [7]. Our results show that

1. LOOM is effective. It can flexibly and safely fix all races we have studied. It does not degrade application availability when installing execution filters. Its evacuation algorithm can install a fix within a second even under heavy workload, whereas a live update approach using function quiescence cannot install the fix in an hour, the time limit of our experiment.
2. LOOM is fast. LOOM has negligible performance overhead and in some cases even speeds up the applications. The one exception is MySQL. Running MySQL with LOOM alone increases response time by 4.11% and degrades throughput by 3.76%.
3. LOOM is scalable. Experiments on a 48-core ma-

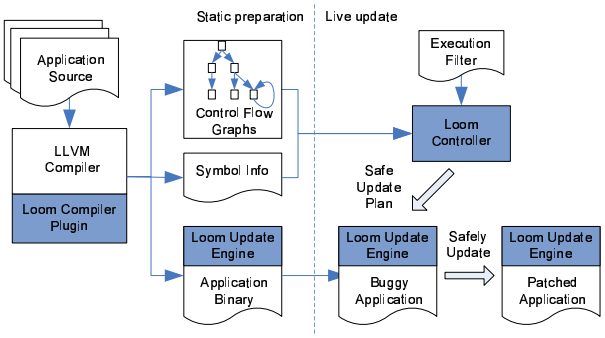


Figure 1: LOOM *overview.* Its components are shaded.

chine show that LOOM scales well as the number of application threads increases.

4. LOOM is easy to use. Execution filters are concise, safe, and flexible (able to fix all races studied, often in more than one way).

This paper is organized as follows. We first give an overview of LOOM (§2). We then describe LOOM's execution filter language (§3), the evacuation algorithm (§4), and the hybrid instrumentation engine (§5). We then present our experimental results (§6). We finally discuss related work (§7) and conclude (§8).

2 Overview

Figure 1 presents an overview of LOOM. To use LOOM for live update, developers first statically transform their applications with LOOM's compiler plugin. This plugin injects a copy of LOOM's update engine into the application binary; it also collects the application's control flow graphs (CFG) and symbol information on behalf of the live update engine.

LOOM's compiler plugin runs within the LLVM compiler [3]. We choose LLVM for its compatibility with GCC and its easy-to-analyze intermediate representation (IR). However, LOOM's algorithms are general and can be ported to other compilers such as GCC. Indeed, for clarity we will present all our algorithms at the source level (instead of the LLVM IR level).

To fix a race, application developers write an execution filter in LOOM's filter language and distribute the filter to application users. A user can then install the filter to immediately protect their application by running

```
% loomctl add <pid> <filter-file>
```

Here `loomctl` is a user-space program called the LOOM controller that interacts with users and initiates live update sessions, `pid` denotes the process ID of a buggy application instance, and `filter-file` is a file containing the execution filter. Under the hood, this controller compiles the execution filter down to a safe update plan using the CFGs and symbol information collected by the compiler plugin. This update plan includes three parts: (1) synchronization operations to enforce the constraints described in the filter and where, in the application, to add the operations; (2) safety preconditions that must hold for installing the filter; and (3) sanity checking code to detect potential errors in the filter itself. The controller sends the update plan to the update engine running as a thread inside the application's address space, which then monitors the runtime states of the application and carries out the update plan only when all the safety preconditions are satisfied.

If LOOM detects a problem with a filter through one of its sanity checks, it can automatically remove the problematic filter. It again waits for all the safety preconditions to hold before removing the filter.

Users can also remove a filter manually, if for example, the race that the filter intends to fix turns out to be benign. They do so by running

```
% loomctl ls <pid>
% loomctl remove <pid> <filter-id>
```

The first command "`loomctl ls`" returns a list of installed filter IDs within process `pid`. The second command "`loomctl remove`" removes filter `filter-id` from process `pid`.

Users can replace an installed filter with a new filter, if for example the new filter fixes the same race but has less performance overhead. Users do so by running

```
% loomctl replace <pid> <old-id> <new-file>
```

where `old-id` is the ID of the installed filter, and `new-file` is a file containing the new filter. LOOM ensures that the removal of the old filter and the installation of the new filter are atomic, so that the application is always protected from the given race.

2.1 Usage Scenarios

LOOM enables users to explicitly describe their synchronization intents and orchestrate thread interleavings of live applications accordingly. Using this mechanism, we envision a variety of strategies users can use to fix races.

Live update At the most basic level, users can translate some conventional patches into execution filters, and use LOOM to install them to live applications.

Temporary workaround Before a permanent fix (*i.e.,* a correct source patch) is out, users can create an execution filter as a crude, temporary fix to a race, to provide immediate protection to highly critical applications.

Preventive fix When a *potential* race is reported (*e.g.,* by automated race detection tools or users of the application), users can immediately install a filter to prevent the race suspect. Later, when developers deem this report false or benign, users can simply remove the filter.

Cooperative fix Users can share filters with each other. This strategy enjoys the same benefits as other cooperative protection schemes [17, 26, 44, 50]. One advantage of LOOM over some of these systems is that it automatically verifies filter safety, thus potentially reducing the need to trust other users.

Site-specific fix Different sites have different workloads. An execution filter too expensive for one site may be fine for another. The flexibility of execution filters allows each site to choose what specific filters to install.

Fix without live update For applications that do not need live update, users can still use LOOM to create quick workarounds, improving reliability.

Besides fixing races, LOOM can be used for the opposite: demonstrating a race by forcing a racy thread interleaving. Compared to previous race diagnosis tools that handle a fixed set of race patterns [25, 41, 42, 49], LOOM's advantage is to allow developers to construct potentially complex "concurrency" testcases.

Although LOOM can also avoid deadlocks by avoiding deadlock-inducing thread interleavings, it is less suitable for this purpose than existing tools (*e.g.,* Dimmunix [26]). To avoid races, LOOM's update engine can add synchronizations to arbitrary program locations. This engine is overkill for avoiding deadlocks: intercepting lock operations (*e.g.,* via `LD_PRELOAD`) is often enough.

2.2 Limitations

LOOM is explicitly designed to work around (broadly defined) races because they are some of the most difficult bugs to fix and this focus simplifies LOOM's execution filter language and safety analysis. LOOM is not intended for other classes of errors. Nonetheless, we believe the idea of high-level and easy-to-verify fixes can be generalized to many other classes of errors.

LOOM does not attempt to fix *occurred* races. That is, if a race has caused bad effects (*e.g.,* corrupted data), LOOM does not attempt to reverse the effects (*e.g.,* recover the data). It is conceivable to allow developers to provide a general function that LOOM runs to recover occurred races before installing a filter. Although this feature is simple to implement, it makes safety analysis infeasible. We thus rejected this feature.

Safety in LOOM terms means that an execution filter and its installation/removal processes introduce no new correctness errors to the application. However, similar to other safe error recovery [46] or avoidance [26, 52] tools, LOOM runs with the application and perturbs timing, thus it may expose some existing application races because it makes some thread interleavings more likely to occur. Moreover, execution filters synchronize code, and may introduce deadlocks and performance problems. LOOM can recover from filter-introduced deadlocks (§3.3) using timeouts, but currently does not deal with performance problems.

At an implementation level, LOOM currently supports a fixed set of synchronization constraint types. Although adding new types of constraints is easy, we have found the existing constraint types sufficient to fix all races evaluated. Another issue is that LOOM uses debugging symbol information in its analysis, which can be inaccurate due to compiler optimization. This inaccuracy has not been a problem for the races in our evaluation because LOOM keeps an unoptimized version of each basic block for live update (§5).

3 Execution Filter Language

LOOM's execution filter language allows developers to explicitly declare their synchronization intents on code. This declarative approach has several benefits. First, it frees developers from the low-level details of synchronization, increasing race fixing productivity. Second, it also simplifies LOOM's safety analysis because LOOM does not have to reverse-engineer developer intents (*e.g.,* what goes into a critical section) from low-level synchronization operations (*e.g.,* scattered `lock()` and `unlock()`), which can be difficult and error-prone. Lastly, LOOM can easily insert error-checking code for safety when it compiles a filter down to low-level synchronization operations.

3.1 Example Races and Execution Filters

In this section, we present two real races and the execution filters to fix them to demonstrate LOOM's execution filter language and its flexibility.

The first race is in MySQL (Bugzilla # 791), which causes the MySQL on-disk transaction log to miss records. Figure 2 shows the race. The code on the left (function `new_file()`) rotates MySQL's transaction log file by closing the current log file and opening a new one; it is called when the transaction log has to be flushed. The code on the right is used by MySQL to append a record to the transaction log. It uses *double-checked locking* and writes to the log only when the log is open. The race occurs if the racy `is_open()` (T2, line 3) catches a closed log when thread T1 is between the `close()` (T1, line 5) and the `open()` (T1, line 6).

Although a straightforward fix to the race exists, performance demands likely forced developers to give up the fix and choose a more complex one instead. The straightforward fix should just remove the racy check (T2, line 3). Unfortunately, this fix creates unnecessary overhead if MySQL is configured to skip logging for speed; this overhead can increase MySQL's response time by more than 10% as observed in our experiments. The concern to this overhead likely forced MySQL developers to use a more involved fix, which adds a new flag field to MySQL's transaction log and modifies the

```
1: // log.cc. thread T1
2: void MYSQL_LOG::new_file(){
3:     lock(&LOCK_log);
4:     ...
5:     close(); // log is closed
6:     open(...);
7:     ...
8:     unlock(&LOCK_log);
9: }
```

```
1: // sql_insert.cc. thread T2
2: // [race] may return false
3: if (mysql_bin_log.is_open()){
4:     lock(&LOCK_log);
5:     if (mysql_bin_log.is_open()){
6:         ... // write to log
7:     }
8:     unlock(&LOCK_log);
9: }
```

Figure 2: *A real MySQL race, slightly modified for clarity.*

```
// Execution filter 1: unilateral exclusion
{log.cc:5, log.cc:6} <> *

// Execution filter 2: mutual exclusion of code
{log.cc:5, log.cc:6} <> MYSQL_LOG::is_open

// Execution filter 3: mutual exclusion of code and data
{log.cc:5 (this), log.cc:6 (this)} <> MYSQL_LOG::is_open(this)
```

Figure 3: *Execution filters for the MySQL race in Figure 2.*

`close()` function to distinguish a regular `close()` call and one for reopening the log.

In contrast, LOOM allows developers to create temporary workarounds with flexible performance and reliability tradeoffs. These temporary fixes can protect the application until developers create a correct and efficient fix at the source level. Figure 3 shows several execution filters that can fix this race. Execution filter 1 in the figure is the most conservative fix: it makes the code region between T1, line 5 and T1, line 6 atomic against all code regions, so that when a thread executes this region, all other threads must pause. We call such a synchronization constraint *unilateral exclusion* in contrast to mutual exclusion that requires participating threads agree on the same lock.[1] Here operator "<>" expresses mutual exclusion constraints, its first operand "{`log.cc:5, log.cc:6`}" specifies a code region to protect, and its second operand "$\star$" represents all code regions. This "expensive" fix incurs only 0.48% overhead (§6.1) because the log rotation code rarely executes.

Execution filter 2 reduces overhead by refining the "$\star$" operand to a specific code region, function `MYSQL_LOG::is_open()`. This filter makes the two code regions mutually exclusive, regardless of what memory locations they access. Execution filter 3 further improves performance by specifying the memory location accessed by each code region.

The second race causes PBZip2 to crash due to a use-after-free error. Figure 4 shows the race. The crash occurs when `fifo` is dereferenced (line 10) after it is freed (line 5). The reason is that the `main()` thread does not wait for the `decompress()` threads to finish. To fix this race, developers can use the filter in Figure 5, which constrains line 10 to run for `numCPU` times before line 5.

[1]Note that unilateral exclusion differs (subtly) from single-threaded execution: unilateral exclusion allows no context switches.

```
// pbzip2.cpp. thread T1
1: main() {
2:     for(i=0;i<numCPU;i++)
3:         pthread_create(...,
4:             decompress, fifo);
5:     queueDelete(fifo);
6: }
```

```
// pbzip2.cpp. thread T2
7 : void *decompress(void *q){
8 :     queue *fifo = (queue *)q;
9 :     ...
10:     pthread_mutex_lock(fifo->mut);
11:     ...
12: }
```

Figure 4: *A real PBZip2 race, simplified for clarity.*

```
pbzip2.cpp:10 {numCPU} > pbzip2.cpp:5
```

Figure 5: *Execution filter for the PBZip2 race in Figure 4.*

3.2 Syntax and Semantics

Table 2 summarizes the main syntax and semantics of LOOM's execution filter language. This language allows developers to express synchronization constraints on *events* and *regions*. An event in the simplest form is "$file : line$," which represents a dynamic instance of a static program statement, identified by file name and line number. An event can have an additional "$(expr)$" component and an "$\{n\}$" component, where $expr$ and n refer to valid expressions with no function calls or dereferences. The $expr$ expression distinguishes different dynamic instances of program statements and LOOM synchronizes the events only with matching $expr$ values. The n expression specifies the number of occurrences of an event and is used in execution order constraints. A *region* represents a dynamic instance of a static code region, identified by a set of entry and exist events or an application function. A region representing a function call can have an additional "$(args)$" component to distinguish different calls to the same function.

LOOM currently supports three types of synchronization constraints (the bottom three rows in Table 2). Although adding new constraint types is easy, we have found existing ones enough to fix all races evaluated. An execution order constraint as shown in the table makes event e_1 happen before e_2, e_2 before e_3, and so forth. A mutual exclusion constraint as shown makes every pair of code regions r_i and r_j mutually exclusive with each other. A unilateral exclusion constraint conceptually makes the execution of a code region single-threaded.

3.3 Language Implementation

LOOM implements the execution filter language using locks and semaphores. Given an execution order constraint $e_i > e_{i+1}$, LOOM inserts a semaphore `up()` operation at e_i and a `down()` operation at e_{i+1}. LOOM implements a mutual exclusion constraint by inserting `lock()` at region entries and `unlock()` at region exits. LOOM implements a unilateral exclusion constraint reusing the evacuation mechanism (§4), which can pause threads at safe locations and resume them later.

Constructs	Syntax
Event (short as e)	$file : line$ $file : line\,(expr)$ $e\{n\}$, n is # of occurrence
Region (short as r)	$\{e_1, ..., e_i; e_{i+1}, ..., e_n\}$ $func\,(args)$
Execution Order	$e_1 > e_2 > ... > e_n$
Mutual Exclusion	$r_1 <> r_2 <> ... <> r_n$
Unilateral Exclusion	$r <> *$

Table 2: *Execution filter language summary.*

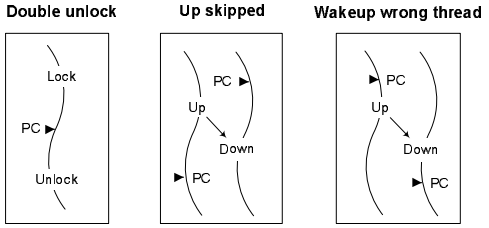


Figure 6: *Unsafe program states for installing filters.*

LOOM creates the needed locks and semaphores on demand. The first time a lock or semaphore is referenced by one of the inserted synchronization operations, LOOM creates this synchronization object based on the ID of the filter, the ID of the constraint, and the value of $expr$ if present. It initializes a lock to an unlocked state and a semaphore to 0. It then inserts this object into a hash table for future references. To limit the size of this table, LOOM garbage-collects these synchronization objects. Freeing a synchronization object is safe as long as it is unlocked (for locks) or has a counter of 0 (for semaphores). If this object is referenced later, LOOM simply re-creates it. The default size of this table is 256 and LOOM never needed to garbage-collect synchronization objects in our experiments.

The `up()` and `down()` operations LOOM inserts behave slightly differently than standard semaphore operations when n, the number of occurrences, is specified. Given $e1\{n_1\} > e2\{n_2\}$, `up()` conceptually increases the semaphore counter by $\frac{1}{n_1}$ and `down()` decreases it by $\frac{1}{n_2}$. Our implementation uses integers instead of floats. LOOM stores the value of n the first time the corresponding event runs and ignores future changes of n.

LOOM computes the values of $expr$ and n using debugging symbol information. We currently allow $expr$ and n to be the following expressions: `a` (constant or primitive variable), `a+b`, `&a`, `&a[i]`, `&a->f`, or any recursive combinations of these expressions. For safety, we do not allow function calls or dereferences. These expressions are sufficient for writing the execution filters in our evaluation.

We implemented this feature using the DWARF library and the `parse_exp_1()` function in GDB. Specifically, we use `parse_exp_1()` to parse the $expr$ or n component into an expression tree, then compile this tree into low level instructions by querying the DWARF library. Note this compilation step is done inside the LOOM controller, so that the live update engine does not have to pay this overhead.

LOOM implements three mechanisms for safety. First, by keying synchronization objects based on filter and constraint IDs, it uses a disjoint set of synchronization objects for different execution filters and constraints, avoiding interference among them. Second, LOOM inserts additional checking code when it generates the update plan. For example, given a code region c in a mutual exclusion constraint, LOOM checks for errors such as c's `unlock()` releasing a lock not acquired by c's `lock()`. Lastly, LOOM checks for filter-induced deadlocks to guard against buggy filters. If a buggy filter introduces a deadlock, one of its synchronization operations must be involved in the wait cycle. LOOM detects such deadlocks using timeouts, and automatically removes the offending filter.

4 Avoiding Unsafe Application States

Figure 6 shows three unsafe scenarios LOOM must handle. For a mutual exclusion constraint that turns code regions into critical sections, LOOM must ensure that no thread is executing within the code regions when installing the filter to avoid "double-unlock" errors. Similarly, for an execution order constraint $e_1 > e_2$, LOOM must ensure either of the following two conditions when installing the filter: (1) both e_1 and e_2 have occurred or (2) neither has occurred; otherwise the `up()` LOOM inserts at e_1 may get skipped or wake up a wrong thread.

Note that a naïve approach is to simply ignore an `unlock()` if the corresponding lock is already unlocked, but this approach does not work with execution order constraints. Moreover, it mixes unsafe program states with buggy filters, and may reject correct filters simply because it tries to install the filters at unsafe program states.

A common safety heuristic called *function quiescence* [2, 13, 21, 39] cannot address this unsafe state problem. This technique updates a function only when no stack frame of this function is active in any call stack of the application. Unfortunately, though this technique can ensure safety for many live updates, it is insufficient for execution filters because their synchronization constraints may affect multiple functions.

We demonstrate this point using a race example. Figure 7 shows the worker thread code of a contrived database. Function `process_client()` is the main thread function. It takes a client socket as input and repeatedly processes requests from the socket. For each request, function `process_client()` opens the corresponding database table by calling `open_table()`, serves the request, and closes the table by calling

```
1 :  // database worker thread
2 :  void handle_client(int fd) {
3 :    for(;;) {
4 :      struct client_req req;
5 :      int ret = recv(fd, &req, ...);
6 :      if(ret <= 0) break;
7 :      open_table(req.table_id);
8 :      ... // do real work
9 :      close_table(req.table_id);
10:    }
11:  }
12:  void open_table(int table_id) {
13:      // fix: acquire table lock
14:      ... // actual code to open table
15:  }
16:  void close_table(int table_id) {
17:      ... // actual code to close table
18:      // fix: release table lock
19:  }
```

Figure 7: *A contrived race.*

`close_table()`. The race in Figure 7 occurs when multiple clients concurrently access the same table.

To fix this race, an execution filter can add a lock acquisition at line 13 in `open_table()` and a lock release at line 18 in `close_table()`. To safely install this filter, however, the quiescence of `open_table()` and `close_table()` is not enough, because a thread may still be running at line 8 and cause a double-unlock error. An alternative fix is to add the lock acquisition and release in function `handle_client()`, but this function hardly quiesces because of the busy loop (line 3-10) and the blocking call `recv()`.

LOOM solves the unsafe state program using an algorithm termed *evacuation* that can proactively quiesce arbitrary code regions. From a high level, this algorithm takes a filter and computes a set of unsafe program locations that may interfere with the filter. It does so conservatively to avoid marking an unsafe location as safe. Then, it "evacuates" threads out of the unsafe locations and blocks them at safe program locations. After that, it installs the filter and resumes the threads.

4.1 Computing Unsafe Program Locations

LOOM uses slightly different methods to compute the unsafe program locations for mutual exclusion and for execution order constraints. To compute unsafe program locations for mutual exclusion constraints, LOOM performs a static reachability analysis on the *interprocedural control flow graph (ICFG)* $\mathcal{G}$ of an application. An ICFG connects each function's control flow graphs by following function calls and returns. Figure 8a shows the ICFG for the code in Figure 7. We say statement s_1 *reaches* s_2 on G or $reachable(G, s_1, s_2)$ if there is a path from s_1 to s_2 on ICFG G. For example, the statement at line 13 reaches the statement at line 8 in Figure 7.

Given an execution filter f with mutual exclusion constraint r_1 <> r_2 <> ... <> r_n, the set of unsafe program locations $unsafe(f)$ includes any statement s potentially inside one of the regions. Specifically, $unsafe(f)$ is the set of statements s such that $reachable(\mathcal{G} \backslash r_i.entries, s, r_i.exits)$ for $i \in [1, n]$, where $r_i.entries$ are the entry statements to region r_i and $r_i.exits$ are the exit statements.

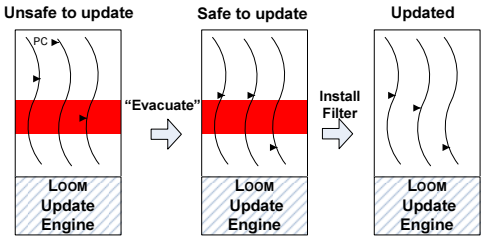


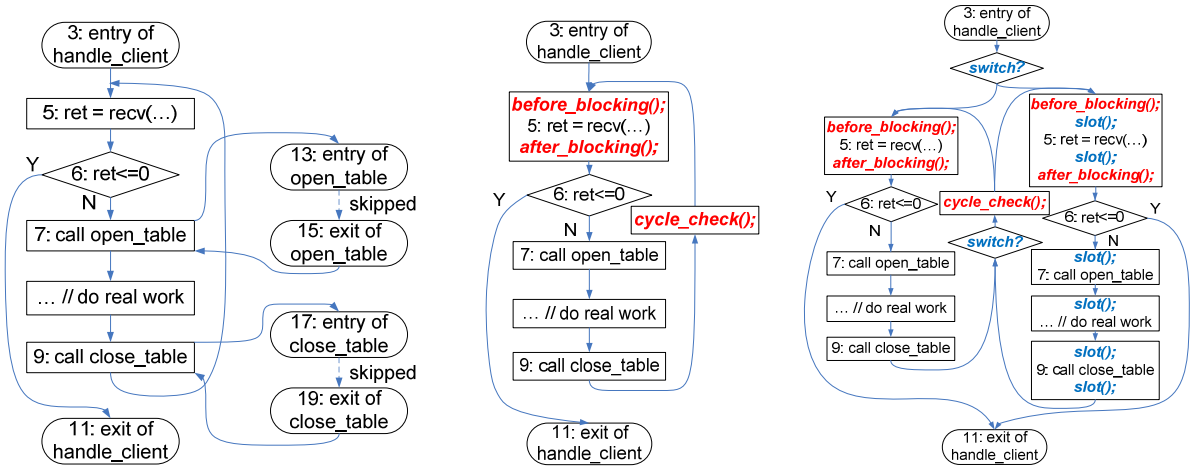
Figure 9: *Evacuation.* Curved lines represent application threads, solid triangles (in black) represents the threads' program counters (PC), and solid stripes (in red) represents an unsafe code region.

LOOM computes unsafe program locations for an execution order constraint by first deriving code regions from the constraint, then reusing the method for mutual exclusion to compute unsafe program locations. Specifically, given a filter f with an execution order constraint $e_1 > e_2 > ... > e_n$, LOOM first computes the set TF of all thread functions (including the main function) that may execute any e_i. It then identifies the entry s_j of each thread function $tf_j \in TF$. It finally computes $unsafe(f)$ as the set of statements between s_j and e_i, denoted by $\{s_j; e_i\}$.

We compute the set $\{s_j; e_i\}$ using a backward reachability analysis. Given the ICFG $\mathcal{G}$ of the program, the set $\{s_j\}$, and the set $\{e_i\}$, our backward reachability analysis first computes $\mathcal{G}^T$, the inversion of $\mathcal{G}$; it then includes s_j, e_i, and every statements that e_i can reach on $\mathcal{G}^T$ in the set $\{s_j; e_i\}$. We only do backward search from e_i, without doing forward search from s_j, because the set s_j of thread entries dominates all statements that these threads may execute.

Our algorithm to compute unsafe locations for execution order filters is safe because it waits until all existing threads to execute beyond e_i. That is, no existing thread will ever run any statements in $\{s_j; e_i\}$. (Threads created after the filter is installed can still run statements in $\{s_j; e_i\}$.)

However, this algorithm is conservative because it may compute a larger-than-necessary set of unsafe locations. That is, to avoid the two unsafe scenarios for order constraints ("up skipped" and "wakeup wrong thread" in Figure 6), it may be unnecessary to require the program counters of every threads to be outside the set. Being conservative may cause a delay when installing a filter. Fortunately, for server applications, which need live-update most, such scenarios rarely occur because most



(a) *ICFG of the code in Figure 7.* (b) *After update-checks inserted.* (c) *After basic blocks cloned.*

Figure 8: *Static transformations that* LOOM *does for safe and fast live update.* Subfigure (a) shows the ICFG of the code in Figure 7; (b) shows the resulting CFG of function `process_client()` after the instrumentation to control application threads (§4); (c) shows the final CFG of function `process_client()` after basic block cloning (§5).

order violations and their fixes only involve with worker threads that run for a short period of time.

Server applications sometimes use thread pools, creating problems for our algorithm. Specifically, during initialization these servers create a fixed set of threads, which loop forever until the application exits to process request. If the loop body contains an event of an execution order filter, the entire loop body will be marked unsafe, and LOOM would have to wait until the application exits to install the filter. Fortunately, we can annotate the request processing function as the conceptual thread function, so that our algorithm does not compute a overly large set of unsafe program locations. In our experiments, we did not (and need not) annotate any request processing function.

4.2 Controlling Application Threads

LOOM needs to control application threads to pause and resume them. It does so using a read-write lock called the update lock. To live update an application, LOOM grabs this lock in write mode, performs the update, and releases this lock. To control application threads, LOOM's compiler plugin instruments the application so that the application threads hold this lock in read mode in normal operation and check for an update once in a while by releasing and re-grabbing this lock.

LOOM carefully places update-checks inside an application to reduce the overhead and ensure a timely update. Figure 8b shows the placement of these update-checks. LOOM needs no update-checks inside straight-line code with no blocking calls because such code can complete quickly. LOOM places one update-check for each cycle in the control flow graph, including loops and recursive function call chains, so that an application thread cycling in one of these cycles can check for an update at least once each iteration. Currently LOOM instruments the backedge of a loop and an arbitrary function entry in a recursive function cycle. LOOM does not instrument every function entry because doing so is costly.

LOOM also instruments an application to release the update lock before a blocking call and re-grab it after the call, so that an application thread blocking on the call does not delay an update. For the example in Figure 7, LOOM can perform the update despite some threads blocking in `recv()`. LOOM instruments only the "leaf-level" blocking calls. That is, if `foo()` calls `bar()` and `bar()` is blocking, LOOM instruments the calls to `bar()`, but not the calls to `foo()`. Currently LOOM conservatively considers calls to external functions (*i.e.,* functions without source), except Math library functions, as blocking to save user annotation effort.

4.3 Pausing at Safe Program Locations

Besides the update lock, LOOM uses additional synchronization variables to ensure that application threads pause at safe locations. LOOM assigns a wait flag for each backedge of a loop and the chosen function entry of a recursive call cycle. To enable/disable pausing at a safe/unsafe location, LOOM sets/clears the corresponding flag. The instrumentation code for each CFG cycle (left of Figure 10) checks for an update only when the corresponding wait flag is set. These wait flags allow application threads at unsafe program locations to run until they reach safe program locations, effectively evacuating the unsafe program locations.

Note that the statement "`if(wait[stmt_id])`" in Figure 10 greatly improves LOOM's performance. With this statement, application threads need not always re-

```
// inserted at CFG cycle
void cycle_check() {
  if(wait[stmt_id]) {
    read_unlock(&update);
    while(wait[stmt_id]);
    read_lock(&update);
  }
}
```

```
// inserted before blocking call
void before_blocking() {
  atomic_inc(&counter[callsite_id]);
  read_unlock(&update);
}
// inserted after blocking call
void after_blocking() {
  read_lock(&update);
  atomic_dec(&counter[callsite_id]);
}
```

Figure 10: *Instrumentation to pause application threads.*

```
volatile int wait[NBACKEDGE] = {0};
volatile int counter[NCALLSITE] = {0};
rwlock_t update;
void evacuate() {
  for each B in safe backedges
    wait[B] = 1; // turn on wait flags
retry:
  write_lock(&update); // pause app threads
  for each C in unsafe callsites
    if(counter[C]) { // threads paused at unsafe callsites
      write_unlock(&update);
      goto retry;
    }
  ... // update
  for each B in safe backedges
    wait[B] = 0; // turn off wait flags
  write_unlock(&update); // resume app threads
}
```

Figure 11: *Pseudo code of the evacuation algorithm.*

lease and re-grab the update lock which can be costly, and hardware cache and branch prediction can effectively hide the overhead of checking these flags. This technique speeds up LOOM significantly (§6) because wait flags are almost always 0 with read accesses.

LOOM cannot use the wait-flag technique to skip a blocking function call because doing so changes the application semantics. Instead, LOOM assigns a counter to each blocking callsite to track how many threads are at the callsites (right of Figure 10). LOOM uses a counter instead of a binary flag because multiple threads may be doing the same call.

Now that LOOM's instrumentation is in place, Figure 11 shows LOOM's evacuation method which runs within LOOM's live update engine. This method first sets the wait flags for safe backedges. It then grabs the update lock in write mode, which pauses all application threads. It then examines the counters of unsafe callsites and if any counter is positive, it releases the update lock and retries, so that the thread blocked at unsafe callsites can wake up and advance to safe locations. Next, it updates the application (§5), clears the wait flags, and releases the update lock.

4.4 Correctness Discussion

We briefly discuss the correctness of our evacuation algorithm in this subsection; for a complete proof, see Appendix A.

In program analysis terms, our reachability analysis (§4.1) is interprocedural and flow-sensitive. We use a crude pointer analysis to discover thread functions, thread join sites, and function pointer targets. We could have refined our analysis to improve precision, but we find it sufficient to compute unsafe locations for all evaluated races because (1) our analysis is sound and never marks an unsafe location safe and (2) execution filters are quite small and slight imprecision does not matter. In the worst case, if our analysis turns out too imprecise for some filters, the flexibility of LOOM allows developers to easily adjust their filters to pass the safety analysis.

Our reachability analysis gives correct results despite compiler reordering. In order to pause application threads at safe locations, our reachability analysis returns only the set of unsafe backedges and external callsites. These locations are instrumented by LOOM; this instrumentation acts as barriers and prevents compilers from reordering instructions across them.

The synchronization between the instrumentation in Figure 10 and the evacuation algorithm in Figure 11 is correct under two conditions: (1) read and write to wait flags are atomic and (2) the operations to the update lock contain correct memory barriers that prevent hardware reordering. Currently we implement wait flags using aligned integers; our update lock operations use atomic operations similar to the Linux kernel's `rw_spinlock`. Thus, our evacuation algorithm works correctly on X86 and AMD64 which do not reorder instructions across atomic instructions. We expect our algorithm to work on other commodity hardware that also provides this guarantee. To cope with more relaxed hardware (*e.g.,* Alpha), we can augment these operations with full barriers.

5 Hybrid Instrumentation

Most previous live update systems update binaries by compiling updated functions and redirecting old functions to the new function binaries using a table or jump instructions. This approach requires source patches to generate the updates, thus it has the limitations described in §1. Moreover, this approach pays the overhead of position independent code (PIC) because application functions must be compiled as PIC for live update. It also suffers the aforementioned function quiescence problem.[2]

Another alternative is to use general-purpose binary instrumentation tools such as vx32 [20], Pin [34] and DynamoRIO [14], but they tend to incur significant runtime overhead just to run their frameworks alone. For example, Pin has been reported to incur 199% overhead [34], and we observed 10 times slowdown on Apache with a

[2]The function quiescence problem can be addressed by transforming loop bodies into functions [38, 39] but only if the CFGs are reducible [23].

```
void slot(int stmt_id) {
  op_list = operations[stmt_id];
  foreach op in op_list
    do op;
}
```

Figure 12: *Slot function.*

CPU-bound workload (§6).

LOOM's hybrid instrumentation engine reduces runtime overhead by combining static and dynamic instrumentation. This engine statically transforms an application's binary to anticipate dynamic updates. The static transformation pre-pads, before each program location, a *slot function* which interprets the updates to this program location at runtime. Figure 12 shows the pseudo code of this function. It iterates though a list of synchronization operations assigned to the current statement and performs each. To update a program location at runtime, LOOM simply modifies the corresponding operation list.

Inserting the slot function at every statement incurs high runtime overhead and hinders compiler optimization. LOOM solves this problem using a basic block cloning idea [29]. LOOM keeps two versions of each basic block in the application binary, an originally compiled version that is optimized, and a hot backup that is unoptimized and padded for live update. To update a basic block at runtime, LOOM simply updates the backup and switches the execution to the backup by flipping a switch flag.

LOOM instruments only function entries and loop backedges to check the switch flags because doing so for each basic block is expensive. Similar to the wait flags in (§4), the switch flags are almost always 0, so that hardware cache and branch predication can effectively hide the overhead of checking them. This technique makes live-update-ready applications run as fast as the original application during normal operations (§6). Figure 8c shows the final results after all LOOM transformations.

Note that the accesses to switch flags are correctly protected by the update lock. An application checks the switch flag when holding the update lock in read mode, and the update engine sets the switch flag when holding the update lock in write mode.

6 Evaluation

We implemented LOOM in Linux. It consists of 4,852 lines of C++ code, with 1,888 lines for the LLVM compiler plugin, 2,349 lines for the live-update engine, and 615 lines for the controller.

We evaluated LOOM on nine real races from a diverse set of applications, ranging from two server applications MySQL [5] and Apache [11], to one desktop application PBZip2 [6], to three scientific applications fft, lu, and barnes in SPLASH2 [7].[3] Table 3 lists all nine races.

[3]We include applications that do not need live update for two reasons. First, as discussed in §1, LOOM can provide quick workarounds for these applications as well. Second, we use them to measure LOOM's overhead and scalability.

Race ID	Description
MySQL-791	Calls to `close()` and `open()` to flush log file are not atomic. Figure 2 shows the code.
MySQL-169	Table update and log write in `mysql_delete()` are not atomic.
MySQL-644	Calls to `prepare()` and `optimize()` in `mysql_select()` are not atomic.
Apache-21287	Reference count decrement and checking are not atomic.
Apache-25520	Threads write to same log buffer concurrently, resulting in corrupted logs or crashes.
PBZip2	Variable `fifo` is used in one thread after being freed by another. Figure 4 shows the code.
SPLASH2-fft	Variable `finishtime` and `initdonetime` are read before assigned the correct values.
SPLASH2-lu	Variable `rf` is read before assigned the correct value.
SPLASH2-barnes	Variable `tracktime` is read before assigned the correct value.

Table 3: *All races used in evaluation.* We identify races in MySQL and Apache as "$\langle application\ name \rangle - \langle Bugzilla\ \# \rangle$", the only race in PBZip2 "$PBZip2$", and races in SPLASH2 "$SPLASH2 - \langle benchmark\ name \rangle$".

Our race selection criteria is simple: (1) they are extensively used in previous studies [31, 42, 43] and (2) the application can be compiled by LLVM and the race can be reproduced on our main evaluation machine, a 2.66 GHz Intel quad-core machine with 4 GB memory running 32-bit Linux 2.6.24.

We used the following workloads in our experiments. For MySQL, we used SysBench [8] (advanced transaction workload), which randomly selects, updates, deletes, and inserts database records. For Apache, we used ApacheBench [1], which repeatedly downloads a webpage. Both benchmarks are multithreaded and used by the server developers. We made both SysBench and ApacheBench CPU bound by fitting the database or web contents within memory; we also ran both the client and the server on the same machine, to avoid masking LOOM's overhead with the network overhead. Unless otherwise specified, we ran 16 worker threads for MySQL and Apache because they performed best with 8-16 threads. We ran four worker threads for PBZip2 and SPLASH2 applications because they are CPU-intensive and our evaluation machine has four cores.

We measured throughput (TPUT) and response time (RESP) for server applications and overall execution time for other applications. We report LOOM's relative overhead, the smaller the better. We compiled the applications down to x86 instructions using `llvm-gcc -O2` and LLVM's bitcode compiler `llc`. For all the performance numbers reported, we repeated the experiment 50

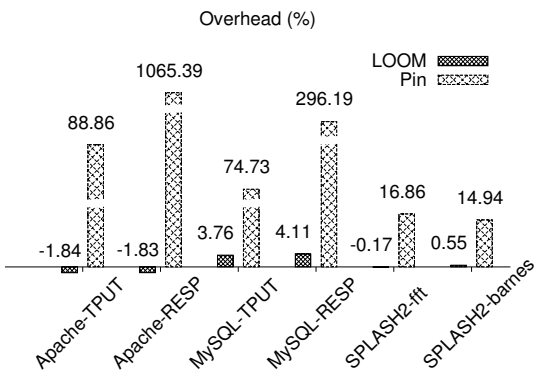


Figure 13: LOOM*'s relative overhead during normal operation.* Smaller numbers are better. We show Pin's overhead for reference. Some Pin bars are broken.

times and take the average.

We focus our evaluation on five dimensions:

1. Overhead. Does LOOM incur low overhead?
2. Scalability. Does LOOM scale well as the number of application threads increases?
3. Reliability. Can LOOM be used to fix the races listed in Table 3? What are the performance and reliability tradeoffs of execution filters?
4. Availability. Does LOOM severely degrade application availability when execution filters are installed?
5. Timeliness. Can LOOM install fixes in a timely way?

6.1 Overhead

Figure 13 shows the performance overhead of LOOM during the normal operations of the applications. We also show the overhead of bare Pin for reference. LOOM incurs little overhead for Apache and SPLASH2 benchmarks. It increases MySQL's response time by 4.11% and degrades its throughput by 3.76%. In contrast, Pin incurs higher overhead for all applications evaluated, especially for Apache and MySQL.

We also evaluated how the optimizations we do reduce LOOM's overhead. Figure 14 shows the effects of these optimizations. Both cloning and wait-flag are very effective at reducing overhead. Cloning reduces LOOM's response-time overhead on Apache from 100% to 17%. It also reduces LOOM's overhead on fft from 15 times to 8 times. Wait-flag actually makes Apache run faster than the original version. Inlining does not help the servers much, but it does help for SPLASH2 applications.

6.2 Scalability

LOOM synchronizes with application threads via a read-write lock. Thus, one concern is, can LOOM scale well as the number of application threads increases? To evaluate LOOM's scalability, we ran Apache and MySQL with LOOM on a 48-core machine with four 1.9 GHz 12-core AMD CPUs and 64 GB memory running 64-bit Linux 2.6.24. In each experiment, we pinned the benchmark to one CPU and the server to the other three to avoid unnecessary CPU contention between them.

Figure 15 shows LOOM's relative overhead vs. the number of application threads for Apache and MySQL. LOOM scales well with the number of threads. Its relative overhead varies only slightly. Even with 32 server threads, the overhead for Apache is less than 3%, and the overhead for MySQL is less than 12%.

Our initial MySQL overhead was around 16%. We analyzed the execution counts of the LOOM-inserted functions and immediately identified two update-check sites (`cycle_check()` calls) that executed exceedingly many times. These update-check sites are in MySQL functions `ptr_compare_1` and `Field_varstring::val_str`. The first function compares two strings, and the second copies one string to another. Each function has a loop with a few statements and no function calls. Such tight loops cause higher overhead for LOOM, but rarely need to be updated. We thus disabled the update-check sites in these two functions, which reduced the overhead of MySQL down to 12%. This optimization can be easily automated using static or dynamic analysis, which we leave for future work.

Race ID	Mutual			Unilateral		
	Events	TPUT	RESP	Events	TPUT	RESP
MySQL-169	2	0.14%	0.15%	1	3.28%	3.37%
MySQL-644	4	0.22%	0.20%	4	32.58%	48.34%
MySQL-791	4	0.23%	0.32%	2	0.33%	0.48%
Apache-21287	16	-0.02%	-0.03%	2	54.03%	118.16%
Apache-25520	1	0.52%	0.55%	1	86.04%	637.03%

Table 4: *Execution filter stats for atomicity errors.* Column Events counts the number of events in each filter.

Race ID	Events	Overhead
PBZip2	6	1.26%
SPLASH2-fft	6	0.08%
SPLASH2-lu	2	1.68%
SPLASH2-barnes	2	1.99%

Table 5: *Execution filter stats for order errors.*

6.3 Reliability

LOOM can be used to fix all races evaluated. (We verified this result by manually inspecting the application binary.) Table 4 shows the statistics for the execution filters that fix atomicity errors. Table 5 shows the statistics for the execution filters that fix order errors.

In all cases, we can fix the race using multiple execution filters, demonstrating the flexibility of LOOM. (The filters for MySQL-791 are shown in Figure 3.) We only show the statistics of one execution filter of each constraint type; other filters of the same type are similar. Our results show that the filters are fairly small, 3.79 events

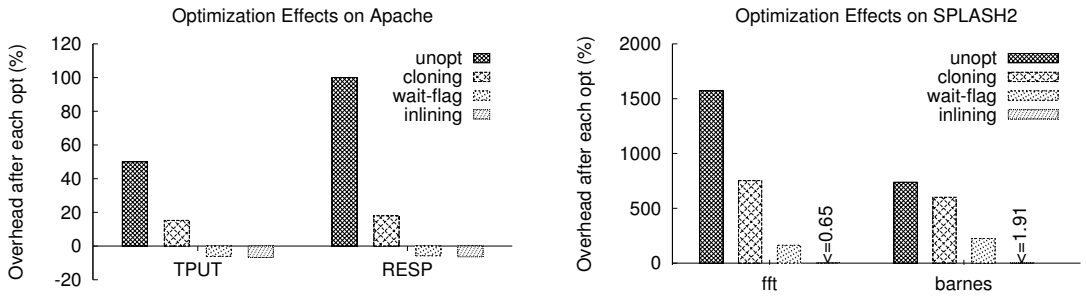


Figure 14: *Effects of* LOOM*'s optimizations.* Label **unopt** represents the versions with no optimizations; **cloning** represents the version with basic block cloning (§5); **wait-flag** represents the version with statement "`if(wait[stmt_id])`" added (§4.2); and **inlining** indicates the version with all LOOM instrumentation inlined into the applications.

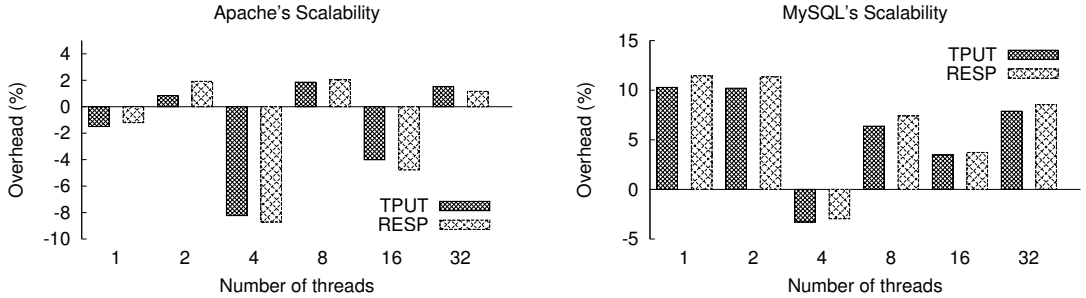


Figure 15: LOOM*'s relative overhead vs. the number of application threads.*

on average and no more than 16 events, demonstrating the ease of use of LOOM. Most filters incur only a small overhead on top of LOOM. Unilateral filters tend to be slightly smaller than mutual exclusion filters, but they can be expensive sometimes. They incur little overhead for two of the MySQL bugs because the code regions protected by the filters rarely run.

These different reliability and performance overheads present an interesting tradeoff to developers. For example, users can choose to install a unilateral filter for immediate protection, then atomically replace it with a faster mutual exclusion filter. Moreover, a user can choose an "expensive" filter as long as their workload is compatible with the filter.

6.4 Availability

We show that LOOM can improve server availability by comparing LOOM to the restart-based software update approach. We restarted a server by running its startup script under `/etc/init.d`. We chose two races, MySQL-791 and Apache-25520, and measured how software updates (conventional or with LOOM) might degrade performance. Note this comparison favors conventional updates because we only compare the installation of the fix, but LOOM also makes it quick to develop fixes. Figure 16 shows the comparison result. Using the restart approach, Apache is unavailable for 4 seconds, and MySQL is unavailable for 2 seconds. Moreover, the restarts also cause Apache and MySQL to lose their internal cache, leading to a ramp-up period after the restart. In contrast, installing an filter using LOOM (at second 5) does not degrade throughput for MySQL and only degrades throughput slightly for Apache.

6.5 Timeliness

The more timely LOOM installs a filter, the quicker the application is protected from the corresponding race. This timeliness is critical for server applications because malicious clients may exploit a known race and launch attacks. In this subsection, we compare how timely LOOM's evacuation algorithm installs an aggressive filter vs. an approach that passively waits for function quiescence. We chose Apache-25520 as the benchmark race. We wrote a simple mutual exclusion filter that fixes the race by making function `ap_buffered_log_writer` a critical region. We then measured the latency from the moment LOOM receives a filter to the moment the filter is installed. We simulated a function quiescence approach by running LOOM without making any `wait_flag` false, so that a thread can pause wherever we insert update-checks. We used the same SysBench and ApacheBench workload. Our results show that LOOM can install the filter within 368 ms. It spends majority of the time waiting for threads to evacuate. In

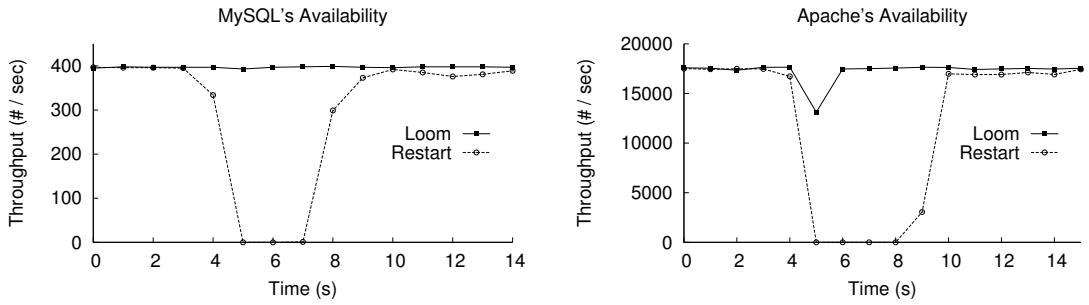


Figure 16: *Throughput degradation for fixing races with* LOOM *vs. with conventional software update.*

contrast, an approach based on function quiescence fails to install the filter in an hour, our experiment's time limit.

7 Related Work

Live update LOOM differs from previous live update systems [10, 12, 15, 35, 38, 39, 51] in that it is explicitly designed for developers to quickly develop temporary workarounds to races. Moreover, it can automatically ensure the safety of the workarounds. In contrast, previous work focuses only on live update after a source patch is available, thus it does not address the automatic-safety and flexibility problems LOOM addresses.

The live update system closest to LOOM is STUMP [38], which can live-update multithreaded applications written in C. Its prior version Ginseng [39] works with single-threaded C applications. Both STUMP and Ginseng have been shown to be able to apply arbitrary source patches and update applications across major releases. Unlike LOOM, both STUMP and Ginseng require source modifications and rely on extensive user annotations for safety because the safety of arbitrary live updates has been proven undecidable [22].

A number of live update systems can update kernels without reboots [12, 15, 35]. The most recent one, Ksplice [12], constructs live updates from object code, and does not require developer efforts to adapt existing source patches. Unlike LOOM, Ksplice uses function quiescence for safety, and is thus prone to the unsafe state problem discussed in §4. Another kernel live update system, DynAMOS [35], requires users to manually construct multiple versions of a function to update non-quiescent functions. This technique is different from basic block cloning (§5): the former is manual and for safety, whereas the later is automatic and for speed.

Error workaround and recovery We compare LOOM to recent error workaround and recovery tools. ClearView [44], ASSURE [50], and Failure-oblivious computing can increase application availability by letting them continue despite errors. Compared to LOOM, these systems are unsafe, and do not directly deal with races. Rx [46] can safely recover from runtime faults using application checkpoints and environment modifications, but it does not fix errors because the same error can re-appear. Vigilante [17] enables hosts to collaboratively contain worms using self-verifiable alerts. By automatically ensuring filter safety, LOOM shares similar benefits.

Two recent systems, Dimmunix [26] and Gadara [52], can fix deadlocks in legacy multithreaded programs. Dimmunix extracts signatures from occurred deadlocks (or starvations) and dynamically avoids them in future executions. Gadara uses control theory to statically transform a program into a deadlock-free program. Both systems have been shown to work on real, large applications. They may possibly be adapted to fix races, albeit at a coarser granularity because these systems control only lock operations.

Kivati [16] automatically detects and prevents atomicity violations for production systems. It reduces performance overhead by cleverly using hardware watch points, but the limited number of watch points on commodity hardware means that Kivati cannot prevent all atomicity violations. Nor does Kivati prevent execution order violations. LOOM can be used to workaround these errors missed by Kivati.

Program instrumentation frameworks Previous work [3, 19, 40] can instrument programs with low runtime overhead, but instrumentation has to be done at compile time. Translation-based dynamic instrumentation frameworks [14, 20, 34] can update programs at runtime but incur high overhead. In particular, vx32 [20] is a novel user-level sandbox that reduces overhead using segmentation hardware; it can be used as an efficient dynamic binary translator. Jump-based instrumentation frameworks [24, 48] have low overhead but automatically ensuring safety for them can be difficult due to low-level issues such as position-dependent code, short instructions, and locations of basic blocks.

One advantage of these instrumentation frameworks over LOOM is that LOOM requires CFGs and symbol information to be distributed to user machines, thus it risks leaking proprietary code information. However, this risk

is not a concern for open-source software. Moreover, LOOM only mildly increases this risk because CFGs can often be reconstructed from binaries, and companies such as Microsoft already share symbol information [4].

The advantage of LOOM is that it combines static and dynamic instrumentation, thus allowing arbitrary dynamic updates issued by execution filters with negligible runtime overhead. LOOM borrows basic block cloning from previous work by Liblit *et al.* [29], but their framework is static only. This idea has also been used in other systems (*e.g.,* LIFT [45]).

Other related work Our work was inspired by many observations made by Lu *et al.* [33]. Aspect-oriented programming (AOP) allows developers to "weave" in synchronizations into code [27, 30]. LOOM's execution filter language shares some similarity to AOP, and can be made more expressive by incorporating more aspects. However, to the best of our knowledge, no existing AOP systems were designed to support race fixing at runtime. We view the large body of race detection and diagnosis work (*e.g.,* [31, 32, 37, 42, 47, 49, 53]) as complimentary to our work and LOOM can be used to fix errors detected and isolated by these tools.

8 Conclusion

We have presented LOOM, a live-workaround system designed to quickly and safely fix application races at runtime. Its flexible language allows developers to write concise execution filters to declare their synchronization intents on code. Its evacuation algorithm automatically ensures the safety of execution filters and their installation/removal processes. It uses hybrid instrumentation to reduce its performance overhead during the normal operations of applications. We have evaluated LOOM on nine real races from a diverse set of applications. Our results show that LOOM is fast, scalable, and easy to use. It can safely fix all evaluated races in a timely manner, thereby increasing application availability.

LOOM demonstrates that live-workaround systems can increase application availability with little performance overhead. In our future work, we plan to extend this idea to other classes of errors (*e.g.,* security vulnerabilities).

Acknowledgement

We thank Cristian Cadar, Jason Nieh, Jinyang Li, Michael Kester, Xiaowei Yang, Vijayan Prabhakaran (our shepherd), and the anonymous reviewers for their tremendous feedback and comments, which have substantially improved the content and presentation of this paper. We thank Shan Lu for providing many of the races used in our evaluation. We thank Jane-Ellen Long for time management.

This work was supported by the National Science Foundation (NSF) through Contract CNS-1012633 and CNS-0905246 and the United States Air Force Research Laboratory (AFRL) through Contract FA8650-10-C-7024 and FA8750-10-2-0253. Opinions, findings, conclusions, and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the US Government.

References

[1] ab - Apache HTTP server benchmarking tool. `http://httpd.apache.org/docs/2.2/programs/ab.html`.

[2] The K42 Project. `http://www.research.ibm.com/K42/`.

[3] The LLVM Compiler Framework. `http://llvm.org`.

[4] Download windows symbol packages. `http://www.microsoft.com/whdc/devtools/debugging/debugstart.mspx`.

[5] MySQL Database. `http://www.mysql.com/`.

[6] Parallel BZIP2 (PBZIP2). `http://compression.ca/pbzip2/`.

[7] Stanford Parallel Applications for Shared Memory (SPLASH). `http://www-flash.stanford.edu/apps/SPLASH/`.

[8] SysBench: a system performance benchmark. `http://sysbench.sourceforge.net`.

[9] G. Altekar and I. Stoica. ODR: output-deterministic replay for multicore debugging. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 193–206, Oct. 2009.

[10] G. Altekar, I. Bagrak, P. Burstein, and A. Schultz. OPUS: online patches and updates for security. In *Proceedings of the 14th USENIX Security Symposium*, 2005.

[11] Apache Web Server. `http://www.apache.org`.

[12] J. Arnold and F. M. Kaashoek. Ksplice: Automatic rebootless kernel updates. In *Proceedings of the 4th ACM European Conference on Computer Systems (EUROSYS '09)*, pages 187–198, Apr. 2009.

[13] A. Baumann, G. Heiser, J. Appavoo, D. Da Silva, O. Krieger, R. W. Wisniewski, and J. Kerr. Providing dynamic update in an operating system. In *Proceedings of the USENIX Annual Technical Conference (USENIX '05)*, pages 32–32, 2005.

[14] D. L. Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, 2004. Supervisor-Amarasinghe, Saman.

[15] H. Chen, R. Chen, F. Zhang, B. Zang, and P.-C. Yew. Live updating operating systems using virtualization. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE '06)*, pages 35–44, 2006.

[16] L. Chew and D. Lie. Kivati: fast detection and prevention of atomicity violations. In *EuroSys '10: Proceedings of the 5th European conference on Computer systems*, pages 307–320, 2010.

[17] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: end-to-end containment of internet worms. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 133–147, 2005.

[18] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *Proceedings of the 4th International Conference on Virtual Execution Environments (VEE '08)*, pages 121–130, Mar. 2008.

[19] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system

rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI '00)*, Sept. 2000.

[20] B. Ford and R. Cox. Vx32: lightweight user-level sandboxing on the x86. In *Proceedings of the USENIX Annual Technical Conference (USENIX '08)*, pages 293–306, 2008.

[21] S. Gilmore and C. Walton. Dynamic ML without dynamic types. Technical report, Lab. for the Foundations of Computer Science, University of Edinburgh, 1997.

[22] D. Gupta, P. Jalote, and G. Barua. A formal framework for online software version change. *IEEE Transactions on Software Engineering*, 22(2):120–131, 1996.

[23] M. S. Hecht and J. D. Ullman. Characterizations of reducible flow graphs. *Journal of the ACM*, 21(3):367–375, 1974.

[24] G. Hunt and D. Brubacher. Detours: Binary interception of win32 functions. In *In Proceedings of the 3rd USENIX Windows NT Symposium*, pages 135–143, 1998.

[25] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation (PLDI '09)*, pages 110–120, June 2009.

[26] H. Jula, D. Tralamazza, Z. Cristian, and C. George. Deadlock immunity: Enabling systems to defend against deadlocks. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 295–308, Dec. 2008.

[27] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. marc Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, 1997.

[28] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '10)*, pages 155–166, June 2010.

[29] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 141–154, 2003.

[30] D. Lohmann, W. Hofer, W. Schrder-Preikschat, J. Streicher, and O. Spinczyk. CiAO: An aspect-oriented operating-system family for resource-constrained embedded systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '09)*, 2009.

[31] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *Twelfth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '06)*, pages 37–48, Oct. 2006.

[32] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. *SIGOPS Oper. Syst. Rev.*, 41(6):103–116, 2007.

[33] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Thirteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '08)*, pages 329–339, Mar. 2008.

[34] 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 *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI '05)*, pages 190–200, 2005.

[35] K. Makris and K. Ryu. Dynamic and adaptive updates of nonquiescent subsystems in commodity operating system kernels. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, page 340, 2007.

[36] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: a software-hardware interface for practical deterministic multiprocessor replay. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 73–84, Mar. 2009.

[37] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 267–280, Dec. 2008.

[38] I. Neamtiu and M. Hicks. Safe and timely dynamic updates for multi-threaded programs. In *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation (PLDI '09)*, pages 13–24, June 2009.

[39] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for C. pages 72–83, June 2006.

[40] G. C. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of Conference on Compilier Construction*, March 2002.

[41] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '08/FSE-16)*, pages 135–145, Nov. 2008.

[42] S. Park, S. Lu, and Y. Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 25–36, Mar. 2009.

[43] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 177–192, Oct. 2009.

[44] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 87–102, 2009.

[45] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 135–148, 2006.

[46] F. Qin, J. Tucek, Y. Zhou, and J. Sundaresan. Rx: Treating bugs as allergies—a safe method to survive software failures. *ACM Trans. Comput. Syst.*, 25(3):7, 2007.

[47] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multithreaded programming. *ACM Trans. Comput. Syst.*, pages 391–411, Nov. 1997.

[48] M. Schulz, D. Ahn, A. Bernat, B. R. de Supinski, S. Y. Ko, G. Lee, and B. Rountree. Scalable dynamic binary instrumentation for blue gene/l. *SIGARCH Comput. Archit. News*, 33(5): 9–14, 2005.

[49] K. Sen. Race directed random testing of concurrent programs.

In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI '08)*, pages 11–21, June 2008.

[50] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis. ASSURE: automatic software self-healing using rescue points. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 37–48, 2009.

[51] S. Subramanian, M. Hicks, and K. S. McKinley. Dynamic software updates: a vm-centric approach. In *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation (PLDI '09)*, pages 1–12, 2009.

[52] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 281–294, Dec. 2008.

[53] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: efficient detection of data race conditions via adaptive tracking. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 221–234, Oct. 2005.

A Proof of the Correctness of the Evacuation Algorithm

To prove our evacuation algorithm is correct, we show that for any mutual exclusion constraint or execution order constraint f, requiring that no threads are running at locations in $unsafe(f)$ prevents all the three unsafe scenarios show in Figure 6. (Our algorithm is conservative because it may prevent some safe scenarios, too.)

When LOOM installs a filter f with mutual exclusion constraint $r_1 <> r_2 <> \ldots <> r_n$ that turns code regions into critical sections, our evacuation algorithm ensures that no threads are running at statement s such that $reachable(\mathcal{G} \backslash r_i.entries, s, r_i.exits)$ for $i \in [1, n]$, *i.e.,* , each thread either cannot reach any critical section at all or has to reach a critical section entry before reaching any exit. Therefore, the "double unlock" scenario in Figure 6, the only unsafe scenario for mutual exclusion constraints, will not happen.

Given a filter f with execution order constraint $e_1 > e_2 > \ldots > e_n$, our evacuation algorithm ensures that no threads are running at locations in $\{s_j; e_i\}$, where s_j are the entries of the thread functions that may execute any event e_i in f. Because the entry of a thread function dominates all statements that this thread function may execute, $\{s_j; e_i\}$ includes every statements that can reach any event e_i on the ICFG of the program. By keeping all currently running threads out of $\{s_j; e_i\}$, LOOM avoids both unsafe scenarios for execution order constraints ("up skipped" and "wakeup wrong thread" in Figure 6) in which some threads can reach e_i.