Optimistic Concurrency Control for Real-world Go Programs

Zhizhou Zhang, University of California, Santa Barbara; Milind Chabbi and Adam Welc, Uber Technologies; Timothy Sherwood, University of California, Santa Barbara

https://www.usenix.org/conference/atc21/presentation/zhang-zhizhou
Optimistic Concurrency Control for Real-world Go Programs

Zhizhou Zhang¹, Milind Chabbi², Adam Welc², and Timothy Sherwood¹

¹University of California, Santa Barbara
¹/zhizhouzhang, sherwood]@cs.ucsb.edu
²Programming Systems Group, Uber Technologies
²/{milind,adam.welc}@uber.com

Abstract
We present a source-to-source transformation framework, GoCC, that consumes lock-based pessimistic concurrency programs in the Go language and transforms them into optimistic concurrency programs that use Hardware Transactional Memory (HTM). The choice of the Go language is motivated by the fact that concurrency is a first-class citizen in Go, and it is widely used in Go programs. GoCC performs rich inter-procedural program analysis to detect and filter lock-protected regions and performs AST-level code transformation of the surrounding locks when profitable. Profitability is driven by both static analyses of critical sections and dynamic analysis via execution profiles. A custom HTM library, using perception, learns concurrency behavior and dynamically decides whether to use HTM in the rewritten lock/unlock points. Given the rich history of transactional memory research but its lack of adoption in any industrial setting, we believe this workflow, which ultimately produces source-code patches, is more apt for industry-scale adoption. Results on widely adopted Go libraries and applications demonstrate significant (up to $10 \times$) and scalable performance gains resulting from our automated transformation while avoiding major performance regressions.

1 Introduction
Golang [46] (or simply Go) is a modern programming language that has gained significant popularity over the last decade. It is being used to write enterprise software [20] (e.g., to implement backend services) in some of the largest technology companies as well as to develop large and widely used open-source applications (e.g., Kubernetes [47]) and libraries (e.g., Tally [88]). The design of Go is inspired by C, but unlike C, it supports concurrency as the first-class language construct. Even more importantly, and unlike other popular languages with first-class concurrency support (e.g., Java), the Go language goes to great lengths to simplify concurrent programming by making concurrency easy to use (and thus frequently used) by the developers [86] — any function in Go can be scheduled to execute concurrently with the rest of the code as a goroutine [46] by simply prefixing its call with the go keyword.

Although Go makes writing concurrent programs easier, it still requires programmers to manage interactions between concurrently executing code — this can be accomplished either via passing messages through channels [46] or explicitly synchronizing accesses to shared memory. Shared memory is used more often than message passing by Go developers, and mutual exclusion via locks [11] remains the most widely-used synchronization mechanism across several applications [86]. It is, therefore, the focus of our work.

Locks may unnecessarily serialize concurrent execution, even if the code operates on disjoint data. Our work aims to improve the performance of concurrent Go code, particularly code hiding behind needlessly held locks. Our goal is to accomplish this while retaining the correctness of concurrent execution. We utilize the concept of transactional memory (TM) [53] to achieve this goal. The general idea behind TM is to decide on whether two (or more) pieces of code can be executed concurrently based on whether their accesses to the underlying data are conflicting [54] or not, that is, if at least one of the accesses is a write. Conflict-free executions are allowed to proceed in parallel. On the other hand, upon encountering a data access conflict, execution effects of at least one piece of code have to be rolled back (i.e., undone), and the computation must be restarted. TM machinery, which originally started in software (STM) [43, 77, 82, 93], is now available in commodity hardware as Hardware Transactional Memory (HTM) [41, 90, 92]. However, despite almost three decades of work in this area, TM’s promise of accelerating concurrent computations for real-life software has not been quite fulfilled. We speculate that there are two reasons why this is the case.

The first reason is that TM, while being a single concept, may have different realizations in terms of algorithms and implementations (e.g., eager vs. lazy versioning [80]) and different integration strategies at the language level (e.g., API-level solutions [77] or the compiler-assisted atomic construct used to demarcate TM-managed concurrent code [52]), resulting in different behavior from the programmer’s perspective. Conse-
quently, attempts to introduce TM as a separate language-level mechanism lead to significant semantic dissonance with respect to existing concurrency-related mechanisms [69, 79]. The second reason is that a lot of TM (particularly STM) work was focusing on designing and implementing TM algorithms but limiting empirical evaluation to synthetic benchmarks (e.g., STM Bench7 [51]) or measuring the performance of only selected concurrent data structures. Unfortunately, unlike what was expected, TM techniques did not easily generalize to real-life applications [93]. A few attempts to apply TM to production code were unsuccessful (e.g., an attempt to rewrite the Quake game server to use TM [97]).

In this work, we attempt to rectify some of these limitations and show that TM can be effective in accelerating real-life concurrent code. We focus less on the algorithmic side of TM (we use state-of-the-art off-the-shelf HTM implementation from Intel), and more on how and when to apply the TM machinery to maximize the benefit. Additionally, we replace Go locks with HTM constructs without changing the code’s behavior in any way, which allows us to completely bypass complications related to transactional memory semantics. More specifically, we employ transactional lock elision (TLE) [73] — a well-known technique that attempts to execute a lock-protected critical section as an atomic hardware transaction, reverting to using the lock if these attempts fail.

Figure 1 depicts our solution. At a high level, our solution starts with using static analysis to identify candidate lock-protected critical sections to be instead protected by the HTM. Then we filter out non-desirable candidates using both static analyses (e.g., to eliminate regions containing I/O operations) and dynamic analysis (to eliminate regions where the application of the HTM would not be beneficial based on profile data collected at runtime). Finally, we rewrite the code to have candidate regions use HTM constructs provided by the HTM library we developed instead of Go locks [11]. Gocc transformations are guaranteed to be safe; developer involvement is optional but highly recommended to let developers ultimately decide whether or not they want to use HTM.

In summary, this paper makes the following contributions:

1. We present the design and implementation of a framework for identifying lock-protected critical sections and select the best candidates for lock elision based on static analysis and execution profiles of Go programs.
2. We describe the source-to-source code transformation to replace mutual-exclusion locks in Go programs with HTM concurrency control constructs.
3. We introduce a library extending vendor-provided HTM primitives with intelligent features such as runtime contention management. Specifically, we devise a lightweight perceptron [59, 84] that learns whether eliding a lock via HTM at a call site [50] is beneficial at runtime.
4. We demonstrate the effectiveness of Gocc for improving performance of real-life concurrent Go code by up to $10 \times$.

Figure 1: Gocc schematic diagram. Static analysis detects three legal lock-unlock pairs in the input file example.go. The top one is a valid replacement candidate. The middle one is filtered since it contains I/O operations in its critical section. The bottom one is dropped due to the infrequent use via the information provided by profiles. The transformed code calls optiLib, which executes the critical section via HTM. The resulting diff is given to the developer for review.

2 Challenges

Locks are widely used in the real-world Go code and a significant amount of execution time can be spent waiting to acquire them [15, 19, 27, 28, 68, 86, 89]. It is possible to replace a lock with a transaction that enables a critical section to be speculatively executed without actually holding the guarding lock. With the support of the HTM, such replacements can result in significant speedups. However, there are several challenges in performing these replacements correctly and robustly, and ensuring that they deliver high performance reliably.

First, automatically and accurately matching a lock with its corresponding unlock operation to precisely identify critical sections is a complex problem. Real-world programs can use locks with nesting intra- or inter-procedurally, which makes it significantly more involved. Additionally, certain lock-compatible instructions (e.g., IO and privileged instructions) will not work with HTM. A critical section including such instructions will not benefit from HTM.

Furthermore, Go provides a keyword that enables delaying lock release operation to all exit points of a function by prefixing the Unlock() operation with the defer [49] keyword. It not only complicates matching an unlock with a lock operation, it may unnecessarily lengthen a critical section, which according to a synthetic benchmark we wrote shows performance degradation. A scan of 21 million lines of industrial Go code, which includes about 8000 Unlock() operations, shows that about 76% are prefixed with the defer keyword. This indicates that handling defer statements is important.

Second, the Go language nuances [46] (e.g., pointer vs. value syntax, anonymous Mutex fields, lambda functions, etc.) make

---

1 A limited study we performed in a large-scale industrial setting using thousands of different Go services showed up to 30% execution time being spent in lock-related code in certain Go programs; 5-10% was quite common.
2 Any function can be deferred in Go.
it non-trivial to transform lock-based code to HTM-based code.

Third, HTM has startup and commit overheads. Even in non-concurrent code, where data-access conflicts do not happen, HTM can fail [14], and locks may outperform HTM, particularly on tiny critical sections [75].

Fourth, the critical section size can be hard to estimate in general. If we make the conservative design choice and do not replace the lock if the critical section size is unknown, we can miss the opportunity to generate significant performance improvement. Thus, we need some runtime mechanism that can handle critical sections of arbitrary sizes with low overhead.

Fifth, when HTM aborts for a genuine data-access conflict, naively falling back to using a lock can be detrimental to performance [42, 64]. Deciding when and how to retry HTM-based executions or fall back to using fine-grained locks must be handled very carefully to avoid pathologies [37, 42, 64].

Our tool, Gocc, attempts to solve the above challenges. Gocc is an end-to-end system for improving the performance of lock-based Go code using HTM. We devise a sophisticated program analysis to identify lock-protected critical sections (§5.2), support lock-to-HTM code transformation including non-trivial Go features (§5.3), and develop an efficient HTM library to handle issues manifested at runtime (§5.4).

3 Related Work

Herlihy and Moss proposed transactional Memory (TM) [53] in 1993 as an alternative to locks. While locks proactively prevent two or more threads from concurrently accessing shared data, TM takes the opposite approach — concurrent accesses are allowed as long as they do not conflict. A lot of work has been done around both software and hardware implementations of transactional memory [41, 43, 71, 77, 82, 90, 92], but only a few [61, 76, 93, 97] focused on evaluating the approach with real-life workloads, and none have done this for Go.

Intel’s TSX extension of x86 instructions set [5] implementing HTM is of specific interest here as it underlies parts of our implementation. It is widely available in modern Intel CPUs and offers software interfaces providing subtly different functionality. The RTM (Restricted Transactional Memory) interface allows programmers to execute arbitrary code as a hardware transaction. All operations within a transaction have atomic execution behavior — they all either appear to happen instantaneously or the entire transaction aborts and reverts the architectural state to before it was started. This can be trivially used to emulate the behavior of mutual-exclusion locks. In fact, this is precisely the kind of functionality that the HLE (Hardware Lock Elision) interface provides. However, HLE has been introduced mainly for backward compatibility with architectures that are not TSX-enabled and is not only very simplistic (e.g., with respect to contention management) but has also been shown to perform poorly compared to RTM [1]. Consequently, our solution uses the RTM interface as the low-level implementation mechanism to build a comprehensive TM-based alternative for mutual-exclusion locks.

Lock elision, whether in software or hardware or a hybrid fashion, including gaining insights into them, has been extensively studied [22, 29, 34, 36–39, 44, 45, 57, 58, 63, 71, 72, 74, 81, 91, 92, 95]. Our work uses many of those techniques; for example, the basic design of our runtime controller was inspired by Wang et al. [91]. Additional possibilities to bring more solutions from the literature to the design and implementation of both our static analysis tool and runtime controller also exist. Other attempts to use transactional memory for emulating mutual-exclusion locks exist as well [70, 96], but they have to cope with higher overheads and semantics-related complications due to using the STM, they target the Java language whose synchronization lock-like primitives (i.e., monitors) are easier to handle due to their lexical scoping and, most importantly, their evaluation is based exclusively on synthetic benchmarks.

4 Gocc Overview

A Go Mutex is a runtime object with Lock() and Unlock() operations on it. Two (or more) critical sections guarded by the same Mutex will not execute concurrently. When transforming locks into HTM, there are two possibilities.

1. A given Mutex guarding a set of critical sections is replaced with another object supporting operations analogous to Lock() / Unlock() but provided by the HTM. As a result, all critical sections previously guarded by the Mutex are now executed under HTM’s control.

2. Lock() / Unlock() operations of the Mutex are replaced with their HTM equivalents on a per critical section basis. As a result, some critical sections for a given Mutex are still guarded by the same Mutex, while the others execute under HTM’s control.

The former is doable only if it is beneficial to transform all Lock() / Unlock() operations using a given Mutex, and the Mutex object is defined in the code that we are rewriting. Assessing the benefit of transforming the Mutex object would require inspecting every critical section it protects. A “may alias” pointer analysis [55, 65] can answer such a question. The “all-or-none” coarse-granularity of this approach makes it unattractive because the imprecision of pointer analysis overapproximates the critical sections protected by a Mutex, disqualifying too many Mutexes from transformation.

This work adopts the latter approach, where we consider pairs of Lock() / Unlock() operations in the code for transformation, which provides fine-grained control over transformation. This approach has to handle pairing a lock with its corresponding unlock and support interoperability of HTM (where the code is transformed) with locks (where the code is not transformed). This kind of interoperability is well-studied in the literature [23, 32, 40, 41, 64] and is handled by our library.

Recall, from Figure 1, that input to Gocc is the source code for a Go package along with its execution profiles. The output
is a source code patch, where candidate \texttt{Lock()}/\texttt{Unlock()} operations are replaced with calls to a custom HTM library. \textsc{Gocc} consists of the following key components:

- Analyzer: performs static analysis on the input program and collects lock-unlock pairs for transformation (§5.2).
- Transformer: rewrites the program by replacing \texttt{Lock()}/\texttt{Unlock()} with \texttt{FastLock()}/\texttt{FastUnlock()}, which elide the lock using HTM (§5.3).
- Adaptive runtime (\texttt{optiLib}): implements HTM in Go and provides required runtime mechanisms including retry and rollback (§5.4).

The source code patch choice, rather than a compiler transformation, is motivated by the desire to keep the developers in the loop. Using HTM without developers’ knowledge can prove unwelcome because developers often demand full visibility into their programs. Developers are becoming performance and variance sensitive [56, 67, 83], and an accidental regression can become hard to diagnose. As a side effect, the choice of source-code patch demands us to be surgical — injecting large, complicated HTM-handling boilerplate code is a non-starter. Consequently, we perform \texttt{Lock()}/\texttt{Unlock()} operations replacements with API calls to HTM logic hidden in the \texttt{optiLib} open-source library and do so only in places where benefits of HTM are likely (e.g., we minimize the number of modified code locations using execution profiles).

## 4.1 \textsc{Gocc} Guarantees and Limitations

- \textsc{Gocc} will transform properly synchronized code (i.e., where every lock operation will have a corresponding unlock operation) into the equivalent code without changing the code’s behavior. Code not meeting this criterion will be either not transformed, or transformed and its runtime behavior will be unchanged.
- \textsc{Gocc} considers only those lock-unlock pairs that seem to operate on the same lock within the same function — inter-procedural \texttt{Lock()}/\texttt{Unlock()} operations are disregarded. Note, however, that in a critical section protected by \textsc{Gocc} transformed lock can make arbitrary function calls. The requirement to have both \texttt{Lock()} and its matching \texttt{Unlock()} operation be present in the same procedure scope is only our implementation choice and pragmatic in nature. Over 70% of the locks we inspected met this criterion.
- \textsc{Gocc} makes no effort to identify critical sections or code reachability in the presence of reflection [10].
- \textsc{Gocc}, as implemented, does not \textit{statically} detect HTM conflicts or capacity limitations (see §5.2 for the details).

## 5 \textsc{Gocc} Design and Implementation

Before diving into the details of \textsc{Gocc}s design and implementation, we define some common terminology.

## 5.1 Terminology

Go’s \texttt{sync} package provides two kinds of shared memory objects: \texttt{Mutex} and \texttt{RWMutex}. \textsc{Gocc} handles them both, but in the following sections, without the loss of generality, we will only use the term \texttt{Mutex} for simplicity. From an HTM transformation viewpoint, an \texttt{RWMutex} is no different from a \texttt{Mutex}, except \texttt{RWMutex} offers additional APIs for read-only accesses.

A \textit{critical section} \textsc{CS} is all code regions protected by a pair of lock and unlock operations on the same mutex object \texttt{m} — the notation for calling lock/unlock operations on \texttt{m} is \texttt{m.Lock()}/\texttt{m.Unlock()} where \texttt{m} is referred to as a \textit{receiver}. \textit{Lock-point}, abbreviated with letter \texttt{L} (\texttt{Unlock-point} abbreviated with letter \texttt{U}), is a static location in the code where the \texttt{Lock()} (\texttt{Unlock()}) function is invoked on a \texttt{Mutex}.

\textit{LU-points} is a set of \texttt{L} and \texttt{U} points. \textit{LU-pair} is a candidate pair of one \textit{lock-point} paired with an \textit{unlock-point}. In the runtime context, fastpath/HTM-path means the use of HTM, and slowpath/fallback-path means the use of the original lock.

We utilize the Abstract Syntax Tree (AST), program Control Flow Graph [85] (CFG), and Static Single Assignment (SSA) [35] form of program representation prevalent in the compiler literature. In a CFG, nodes are basic blocks [85] of straight-line code, and edges are control flow relationships among them. \textsc{Gocc} first transforms the source code to the AST form (which is also used for code transformation as described in §5.3) and then to the SSA form for CFG construction.

## 5.2 Analyzer

The goal of the analyzer is to find as many LU-pairs as possible. The LU-pairs that protect HTM-incompatible critical sections (e.g., those including IO operations) must be pruned. This filtering serves two purposes: it reduces the number of code changes and non-beneficial HTM transformations. Complicated lock usage patterns, several Go language quirks, and pointer imprecision complicate the static analysis. A comprehensive call-graph analysis is vital because critical sections often contain function calls.

\textbf{Conflicts:} A sophisticated static analysis may detect whether transactions conflict. Answering this question, however, is unlikely to be valuable because developers typically do not use a lock if a conflict is impossible. Assuming conflicts happen, there is no easy way to statically determine whether transactions do not “typically” conflict. We do not try to solve this problem and leave conflict resolution to \texttt{optiLib}.

\textbf{Capacity:} Although one can perform static analysis to estimate the memory footprint of a critical section, it may not be possible if the bounds of a loop are unknown. Also, without knowing the target architecture’s HTM capacity, it would be premature to filter out candidate critical sections this way. We leave the capacity-related decisions also to \texttt{optiLib}.

In the rest of this section, we, first, define the scope of our transformation (§5.2.2); then, describe the process of
matching a lock with an unlock operation within a code region assuming no lock nesting and no function calls in a CS; extend our analysis to include nested locks (§ 5.2.3); expand the analysis scope to CSs that may contain function calls (§ 5.2.4); detail special case of Go’s defer statement (§ 5.2.5); and finally discuss profile-based filtering (§ 5.2.6).

5.2.1 Scope of Transformation

To simplify the analysis, if a \texttt{Lock()}/\texttt{Unlock()} operation is executed in the middle of a basic block, we break such basic blocks in the CFG so that each lock-point begins a new basic block and each unlock-point ends a basic block. A single-entry single-exit (SESE) region [60] (simply region) of a CFG is our smallest granularity of lock transformation. A region is a sub-graph of a CFG. Control reaching any basic block in a region is guaranteed to have already executed a designated entry basic block; control leaving from any basic block in the region is guaranteed to eventually pass through a designated exit basic block.

A function is the largest granularity of our lock transformation; a function always forms a region because all exits from a function are considered to go through a dummy basic block.

Regions can be nested within one another. A Program Structure Tree (PST) organizes regions into a hierarchical tree [60]. We visit regions inside out from most-nested to least-nested. Appendix B in the extended version of this paper [94] describes the region identification and visiting strategies, which are not central to this paper.

5.2.2 Matching LU-pair in the Absence of Nested Locks

This subsection discusses analyzing a candidate region \( R \).

LU-points in \( R \) may be operating on different locks, which should be pruned. Some lock (unlock) operations may escape \( R \), without a corresponding unlock (lock) operation in \( R \), which should also be pruned. Below, we formalize these aspects.

Definition 5.1 (Points-to set \( M(L) \) of a Lock point \( L \)). Every lock-point \( L \) operates on some receiver mutex pointer \( p \).\(^3\) Such a mutex pointer may point to one or more mutex objects allocated in the program. The set of all possible Mutex objects that \( p \) may point to in the program is the Points-to Set of \( L \), denoted by \( M(L) \).

\(^3\)At the source level \( p \) can be either a pointer or an actual object value, but at the SSA level it is always a pointer.

Similarly, the Points-to set of an Unlock point \( U \) is \( M(U) \). We employ Anderson’s flow-insensitive may-alias analysis [25] to obtain \( M(L) \) and \( M(U) \) on the whole program.

Definition 5.2 (Downward Exposed Lock-point (DELock)). A lock-point, \( L \), with points-to set \( M(L) \), is downward exposed in region \( R \), if there exists at least one path from \( L \) to \( R \)’s exit without any unlock-point on any mutex in \( M(L) \).

Definition 5.3 (Upward Exposed Unlock-point (UEUnlock)). An unlock-point, \( U \), with points-to set \( M(U) \), is upward exposed in region \( R \), if there exists at least one path from \( R \)’s entry to \( U \) without a lock-point on any mutex in \( M(U) \).

DELock identifies lock-points that definitely do not have any corresponding unlock-points in some execution paths in \( R \); and UEUnlock identifies unlock-points that definitely do not have corresponding lock-points in some execution paths in \( R \).

Figure 2 exemplifies a downward exposed lock-point. Mutex pointer \( a \)’s points-to set \( \{m\} \), has an empty intersection with \( b \)’s points to set \( \{p\} \); although it has a non-empty intersection with \( c \)’s points-to set \( \{n\} \). Figure 3 exemplifies an unlock-point that is not upward exposed. Mutex pointer \( c \)’s points-to set \( \{m\} \), has non-empty intersection with \( a \)’s points-to set \( \{m,n\} \) and \( b \)’s points-to set \( \{m,k\} \).

We eliminate all DELock(\( R \)) and UEUnlock(\( R \)) from the transformation in \( R \). The remaining lock-points in \( R \) are the complement of DELock(\( R \)), which is denoted by DELock(\( R \)). Similarly, the remaining unlock-points in \( R \) are the complement of UEUnlock(\( R \)), which is denoted by UEUnlock(\( R \)).

Definition 5.4 (Feasible-HTM-Pair). Let \( L \in \text{DELock}(\overline{R}) \). Let \( U \in \text{UEUnlock}(\overline{R}) \). \( L \) and \( U \) form a feasible HTM pair if all of the following conditions are true,
(1) \( M(L) \cap M(U) \neq \emptyset \),
(2) \((L \text{ DOM } U) \cup (U \text{ PDOM } L)\),
(3) The critical section \( C \subseteq R \) guarded by \( L \) and \( U \) contains no LU-point \( X \) such that \( M(X) \cap (M(L) \cup M(U)) \neq \emptyset \), and
(4) \( C \) contains no HTM-unfriendly instructions.

Condition (1) filters out those LU-points that are guaranteed to be operating on different Mutexes.

Condition (2) filters out infeasible control flows where unlock happens before lock and vice-versa. DOM and PDOM respectively represent dominator \([85]\) and post-dominator \([85]\) relationships in a CFG. Figure 4 shows an example, where all paths from lock-point \( a.\text{Lock()} \) are post-dominated by unlock-point \( b.\text{Unlock()} \), whose all incoming paths are dominated by \( a.\text{Lock()} \). Additionally, the set-intersection of the points-to set of mutex pointers \( a = \{m.p\} \) and \( b = \{m.n\} \) is non-empty. Any Feasible-HTM-Pair on \( L \) and \( U \), forms an SESE-region by itself, where the entry basic block has \( L \) as its first instruction and the exit basic block has \( U \) as its last instruction. Condition (2) intuitively finds correct candidate LU-pairs in the absence of nested locks because if a lock operation \( L \) is performed on every path reaching any code in \( C \) and an unlock operation \( U \) is performed on every path exiting \( C \), then \( L \cup U \) must be operating on the same Mutex. Appendix A in the extended version of this paper \([94]\) justifies our choice of DOM/POD domination relationships.

Condition (3) ensures that if we match an \( L \) with a \( U \), there does not exist another lock-point or unlock-point in the same region that may operate on a Mutex in the same points-to set as that of \( L \) or \( U \). The next subsection elaborates on lock nesting.

Condition (4) is an obvious requirement to ensure HTM does not abort. A region is unsafe if it contains any IO instructions.

Since we use “may alias” to match a lock-point with unlock-point, it is possible (but less likely) for our transformation to pair a lock with an unlock that may be operating on two different mutex objects at runtime. However, at runtime, we can obtain and memorize the address of the mutex object used at the lock-point, and compare it against the mutex object used at the runtime of the unlock-point. In case of an address mismatch of the mutex objects accessed in the same LU-pair, we can abort the transaction and revert to a safe state and fall back to using the lock. A mismatch is impossible without nested locks because of the dominance and post-dominance relationship between the lock and unlock in an LU-pair.

### 5.2.3 Lock Nesting

Go supports nested locks, but reentrant \([13]\) locks are not allowed. Condition (3) in Definition 5.4 disqualifies a candidate LU-pair from the transformation in region \( R \) if there exists any other lock or unlock point whose lock/unlock operation may be operating on the same mutex as those in the LU-pair.

As an example, in Listing 3, assume the mutex pointers \( a \) and \( b \) point to the same points-to set. When inspecting the “inner region”, we find only one LU-pair, which obeys all Feasible-HTM-Pair conditions in Definition 5.4. Consequently, the lock usage on \( b \) in the inner region can be transformed to HTM. When inspecting the “outer region”, however, we see conflicting LU-points, and hence the locking operations on \( a \) will not be transformed. The resulting transformed code is shown in Listing 4, which is correct.

This approach complicates hand-over-hand locking \([33, 62]\), sometimes used in the concurrent linked-list traversal, shown in Listing 5. As before, assume all four LU-points have a non-empty intersection of their points-to sets. When inspecting the inner region, the LU-pair \( b.\text{Lock()} \) and \( a.\text{Unlock()} \) passes all tests in Definition 5.4. Hence, they will be, incorrectly, paired and transformed to use HTM, as shown in Listing 6. This transformation violates the programmer’s intention. Subsequently, when visiting the outer region, condition (3) is violated, and hence the outer LU-pair will not be transformed. One could have discarded the transformation of the inner region when the conflict is visible in the enclosing region. However, we cannot distinguish this incorrect pairing from the correct pairing in the previous case. Our solution is to always apply the transformations on the candidates found in inner regions, and handle mismatches at runtime via HTM aborts if executing on the fastpath. As mentioned at the end of § 5.2.2, a mismatch is easy to recognize at runtime by, first, making \( \text{FastLock()} \) store the address of the Mutex used at the lock-point in a field in \( \text{OptiLock} \) and, second, checking whether the Mutex passed to \( \text{FastUnlock()} \) is the one present in \( \text{OptiLock} \). The transactional abort is needed (and possible) only on the fastpath. Appendix C in the extended version of this paper \([94]\) details the correctness of transforming nested locks into HTM via Gocc.
5.2.4 Critical Sections with Function Calls

When the critical section protected by a candidate LU-pair contains function calls, we need to extend the analysis beyond the current function. Conditions (1) and (2) in Definition 5.4 are local to \( R \). Conditions (3) and (4) require inter-procedural analysis.

We need to ensure that the transitive-closure of all code regions protected by a candidate LU-pair, including the blocks reachable via function calls, neither contains any HTM-unfriendly instructions nor contains any LU-points whose points-to set may overlap with the points-to sets of \( L \) or \( U \). Otherwise, we discard the candidate LU-pair.

To accomplish this, we first build a static call graph using rapid type analysis [7, 26]. Next, we precompute summary information for each function on its own without its transitive closure of function calls; the summary contains (a) the fit of the function for HTM-based-execution (i.e., no HTM-unfriendly instructions), and (b) the union of all points-to sets of all LU-points in the function, denoted by \( \mathcal{P} \).

For a candidate LU-pair meeting all the conditions in Definition 5.4 within the region \( R \), we proceed to do inter-procedural analysis. Let \( F^* \) be the transitive closure of all the function calls invoked inside the critical section \( C \subseteq R \) protected by a candidate LU-pair. LU-pair is discarded if (a) \( \exists F' \in F^* \text{s.t. } F' \text{’s summary contains HTM-unfriendly instructions or } (b) \exists F' \in F^* s.t. \mathcal{P} \cup (\mathcal{M}(L) \cup \mathcal{M}(U)) \neq \emptyset \). The former is simply the condition (4) expanded to all functions, and the latter is condition (4) expanded to all functions. We note that nested locks discussed in § 5.2.3 can be in different functions.

5.2.5 The defer Statement

The defer [49] statement in Go, introduced in § 2, needs special attention. Go defers the execution of functions prefixed with the defer keyword to the calling function’s return point. The presence of defer Unlock() complicates our CFG-based dominance/post-dominance analysis. Deferred unlocks extend the critical sections till function exit points. Listing 7 shows a legal Go code, where the defer m.Unlock() appears even before the call to m.Lock(). Condition (2) in Feasible-HTM-Pair will treat this pair as an invalid candidate for transformation because the lock-point does not dominate the unlock-point.

We address this case by interpreting defer m.Unlock() in a CFG by (a) introducing a synthetic m.Unlock() statement at the end of each basic block that returns control from the function, and (b) discarding the presence of m.Unlock() in its original position during the analysis. This allows us to reuse the previously described dominance relationship. During transformation, however, GoCC simply replaces a defer Unlock() with a defer FastUnlock() in its original place, as shown in Listing 8.

Multiple defer calls are executed in a last-in first-out (LIFO) order of encountering the defer statement at runtime. This complicates the dominance and post dominance relationship; for simplicity, we discard any function that contains multiple defer Unlock() statements. We found none in the packages used in our evaluation.

5.2.6 Profiles to Filter Hot Critical Sections

Profiling is a built-in feature in Go, which takes callstack samples via timer [48] or hardware performance counter [30] interrupts. One can take CPU profiles of a go program either at launch time by simply passing a -cpuprofile flag or from an already running program, for a specified duration, by accessing an exposed profiling port.

Go profiles are in the pprof format. The pprof Go package [48] allows us to programmatically navigate the callstack samples presented as weighted call graphs, where the nodes represent functions and edges represent caller-callee relationships. The functions are annotated with their inclusive and exclusive execution times.

When profiles are available, we use them to filter the regions where negligible execution time is spent, even before applying the static analysis. In fact, this is the first filtering step we perform before making the aforementioned LU-pair identification. We treat any critical section (including the entry and exit) where the aggregated execution time is less than 1% of the total execution time as insignificant.

5.3 Transformer

Since our end product is a code patch, we perform the transformation on the AST form of the program. Go AST can be serialized into source code via Go format [8] package. To this end, the transformer maps the candidate set of LU-pair operations found during the SSA-based analysis phase (described in § 5.2) to AST nodes [6]. It then replaces the LU-pair operations with calls to FastLock()/FastUnlock() in optILib. It also passes the original Mutex object as a pointer to the calls to FastLock() / FastUnlock() since the underlying lock object is necessary for lock elision (fastpath) as well as for slowpath. Figure 5 shows an example AST transformation. The transformation itself is mechanical but challenging. In the following sections, we discuss several Go features that pose special challenges in transforming the AST.

Go pointer vs. value: Go syntax does not distinguish accessing a field of a composite type (e.g., struct) via an object-pointer or an object-value; both use the same

Listing 7: defer Unlock.

Listing 8: HTMized.
Go anonymous fields: Go allows programmers to define a struct that has fields without names. For instance, Listing 11 shows a struct `AStruct` that has an anonymous `*sync.Mutex` field. Operations on this anonymous mutex are performed by simply using the name of the enclosing struct variable as shown on Line 8. Hence, our transformation needs to be cognizant about whether the original LU-pair operations are performed on an anonymous mutex.

By inspecting the type information of the access path used to invoke the lock/unlock operation in the AST, we determine whether or not the operation is performed on an anonymous mutex field. Upon determining the operations to be on an anonymous mutex, we pass the address of the anonymous `Mutex` to `optiLib` by simply suffixing the operation access path with `Mutex` as shown in Listing 12, starting from Line 8 (where access path simply consists of variable `a`). This transformation composes with the previously described pointer vs. value operations.

Anonymous goroutines: Go supports anonymous goroutines [2], which are nested inside other functions as shown in Listing 13; these goroutines run concurrently. Our transformation introduces a new `OptiLock` variable in transformed functions. `OptiLock`’s declaration should be outside the scope that encloses both Lock and Unlock operations, as well as its placement are carefully planned to minimize any overhead. It is responsible for detecting and recovering from the mistake. Finally, it is responsible for understanding and dynamically adjusting to changing contention.

We implement `optiLib` using TSX [5] for Intel platforms. `optiLib` is carefully implemented to ensure correctness under all circumstances. Equally important, it is implemented with the utmost attention to performance. Every instruction and its placement are carefully planned to minimize any overhead of its own. `optiLib` uses Intel RTM: it does not use the Hardware Lock Elision (HLE) [1] because it does not provide

---

Listing 9: Both Mutex pointer

```go
Listing 10: GOCC transforms and Mutex value to invoke the lock/unlock operation in the AST, and pass it to FastLock() and FastUnlock().

OptiLock

Figure 5: Example of AST transformation from m.Lock() to optiLib.FastLock(m). Some AST nodes are omitted for brevity.

```go

Listing 11: Locking on an unnamed field of a struct.

```go
Listing 12: Unnamed mutex transformed to HTM.

```go

Listing 13: Anonymous goroutine

```go
Listing 14: The OptiLock outlines create concurrent units needed for the transformation of execution on anonymous should be placed in the functions. innermost function scope.

---

5.4 Adaptive HTM Runtime: `optiLib`

`optiLib` implements all the intelligent runtime control needed to perform HTM in lieu of locks. It is in charge of starting and committing transactions in critical sections, as well as falling back to the LSE when necessary. It is responsible for inter-operating with locking operations on the same mutex that may not be transformed to use HTM. In the event of aborts, it is responsible for determining the cause of the abort and deciding whether and how many times to retry. If, accidentally, the code rewriting matches lock-point with a programmer-unintended unlock-point, `optiLib` is responsible for detecting and recovering from the mistake. Finally, it is responsible for understanding and dynamically adjusting to changing contention.

We implement `optiLib` using TSX [5] for Intel platforms. `optiLib` is carefully implemented to ensure correctness under all circumstances. Equally important, it is implemented with the utmost attention to performance. Every instruction and its placement are carefully planned to minimize any overhead of its own. `optiLib` uses Intel RTM: it does not use the Hardware Lock Elision (HLE) [1] because it does not provide...
the fine-grained control we need.

optiLib introduces a data structure: OptiLock, which has two fields: a boolean slowPath and a *sync.Mutex lkMutex. slowPath is set to true if the lock is not elided at runtime. lkMutex always holds the address of the fine-grained lock being elided. OptiLock supports FastLock()/FastUnlock() operations, both need a *sync.Mutex argument, which is the mutex receiver object being elided at the original call sites of Lock()/Unlock(). Try locks and timed locks [31, 78] are absent in Go; it is trivial to support them in optiLib.

The FastLock() implementation uses sophisticated mechanisms described previously by others [23, 32, 64] to interoperate slow and fast paths concurrently. Stated succinctly, the FastLock(), waits for the fine-grained lock to be available before starting the hardware transaction; on starting a transaction, it first checks if the fine-grained lock is already held and unconditionally aborts if it is already held; if it is not held, the act of checking adds the lock word to the transaction read-set, and hence, if a concurrent execution on the slowpath acquires the same lock during the transaction, the fastpath immediately aborts ensuring mutual exclusion. Any two threads in the fastpath can run concurrently as long as they do not conflict in their memory accesses.

Reading the internals of the original sync.Mutex object is straightforward and has no performance penalty; FastLock() simply de-references the first word of the Mutex pointer passed into the function, which contains the lock status.

The FastUnlock(), based on slowPath value, either commits the transaction or invokes the unlock on the mutex object. It also safeguards against accidental incorrect code patches by ensuring that the mutex object passed into FastLock() and stored in the lkMutex field of OptiLock matches the mutex object presented to FastUnlock(). In case of a mismatch, FastUnlock() restores safety by aborting the transaction (iff on fastpath), and subsequently enforces the slowpath.

### 5.4.1 Dynamic adjustment via perceptron

It would not be fruitful to attempt HTM if doing so has already proved to be unsuccessful for whatever reason. GOcc learns and adapts to HTM behavior and decides whether to use HTM for the already transformed LU-points, the fallback being the original lock. For this purpose, GOcc uses a featherlight, hardware-inspired “hashed perceptron” [84].

The hashed perceptron predictor hashes feature weights into one or more tables. Then at the prediction time, it uses indexes to access feature weights from the tables and adds up all the relevant weights. If the sum exceeds a threshold, the prediction will be regarded as positive (e.g., “HTM should be taken”). Otherwise, the result will be viewed as negative (e.g., “HTM should not be used”). The weights will be updated based on the correctness of the predictions.

If operations on a given Mutex have been HTM-friendly/unfriendly, we want to utilize this information. Similarly, if a code location has been HTM-friendly/unfriendly irrespective of the Mutex used, we want to use this information as well. Hence, the two input features for the perceptron are the Mutex and the calling context [24, 50] of lock/unlock invocation. The address of the Mutex serves as the Mutex feature, and the address of OptiLock serves as a unique identifier for the calling context feature. Updating the same perceptron weight for the Mutex feature by different goroutines would create a conflict (and potentially a performance collapse). Hence, we instead XOR the Mutex address with the address of the OptiLock to produce a conflict-free feature input.

Our perceptron implementation creates two 4K-entry arrays as the global weight tables (GWT). The weights take an integer number ranging from -16 to 15. At runtime, FastLock() and FastUnlock() functions index into GWT by taking the lower-12 bits of the two features. Perceptron operations are done outside the transaction. The updates and reads from GWT are lock-free but racy — perfection is not required here, but high-performance is necessary. Experiment results from § 6.2 show the effectiveness of perceptron learning in protecting against poor HTM performance.

**Perceptron weight update:** Perceptron weight updates happen in the FastUnlock() function after successfully finishing the critical section, whether on fast and slow path.

If the perceptron decides that the lock should be used, there will be no update to the weights as the lock will always succeed. When the perceptron indicates to use the HTM and the execution finishes on the fastpath, the corresponding weight in the cell will be increased (because the perceptron makes a correct decision, it should be encouraged to use the HTM more frequently). On the other hand, if perceptron determines to use HTM, but HTM fails and falls back to slowpath, the weights will be decreased (because HTM does not work for the current call, perceptrons should be penalized for incorrect recommendation to improve future predictions).

**Weight decay:** We keep a counter, in each cell in GWT, to record the number of lock calls that go to the slowpath directly as a result of perceptron decision. If a lock has been used consecutively for a certain number of times and exceeds the threshold, we reset the weight of the perceptron cell and subsequently try HTM. Without this reset, perceptron would get stuck on the slowpath, preventing the benefits of the HTM execution in the future. We set this threshold to 1000 continuous decisions. Appendix D in the extended version of this paper [94] summarizes our FastLock()/FastUnlock() implementations including the perceptron logic.

### 5.4.2 Alleviating HTM overhead

HTM brings overhead for very short critical sections as described in § 2 above, even under single-core execution. optiLib avoids using HTM if it recognizes a single OS-thread in a Go process. optiLib employs runtime.GOMAXPROCS(0) API for this purpose.
6 GOCC Evaluation

We evaluate GOCC on an 8-core (×2-way SMT [87]) Intel Coffee Lake CPU with a total 32GB memory, running Linux 5.4.0. The CPU has 32KB L1I and L1D cache, 256KB L2 cache, and 16MB L3 cache. The Go version is 1.15.2.

Table 1 shows the list of applications and libraries we employ. In the absence of standard benchmarking for Go, we selected packages that are popular open-source Go projects (column 2 in Table 1), focus on high performance, utilize lock-based Go concurrency, and provide thread-safe APIs. In particular, Zap and Tally are foundational logging and metrics collection packages used in production Go programs by many organizations. Additionally, since we evaluate the projects using their own benchmark suites (more on this below), we only selected projects that feature concurrent benchmarks or whose benchmarks could be straightforwardly converted to be concurrent.

From a static analysis viewpoint, we see that all applications contain several locks. Defer unlocks are common (column 7). The “violates dominance” column shows how many LU-points were discarded since they did not meet the dominance relationship. The number is typically low except for go-cache, which has several functions with the repeating pattern of unlocks that do not post-dominate the candidate lock. The “candidate pairs” column shows how many LU-points remain for further analysis. Each column to the right progressively shows the reasons for which a candidate LU-pair was rejected. Rejection due to nested aliased locks is not found in these packages. The second-to-last column shows how many LU-pairs were finally rewritten to use OptiLock, including how many of them contain defer Unlock()s. The last column shows the numbers after we retain only those locks where the functions contain at least 1% of execution time in execution profiles. Overall, GOCC transforms several LU-pairs in each application. Using profiles significantly reduces the number of transformed LU-pairs.

We run all the benchmarks within each repository five times and report the median. We believe the benchmarks accompanying the code best represent its desired characteristics. As some benchmarks are written for a single thread setting, we rewrite them to introduce concurrency to utilize HTM-enabled parallelism fully. We adopt the standard testing package from Go [16], which runs each benchmark for a certain amount of time and reports the throughput as nanoseconds per operation. We wrap the benchmark codes with RunParallel [4] helper function to get parallel performance if it was not already done so. Using more CPU cores, ideally, increases throughput (i.e., reduces average nanoseconds per operation). Then we compare the throughput from locks vs. HTM — a positive percentage means GOCC’s rewrite did better, and a negative percentage means the baseline did better. We vary the number of CPU cores available for benchmarks from 1 to 8. Unlike HPC codes that run on all cores on a server, Go services often use 2–4 cores.

6.1 Results on Popular Go Programs

We categorize the benchmarks in each package into two groups:

1. Concurrency non-sensitive benchmarks either have no locks or do not spend much time in critical sections, or our transformation does not result in any performance difference. For these benchmarks, we only show the aggregate (geomean) results unless noted otherwise. They appear as “non sensitive” in our charts, and the number in the parenthesis indicates how many benchmarks are in this group.

2. Concurrency sensitive benchmarks exercise modified locks non-trivially. We might have impacted them positively or negatively. For these, we present data from each benchmark and also present an aggregate result (“sensitive” in our charts).

The “all” part of our charts is the geometric mean taken over all benchmarks. Sometimes this number looks small because of a large number of non-sensitive benchmarks.

In what follows, we provide details of performance evaluation on the aforementioned Go packages. The total number of benchmarks is large; hence, we dive deep only into benchmarks with surprisingly good speedups.

Tally [88] is a fast, buffered stats collection library and Figure 6 shows its results. For the HistogramExisiting benchmark, GOCC achieves more than 660% speedup on 8 cores reducing the original time per operation from 65 ns/ops down to around 8.47 ns/ops at 8 cores. Moreover, the HTM delivers scalable performance. This benchmark uses a Mutex lock on a read-only Exists operation, and hence, is a natural candidate to demonstrate speedup as HTM eliminates the unnecessary serialization. Conversely, the baseline has a scalability collapse, where the time per operation increases from 20.4 ns/ops to 65 ns/ops for 1 to 8 cores. ScopeReporting1 holds three independent RMutexes at different points in time and accesses read-
only data. However, since the RWMutex also involves a counter increment and a decrement, its overhead as a result of cache invalidation does not scale well. Thus, even eliding RWMutex proves highly beneficial. The speedup for ScopeReporting10 is lower than that for ScopeReporting1 because it performs 10x more work inside the critical section. Overall, in the sensitive group, we see a 10% performance drop with a single CPU but 145%, 283%, and 267% improvements with 2, 4, and 8 CPUs, respectively. In the non-sensitive group, the overall performance drop is within the margin of error. Among all the 27 benchmarks of tally, we see up to 18.7% speedup at 8-CPU.

go-cache [12] is an in-memory key-value store. It contains benchmarks that exercise repeatedly accessing the same item in a small map. The benchmarks contain both non-cached accesses, similar to how go programmers often use a map, and cached accesses provided by the go-cache layer to demonstrate the effectiveness of the library. All benchmarks employ RWMutexes for concurrent map read access. Unlike the rest of the use cases, the benchmark files themselves contain locks, which GOCCL transforms into using HTM.

Figure 6 shows our empirical results. GOCCL speeds up four benchmarks in go-cache that were directly accessing the map without the library-provided cache. In each case, we can see more than 100% speedup; the biggest speedup is 742%. The speedups come from eliminating contended atomic operations involved in entering and exiting from a reader lock. The performance scales well with increased parallelism because while the lock-based approach incurs more and more contention, the HTM approach remains conflict-free throughout. The other benchmarks, the majority of which employ the go-cache, are mildly improved, but more importantly, they were not degraded as a result of transformation via GOCCL.

go-datastructures [21] is a collection of performant, thread-safe data structures. We apply GOCCL on the set subdirectory, which contains concurrency benchmarks. The results are shown in Figure 8. The Len benchmark computes the length of the set, and it is sped up by ~1000% in the 8 cores setting. Len has a short critical section that has a higher entry and exit cost due to atomic operations when using RWMutex. HTM performance shows scalability since the HTM version remains conflict-free, whereas the lock-based version collapses with increased contention. The Exists benchmark is similar to Len, where each goroutine searches one item in a set containing only one item. It scales almost as well as Len, but more work is done in the critical section, which amortizes RWMutex’s overhead, and slightly reduces HTM’s advantage. The Flatten benchmark reads 50 elements from a shared map into a private array, with a layer of caching that eliminates repeated map scanning. It holds a Mutex to serialize concurrent accesses to the map/cache. The HTM version avoids the serialization and shows scalable performance for 1-4 cores. At 8 cores, the number of conflicts resulting from updating the cache rises, which makes perceptron not use the HTM, and hence there is no speedup. The Clear benchmark has true conflicts, and there is no speedup, but the HTM does not significantly degrade the performance. Overall, utilizing GOCCL results in more than 100% geomean performance gain while introducing less than 4% slowdown in a single core setting.

Zap [17] is a library that implements fast and structured logging in Go. Being a logging library, it has several IO operations,
and hence GOCC rewrote fewer locks. Compared with other repositories, the improvement on zap is relatively mild. Due to arguably mild speedups on zap, a large number of benchmarks, we omit a deeper analysis of zap results. Slowdowns are rare, the biggest being 7%. Overall, we observed a mild ~ 4% geometric mean speedup with the best case 28% speedup.

**Fastcache** [18] is a fast, scalable, in-memory cache. The transformed code delivers a maximum of 35.60% speedup and a geomean of 15.65% speedup across all benchmarks.

In the CacheGet benchmark, goroutines repeatedly invoke the Get function, which uses an RWMutex to protect a shared map. Get has inter-procedural nested but non-conflicting locks, all of which are transformed into HTM. Get looks up a key in the map and returns a value blob. The critical section of Get contains a few atomic add instructions, which update shared variables. Transactional conflicts on the shared atomic adds are fewer at low core numbers, and the speedup is visible; however, at larger core counts, the conflicts increase, and the speedup vanishes. Fortunately, the perceptron kicks in and avoids any performance collapse.

The CacheHas benchmark is virtually the same as CacheGet, but its critical section is shorter since it does not return a populated value buffer. Hence, the speedups are higher due to fewer conflicts, but it follows the same performance pattern as CacheGet.

In the CacheSetGet benchmark, each goroutine has two loops: the first loop repeatedly invokes Set and the second loop repeatedly invokes Get. The Set function, which inserts a key-value pair into the map, may raise a panic if certain constraints are violated. Hence, GOCC does not modify a Lock() present in Set. The Get function is already described previously.

Since all goroutines first attempt Set, where Go’s default locks are being used, the runtime recognizes it as a starved mutex and takes away the time slice of some of the goroutines. This runtime behavior reduces the number of lock contenders and, as a result, a few goroutines monopolize the lock. These goroutines quickly finish their series of Set operations and proceed into calling Get in a loop. The contention is lower on Get also since the load is now split between Get and Set with some goroutines on hold. The net effect is a high throughput for the whole benchmark.

It is worth noting that the only other benchmark which invokes the Set function is the non-sensitive benchmark CacheSet. Even though CacheSet exhibits no performance improvement, and CacheGet shows mild performance improvement, their composition in CacheSetGet leads to secondary effects causing much higher performance gain at higher core counts.

### 6.2 Perceptron Evaluation

We assess the effectiveness of perceptron using the Tally benchmarks. We compare the performance with and without the perceptron machinery. In the absence of the perceptron, we always attempt HTM. In the results presented in Figure 10, we can observe that the perceptron is effective in eliminating any performance loss. For example, CounterAllocation and SanitizedCounterAllocation are HTM-unfriendly benchmarks and cause aborts frequently. Perceptron quickly learns to move away from HTM and keeps using the slowpath. Therefore, there is minimal performance loss for the perceptron case.

Finally, we setup a synthetic benchmark — a conflict-free critical section with 1000 counter updates — to evaluate the overhead of the perceptron machinery. We measured the perceptron prediction overhead to be 0.65% and weight update overhead to be 0.73% for a total of only 1.38%.

### 7 Conclusions

GOCC is a source-to-source transformation tool to speed up lock-based pessimistic concurrency control in Go programs with Hardware Transactional Memory. GOCC combines thorough static analysis with intelligent runtime control to expose additional parallelism available in Go programs. GOCC keeps the developer in the loop, minimizes code changes via execution profiles, and targets only those critical sections that are likely to improve with HTM. The experimental results from real-world Go packages show that GOCC delivers significant (up to 10×), scalable performance for concurrent Go code that uses locks while exhibiting rare and relatively small slowdowns.

### 8 Availability

GOCC is available as an open-source tool [3].

### 9 Acknowledgement

This material is based upon work supported in part by the National Science Foundation under Grant No. 1763699, 1717779, 1563935. We thank our shepherd Michael Spear and the anonymous reviewers for their feedback.
References


[38] Dave Dice, Alex Kogan, and Yossi Lev. Refined Transactional Lock Elision. SIGPLAN Not., 51(8), February 2016.


