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

This paper is included in the Proceedings of the 2021 USENIX Annual Technical Conference.

July 14–16, 2021

978-1-939133-23-6

Open access to the Proceedings of the 2021 USENIX Annual Technical Conference is sponsored by USENIX.

Optimistic Concurrency Control for Real-world Go Programs

Zhizhou Zhang[1], Milind Chabbi[2], Adam Welc[2], and Timothy Sherwood[1]

[1]University of California, Santa Barbara
[1]{zhizhouzhang, sherwood}@cs.ucsb.edu
[2]Programming 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 interprocedural 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 perceptron, 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×) 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., STMBench7 [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×.

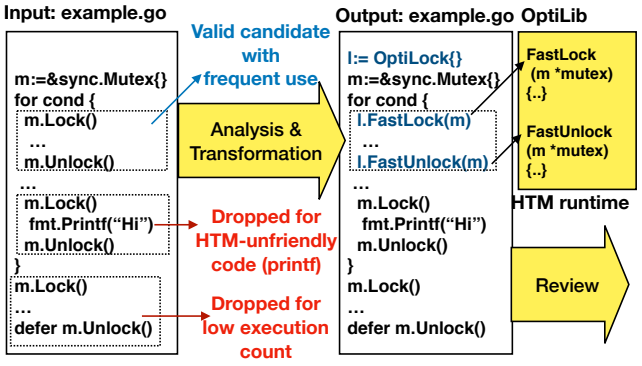

Figure 1: GOCC schematic diagram. Static analysis detects three legal lock-unlock pairs in the input file `example.go`. The top one is a valid replacement candidate. The middle one is filtered since it contains I/O operations in its critical section. The bottom one is dropped due to the infrequent use 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] [1]. 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[2]. 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, 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 `Mutex`es 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 `Lock()/Unlock()` operations are replaced with calls to a custom HTM library. 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 `Lock()/Unlock()` with `FastLock()/FastUnlock()`, which elide the lock using HTM (§ 5.3).
- Adaptive runtime (`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 `Lock()/Unlock()` operations replacements with API calls to HTM logic hidden in the `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 GOCC Guarantees and Limitations

- 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.
- GOCC considers only those lock-unlock pairs that seem to operate on the same lock within the same function — inter-procedural `Lock()/Unlock()` operations are disregarded. Note, however, that in a critical section protected by GOCC transformed lock can make arbitrary function calls. The requirement to have both `Lock()` and its matching `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.
- GOCC makes no effort to identify critical sections or code reachability in the presence of reflection [10].
- GOCC, as implemented, does not *statically* detect HTM conflicts or capacity limitations (see § 5.2 for the details).

5 GOCC Design and Implementation

Before diving into the details of GOCCs design and implementation, we define some common terminology.

5.1 Terminology

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

A *critical section* **CS** is all code regions protected by a pair of lock and unlock operations on the same mutex object `m` — the notation for calling lock/unlock operations on `m` is `m.Lock()`/`m.Unlock()` where `m` is referred to as a *receiver*. *Lock-point*, abbreviated with letter *L* (*Unlock-point* abbreviated with letter *U*), is a static location in the code where the `Lock()` (`Unlock()`) function is invoked on a `Mutex`. **LU-points** is a set of *L* and *U* points. **LU-pair** is a candidate pair of one *lock-point* paired with an *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. 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.

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 `optiLib`.

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 `optiLib`.

In the rest of this section, we, first, define the scope of our transformation (§ 5.2.2); then, describe the process of

```
1
2 m := &sync.Mutex{}
3 m.Lock()
4 m.Unlock()
```

Listing 1: Original lock-based code.

```
1 l := OptiLock{}
2 m := &sync.Mutex{}
3 l.FastLock(m)
4 l.FastUnock(m)
```

Listing 2: Transformed HTM code.

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 **CS**s 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 `Lock()/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 subgraph 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. This choice is pragmatic in nature since LU-pairs spanning multiple functions are uncommon.

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 $\mathcal{M}(L)$ 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 $\mathcal{M}(L)$.*

[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.

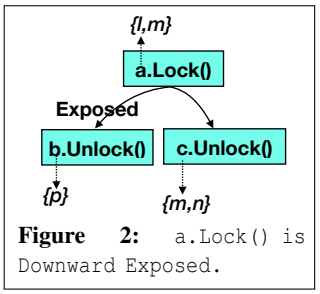

Figure 2: `a.Lock()` is `Downward Exposed`.

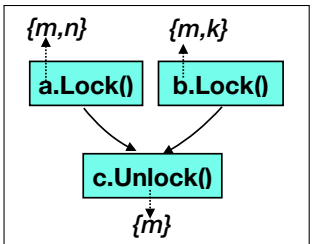

Figure 3: `c.Unlock()` is **not** Upward Exposed.

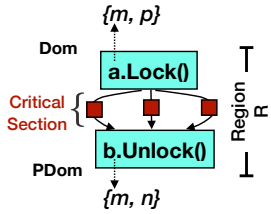

Figure 4: Region dominated by lock and post-dominated by unlock.

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

Definition 5.2 (Downward Exposed Lock-point (**DELock**))**.** *A lock-point, L, with points-to set $\mathcal{M}(L)$, 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 $\mathcal{M}(L)$.*

Definition 5.3 (Upward Exposed Unlock-point (**UEUnlock**))**.** *An unlock-point, U, with points-to set $\mathcal{M}(U)$, 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 $\mathcal{M}(U)$.*

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 $\{l,m\}$, has an empty intersection with b's points to set $\{p\}$; although it has a non-empty intersection with c's points-to set $\{m,n\}$. 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 $\overline{DELock(R)}$. Similarly, the remaining unlock-points in R are the complement of $UEUnlock(R)$, which is denoted by $\overline{UEUnlock(R)}$.

Definition 5.4 (Feasible-HTM-Pair)**.** *Let $L \in \overline{DELock(R)}$. Let $U \in \overline{UEUnock(R)}$. L and U form a feasible HTM pair if all of the following conditions are true,*

(1) $\mathcal{M}(L) \cap \mathcal{M}(U) \neq \phi$,
(2) $(L \text{ DOM } U) \bigwedge (U \text{ PDOM } L)$,
(3) The critical section $C \subseteq R$ *guarded by L and U contains no LU-point X such that* $\mathcal{M}(X) \cap (\mathcal{M}(L) \cup \mathcal{M}(U)) \neq \phi$, *and*
(4) C contains no HTM-unfriendly instructions.

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

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.Lock()` are post-dominated by unlock-point `b.Unlock()`, whose all incoming paths are dominated by `a.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 LU must be operating on the same `Mutex`. Appendix A in the extended version of this paper [94] justifies our choice of DOM/PDOM 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 offered to the runtime at the unlock-point. In case of an address mismatch of the mutex objects used in the same LU-pair, we can abort the transaction and revert to a safe state and fall back to using the locks. 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 allows nested locks but demands that they operate on disjoint `Mutex` objects.

HTM via Intel TSX allows nesting: if a nested transaction succeeds, hardware does not commit it until the outermost transaction commits. If a nested transaction fails, the control jumps to the starting code address of the nested transaction.

```
1 a.Lock() //outer region start
2
3
4 b.Lock()  // inner region start
5 b.Unlock() // inner region end
6
7 a.Unlock() //outer region end
```

Listing 3: Nested Locks.

```
1 a.Lock()
2
3 l := OptiLock{}
4 l.FastLock(b)
5 l.FastUnlock(b)
6
7 a.Unlock()
```

Listing 4: HTMized.

```
1 a.Lock() //outer region start
2
3
4 b.Lock()  // inner region start
5 a.Unlock() // inner region end
6
7 b.Unlock() //outer region end
```

Listing 5: Hand-over-hand lock.

```
1 a.Lock()
2
3 l := OptiLock{}
4 l.FastLock(b)
5 l.FastUnlock(a)
6
7 b.Unlock()
```

Listing 6: HTMized.

This facility allows us to safely transform locks into HTM even when they are nested.

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.Lock()` and `a.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 iff executing on the fastpath. As mentioned at the end of § 5.2.2, a mismatch is easy to recognize at runtime by, first, making `FastLock()` store the address of the `Mutex` used at the lock-point in a field in `OptiLock` and, second, checking whether the `Mutex` passed to `FastUnlock()` is the one present in `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) $\exists F \in F^* s.t.$ F's summary contains HTM-unfriendly instructions or (b) $\exists F \in F^* s.t.$ $\mathcal{P} \cup (\mathcal{M}(L) \cup \mathcal{M}(U)) \neq \phi$. 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.

```
1 func DeferExample() {
2
3   m := &sync.Mutex{}
4   defer m.Unlock()
5   m.Lock()
6   // critical section
7 }
```

Listing 7: defer Unlock.

```
1 func DeferExample() {
2   l := OptiLock{}
3   m := &sync.Mutex{}
4   defer l.FastUnlock(m)
5   l.FastLock(m)
6   // inside HTM
7 }
```

Listing 8: HTMized.

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

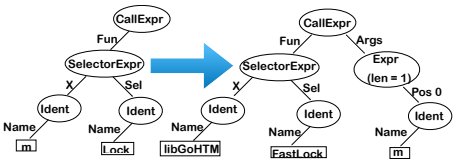

Figure 5: Example of AST transformation from `m.Lock()` to `optiLib.FastLock(m)`. Some AST nodes are omitted for brevity.

```
1
2 // pointer form
3 m := &sync.Mutex{}
4 m.Lock()
5 m.Unlock()
6
7 // value form
8 n := sync.Mutex{}
9 n.Lock()
10 n.Unlock()
```

Listing 9: Both Mutex pointer `m` and Mutex value `n` invoke Lock/Unlock using the same `dot` dereferencing operator.

```
1 l := Optilock{}
2 // pointer form
3 m := &sync.Mutex{}
4 l.FastLock(m)
5 l.FastUnlock(m)
6
7 // value form
8 n := sync.Mutex{}
9 l.FastLock(&n)
10 l.FastUnlock(&n)
```

Listing 10: GOCC transformation passes `m` as-is to FastLock()/FastUnlock() but `&n` to FastLock()/FastUnlock().

```
1 type AStruct struct {
2   x int //not anonymous
3   *sync.Mutex //anonymous
4 }
5 func main{} {
6
7   a := AStruct{}
8   a.Lock()
9   a.Unlock()
10 }
```

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

```
1 type AStruct struct {
2   x int // not anonymous
3   *sync.Mutex //anonymous
4 }
5 func main{} {
6   l := OptiLock{}
7   a := AStruct{}
8   l.FastLock(a.Mutex)
9   l.FastUnlock(a.Mutex)
10 }
```

Listing 12: Unnamed mutex transformed to HTM.

```
1  m := &sync.Mutex{}
2  for i:=0;i<10;i++ {
3   go func() {
4
5       m.Lock()
6       // CS
7       m.Unlock()
8    }()
9   }
10 //wait all
```

Listing 13: Anonymous goroutines create concurrent units of execution on anonymous functions.

```
1  m := &sync.Mutex{}
2  for i:=0;i<10;i++ {
3  go func() {
4       l := OptiLock{}
5       l.FastLock(m)
6       // CS
7       l.FastUnlock(m))
8    }()
9   }
10 //wait all
```

Listing 14: The `OptiLock` needed for the transformation should be placed in the innermost function scope.

AST `dot` operator as exemplified in Listing 9. However, `FastLock()` and `FastUnlock()` must receive a pointer to the `Mutex` object to perform the elision correctly. Hence, if the LU-pair uses a `Mutex` value, its address needs to be passed to `FastLock()/FastUnlock()`, and if the LU-pair uses a `Mutex` pointer, it should be passed as is.

We address this issue in the transformer by querying the type information [9] for each receiver object subject to transformation. If the receiver identifier is a `Mutex` value type, we insert the additional `address-of` operator before the `Mutex` identifier in the AST and pass it to `FastLock()/FastUnlock()`. If the receiver identifier is a pointer to a `Mutex` type, we pass it as is.

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 [9] of the *access path* [66] 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, 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 in the scope that encloses both Lock and Unlock operations, but it should not be shared by other concurrent executions because it maintains goroutine-specific state. Hence, we make `OptiLock` a variable in the stack of each goroutine. We add the declaration to the innermost function body as shown on line 4 in Listing 14. A bottom-up AST walk from LU-pair to be transformed allows us to easily detect the innermost enclosing anonymous 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 lock when necessary. It is responsible for interoperating 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.

repo	stars	contributors	commits	LoC	Lock points	Unlock points total (defer)	violates dominace	Candidate pairs	unfit for HTM intra/interproc	Nested alias locks intra/interproc	Transformed Pairs w/o profiles total (defer)	Transformed Pairs w/ profiles total (defer)
tally	450*	27	95	2.4k	54	56 (28)	2	52	2/29	0/0	21 (14)	7 (7)
zap	4.5k*	7	163	3.3k	8	8 (4)	0	8	3/0	0/0	5 (1)	6 (0)
go-cache	11.6k*	71	322	18k	96	230 (6)	68	28	0/2	0/0	26 (4)	10 (2)
fastcache	59k*	40	673	33k	24	24 (2)	2	22	2/2	0/0	18 (0)	7 (4)
set	967*	8	48	2.4k	16	16 (10)	0	16	0/2	0/0	14 (8)	8 (2)

Table 1: Go package characteristics and their behavior using GOCC

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()`. 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 geomean 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 `RWMutex`es at different points in time and accesses read-

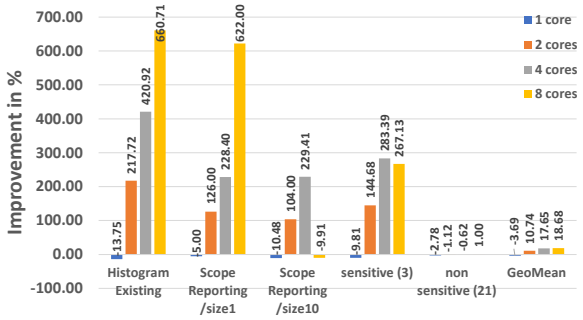

Figure 6: Results on Tally with different core numbers.

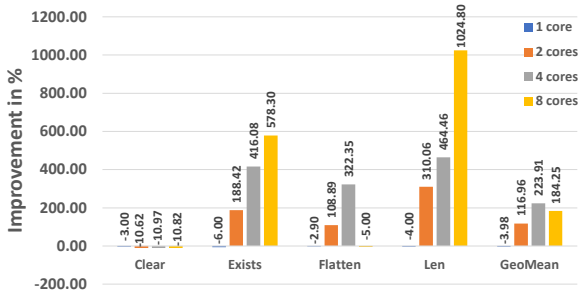

Figure 8: Results on concurrent set with different core numbers.

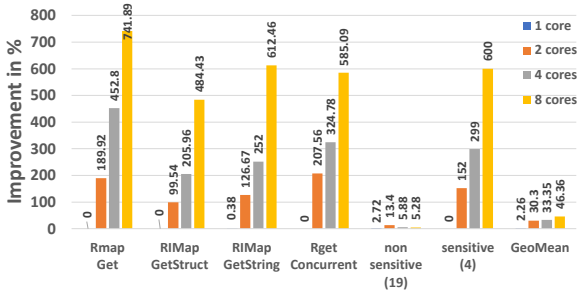

Figure 7: Results on go-cache with different core numbers. (benchmark names reflect abbreviated names of go-cache's benchmark functions).

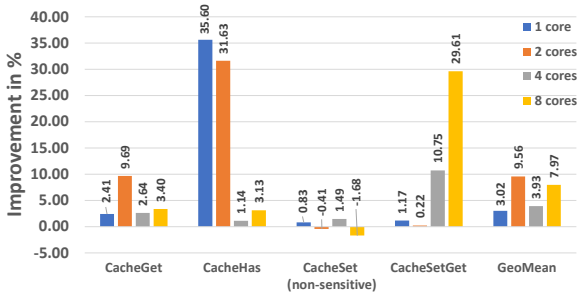

Figure 9: Results on fastcache with different core numbers.

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-CPUs.

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 `RWMutex`s for concurrent map read access. Unlike the rest of the use cases, the benchmark files themselves contain locks, which GOCC transforms into using HTM.

Figure 7 shows our empirical results. GOCC 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 GOCC.

go-datastructures [21] is a collection of performant, thread-safe data structures. We apply GOCC 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 a`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 GOCC 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.

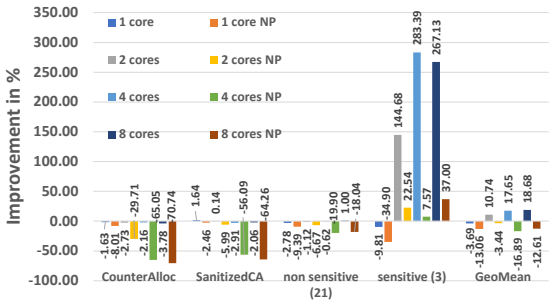

Figure 10: Results on `Tally` to show the effectiveness of perceptron. NP ⟹ No Perceptron.

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 Gran No. 1763699, 1717779, 1563935. We thank our shepherd Michael Spear and the anonymous reviewers for their feedback.

References

[1] GitHub - linux4life798/safetyfast: An Go library of synchronization primitives to help make use of hardware transactional memory (HTM). https://github.com/linux4life798/safetyfast.

[2] Go by Example: Closures. https://gobyexample.com/closures.

[3] Go Optimistic Concurrency Control (GOCC). https://github.com/uber-research/GOCC.

[4] Golang benchmark RunParallel API. https://golang.org/pkg/testing/#B.RunParallel.

[5] Intel 64 and IA-32 Architectures Optimization Reference Manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[6] Package astutil. https://godoc.org/golang.org/x/tools/go/ast/astutil.

[7] Package callgraph. https://pkg.go.dev/golang.org/x/tools/go/callgraph.

[8] Package format. https://golang.org/pkg/go/format/.

[9] Package go/types. https://golang.org/pkg/go/types/.

[10] Package reflect. https://golang.org/pkg/reflect/.

[11] Package sync. https://golang.org/pkg/sync/.

[12] patrickmn/go-cache: An in-memory key:value store/cache (similar to Memcached) library for Go, suitable for single-machine applications. https://github.com/patrickmn/go-cache.

[13] Reentrant Mutex. https://en.wikipedia.org/wiki/Reentrant_mutex.

[14] Solved: A low background number of - Intel Community. https://community.intel.com/t5/Software-Tuning-Performance/TSX-conflict-aborts-for-single-threaded-applications/m-p/983986#M3190.

[15] sync: Mutex performance collapses with high concurrency. https://github.com/golang/go/issues/33747.

[16] testing - The Go Programming Language. https://golang.org/pkg/testing/.

[17] uber-go/zap: Blazing fast, structured, leveled logging in Go. https://github.com/uber-go/zap.

[18] VictoriaMetrics/fastcache: Fast thread-safe inmemory cache for big number of entries in Go. Minimizes GC overhead. https://github.com/VictoriaMetrics/fastcache.

[19] Why Locking in Go much slower than Java? https://stackoverflow.com/questions/39815723/why-locking-in-go-much-slower-than-java-lots-of-time-spent-in-mutex-lock-mut.

[20] Why Use the Go Language for Your Project? https://nix-united.com/blog/why-use-the-go-language-for-your-project/.

[21] Workiva/go-datastructures: A collection of useful, performant, and threadsafe Go datastructures. https://github.com/Workiva/go-datastructures.

[22] Yehuda Afek, Amir Levy, and Adam Morrison. Software-Improved Hardware Lock Elision. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, page 212–221, New York, NY, USA, 2014. Association for Computing Machinery.

[23] Yehuda Afek, Alexander Matveev, Oscar R. Moll, and Nir Shavit. Amalgamated lock-elision. In Yoram Moses, editor, *Distributed Computing*, pages 309–324, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[24] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, page 85–96, New York, NY, USA, 1997. Association for Computing Machinery.

[25] Lars Ole Andersen. Program Analysis and Specialization for the C Programming Language. Technical report, University of Copenhagen, 1994.

[26] David F. Bacon and Peter F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, page 324–341, New York, NY, USA, 1996. Association for Computing Machinery.

[27] Vincent Blanchon. Go: Mutex and Starvation. https://medium.com/a-journey-with-go/go-mutex-and-starvation-3f4f4e75ad50, Sep 2019.

[28] Vincent Blanchon. Go: How to Reduce Lock Contention with the Atomic Package. https://medium.com/a-journey-with-go/go-

how-to-reduce-lock-contention-with-the-atomic-package-ba3b2664b549, Aug 2020.

[29] Irina Calciu, Tatiana Shpeisman, Gilles Pokam, and Maurice Herlihy. Improved single global lock fallback for best-effort hardware transactional memory. In *Transact 2014 Workshop. ACM*, page 54, 2014.

[30] Milind Chabbi. pprof++: A Go Profiler with Hardware Performance Monitoring. https://eng.uber.com/pprof-go-profiler/, May 2020.

[31] Milind Chabbi, Abdelhalim Amer, Shasha Wen, and Xu Liu. An Efficient Abortable-locking Protocol for Multi-level NUMA Systems. In Vivek Sarkar and Lawrence Rauchwerger, editors, *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Austin, TX, USA, February 4-8, 2017*, pages 61–74. ACM, 2017.

[32] Milind Chabbi and John Mellor-Crummey. Contention-Conscious, Locality-Preserving Locks. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, New York, NY, USA, 2016. Association for Computing Machinery.

[33] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging Locks for Non-Volatile Memory Consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '14, page 433–452, New York, NY, USA, 2014. Association for Computing Machinery.

[34] Keith Chapman, Antony L. Hosking, and J. Eliot B. Moss. Hybrid STM/HTM for Nested Transactions on OpenJDK. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, page 660–676, New York, NY, USA, 2016. Association for Computing Machinery.

[35] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[36] Luke Dalessandro, Franois Carouge, Sean White, Yossi Lev, Mark Moir, Michael L. Scott, and Michael F. Spear. Hybrid NOrec: A Case Study in the Effectiveness of Best Effort Hardware Transactional Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, page 39–52, New York, NY, USA, 2011. Association for Computing Machinery.

[37] Dave Dice, Maurice Herlihy, Doug Lea, Yossi Lev, Victor Luchangco, Wayne Mesard, Mark Moir, Kevin Moore, and Dan Nussbaum. Applications of the adaptive transactional memory test platform. In *3rd ACM SIGPLAN Workshop on Transactional Computing*, pages 1–10, 2008.

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

[39] Dave Dice, Alex Kogan, Yossi Lev, Timothy Merrifield, and Mark Moir. Adaptive Integration of Hardware and Software Lock Elision Techniques. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, page 188–197, New York, NY, USA, 2014. Association for Computing Machinery.

[40] Dave Dice, Alex Kogan, Yossi Lev, Timothy Merrifield, and Mark Moir. Adaptive Integration of Hardware and Software Lock Elision Techniques. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, page 188–197, New York, NY, USA, 2014. Association for Computing Machinery.

[41] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early Experience with a Commercial Hardware Transactional Memory Implementation. *SIGARCH Comput. Archit. News*, 37(1):157–168, March 2009.

[42] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, pages 157–168, 2009.

[43] Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, page 194–208, Berlin, Heidelberg, 2006. Springer-Verlag.

[44] Nuno Diegues and Paolo Romano. Self-Tuning Intel Transactional Synchronization Extensions. In *11th International Conference on Autonomic Computing (ICAC 14)*, pages 209–219, Philadelphia, PA, June 2014. USENIX Association.

[45] Nuno Diegues, Paolo Romano, and Luís Rodrigues. Virtues and Limitations of Commodity Hardware Transactional Memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, page 3–14, New York, NY, USA, 2014. Association for Computing Machinery.

[46] Google. Effective Go - The Go Programming Language. https://golang.org/doc/effective_go.html.

[47] Google. Kubernetes. https://kubernetes.io/.

[48] Google. Profiling Go Programs. https://blog.golang.org/pprof.

[49] Google. The Go Programming Language Specification. https://golang.org/ref/spec#Defer_statements.

[50] Susan L Graham, Peter B Kessler, and Marshall K McKusick. An execution profiler for modular programs. *Software: Practice and Experience*, 13(8):671–685, 1983.

[51] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. STMBench7: A Benchmark for Software Transactional Memory. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, page 315–324, New York, NY, USA, 2007. Association for Computing Machinery.

[52] Tim Harris and Keir Fraser. Language Support for Lightweight Transactions. *SIGPLAN Not.*, 38(11):388–402, October 2003.

[53] Maurice Herlihy and J Eliot B Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, pages 289–300, 1993.

[54] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[55] Michael Hind. Pointer Analysis: Haven't We Solved This Problem Yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, page 54–61, New York, NY, USA, 2001. Association for Computing Machinery.

[56] Jiamin Huang, Barzan Mozafari, and Thomas F. Wenisch. Statistical Analysis of Latency Through Semantic Profiling. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 64–79, New York, NY, USA, 2017. Association for Computing Machinery.

[57] Joseph Izraelevitz, Alex Kogan, and Yossi Lev. Implicit acceleration of critical sections via unsuccessful speculation. *11th ACM SIGPLAN Wkshp. on Transactional Computing, TRANSACT*, 16, 2016.

[58] Christian Jacobi, Timothy Slegel, and Dan Greiner. Transactional Memory Architecture and Implementation for IBM System Z. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, page 25–36, USA, 2012. IEEE Computer Society.

[59] Daniel A Jiménez and Calvin Lin. Dynamic branch prediction with perceptrons. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, pages 197–206. IEEE, 2001.

[60] Richard Johnson, David Pearson, and Keshav Pingali. The Program Structure Tree: Computing Control Regions in Linear Time. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, page 171–185, New York, NY, USA, 1994. Association for Computing Machinery.

[61] Tomas Karnagel, Roman Dementiev, Ravi Rajwar, Konrad Lai, Thomas Legler, Benjamin Schlegel, and Wolfgang Lehner. Improving in-memory database index performance with Intel® Transactional Synchronization Extensions. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 476–487. IEEE, 2014.

[62] Terrance Kelly. Programming Workbench Hand-Over-Hand Locking for Highly Concurrent Collections. https://www.usenix.org/system/files/login/articles/login_fall20_14_kelly.pdf, 2020.

[63] Andi Kleen. Lock elision in the GNU C library. https://lwn.net/Articles/534758/, January 2013.

[64] Andi Kleen. Scaling Existing Lock-Based Applications with Lock Elision: Lock Elision Enables Existing Lock-Based Programs to Achieve the Performance Benefits of Nonblocking Synchronization and Fine-Grain Locking with Minor Software Engineering Effort. *Queue*, 12(1):20–27, January 2014.

[65] William Landi and Barbara G. Ryder. Pointer-Induced Aliasing: A Problem Classification. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, page 93–103, New York, NY, USA, 1991. Association for Computing Machinery.

[66] Johannes Lerch, Johannes Späth, Eric Bodden, and Mira Mezini. Access-Path Abstraction: Scaling Field-Sensitive Data-Flow Analysis with Unbounded Access Paths. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ASE '15, page 619–629. IEEE Press, 2015.

[67] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. Taming Performance Variability. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 409–425, USA, 2018. USENIX Association.

[68] Dmitri Melikyan. Detecting Lock Contention in Go. https://www.instana.com/blog/detecting-lock-contention-in-go/, March 2016.

[69] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Bratin Saha, and Adam Welc. Practical Weak-Atomicity Semantics for Java STM. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, page 314–325, New York, NY, USA, 2008. Association for Computing Machinery.

[70] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Bratin Saha, and Adam Welc. Single Global Lock Semantics in a Weakly Atomic STM. *SIGPLAN Not.*, 43(5):15–26, May 2008.

[71] Takuya Nakaike, Rei Odaira, Matthew Gaudet, Maged M. Michael, and Hisanobu Tomari. Quantitative Comparison of Hardware Transactional Memory for Blue Gene/Q, ZEnterprise EC12, Intel Core, and POWER8. *SIGARCH Comput. Archit. News*, 43(3S):144–157, June 2015.

[72] Martin Pohlack and Stephan Diestelhorst. From lightweight hardware transactional memory to lightweight lock elision. https://www.cs.purdue.edu/sss/projects/transact11/papers/Pohlack.pdf, January 2011.

[73] Ravi Rajwar and James R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, page 294–305, USA, 2001. IEEE Computer Society.

[74] Torvald Riegel, Patrick Marlier, Martin Nowack, Pascal Felber, and Christof Fetzer. Optimizing Hybrid Transactional Memory: The Importance of Nonspeculative Operations. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, page 53–64, New York, NY, USA, 2011. Association for Computing Machinery.

[75] Carl RITSON and Frederick BARNES. An Evaluation of Intel's Restricted Transactional Memory for CPAs. https://core.ac.uk/download/pdf/18531106.pdf.

[76] Wenjia Ruan, Trilok Vyas, Yujie Liu, and Michael Spear. Transactionalizing legacy code: An experience report using GCC and memcached. *ACM SIGARCH Computer Architecture News*, 42(1):399–412, 2014.

[77] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: A High Performance Software Transactional Memory System for a Multi-Core Runtime. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, page 187–197, New York, NY, USA, 2006. Association for Computing Machinery.

[78] Michael L. Scott and William N. Scherer III. Scalable queue-based spin locks with timeout. In Michael T. Heath and Andrew Lumsdaine, editors, *Proceedings of the 2001 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'01), Snowbird, Utah, USA, June 18-20, 2001*, pages 44–52. ACM, 2001.

[79] Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Robert Geva, Yang Ni, and Adam Welc. Towards Transactional Memory Semantics for C++. In *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, page 49–58, New York, NY, USA, 2009. Association for Computing Machinery.

[80] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. Enforcing Isolation and Ordering in STM. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 78–88, New York, NY, USA, 2007. Association for Computing Machinery.

[81] Gustavo Sousa and Alexandro Baldassin. FGSCM: A fine-grained approach to transactional lock elision. In *29th International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2017, Campinas, Brazil, October 17-20, 2017*, pages 113–120. IEEE Computer Society, 2017.

[82] Michael F. Spear, Maged M. Michael, and Christoph von Praun. RingSTM: Scalable Transactions with a Single Atomic Instruction. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, page 275–284, New York, NY, USA, 2008. Association for Computing Machinery.

[83] Pengfei Su, Shuyin Jiao, Milind Chabbi, and Xu Liu. Pinpointing Performance Inefficiencies via Lightweight Variance Profiling. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[84] David Tarjan and Kevin Skadron. Merging path and gshare indexing in perceptron branch prediction. *ACM transactions on architecture and code optimization (TACO)*, 2(3):280–300, 2005.

[85] Linda Torczon and Keith Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2007.

[86] Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. Understanding Real-World Concurrency Bugs in Go. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 865–878, New York, NY, USA, 2019. Association for Computing Machinery.

[87] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings 22nd Annual International Symposium on Computer Architecture*, pages 392–403, 1995.

[88] Uber. Tally: A Go metrics interface with fast buffered metrics and third party reporters. https://github.com/uber-go/tally.

[89] Filippo Valsorda. Creative foot-shooting with Go RWMutex. https://blog.cloudflare.com/creative-foot-shooting-with-go-rwmutex/, Oct 2015.

[90] Amy Wang, Matthew Gaudet, Peng Wu, José Nelson Amaral, Martin Ohmacht, Christopher Barton, Raul Silvera, and Maged Michael. Evaluation of Blue Gene/Q Hardware Support for Transactional Memories. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, page 127–136, New York, NY, USA, 2012. Association for Computing Machinery.

[91] Qingsen Wang, Pengfei Su, Milind Chabbi, and Xu Liu. Lightweight Hardware Transactional Memory Profiling. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, page 186–200, New York, NY, USA, 2019. Association for Computing Machinery.

[92] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. Performance Evaluation of Intel Transactional Synchronization Extensions for High-Performance Computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[93] Richard M. Yoo, Yang Ni, Adam Welc, Bratin Saha, Ali-Reza Adl-Tabatabai, and Hsien-Hsin S. Lee. Kicking the Tires of Software Transactional Memory: Why the Going Gets Tough. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, page 265–274, New York, NY, USA, 2008. Association for Computing Machinery.

[94] Zhizhou Zhang, Milind Chabbi, Adam Welc, and Timothy Sherwood. Optimistic Concurrency Control for Real-world Go Programs (Extended Version with Appendix), 2021.

[95] L. Zheng, X. Liao, H. Jin, and H. Liu. Exploiting the Parallelism Between Conflicting Critical Sections with Partial Reversion. *IEEE Transactions on Parallel and Distributed Systems*, 28(12):3443–3457, 2017.

[96] Lukasz Ziarek, Adam Welc, Ali-Reza Adl-Tabatabai, Vijay Menon, Tatiana Shpeisman, and Suresh Jagannathan. A Uniform Transactional Execution Environment for Java. In *Proceedings of the 22nd European Conference on Object-Oriented Programming*, ECOOP '08, page 129–154, Berlin, Heidelberg, 2008. Springer-Verlag.

[97] Ferad Zyulkyarov, Vladimir Gajinov, Osman Unsal, Adrian Cristal, Eduard Ayguadé, Tim Harris, and Mateo Valero. Atomic Quake: Using Transactional Memory in an Interactive Multiplayer Game Server. volume 44, pages 25–34, 04 2009.