Reasoning About Lock Placements

Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv

Stanford University, AT&T Labs Research, MIT, Tel Aviv University

Abstract. A *lock placement* describes, for each heap location, which lock guards the location, and under what circumstances. We formalize methods for reasoning about lock placements, making precise the interacting obligations between the program, the organization of the heap, and the placement of locks. Our methods capture realistic and subtle situations, such as the placement and correct use of speculative locks and lock assignments that change dynamically with updates to the heap. We present results for flat heaps with no structure, tree-structured heaps and a language of DAG-shaped heaps with bounded in-degree.

1 Introduction

Most concurrent software is written using *locks*. A lock can be *acquired* by a thread and subsequently *released*; between the two operations the lock is said to be *held* by the thread. The characteristic property of locks is that a lock may be held by only one thread at a time, thereby providing a primitive for mutual exclusion between threads.

These definitions are standard and describe what locks do, but they fail to convey the higher-level purposes for which programmers use locks. Universally, locks are used to protect data, guaranteeing that only one thread operates on particular parts of the store at a time. The association between locks and the data they protect is, however, implicit, and in the presence of mutable data structures it is not even clear how to describe the relationship between a possibly changing set of locks and the changing heap the locks protect. Indeed, in many programs the association between locks and the data they protect changes over time.

A widely-used approach to concurrency originating from the database community is to use two-phase locking [9] to guarantee conflict serializability. This paper shows how to apply two-phase locking to programs with potentially shared dynamically-allocated data structures. We show that we can achieve conflict serializability by performing two-phase locking on *logical locks* and we develop techniques for mapping logical locks into concrete implementations in a way that captures the range of ways in which locks are used in practice. Logical locks allow us to reason about *speculative locks*, and the common situation in which updates to the heap change the association between locks and the data they protect. Our approach also allows us to reason about concurrent transactions on heaps with apparently-complex patterns of sharing in a simple way.

To explain our results, we begin with a slightly informal, simple, obviously correct, but impractical locking protocol. We assume the heap consists of a set of *objects*, each of which has a set of *fields* that can hold pointers to other objects. Equivalently, we can view the heap as a graph of nodes (the objects) and edges (the fields). We also assume that concurrent operations are expressed as transactions that execute atomically (e.g., database transactions, atomic blocks, or a similar atomicity primitive). Every heap edge has a *logical lock*. For each transaction t, we use a standard two-phase locking protocol:

1. Acquire all logical locks of every edge read or written by t.
2. Perform the reads and writes of t.
3. Release all of t's logical locks.

Because the transaction holds locks on every data item it touches before any writes are performed, it is easy to prove that any interleaving of concurrent transactions is *serializable* (equivalent to some sequential schedule of the transactions) [9]. The disadvantage of this approach is that it is slow: acquiring locks is expensive, and in practice acquiring a lock on every field of every object touched by a transaction is exorbitantly expensive. Thus, practical locking protocols use fewer locks. For example, a tree data structure might have a single lock at the root node, or a hash table may have one lock per hash bucket, with no locks on the contents of the buckets. The key insight is that in such designs the programmer has made an optimization: many logical locks are represented by a single *physical lock*. That is, we can still think of a transaction as acquiring all of the logical locks required, but now instead of acquiring the lock on the actual edge e it must instead acquire the physical lock $\pi(e)$ assigned to the edge by the *lock placement* π. So, for example, in the tree case $\pi(e) = \rho$, where ρ is the tree's root, for every edge in the tree. For the hash table, $\pi(e) = l_i$, where the i-th bucket has an associated physical lock l_i for every field e in the i-th bucket. When multiple logical locks are represented by a single physical lock, transactions need only acquire one physical lock to obtain access to multiple heap locations.

Thus, the starting point for our formal development is to distinguish between logical and physical locks, and to make the mapping π from logical locks to their corresponding physical locks explicit. In simple cases the lock placement π can be just a function from logical locks to physical locks, but most interesting applications require more involved mappings. The main complication is that the lock for an edge e can depend on the state of the heap; that is, which physical lock guards e may change depending on heap updates. For example, in the hash table example above, if an object o is moved from bucket b_i to bucket b_j the lock guarding the fields of o also changes from l_i to l_j. Therefore, the mapping $\pi(e)$ must encode a set of possible physical locks for e, each of which applies in a different heap state. In general the heap and the lock mapping are interdependent: which locks to acquire depends on the state of the heap, and any transaction t must take care to lock those parts of the heap that, if changed by another transaction, could alter the association between fields and the physical locks t holds. We introduce the idea of stability to make this notion precise. A set of locks and object fields is mutually *stable* if, given the current contents of the fields, the locks are sufficient to guard those fields under π.

Lock placements provide a unifying explanation for common programming idioms with locks:

- Locking at different granularities corresponds to altering the granularity of the lock placement. For example, each element of a tree may have its own lock, or every element of the tree may be associated with a single lock at the root. The lock placement makes explicit which locations are guarded by the same lock, and where that lock is placed.
- It is sometimes beneficial to place the lock guarding an object o in a field of o itself, which means that o cannot be locked without first accessing o in an unlocked state. The only safe way to acquire such *speculative locks* is to first read o, acquire the lock, and then reread o to validate that the link to the lock was not altered by a competing transaction before the lock was acquired. Our approach naturally handles speculative locks. Lock placements are general (the lock can be placed anywhere, including in the target of the field it is meant to protect) and the stability condition requires the revalidation of any unstable reads.
- Which locks guard which fields often changes over time. As a simple example, consider a heap in which all `nil` fields are guarded by a global lock, and all non-`nil` fields are guarded by a speculative lock in the object the field points to. When a `nil` field is assigned an object the global lock is *split* and no longer guards the field, and when a pointer field is assigned `nil` that field is *merged* into the global

lock. Lock placements can depend on the state of the heap and so naturally capture lock splitting and merging.

We develop our results incrementally, beginning with flat "heaps" that are just a set of global variables with no pointers (Section 2). In this simple setting we formalize the key notions of lock placements and stability, and we give a proof system for showing that transactions are *well-locked*. We also prove our main result, that well-locked transactions are serializable. We then consider heaps that are mutable trees (Section 3), where the main complication is that logical locks are now named by paths in the tree, which may be updated concurrently.

Finally, we consider mutable heaps with sharing (Section 4), where we extend our methods to heaps that are DAGs—i.e., where there may be multiple access paths to objects. To acquire a logical lock in the presence of sharing we must be sure that the appropriate physical lock is held on all potential access paths to the field in question, which means we must have some description of and be able to enumerate all such paths. We use a recent proposal for describing a large class of heaps with sharing called *decompositions* [13], though other approaches (e.g., a standard points-to or shape analysis) could be used just as well. The main restriction of our approach is that our proof technique applies only to DAGs where objects have bounded in-degree.

Because our focus is on formalizing and explaining lock placements and the safety conditions under which they are used correctly, we do not consider protocols that vary the strategy or implementation techniques for acquiring and releasing locks. In particular, we do not concern ourselves with the order in which locks are acquired or released. Thus, we do not discuss liveness properties such as deadlock or optimizations such as early release of locks, as these issues are orthogonal to the ones we explore. The standard techniques for ensuring deadlock-freedom can be applied in our context, including both static techniques (imposing a total ordering on locks), and dynamic techniques (using a contention manager to resolve deadlocks at runtime).

To summarize, our contributions are:

– We describe a novel two-level locking protocol in which we attach a logical lock to every field in a heap and use a *lock placement* to map the possibly unbounded set of logical locks to a smaller set of physical locks.
– We introduce the notion of heap *stability*, which describes what set of heap fields are protected by a set of locks under a given lock placement.
– We describe *well-locked* transactions, which characterize serializable transactions on the heap.
– To apply lock placements to dynamically-changing heaps, we propose *path locking*, in which we use a placement scheme based on graph paths to describe a space of locking strategies for tree- and DAG-structured heaps.
– We show that lock placements provide a simple explanation for standard programming techniques, including locking at different granularities, speculative locks, and lock splitting and merging.

2 Flat Maps

We first consider a simple class of heaps defined over a fixed set of *memory locations* $\mathcal{M}$. A *flat map heap* is a set of mappings $\{m \mapsto b\}_{m \in \mathcal{M}}$ from each location $m \in \mathcal{M}$ to a boolean value b. Let $\mathcal{L}$ be a fixed set of *physical locks*; in this section we assume that memory locations and locks are disjoint. For ease of exposition we consider only exclusive locks — that is, if a transaction holds a lock then no other transaction may acquire concurrent access to the same lock.

A common correctness criterion for concurrent transactions is *serializability*. Informally a concurrent execution of a set of transactions is serializable if the reads and writes transactions make to the heap are equivalent to the reads and writes in some

$m \in \mathcal{M}$	memory locations	$l \in \mathcal{L},\ L \subseteq \mathcal{L}$	locks, lock sets
$b ::= \mathsf{F} \mid \mathsf{T}$	booleans	$\omega ::= m \mapsto b$	heap assertions
$\pi \subseteq \mathcal{M} \to 2^{L \times \Phi}$	lock placements	$\Phi \ni \phi ::= b \mid \omega \mid \phi \vee \phi \mid \phi \wedge \phi$	guards
$t ::= \mathsf{write}(m, b) \mid \mathsf{observe}(m) = b \mid \mathsf{read}(m) = b \mid \mathsf{lock}\ l \mid \mathsf{unlock}\ l$	transaction ops.		

Fig. 1. Locations, Lock Placements, Transaction Operations

serial schedule of the same set of transactions. By showing that concurrent executions are serializable we can reason about our code as if only one transaction executes at a time.

A *transaction* $\mathbf{T}$ is a sequence $t^1 t^2 \ldots$ of the atomic *transaction operations* given in Figure 1: a possibly unstable read of location m yielding b ($\mathsf{read}(m) = b$), a stable read of location m yielding b ($\mathsf{observe}(m) = b$), a write of b to location m ($\mathsf{write}(m, b)$), a lock of a physical lock l ($\mathsf{lock}\ l$), or an unlock of physical lock l ($\mathsf{unlock}\ l$). With the exception of the read and observe operations the concrete semantics of transaction operations are standard; the details are in Appendix A. We assume the execution of transaction operations is sequentially consistent.

The transaction language distinguishes between between high-level observe operations, which are observations of the state of memory that affect the outcome of a transaction and for which the locking protocol must ensure serializability, and low-level read operations, which do not directly affect the outcome of a transaction and need not be serializable. A transaction may freely perform a read operation on any memory location at any time, regardless of the locks that it holds, however there is no guarantee that the value read will remain stable—other concurrent transactions may update it. If a transaction holds locks that ensure that the value returned by a read operation is stable and cannot be altered by concurrent transactions, then a transaction may observe the result of the read operation and use that value to perform computation. The distinction between stable and unstable reads is key to reasoning about *speculative locking*, which we discuss in Section 2.1.

2.1 Lock Placements

We associate a *logical lock* with every heap location $m \in \mathcal{M}$. Whenever a transaction observes or changes the value of a memory location it must hold the associated logical lock. Unfortunately it is inefficient to attach a distinct lock to every memory location. Instead we use a smaller set of *physical locks* (or simply *locks*) $\mathcal{L}$ to implement logical locks; a *lock placement* describes the mapping from logical locks to physical locks. Different choices of placement function describe different granularities of locking.

Formally, a *lock placement* π for a boolean heap is a mapping from each location $m \in \mathcal{M}$ to a guarded set of locks that protect it. Each entry in $\pi(m)$ is a pair of a lock $l \in \mathcal{L}$ and a *guard* ϕ, which is a condition under which l protects m. A guard is a boolean combination of heap assertions $m \mapsto b$; for a given memory location each lock may only appear at most once on the left hand side of a guarded lock pair, and the set of guards must be mutually exclusive, and total, that is, exactly one guard is true for any given heap state.

For example, suppose $\mathcal{M} = \{m_0, \ldots, m_{k-1}\}$. Different placements allow us to describe a range of different locking granularities:

- A coarse-grain locking strategy protects every memory location with the same lock, that is, set $L = \{l\}$ and set $\pi(m_i) = \{(l, \mathsf{T})\}$ for all i. To observe or write to any memory location a transaction must hold lock l.

– An medium-grain locking strategy stripes different memory locations across a small set of locks. Set $L = \{l_0, \ldots, l_{p-1}\}$, and then set $\pi(m_i) = \{(l_{(i \bmod p)}, \mathsf{T})\}$ for all i. To observe or write to memory location m_i, we must hold lock $l_{(i \bmod p)}$.
– A fine-grain strategy associates a distinct lock with every memory location. Set $L = \{l_0, \ldots, l_{k-1}\}$ and set $\pi(m_i) = \{(l_i, \mathsf{T})\}$ for all i. To observe or write to memory location m_i we must hold lock l_i.

Coarse	**Intermediate**	**Fine**
$\mathsf{lock}\ l$	$\mathsf{lock}\ l_1$	$\mathsf{lock}\ l_1$
$\mathsf{read}(m_1) = \mathsf{T}$	$\mathsf{read}(m_1) = \mathsf{T}$	$\mathsf{read}(m_1) = \mathsf{T}$
$\mathsf{observe}(m_1) = \mathsf{T}$	$\mathsf{observe}(m_1) = \mathsf{T}$	$\mathsf{observe}(m_1) = \mathsf{T}$
$\mathsf{read}(m_3) = \mathsf{F}$	$\mathsf{read}(m_3) = \mathsf{F}$	$\mathsf{lock}\ l_3$
$\mathsf{observe}(m_3) = \mathsf{F}$	$\mathsf{observe}(m_3) = \mathsf{F}$	$\mathsf{read}(m_3) = \mathsf{F}$
$\mathsf{read}(m_4) = \mathsf{F}$	$\mathsf{lock}\ l_0$	$\mathsf{observe}(m_3) = \mathsf{F}$
$\mathsf{observe}(m_4) = \mathsf{F}$	$\mathsf{read}(m_4) = \mathsf{F}$	$\mathsf{lock}\ l_4$
$\mathsf{unlock}\ l$	$\mathsf{observe}(m_4) = \mathsf{F}$	$\mathsf{read}(m_4) = \mathsf{T}$
	$\mathsf{unlock}\ l_1$	$\mathsf{observe}(m_4) = \mathsf{T}$
	$\mathsf{unlock}\ l_0$	$\mathsf{unlock}\ l_4$
		$\mathsf{unlock}\ l_3$
		$\mathsf{unlock}\ l_1$

Fig. 2. Three transaction traces that observe the values of memory locations m_1, m_3, and m_4 under three different lock placements.

Figure 2 shows three variants of a transaction that reads memory location m_1, m_3 and m_4, observing values T, F, and F respectively. The figure shows a variant of the transaction for each locking granularity, using $p = 2$ physical locks in the medium-grain case.

A *speculative lock placement* is a placement in which the identity of a lock that protects a memory location depends on the memory location itself. For example a simple speculative placement is as follows. Let $L = \{l_f, l_t\}$ and $\mathcal{M} = \{m\}$. Set

$$\pi(m) = l_f \text{ if } m \mapsto \mathsf{F}, \text{ or } l_t \text{ if } m \mapsto \mathsf{T} \qquad (1)$$

Under this placement, lock l_f protects memory location m if m contains the value F, whereas lock l_t protects memory location m if m contains the value T.

A more realistic example of speculative lock placement is motivated by transactional predication [4] which uses a speculative placement of STM metadata. We use a collection $\mathcal{M} = \{m_1, \ldots, m_k\}$ of memory locations to model a concurrent set. Location m_i has value T if value i is present in the set. We use $L = \{l_\perp, l_1, \ldots, l_k\}$ and the placement

$$\pi(m_i) = l_\perp \text{ if } m_i \mapsto \mathsf{F}, \text{ or } l_i \text{if } m_i \mapsto \mathsf{T}$$

The speculative placement allows us to attach a distinct lock to every entry present in the set, without also requiring that we keep around a distinct lock for every entry that is absent from the set. Two transactions that operate on keys present in the set only contend on the same lock if they are accessing the same key. Transactions that operate on keys that are absent will however contend on $l_\perp$; this strategy is effective if we expect sets to have at most a small fraction of all possible elements at any one time. If contention on absent entries becomes a problem we can reduce contention to arbitrarily low levels by striping the logical locks protecting absent entries across a set of physical locks $l_\perp^1, l_\perp^2, \ldots$ as discussed earlier.

	(a)	**(b)**	**(c)**
1:	$\mathsf{read}(m) = \mathsf{T}$	$\mathsf{read}(m) = \mathsf{T}$	$\mathsf{lock}\ l_f$
2:	$\mathsf{lock}\ l_t$	$\mathsf{lock}\ l_t$	$\mathsf{lock}\ l_t$
3:	$\mathsf{read}(m) = \mathsf{T}$	$\mathsf{read}(m) = \mathsf{F}$	$\mathsf{read}(m) = \mathsf{T}$
4:	$\mathsf{observe}(m) = \mathsf{T}$	$\mathsf{unlock}\ l_t$	$\mathsf{write}(m, \mathsf{F})$
5:	$\mathsf{unlock}\ l_t$	$\mathsf{lock}\ l_f$	$\mathsf{unlock}\ l_t$
6:		$\mathsf{read}(m) = \mathsf{F}$	$\mathsf{unlock}\ l_f$
7:		$\mathsf{observe}(m) = \mathsf{F}$	
8:		$\mathsf{unlock}\ l_f$	

Fig. 3. Traces that read and write a memory location m under the speculative lock placement $\pi(m) = \{(l_t, m \mapsto \mathsf{T}), (l_f, m \mapsto \mathsf{F})\}$. In (a) the trace observes the value of m; in (b) the trace incorrectly speculates the lock protecting m due to a concurrent update, and transaction (c) writes to m by taking both locks.

It may not be immediately obvious how to acquire a lock on a memory location when we do not know which

$$\text{(FLock)}\quad \frac{l \notin L}{\varOmega, L \vdash_\pi \mathsf{lock}\ l; \varOmega, L \cup \{l\}} \qquad \text{(FUnlock)}\quad \frac{l \in L \qquad L' = L \setminus \{l\} \qquad \varOmega' = \lceil \varOmega \mid L'; \pi \rceil}{\varOmega, L \vdash_\pi \mathsf{unlock}\ l; \varOmega', L'}$$

$$\text{(FReadUnstable)}\quad \frac{\varOmega' = \varOmega \cup \{m \mapsto b\} \qquad \neg\mathsf{locked}_\pi(m, \varOmega', L)}{\varOmega, L \vdash_\pi \mathsf{read}(m) = b; \varOmega, L} \qquad \text{(FReadStable)}\quad \frac{\varOmega' = \varOmega \cup \{m \mapsto b\} \qquad \mathsf{locked}_\pi(m, \varOmega', L)}{\varOmega, L \vdash_\pi \mathsf{read}(m) = b; \varOmega', L}$$

$$\text{(FObserve)}\quad \frac{(m \mapsto b) \in \varOmega}{\varOmega, L \vdash_\pi \mathsf{observe}(m) = b; \varOmega, L} \qquad \text{(FWrite)}\quad \frac{\begin{array}{c} m \in \mathrm{dom}\, \varOmega \qquad \varOmega' = \varOmega[m \mapsto b] \\ \Big(\forall m', l, \phi.\ (l, \phi) \in \pi(m') \land m \text{ appears in } \phi \implies l \in L\Big) \end{array}}{\varOmega, L \vdash_\pi \mathsf{write}(m, b); \varOmega', L}$$

Fig. 4. Well-locked transaction operations: $\varOmega, L \vdash_\pi t; \varOmega', L'$

lock to take without knowing the value of the memory location. The key to this apparent circularity is that a transaction can use unstable reads to guess the identity of the correct lock; once the transaction has acquired the lock it can redo the read to verify that its guess was correct. If the transaction guesses correctly, then the second read is stable. If the transaction guesses incorrectly it can release the lock and repeat the process. Figure 3(a) shows a transaction that observes the state of m under the speculative lock placement of Equation (1). If another transaction had raced, we might have had to retry the read, as shown in Figure 3(b). Finally, to perform an update, we must hold both locks, as shown in Figure 3(c); otherwise by changing m we might implicitly release a lock that another transaction holds on a particular state of m.

2.2 Well-Locked Transactions

We represent the state of a transaction as two sets: the *observation set* $\varOmega$ and a *lock set* L. The observation set $\varOmega$ is a set of heap assertions $m \mapsto b$ that represent a transaction's local picture of the heap. The lock set L is a set of *physical locks* held by the transaction. Every heap assertion in the observation set must be *stable*; informally, the facts in the observation set are logically locked and cannot be invalidated by a concurrent interfering transaction. We write $\varOmega[m \mapsto b]$ to denote the result of adding or updating the heap observation $m \mapsto b$ to $\varOmega$, replacing any existing observations about m. The predicate $\mathsf{locked}_\pi(m, \varOmega, L)$ holds for heap location m if a transaction with heap observations $\varOmega$ and locks L has logically locked location m under lock placement π:

$$\mathsf{locked}_\pi(m, \varOmega, L) = \exists (l, \phi) \in \pi(m).\ \ l \in L \land \varOmega \vdash \phi$$

The judgement $\varOmega, L \vdash_\pi t; \varOmega', L'$ defined in Figure 4 characterizes *well-locked operations.* The judgment holds if when transaction operation t is executed by a transaction with observations $\varOmega$ and holding locks L, then on completion of the operation the transaction has new observations $\varOmega'$ and locks L'. Given the set of physical reads, writes and locks that a transaction performs, the well-lockedness judgement computes the set of stable observations of the transaction, and ensures that a transaction's logical observations and writes only occur on locations on which a transaction holds logical locks.

The (FLock) rule allows a transaction to acquire a lock l if the transaction does not already have l in its set of locks L; acquiring a lock has no affect on the observation set Ω. The (FUnlock) rule allows a transaction to release any lock l in its lock set L; any facts in Ω that were protected by l are no longer stable, so the rule uses the *stabilization operator* to compute a new stable set of observations Ω'. The *stabilization* of a set of observations Ω_0 under locks L and placement π, written $\lceil \Omega_0 \mid L; \pi \rceil$, is the limit of the monotonic sequence:

$$\Omega_{i+1} = \{m \mapsto b \in \Omega_i \mid \mathsf{locked}_\pi(m, \Omega_i, L)\}$$

Note that the limit always exists, because Ω_0 is finite (since it is constructed by a finite transaction execution) and the empty set is always a fixed point of the equation if no larger set is. A set of observations Ω is *stable* under locks L and placement π if Ω is its own stabilization, that is, $\Omega = \lceil \Omega \mid L; \pi \rceil$.

Rule (FObserve) states that a transaction may logically observe any stable fact from its stable observation set Ω. The (FReadUnstable) rule allows a transaction to perform a speculative read on a memory location on which the transaction does not hold a lock; however since the result may not be stable the rule does not update the set Ω. To enable reasoning about speculation, the determination whether the read is stable or not occurs in a context that includes the read of m; since we assume that reads are atomic, there is an instant in time at which both the old stable facts in Ω and the newly read value of m hold, and it is in that context that we determine stability. The (FReadStable) rule allows a transaction to read memory locations on which it holds a lock; since such a read is stable the rule updates the set of observations Ω with the newly read information about the heap. Finally the (FWrite) rule requires that a transaction can only update a location m if it holds the lock on m; furthermore the lock for any location m' for which m appears in a guard must also be held by the transaction—hence no transaction can destabilize the observations of another transaction. The last condition together with the $\mathsf{locked}_\pi(m, \Omega, L)$ also implies that $\mathsf{locked}_\pi(m, \Omega', L)$ holds, which is why the latter is not listed as a precondition of the rule.

A transaction $\mathbf{T} = t^1 \ldots t^k$ is *well-locked* if there exists a sequence of lock sets L^i and observation sets Ω^i such that

$$L^0 = L^k = \emptyset, \quad \Omega^0 = \Omega^k = \emptyset, \text{ and } \Omega^{i-1}, L^{i-1} \vdash_\pi t^i; \Omega^i, L^i \text{ for } 1 \le i \le k.$$

As an example of applying the rules, consider again the speculative read transaction shown in Figure 3(b). Let Ω^i and L^i denote the lock sets of the transaction after line i. Initially we have $\Omega^0 = \emptyset$ and $L^0 = \emptyset$. The read on line 1 is unstable, so $\Omega^1 = \emptyset$ and $L^1 = \emptyset$. The lock on line 2 adds an entry to the lock set l_t, so $\Omega^2 = \emptyset$ and $L^2 = \{l_t\}$. The read on line 3 yields $m \mapsto \mathsf{F}$, however the read would only be stable if $\mathsf{locked}_\pi(m, \{m \mapsto \mathsf{F}\}, \{l_t\})$ holds, which it does not; once again we have $\Omega^3 = \emptyset$ and $L^3 = \{l_t\}$. Lines 4 and 5 update the lock set; we have $\Omega^4 = \Omega^5 = \emptyset$, $L^4 = \emptyset$, and $L^5 = \{l_f\}$. The read on line 6 once again yields $m \mapsto \mathsf{F}$, but this time the predicate $\mathsf{locked}_\pi(m, \{m \mapsto \mathsf{F}\}, \{l_f\})$ holds and the read is stable, yielding $\Omega^6 = \{m \mapsto \mathsf{F}\}$ and $L^6 = \{l_f\}$. The logical observation of $m \mapsto \mathsf{F}$ on line 7 is permitted by the judgement since we know $m \mapsto \mathsf{F}$ is part of the stable heap; the observation and lock sets are unchanged ($\Omega^7 = \Omega^6$, $L^7 = L^6$). Finally, line 8 releases lock l_f, so we have $L^8 = \emptyset$. The assertion about m in Ω^7 is no longer stable, so the stabilization operator removes it from the observation set, finally yielding $\Omega^8 = \emptyset$.

2.3 Serializability of Well-Locked Transactions

A *schedule* $\mathbf{s}$ for a set of transactions $\mathbf{T}_1, \ldots, \mathbf{T}_k$ is a permutation of the concatenation of all transactions in the set, such that each transaction $\mathbf{T}_i$ is a subsequence of $\mathbf{s}$. Formally, a schedule is *valid* if it corresponds to an execution of the concrete semantics (see Appendix A for details). Informally, validity requires the execution respect the mutual exclusion property of locks, and memory accesses must accurately reflect the state of the global heap. A schedule is *serial* if operations of different transactions are not interleaved.

Lemma 1. *Let* $\mathbf{s}$ *be a valid schedule of a set of well-locked transactions* $\{\mathbf{T}_1, \ldots, \mathbf{T}_k\}$. *Let* Ω_i^j *and* L_i^j *be the set of observations and locks of each transaction after schedule step* j. *Let* h^j *be the heap after schedule step* j. *Then for all time steps* j:

- *the lock sets* $\{L_i^j\}_{i=1}^k$ *are disjoint, and*
- *the observation sets* $\{\Omega_i^j\}_{i=1}^k$ *are stable, have disjoint domains, and heap* h^j *is an extension of each* $\{\Omega_i^j\}_{i=1}^k$.

Proof. By induction on the length of the schedule (see appendix for full proof).

The disjointness of observation sets in Lemma 1 is a consequence of the fact that our physical locks are exclusive. If we allowed shared/exclusive locks, then we would also need to allow observation sets to overlap on values protected by shared locks.

A well-locked transaction $\mathbf{T} = (t^i)_{i=1}^k$ is *logically two-phase* if the domains of the observation sets of the transaction have a growing phase and a shrinking phase, that is, there exists some j such that for all i where $1 \leq i \leq j$, we have $\mathrm{dom}\,\Omega^{i-1} \subseteq \mathrm{dom}\,\Omega^i$ and for all i where $j < i \leq k$ we have $\mathrm{dom}\,\Omega^{i-1} \supseteq \mathrm{dom}\,\Omega^i$.

A *logical schedule* $\hat{\mathbf{s}}$ is the subsequence of a schedule $\mathbf{s}$ consisting of all the $\mathsf{observe}$ and write operations. Two operations *conflict* if they access the same memory location m. Two schedules $\mathbf{s}_1$ and $\mathbf{s}_2$ are *conflict-equivalent* if the logical schedule $\hat{\mathbf{s}}_1$ can be turned into the logical schedule $\hat{\mathbf{s}}_2$ by a sequence of swaps of adjacent non-conflicting operations.

Lemma 2. *Any valid schedule of a set of well-locked, logically two-phase transactions* $\{\mathbf{T}_1, \ldots, \mathbf{T}_k\}$ *is conflict-equivalent to a serial logical schedule.*

Proof. Identical to the usual proof of serializability of two-phase locking [9]. A fact is "locked" when it is added to the observation set of a transaction, and "unlocked" when it is removed from the observation set of the transaction. Disjointness of observation sets is guaranteed by Lemma 1. The "read" and "write" operations of the two-phase locking proof correspond to $\mathsf{observe}$ and write operations on the observation set.

2.4 Shared/Exclusive Logical Locks

A limitation of the protocol just presented is that locks are exclusive — holding a lock gives a transaction sole access to an edge, even if the transaction only wants to read the edge. Lock placement is a separate issue from whether non-exclusive locks exist for reading. Exclusive locks are sufficient to illustrate all of the important features of our techniques and have the advantage of not introducing the extra and extraneous complications of supporting non-exclusive access. However, non-exclusive locks are important, and so we briefly illustrate how to extend our approach to locks providing shared read access.

To allow shared access to fields we relax the requirement that guards must be mutually exclusive, thereby allowing each logical lock to map to many physical locks at the same time. Under the relaxed definition of placement, a transaction has shared access to a memory location m if it holds at least one of the locks that protect m, whereas a transaction has exclusive access to m if it holds all of the locks that protect m. Formally, a transaction has shared access to a memory location m if $\mathsf{locked}_\pi(m, \Omega, L)$ holds. We also define a new predicate $\mathsf{exclusive}_\pi(m, \Omega, L)$ which holds for heap location

$$\begin{array}{rlrl}
f, \mathbf{f}, \mathcal{F} & \text{fields} & x, y, \rho & \text{object names} \\
e ::= \mathsf{nil} \mid x & \text{expressions} & \omega ::= x.f \mapsto e & \text{heap assertions} \\
\pi \subseteq 2^{\mathbf{f}} \to 2^{\mathbf{f}} & \text{placements} & & \\
\multicolumn{3}{l}{t ::= \mathsf{write}(x.f, e) \mid \mathsf{observe}(x.f) = e \mid \mathsf{read}(x.f) = e} & \\
\multicolumn{3}{l}{\quad \mid x = \mathsf{new}() \mid \mathsf{lock}\ x \mid \mathsf{unlock}\ x} & \text{transaction ops.}
\end{array}$$

Fig. 5. Tree transactions

m if a transaction with heap observations Ω and locks L has an exclusive logical lock on location m under lock placement π:

$$\mathsf{exclusive}_\pi(m, \Omega, L) = \forall (l, \phi) \in \pi(m).\ \ l \in L \lor \Omega \vdash \neg\phi$$

To show serializability, we need to add an additional precondition to the (FWrite) rule requiring that a transaction have exclusive access to any memory location it writes. The statement of the proof of Lemma 1 must be altered since different observation sets may share fields on which they hold a shared lock—only the exclusively held fields must be disjoint between transactions. Finally we must update the definition of a two-phase transaction to ensure that transactions only release exclusive access to a field in the shrinking phase of a transaction.

3 Mutable Tree-Structured Heaps

In Section 2 we described a locking protocol for a class of flat heaps with a fixed set of memory locations and locks. In this section we extend our results to dynamically allocated, mutable tree-shaped heaps with a placement function based on paths.

A *tree heap* h consists of a set of objects, each with a unique name, usually denoted x or y. Every object has a fixed set of fields $\mathcal{F}$. Each object field $x.f$ contains a pointer either to some object y or nil. The heap contains a distinguished *root object*, named ρ. In a quiescent state, that is, in the absence of running transactions, we require that the heap be a forest.

As in the case of flat heaps, we associate a logical lock with every field of every object in the heap. Unlike the flat heaps of Section 2 we do not assume that we have a separate set of locks distinct from the set of memory locations; instead, following the practice of languages such as Java, we require that every heap object can function as a physical lock, and we use a lock placement function to describe a policy for mapping the logical locks attached to fields onto the physical locks (the objects). To define the lock placement, we use access paths from the root ρ to name both the fields we want to protect and the objects whose physical locks protect them.

We extend the set of transaction operations of Section 2 to read from and write to fields of objects, to handle dynamic allocation of new objects, and to apply lock and unlock operations to objects rather than a separate set of locks. The transaction operations, shown in Figure 5, are: write an expression e (either an object y or nil) to field f of object x ($\mathsf{write}(x.f, e)$)), a possibly unstable read of field f of object x yielding result e ($\mathsf{read}(x.f) = e$), a stable observation of field f of object x yielding e ($\mathsf{observe}(x.f) = e$), allocation of a fresh object ($x = \mathsf{new}()$), locking an object ($\mathsf{lock}\ \ x$), and unlocking an object ($\mathsf{unlock}\ \ x$).

3.1 Lock Placements

We name edges in the heap as a non-empty field path (a sequence of field names) $\mathbf{f} = f_1 f_2 \ldots$ from the root, ending in the edge in question. Since the path names a field in the heap, the path must be non-empty. We also name objects using fields, except that the path ends at the object the field points to; note that in the case of objects the empty path names the root of the heap. A *lock placement* π is a function from non-empty paths to paths, which maps every edge in a heap to an object whose attached physical lock protects it.

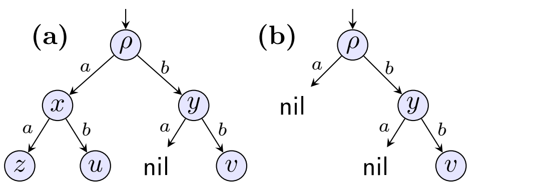

Fig. 6. Two examples of tree-structured heaps. Nodes (e.g., x, y) represent objects, whereas edges (labeled a, b) represent fields. Node ρ is the root object of the heap.

Consider heaps with field labels drawn from the set $\mathcal{F} = \{a, b\}$. We can protect every edge of the heap with a single coarse-grain lock at the root by setting $\pi_1(\mathbf{f}) = \epsilon$ for all $\mathbf{f}$. If we want different locks for the a and b subtrees, we can use the lock placement

$$\pi_2(\mathbf{f}) = \begin{cases} a & \text{if } a \prec \mathbf{f} \\ b & \text{if } b \prec \mathbf{f} \\ \epsilon & \text{if } \mathbf{f} = a \text{ or } \mathbf{f} = b \end{cases} \qquad (2)$$

where $\mathbf{g} \prec \mathbf{f}$ denotes that $\mathbf{g}$ is a prefix of $\mathbf{f}$. For example, in Figure 6(a), under placement π_2 the lock at ρ protects the edges from ρ to x and from ρ to y, the lock at x protects the edges from x to z and from x to u, and the lock at y protects the edge from y to v.

If for an edge $\mathbf{f}$ the placement path $\pi(\mathbf{f})$ leads to nil in the heap, we use the lock on the object immediately preceding the edge to nil in the placement path. For example, consider the heap of Figure 6(b) under the placement π_2. The lock that protects the edge named by the path ab according to the placement is $\pi_2(ab) = a$, however the edge a from the root node ρ points to nil. In this case, we use the lock on the longest non-nil prefix of $\pi_2(ab)$ to protect the edge ab, namely ρ itself.

Modifications to the heap may implicitly alter the mapping from logical locks to physical locks. If a transaction updates an edge, then the transaction must hold all logical locks whose mapping to physical locks may change both before and after the update. For example consider again the lock placement π_2 in the context of the tree heap shown in Figure 6(b). According to the placement the lock on ρ protects the edge a from the root; however since edge a points to nil, edges on any path that begins with a are also protected by the lock at a. If a transaction were to set $\rho.a$ to point to a fresh vertex w, the lock at w would now protect the edges on paths that begin with a; the transaction has *split* the logical roles of the lock at ρ before the write between the lock at ρ and the lock on new node w. Whenever a transaction splits or merges locks (e.g., by setting the field $\rho.a$ to nil again), it must hold every lock involved.

Figure 7(a) shows a trace of a transaction that adds a new edge labeled a from object y to a fresh object w to the heap of Figure 6(b) under placement π_2. The transaction acquires two locks, namely lock at ρ that protects the edge from ρ to z, and the lock at y that protects the entire subtree rooted at y. We need not hold a lock on w when adding w into the tree since no path in the range of the placement function is a suffix of the path to the updated edge ba.

If we desired a finer-grain locking, we can use a lock attached to every object to protect the fields of that object by using the placement function

$$\pi_3(\mathbf{g}f) = \mathbf{g} \text{ for any } \mathbf{g}, f. \qquad (3)$$

The lock placement π_3 places the lock that protects each edge f on the object at the head of the edge. Figure 7(b) shows a trace of a transaction that again adds the edge labeled a from node z to a fresh node x to the heap of Figure 6(b), this time under lock placement π_3. Unlike the transaction of Figure 7(a), we need to ensure that by adding the new edge the write does not implicitly change the mapping from edges to locks.

The well-lockedness conditions, which we introduce shortly, require that a transaction hold all physical locks which may map to different logical locks before and after a write. The operation read$(y.a)$ = nil verifies that there is no existing subtree of y reachable via edge a. Before the update the lock at y protects every possible edge reachable from $y.a$, however after the write the lock y only protects the edge $y.a$ itself, whereas the lock at w protects everything reachable from node w. Hence we must hold lock w when performing the write, since adding the edge splits the lock at y. (In general one must hold locks when connecting objects into the heap, however in this specific case, since the write which links w to the heap is the last write in the transaction it would be possible to optimize away the lock and unlock.)

(a)
```
lock $\rho$
read($\rho.b$) = $y$
observe($\rho.b$) = $y$
lock $y$
$w$ = new ()
write($y.a, w$)
unlock $y$
unlock $\rho$
```

(b)
```
lock $\rho$
read($\rho.b$) = $z$
observe($\rho.b$) = $z$
lock $y$
read($y.a$) = nil
$w$ = new ()
lock $w$
write($y.a, w$)
unlock $w$
unlock $y$
unlock $\rho$
```

(c)
```
read($\rho.b$) = $y$
lock $y$
read($\rho.b$) = $y$
observe($\rho.b$) = $y$
read($y.a$) = nil
$w$ = new ()
lock $w$
write($y.a, w$)
unlock $w$
unlock $y$
```

Fig. 7. Three transaction traces that add a new outgoing edge labelled a from node z to the tree of Figure 6(b) under: (a) the lock placement π_2 defined in Equation (2), (b) the lock placement π_3 defined in Equation (3), and (c) the lock placement π_4 defined in Equation (4).

Finally, we can use a speculative placement, as in the last section. If we set

$$\pi_4(\mathbf{f}) = \mathbf{f} \qquad (4)$$

the lock that protects each edge is located at the target of the edge. Figure 7(c) once again shows a transaction that adds a fresh edge labeled a to node z, this time using lock placement π_4. The transaction begins by performing a speculative read to guess that the identity of the object whose lock protects $\rho.b$ is y. After locking y, the transaction performs the read again; since the read still returns y, we know that the read is stable since the transaction already holds lock y. The transaction then performs a read of $y.a$ which returns nil. The value of the placement function for edge $y.a$ is $\pi(ba) = ba$, however since edge ba points to nil, the lock on the longest non-nil prefix of ba protects ba, in this case path b (node y). Since we hold the lock on y already, we know that the read of $y.a$ is also stable. Finally, the transaction must hold the lock on w when adding it to the heap to maintain the invariant that a transaction must hold all physical locks whose logical/physical mapping changes as a consequence of a write.

3.2 Well-Locked Transactions

We represent a transaction's state by three sets. As before, L is the set of locks that transaction holds, and Ω is a set of stable heap observations of the form $x.f \mapsto e$. We do not require Ω be a forest; a transaction may create any heap shapes it desires within its local heap. However, the forest invariant must be restored when the transaction releases objects in its local heap back into the global heap. Enforcing this condition is the purpose of the set Γ. An object x is a member of Γ if the transaction has shown that there is no globally visible path from the root to x (i.e., the transaction has locked the edge to x). The well-lockedness rules for tree heaps ensure that there is at most one globally-visible edge to any node and hence the globally-visible part of the heap is a forest. At the start of every transaction Γ is the empty set. Transactions add entries to Γ by discovering global edges to nodes and transferring them into their local heap Ω; entries are removed from Γ when pointers to objects are released from the stable heap Ω back into the global heap.

The *path alias* judgement $\Omega \vdash \mathbf{f} \sim x$ holds if $\mathbf{f}$ is a path in Ω from the root to location x; that is, if $|\mathbf{f}| = k$, then there is a sequence of vertices $\mathbf{v} = v_0 v_1 \cdots$ such that

$$\text{(TLock)}\ \frac{x \notin L}{\Omega, \Gamma, L \vdash_\pi \mathsf{lock}\ x; \Omega, \Gamma, L \cup \{x\}} \qquad \text{(TUnlock)}\ \frac{x \in L \qquad L' = L \setminus \{x\} \qquad (\Omega', \Gamma') = \lceil \Omega; \Gamma \mid L'; \pi \rceil \qquad \mathsf{forest}(\Omega, \Omega', \Gamma, \Gamma')}{\Omega, \Gamma, L \vdash_\pi \mathsf{unlock}\ x; \Omega', \Gamma', L'}$$

$$\text{(TNew)}\ \frac{\Omega' = \Omega \cup \{x.f \mapsto \mathsf{nil} \mid f \in \mathcal{F}\} \qquad x \notin \mathrm{dom}\,\Omega \qquad x \notin \Gamma \qquad \Gamma' = \Gamma \cup \{x\}}{\Omega, \Gamma, L \vdash_\pi x = \mathsf{new}(); \Omega', \Gamma', L} \qquad \text{(TObserve)}\ \frac{(x.f \mapsto e) \in \Omega}{\Omega, \Gamma, L \vdash_\pi \mathsf{observe}(x.f) = e; \Omega, \Gamma, L}$$

$$\text{(TReadUnstable)}\ \frac{x.f \notin \mathrm{dom}\,\Omega \qquad \Omega' = \Omega \cup \{x.f \mapsto e\} \qquad \neg\mathsf{locked}_\pi(x.f, \Omega', \Gamma, L)}{\Omega, \Gamma, L \vdash_\pi \mathsf{read}(x.f) = e; \Omega, \Gamma, L}$$

$$\text{(TReadStable)}\ \frac{x.f \notin \mathrm{dom}\,\Omega \qquad \Omega' = \Omega \cup \{x.f \mapsto e\} \qquad \mathsf{locked}_\pi(x.f, \Omega', \Gamma, L) \qquad \Gamma' = \begin{cases} \Gamma & \text{if } e = \mathsf{nil} \\ \Gamma \cup \{y\} & \text{if } e = y \end{cases}}{\Omega, \Gamma, L \vdash_\pi \mathsf{read}(x.f) = e; \Omega', \Gamma', L}$$

$$\text{(TWrite)}\ \frac{x.f \in \mathrm{dom}\,\Omega \qquad \Omega' = \Omega[x.f \mapsto e] \qquad \Big(\forall \mathbf{g}, \mathbf{h}.\ (\Omega \vdash \mathbf{g} \sim x) \wedge \mathbf{g}f \preceq \pi(\mathbf{h}) \implies \mathsf{pathlocked}_\pi(\mathbf{h}, \Omega, L) \wedge \mathsf{pathlocked}_\pi(\mathbf{h}, \Omega', L)\Big)}{\Omega, \Gamma, L \vdash_\pi \mathsf{write}(x.f, e); \Omega', \Gamma, L}$$

Fig. 8. Well-locked tree operations: $\Omega, \Gamma, L \vdash_\pi t; \Omega', \Gamma', L'$

$(\rho.f_0 \mapsto v_0) \in \Omega$, $(v_{i-1}.f_{i-1} \mapsto v_i) \in \Omega$ for all $1 < i < k-1$, and $v_{k-1}.f_{k-1} \mapsto x$. We write $\mathbf{f} \in \Omega$ if the path $\mathbf{f}$ from the root vertex exists in Ω, that is, $\Omega \vdash \mathbf{f} \sim x$ holds for some object x.

The *restriction of a path* $\mathbf{f}$ to a local heap Ω, written $\mathbf{f}|_\Omega$ is defined as:

$$\mathbf{f}|_\Omega = \begin{cases} \mathbf{f} & \text{if } \mathbf{f} \in \Omega \\ \mathbf{g} & \text{where } \exists \mathbf{g}, h.\ \mathbf{g}h \preceq \mathbf{f} \wedge \Omega \vdash \mathsf{nil} \sim \mathbf{g}h \\ \text{undefined} & \text{otherwise} \end{cases}$$

The restriction of path $\mathbf{f}$ is either $\mathbf{f}$ itself if present in the heap, or the longest prefix of the path present in the heap where no edge points to nil. The restriction of a path is undefined if the path $\mathbf{f}$ leaves the stable local heap Ω.

We hold the lock on an edge reached via a path if we hold the corresponding lock placement, restricted to the heap:

$$\mathsf{pathlocked}_\pi(\mathbf{f}, \Omega, L) ::= \exists x \in L.\ \Omega \vdash \pi(\mathbf{f})|_\Omega \sim x$$

We hold the lock on a field f of an object x under observations Ω, objects Γ and locks L, written $\mathsf{locked}_\pi(x.f, \Omega, \Gamma, L)$, if we hold a lock on field f on every path in the local heap, and furthermore there are no paths to x outside the local heap. Formally,

$$\mathsf{locked}_\pi(x.f, \Omega, \Gamma, L) ::= x \in \Gamma \ \wedge\ \forall \mathbf{g}.\ (\Omega \vdash x \sim \mathbf{g} \implies \mathsf{pathlocked}_\pi(\mathbf{g}f, \Omega, L))$$

If the local heap Ω contains cycles, observe that there may be infinitely many paths $\mathbf{g}$ and the predicate is well-defined in this case. To verify the absence of paths to x from outside the local heap, it is sufficient to check that $x \in \Gamma$, because any object $y \notin \Gamma$ is outside the local heap and thus has no stable fields and cannot form part of a path to x. Further, the definition of the locked predicate implies that if there is no path from the root ρ to a node x, then the fields of x are locked for any transaction with $x \in \Gamma$;

thus newly allocated objects can be immediately added to Γ without taking an explicit lock since they are created disconnected from the global heap.

The judgement $\Omega, \Gamma, L \vdash_\pi t; \Omega', \Gamma', L'$ defined in Figure 8 captures the class of *well-locked tree operations*. If the judgement holds, then a transaction that executes operation t under stable observation set Ω, objects Γ, and lock set L yields a new stable observation set Ω', objects Γ' and lock set L'. The (TNew) rule states that all of the fields of a newly allocated object x point to nil, and since there can be no path to x in the heap all of x's fields are stable and $x \in \Gamma$. As before, the (TLock) rule allows a transaction to acquire a lock it does not yet hold and has no affect on either Ω or Γ.

In the (TUnlock) rule, the stabilization operator is slightly more involved that in the case of flat heaps, because we must compute not just the stable set of heap facts, but also the set of objects for which the transaction has locked the incoming path: if an edge $x.f \mapsto y$ drops out of the stable observation set because a lock is released, the transaction can no longer assume it holds locks on all of the paths to object y. The stabilization (Ω', Γ') of a local heap Ω_0 and global heap Γ_0 under locks L and placement π, written $(\Omega', \Gamma') = \lceil \Omega_0; \Gamma_0 \mid L; \pi \rceil$, is the limit of the monotonically decreasing sequence:

$$\Omega_{i+1} = \{x.f \mapsto e \in \Omega_i \mid \mathsf{locked}_\pi(x.f, \Omega_i, \Gamma_i, L)\} \quad \Gamma_{i+1} = \Gamma_i \setminus \{y \mid x.f \mapsto y \in \Omega_i \setminus \Omega_{i+1}\}$$

In addition, rule (TUnlock) requires that transactions maintain the *forest condition*

$$\mathsf{forest}(\Omega, \Omega', \Gamma, \Gamma') \quad ::= \quad \forall y. \quad |\{x.f \mid (x.f \mapsto y) \in \Omega \setminus \Omega'\}| \quad = \quad \begin{cases} 1 & \text{if } y \in \Gamma \setminus \Gamma' \\ 0 & \text{otherwise.} \end{cases}$$

The forest condition ensures that a transaction may only release a pointer to a node y into the global heap if there are no other references to y in the global heap ($y \in \Gamma$). Furthermore, the condition also ensures that a transaction cannot release two or more pointers to the same location y into the global heap.

The rules (TObserve), (TReadUnstable), and (TReadStable) are similar to the rules in Section 2, updated to reflect that the heap now involves objects and fields. Note that (TReadStable) adds the object that is the target of the read to Γ in the case that the field is not nil.

The most interesting rule is (TWrite). Writing a field $x.f \mapsto y$ not only changes the paths to y, it changes the paths a to every object reachable from y. Thus, as a result of a single field update, the lock placements may change for y and every edge reachable from y. Furthermore, fields no longer reachable from $x.f$ after the update also may have altered lock placements. For this reason a transaction must hold locks on every edge reachable from $x.f$ both before and after the update. These conditions are necessary for safety, but need not be burdensome if the lock placement has a suitable granularity. For example, if the subtrees rooted at x and y are locked by the locks at x and y respectively, the update requires two locks.

A transaction $\mathbf{T} = t^1 \ldots t^k$ is *well-locked* if there exists a sequence of lock sets L^i, observation sets Ω^i, and object sets Γ^i such that

$$L^0 = L^k = \emptyset, \quad \Omega^0 = \emptyset, \quad \Gamma^0 = \emptyset, \text{ and } \Omega^{i-1}, \Gamma^{i-1}, L^{i-1} \vdash_\pi t^i; \Omega^i, \Gamma^i, L^i \text{ for } 1 \le i \le k.$$

A well-locked must begin with all three sets Ω, Γ, L empty. Furthermore at the end of the transaction the set of locks L must be empty again, and hence a transaction must release all of its locks. We do not require that Ω or Γ be empty at the conclusion of a transaction; however since the transaction may not hold any locks on termination, any part of the heap that is stable and in Ω with an empty lock set cannot be reachable from the global and can be garbage-collected.

Lemma 3. *Let* $\mathbf{s}$ *be a valid schedule of a set of well-locked transactions* $\{\mathbf{T}_1, \dots, \mathbf{T}_k\}$*. Let* Ω_i^j*,* Γ_i^j*, and* L_i^j *be the set of observations, objects, and locks of each transaction after schedule step* j*. Let* h^j *be the heap after schedule step* j*, and suppose* h^0 *is a forest. Then for all time steps* j*:*

– *the lock sets* $\{L_i^j\}_{i=1}^k$ *are disjoint.*
– *the observation sets* $\{\Omega_i^j\}_{i=1}^k$ *are stable, have disjoint domains, and heap* h^j *is an extension of each* $\{\Omega_i^j\}_{i=1}^k$*.*
– *the sets* $\{\Gamma_i^j\}_{i=1}^k$ *are disjoint, and*
– *the global heap* h^j *less edges present in the local heaps* $\{\Omega_i^j\}_{i=1}^k$ *is a forest. Furthermore if* $x \in \Gamma_i^j$ *then every pointer to node* x *in the heap is an element of some local heap* $\Omega_{i'}^j$*.*

The proof of Lemma 3 is in the appendix. Finally, we have a logical serializability lemma analogous to Lemma 2:

Lemma 4. *Any valid schedule of a set of well-locked, logically two-phase tree transactions* $\{\mathbf{T}_1, \dots, \mathbf{T}_k\}$ *is conflict-equivalent to a serial logical schedule.*

As in the flat heap case, these results can be extended to shared/exclusive locks using the approach in Section 2.4.

4 Transactions on DAGs of Bounded Degree

At the core of the locking protocol of Section 3 is the invariant that the global heap is a forest. Since lock placements are defined using access paths, for soundness the locking protocol must verify that a transaction holds the locks that protect an edge on every possible access path. In a forest there can be at most one path between nodes.

In this section we show how to relax the forest restriction and apply lock placements to a class of directed acyclic graph heaps with a bounded number of paths to each node. The technical machinery developed so far remains almost unchanged, with the exception that the forest condition is replaced by a condition that allows for more paths to an object. One hurdle, however, is that we need to define some language of heaps with sharing—we need some way to describe the aliasing patterns in the heap, for otherwise it is not possible to define what it means to be sound for any locking protocol. We use a recent proposal for describing a large class of heaps with sharing [13], which describe *decomposition heaps* whose shape matches a static description given by a *decomposition heap shape*. We stress that our results are not limited to class of heaps described in [13]; our techniques could be applied analogously to any number of other methods for describing the possible shapes of the heap. The point is that we need some description of the sharing patterns in the heap, and one good choice is to use decomposition heaps.

A *decomposition heap shape* $\hat{h}$ is a rooted, connected, directed acyclic graph $(\hat{V}, \hat{E})$ consisting of a set of vertices $\hat{V} = \{\hat{u}, \hat{v}, \dots\}$ and a set of edges $\hat{E} \subseteq \hat{V} \times \hat{F} \times \hat{V}$ labeled with field names drawn from a set $\hat{F}$. We require that every edge in a decomposition shape have a unique field label. Figure 9(a) gives a decomposition heap shape describing the data structures of a simple process scheduler. Every process has associated fields *pid* (process id), *ns* (name space), and the process' assigned *cpu*. To find the cpu of a particular process, we can first look up the the process id by following edge $\hat{a}_{pid}$ and then the process' name space by following edge $\hat{c}_{ns}$, or we can first look up the name space by following edge $\hat{b}_{ns}$ and then the process id by following edge $\hat{d}_{pid}$. For a given pair of process id and name space, the shared node $\hat{w}$ in the decomposition shape assures us we will get the same result regardless of which path we take.

$\hat{u}, \hat{v}$	decomposition vertices	$u_i, v_j, \mathcal{V}$	object names
$\hat{f}, \hat{\mathbf{f}}$	abstract fields	$f_i, f_j, \mathbf{f}$	concrete fields
$e ::= \mathsf{nil} \mid v_i$	expressions	$\omega ::= u_i.f \mapsto e$	heap assertions
$\pi \subseteq 2^{\hat{\mathbf{f}}} \to 2^{\hat{\mathbf{f}}}$	placements		

$$t ::= \mathsf{write}(u_i.f, e) \mid \mathsf{observe}(u_i.f) = e \mid \mathsf{read}(u_i.f) = e$$
$$\mid v_i = \mathsf{new}\ \hat{v} \mid \mathsf{lock}\ v_i \mid \mathsf{unlock}\ v_i \qquad \text{transaction ops.}$$

Fig. 10. Decomposition transactions

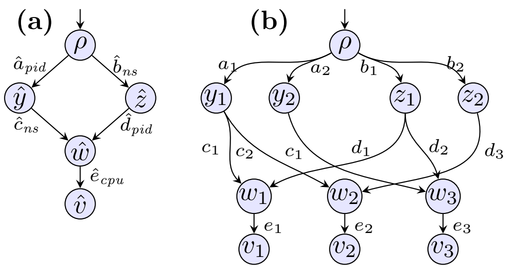

Fig. 9. (a): A decomposition heap shape, and **(b)**: a decomposition heap that is an instance of decomposition heap shape (a).

A decomposition shape is a static description of a class of heaps. Let $\mathsf{in}(\hat{v})$ be the set of field names incoming to $\hat{v}$ in a decomposition shape and let $\mathsf{out}(\hat{v})$ be the set of outgoing field names. A heap (V, E) is an *instance* of a decomposition shape $\hat{d}$ if

- every vertex in V is an instance v_i of some vertex $\hat{v} \in \hat{V}$,
- every edge $(u_i, f_j, v_k) \in E$ is an instance of some $(\hat{u}, \hat{f}, \hat{v}) \in \hat{E}$, and
- every vertex v_i has exactly one instance f_i of every incoming edge $\hat{f} \in \mathsf{in}(\hat{v})$.

The last condition provides a bound on the in-degree of a vertex, which is the key to applying path-based lock placements to decomposition heaps. Figure 9(b) shows a heap that is an instance of the process scheduler decomposition shape of Figure 9(a). The nodes are objects in memory. Every edge $\hat{f}$ from a vertex $\hat{u}$ to a vertex $\hat{v}$ of the decomposition shape has a corresponding set of edges $\{f_1, f_2, \cdots\}$ outgoing from any instance u_i of $\hat{u}$ in a decomposition heap. Intuitively, each vertex (object) u has a container data structure called f that contains references to a set of instances of $\hat{v}$. For example in Figure 9(b), the root object ρ has a set of process id's (the a_i) and a set of name spaces (the b_i). Note how the decomposition shape in Figure 9(a) is replicated across a number of different instances in Figure 9(b) with the stated sharing properties.

The well-lockedness rules defined below quantify over all paths that are a suffix of a particular path $\mathbf{f}$. To keep our transaction-language small, we impose an additional requirement that each the set of possible instances f_i of each abstract edge $\hat{f}$ be drawn from a bounded set; that is $i \in \{1, \ldots, k\}$ for some k. The bounded set restriction can be lifted by extending the transaction language with an iteration operation that allows a transaction to iterate over all instances of an edge from a vertex; the addition of iteration gives another way for the rules to conclude the a fact holds for all instances of an abstract edge $\hat{f}$.

The transaction operations on DAGs are almost identical to the set of operations on trees (Section 3). The transaction operations, shown in Figure 10, are: write an expression e (either nil or some v_k to field f_j of object u_i ($\mathsf{write}(u_i.f_j, e)$)), a possibly unstable read of field f_j of object u_i yielding result e ($\mathsf{read}(u_i.f_j) = e$), a stable observation of field f_j of object u_i yielding e ($\mathsf{observe}(u_i.f_j) = e$), allocation of a fresh object of type $\hat{v}$ ($v_i = \mathsf{new}\ \hat{v}$), locking an object ($\mathsf{lock}\ \ v_i$), and unlocking an object ($\mathsf{unlock}\ \ v_i$).

4.1 Lock Placements

Lock placements are defined exactly as in the tree case: π is a function from non-empty heap paths to paths, which maps every edge in a heap to an object whose lock protects it. Because edges may now have multiple paths that reach them, a transaction must hold locks on all paths to an edge to perform a stable read or to write the edge.

Line	(a)	(b)
1:	lock ρ	read$(\rho.a_2) = y_2$
2:	read$(\rho.a_2) = y_2$	lock y_2
3:	read$(\rho.b_2) = z_2$	read$(\rho.a_2) = y_2$
4:	observe$(\rho.a_2) = y_2$	observe$(\rho.a_2) = y_2$
5:	observe$(\rho.b_2) = z_2$	read$(\rho.b_2) = z_2$
6:	read$(y_2.c_7) =$ nil	lock z_2
7:	read$(z_2.d_5) =$ nil	read$(\rho.b_2) = z_2$
8:	$w_4 =$ new $\hat{w}$	observe$(\rho.b_2) = z_2$
9:	write$(y_2.c_7, w_4)$	read$(y_2.c_7) =$ nil
10:	write$(z_2.d_5, w_4)$	read$(z_2.c_5) =$ nil
11:	unlock ρ	$w_4 =$ new $\hat{w}$
12:		write$(y_2.c_7, w_4)$
13:		write$(z_2.d_5, w_4)$
14:		unlock z_2
15:		unlock y_2

Fig. 11. Example transactions that add a new node w_4 with access paths a_2c_7 and b_2d_5 to the decomposition heap instance shown in Figure 9(b), under (a) lock placement π_1 defined in Equation (5), and (b) lock placement π_3 defined in Equation (6).

We now illustrate some of the possibilities for lock placements on decomposition heaps. For our standard first example, by setting

$$\pi_1(\mathbf{f}) = \epsilon \text{ for all } \mathbf{f} \tag{5}$$

we can use a single lock at the root of the heap to protect every edge in a decomposition instance. Figure 11(a) shows a well-locked transaction that adds a fresh instance of $\hat{w}$, namely w_4, to the heap of Figure 9(b) under lock placement π_1. Acquiring the lock on ρ protects the entire heap graph; the transaction then adds w_4 under both the access path a_2c_7 and b_2d_5.

Another possibility is to use the placement

$$\pi_2(\mathbf{f}) = \begin{cases} \epsilon & \text{if } \mathbf{f} \in \{a_i, b_i, a_ic_j, a_id_j\} \\ a_ic_j & \text{if } \mathbf{f} = a_ic_je_k \\ b_id_j & \text{if } \mathbf{f} = b_id_je_k \end{cases}$$

which uses a lock at the root to protect instances of edges $\hat{a}$, $\hat{b}$, $\hat{c}$, and $\hat{d}$, and locks at instances of node $\hat{w}$ to protect instances of edge $\hat{e}$. Since instances of edge $\hat{e}$ can be reached by two different paths, and thus to observe $\hat{e}$ a transaction must acquire locks on both paths.

Finally, we can use a speculative lock placement. For example, we could protect instances of edges $\hat{a}$ and $\hat{b}$ using speculative locks placed at their targets, and use locks at y and z to protect edges $\hat{c}$, $\hat{d}$, and $\hat{e}$, via the lock placement

$$\pi_3(\mathbf{f}) = a_i \text{ if } a_i \preceq \mathbf{f}, \text{ and } b_j \text{ if } b_j \preceq \mathbf{f} \tag{6}$$

Figure 11(b) again shows a transaction that adds a fresh instance w_4 of node $\hat{w}$, this time under the speculative locking placement π_3.

4.2 Well-Locked Transactions

As in the case of tree heaps we represent the state of a transaction using three sets: Ω (the local stable heap), L (the held set of locks), and Γ. Sets Ω and L are defined as for trees, but we extend the definition of Γ to DAGs with bounded sharing.

The purpose of Γ is to track objects for which the transaction holds locks on incoming edges. In particular, if a transaction does not hold locks on some incoming edges to an object o, then there may be a path from the global heap to o and the transaction cannot rely on the stability of o's fields. Thus Γ is the transaction's view of the global heap and what other transactions might be able to do to objects of interest to the transaction.

(DNew)
$$\frac{\Omega' = \Omega \cup \{v_i.f_j \mapsto \mathsf{nil} \mid \hat{f} \in \mathsf{out}(\hat{v})\} \quad v_i \notin \mathrm{dom}\,\Omega \qquad \Gamma' = \Gamma[v_i \mapsto \mathsf{in}(\hat{v})] \qquad v_i \notin \mathrm{dom}\,\Gamma}{\Omega, \Gamma, L \vdash_\pi v_i = \mathsf{new}\ \hat{v}; \Omega', \Gamma', L}$$

(DLock)
$$\frac{v_i \notin L}{\Omega, \Gamma, L \vdash_\pi \mathsf{lock}\ v_i; \Omega, \Gamma, L \cup \{v_i\}}$$

(DUnlock)
$$\frac{v_i \in L \qquad L' = L \setminus \{v_i\} \quad (\Omega', \Gamma') = \lceil \Omega; \Gamma \mid L'; \pi \rceil \qquad \mathsf{balias}(\Omega, \Omega', \Gamma, \Gamma')}{\Omega, \Gamma, L \vdash_\pi \mathsf{unlock}\ v_i; \Omega', \Gamma', L'}$$

(DObserve)
$$\frac{(u_i.f_j \mapsto e) \in \Omega}{\Omega, \Gamma, L \vdash_\pi \mathsf{observe}(u_i.f_j) = e; \Omega, \Gamma, L}$$

(DReadUnstable)
$$\frac{u_i.f_j \notin \mathrm{dom}\,\Omega \quad \Omega' = \Omega \cup \{u_i.f_j \mapsto e\} \quad \neg\mathsf{locked}_\pi(u_i.f_j, \Omega', \Gamma, L)}{\Omega, \Gamma, L \vdash_\pi \mathsf{read}(u_i.f_j) = e; \Omega, \Gamma, L}$$

(DReadStable)
$$\frac{u_i.f_j \notin \mathrm{dom}\,\Omega \qquad \Omega' = \Omega \cup \{u_i.f_j \mapsto e\} \quad \mathsf{locked}_\pi(u_i.f_j, \Omega', \Gamma, L) \qquad \Gamma' = \begin{cases} \Gamma & \text{if } e = \mathsf{nil} \\ \Gamma[v_i \mapsto \Gamma(v_i) \cup \{\hat{f}\}] & \text{if } e = v_i \end{cases}}{\Omega, \Gamma, L \vdash_\pi \mathsf{read}(u_i.f_j) = e; \Omega', \Gamma', L}$$

$$(\text{DWrite})\ \frac{u_i.f_j \in \mathrm{dom}\,\Omega \qquad \Omega' = \Omega[u_i.f_j \mapsto e] \quad \Big(\forall \mathbf{g}, \mathbf{h}.\ (\Omega \vdash \mathbf{g} \sim u_i) \wedge \mathbf{g}f \preceq \pi(\mathbf{h}) \implies \mathsf{pathlocked}_\pi(\mathbf{h}, \Omega, L) \wedge \mathsf{pathlocked}_\pi(\mathbf{h}, \Omega', L)\Big)}{\Omega, \Gamma, L \vdash_\pi \mathsf{write}(u_i.f_j, e); \Omega', \Gamma, L}$$

Fig. 12. Well-locked decomposition operations: judgement $\Omega, \Gamma, L \vdash_\pi t; \Omega', \Gamma', L'$.

The global heap view Γ is a mapping from each vertex v_i in the heap to the subset of the incoming edge labels of the decomposition $\mathsf{in}(\hat{v})$ known to be absent from the global heap (i.e., either non-existent or locked by the transaction). We maintain the invariant that in the global heap there is at most one edge to any instance of a decomposition vertex $\hat{v}$ labeled with an instance of each $\hat{f} \in \mathsf{in}(\hat{v})$. If $\Gamma(v_i) = \emptyset$, then v_i may have an instance of each incoming edge in $\mathsf{in}(\hat{f})$ in the global heap. If $\Gamma(v_i) = \{\hat{f}\}$ then v has no incoming edge in the global heap labeled with an instance of $\hat{f}$. If $\Gamma(v) = \mathsf{in}(\hat{v})$ then v has no incoming edges from the global heap.

As before, we hold the lock on an edge reached via a path if we hold the path's corresponding lock placement, restricted to the heap:

$$\mathsf{pathlocked}_\pi(\mathbf{f}, \Omega, L) ::= \exists v_i \in L.\ \Omega \vdash \mathbf{g} \sim v_i \wedge \pi(\mathbf{f})|_\Omega = \mathbf{g},$$

where $\mathbf{f}_\Omega$ is the restriction of path $\mathbf{f}$ to heap Ω, defined in Section 3.

The judgement $\Omega, \Gamma \vdash \mathsf{exposed}(x)$ holds if there may be a path to vertex x in the heap that does not lie entirely in the stable observation set Ω; the judgement is defined by the inference rules:

$$\frac{\Gamma(v_k) \neq \mathsf{in}(\hat{v})}{\Omega, \Gamma \vdash \mathsf{exposed}(v_k)} \qquad \frac{\Omega, \Gamma \vdash \mathsf{exposed}(u_i) \wedge (u_i.f_j \mapsto v_k) \in \Omega}{\Omega, \Gamma \vdash \mathsf{exposed}(v_k)}$$

We hold the lock on a field $x.f$ if we hold a lock on that field on every path in the local heap, and there are no paths to x outside the local heap.

$$\mathsf{locked}_\pi(v_i.f_j, \Omega, \Gamma, L) ::= \neg\mathsf{exposed}(v_i) \wedge \forall \mathbf{g}.\ (\Omega \vdash \mathbf{g} \sim v_i \implies \mathsf{pathlocked}_\pi(\mathbf{g}f_j, \Omega, L))$$

The judgement $\Omega, \Gamma, L \vdash_\pi t; \Omega', \Gamma', L'$ defined by the rules in Figure 12 describes the class of *well-locked decomposition operations*, analogous to the class of well-locked tree operations of Section 3. The judgement holds if a transaction executing operation t under local heap Ω, global heap approximation Γ, and locks L yields an updated local heap Ω', global heap approximation Γ', and lock set L'. The (DNEW) rule states that the fields of a newly allocated object v_i point to nil; furthermore there can be no heap paths to a freshly allocated object so assertions about the fields of v_i are stable and $\Gamma(v_i) = \mathsf{in}(\hat{v})$. The (DLOCK) rule allows a transaction to acquire a lock that it does not hold at any time.

The (DUNLOCK) rule allows a transaction to release any lock that it holds; the rule applies the stabilization operation to remove any newly unstable facts from Ω. Similar to the tree case, the *stabilization* (Ω', Γ') of a local heap Ω_0 and global heap Γ_0 under locks L and placement π, written $(\Omega', \Gamma') = \lceil \Omega_0; \Gamma_0 \mid L; \pi \rceil$, is the limit of the monotonically decreasing sequence:

$$\Omega_{i+1} = \{u_j.f_k \mapsto e \in \Omega_i \mid \mathsf{locked}_\pi(u_j.f_k, \Omega_i, \Gamma_i, L)\}$$
$$\Gamma_{i+1} = \Gamma_i \setminus \{v_k \mapsto \hat{f} \mid u_i.f_j \mapsto v_k \in \Omega_i \setminus \Omega_{i+1}\}$$

To ensure that there is at most instance of any edge label $\hat{f} \in \mathsf{in}(\hat{v})$ in the global heap, the rule requires the *bounded alias* condition

$$\mathsf{balias}(\Omega, \Omega', \Gamma, \Gamma') ::= \forall v_k.\ |\{u_i.f_j \mid (u_i.f_j \mapsto v_k) \in \Omega \setminus \Omega'\}| = \begin{cases} 1 & \text{if } \hat{f} \in \Gamma(v_k) \setminus \Gamma'(v_k) \\ 0 & \text{otherwise.} \end{cases}$$

The bounded alias condition ensures that a transaction may only release an edge with abstract label $\hat{f}$ to a node v_k into the global heap if there are no other edges to v_k labeled $\hat{f}$ in the global heap ($\hat{f} \in \Gamma(v_k)$). Further the condition forbids releasing two pointers with the same label $\hat{f}$ to the same node v_k into the global heap.

Rule (DOBSERVE) states that a transaction may logically observe stable facts about the heap. The (DREADUNSTABLE) rule allows a transaction to read a value speculatively at any time, however unstable reads do not update Ω or Γ. A transaction may perform a stable read of a pointer if it holds the appropriate lock, transferring the pointer from the global heap into Ω and updating Γ accordingly. Finally, a transaction may write to a field if it holds the associated lock, and further holds locks on any edges whose logical/physical mapping may implicitly change as a result of the update.

A transaction $\mathbf{T} = t^1 \dots t^k$ is *well-locked* if there exists a sequence of lock sets L^i, observation sets Ω^i, and global heap sets Γ^i such that

$$L^0 = L^k = \emptyset, \quad \Omega^0 = \emptyset, \quad \Gamma^0 = \emptyset, \text{ and } \Omega^{i-1}, \Gamma^{i-1}, L^{i-1} \vdash_\pi t^i; \Omega^i, \Gamma^i, L^i \text{ for } 1 \le i \le k.$$

Lemma 5. *Let* $\mathbf{s}$ *be a valid schedule of a set of well-locked transactions* $\{\mathbf{T}_1, \dots, \mathbf{T}_k\}$. *Let* Ω_i^j, Γ_i^j, *and* L_i^j *be the set of observations, global heaps, and locks of each transaction after schedule step* j. *Let* h^j *be the heap after schedule step* j, *and suppose the part of* h^0 *reachable from the root is a tree. Then for all time steps* j:

- *the lock sets* $\{L_i^j\}_{i=1}^k$ *are disjoint,*
- *the observation sets* $\{\Omega_i^j\}_{i=1}^k$ *are stable, disjoint, and heap* h^j *is an extension of each* $\{\Omega_i^j\}_{i=1}^k$, *and*
- *the global non-alias sets* $\{\Gamma_i\}_{i=1}^k$ *are disjoint.*
- *Let heap* h *be the heap* h^j *less edges present in the local heaps* $\{\Omega_i^j\}_{i=1}^k$. *Then for every vertex* $v \in h$ *and edge label* $\hat{f} \in \mathsf{in}(\hat{v})$ *either there is exactly one edge labeled*

with an instance of $\hat{f}$ pointing to v in h, or $\hat{f} \in \Gamma_i^j$ for some i and there are no edges labeled with an instance of $\hat{f}$ pointing to v in h.

Finally, we have a logical serializability similar to Lemma 2 and Lemma 4.

Lemma 6. *Any valid schedule of a set of well-locked, logically two-phase decomposition transactions $\{\mathbf{T}_1, \ldots, \mathbf{T}_k\}$ is conflict-equivalent to a serial logical schedule.*

And, again, as in the flat heap case, these results can be extended to shared/exclusive locks using the approach in Section 2.4.

5 Related Work

Two-phase locking was originally introduced in the context of transactions operating over abstract entities, each with its own associated lock [9]. The core technical idea of this paper is that we can use two-phase locking to show serializability of a wide class of locking strategies by adding a layer of indirection between logical locks, which are the entities that are the subject of the original two-phase locking protocol, and the physical locks that implement them.

Various authors have investigated techniques for inferring locks to implement atomic sections [16,14,8,12,5,6,20]. A related problem is automatically optimizing programs with explicit locking by combining multiple locks into one [7]. A key part of this class of work is constructing a mapping from program objects to the locks that protect them, similar to our lock placement language. The lock placements we propose are much more flexible; in particular existing formalisms cannot handle the class of path placements we propose in this paper, such as speculative locks, or lock placements that vary with heap updates. A possible future application of our methods is extending lock inference techniques to take advantage of the additional expressive power of our techniques.

A novel feature of our proposal is that we can reason about speculative lock placements. Speculative locking is used in practice in highly concurrent libraries and has appeared in the literature in the context of software transactional memory [4,2,3]. Although we present our ideas in the context of speculative placements of pessimistic locks, the idea of a lock placement can also be used to reason about speculative placements of optimistic STM metadata.

A variety of locking protocols have been proposed in the literature that extend two-phase locking to handle dynamically changing heaps and to allow early release. Notable examples include the dynamic tree locking and dynamic DAG locking protocols [1], and domination locking [10]. Existing protocols use the lock on each object to protect that object's fields, whereas a primary goal of our work is to investigate a more flexible space of mappings. We do not address early release in this paper; early release is orthogonal to the issues of lock placement.

Concurrent extensions of separation logic, such as Concurrent Separation Logic [17], RGSep [18] and work on storable locks [11] allow local reasoning about programs with shared mutable state that is accessed concurrently. The concept of a stable set and stabilization is related to rely-guarantee logic [15] and its subsequent developments [19]. Our work complements work on direct reasoning about concurrent code; we propose a locking protocol, parameterized by a declarative lock placement, by which we can show conflict-serializability, thereby removing the need for direct concurrent reasoning for programs that obey the locking protocol.

6 Conclusion

We have presented a formalization of lock placements, showing that such diverse concepts as lock granularity, speculative locks, lock splitting and merging, and dynamically changing lock assignments can all be understood as examples of a lock placement that maps each heap field to a lock that guards it. We have also identified the key concept of a stable set of mutually supporting locks and heap facts, where the set of locks protect the heap facts and the set of heap facts preserve the lock placement. We have used these two concepts to develop a series of proof system for showing that transactions are well-locked and therefore serializable, applying these technique to flat heaps, tree-structured heaps, and a family of DAG-structured heaps with bounded degree.

References

1. Attiya, H., Ramalingam, G., Rinetzky, N.: Sequential verification of serializability. In: POPL. pp. 31–42. ACM, New York, NY, USA (2010)
2. Bronson, N.G.: Composable Operations on High-Performance Concurrent Collections. Ph.D. thesis, Stanford University (2011)
3. Bronson, N.G., Casper, J., Chafi, H., Olukotun, K.: A practical concurrent binary search tree. In: PPoPP (2010)
4. Bronson, N.G., Casper, J., Chafi, H., Olukotun, K.: Transactional predication: high-performance concurrent sets and maps for STM. In: PODC. pp. 6–15. ACM, New York, NY, USA (2010)
5. Cherem, S., Chilimbi, T., Gulwani, S.: Inferring locks for atomic sections. In: PLDI. pp. 304–315. ACM, New York, NY, USA (2008)
6. Cunningham, D., Gudka, K., Eisenbach, S.: Keep off the grass: Locking the right path for atomicity. In: Hendren, L. (ed.) Compiler Construction, LNCS, vol. 4959, pp. 276–290. Springer Berlin / Heidelberg (2008)
7. Diniz, P.C., Rinard, M.C.: Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs. Journal of Parallel and Distributed Computing 49(2), 218–244 (1998)
8. Emmi, M., Fischer, J.S., Jhala, R., Majumdar, R.: Lock allocation. In: POPL. pp. 291–296. ACM, New York, NY, USA (2007)
9. Eswaran, K.P., Gray, J.N., Lorie, R.A., Traiger, I.L.: The notions of consistency and predicate locks in a database system. Commun. ACM 19, 624–633 (Nov 1976)
10. Golan-Gueta, G., Bronson, N., Aiken, A., Ramalingam, G., Sagiv, M., Yahav, E.: Automatic fine-grained locking using shape properties. In: OOPSLA (2011)
11. Gotsman, A., Berdine, J., Cook, B., Rinetzky, N., Sagiv, M.: Local reasoning for storable locks and threads. In: APLAS. pp. 19–37 (2007)
12. Halpert, R.L., Pickett, C.J.F., Verbrugge, C.: Component-based lock allocation. In: PACT. pp. 353–364. IEEE Computer Society, Washington, DC, USA (2007)
13. Hawkins, P., Aiken, A., Fisher, K., Rinard, M., Sagiv, M.: Data representation synthesis. In: PLDI. pp. 38–49. ACM, New York, NY, USA (2011)
14. Hicks, M., Foster, J.S., Pratikakis, P.: Lock inference for atomic sections. In: Workshop on Languages, Compilers and Hardware Support for Transactional Computing (2006)
15. Jones, C.: Development methods for computer programs including a notion of interference. Ph.D. thesis, Oxford University (1981)
16. McCloskey, B., Zhou, F., Gay, D., Brewer, E.: Autolocker: synchronization inference for atomic sections. In: POPL. pp. 346–358. ACM, New York, NY, USA (2006)
17. O'Hearn, P.W.: Resources, concurrency, and local reasoning. Theoretical Computer Science 375(1–3), 271–307 (2007)
18. Vafeiadis, V., Parkinson, M.J.: A marriage of rely/guarantee and separation logic. In: CONCUR (2007)
19. Wickerson, J., Dodds, M., Parkinson, M.: Explicit stabilisation for modular rely-guarantee reasoning. LNCS, vol. 6012, pp. 610–629 (2010)
20. Zhang, Y., Sreedhar, V., Zhu, W., Sarkar, V., Gao, G.: Minimum lock assignment: A method for exploiting concurrency among critical sections. In: Languages and Compilers for Parallel Computing, LNCS, vol. 5335, pp. 141–155 (2008)

A Proofs

Proof (Lemma 1). The result is immediate for a schedule of length 0 since the observation sets of each transaction must necessarily be empty by the definition of well-lockedness. For the inductive case, assume the result holds for all schedules of length n and consider a schedule with length $n+1$. Suppose without loss of generality that operation $n+1$ is an operation t_1 performed by transaction 1. Since transaction 1 is well-locked, we know that $\Omega_1^n, L_1^n \vdash_\pi t_1; \Omega_1^{n+1}, L_1^{n+1}$ holds and we perform a case analysis of the possible derivations:

- Rule (FLock): The disjointness of lock sets is an immediate consequence of the validity of schedule s. Acquiring a lock does not change the stability of a transaction's observation set, and cannot destabilize the stable observation set of any transaction, so the stability, disjointness, and extension properties follow from the induction hypothesis.
- Rule (FUnlock): Releasing a lock yields disjoint lock sets by the induction hypothesis. It is always sound to discard anything from a transaction's stable observation set Ω, and stability of the resulting observations Ω' holds by definition of the stabilization operator. Disjointness and extension hold by the induction hypothesis and since $\Omega' \subseteq \Omega$.
- Rule (FObserve): Immediate.
- Rule (FReadUnstable): Immediate.
- Rule (FReadStable): The observation sets of each transaction must be the same before and after the read, with the exception of Ω_1, which contains the additional result of the reading m. We must have $\mathsf{locked}_\pi(m, \Omega_1, L_1)$ so by definition we have $(l, \phi) \in \pi(m)$ such that $l \in L_1$ and $\Omega_1 \vdash \phi$. Disjointness follows by the disjointness of the guards ϕ and the disjointness of local heaps due to the induction hypothesis. Extension follows by the validity of the schedule and the induction hypothesis.
- Rule (FWrite): We have $\mathsf{locked}_\pi(m, \Omega_1, L_1)$ because the rule requires that $m \in \operatorname{dom} \Omega$ and hence m must be stable; hence we know that no other transaction may have m in its stable observation set by the disjointness of guards and heaps. Further, the rule requires that the transaction must hold any locks whose association with memory locations may change as a consequence of the write, and hence the stability of each Ω_i holds. Finally, extension follows from the inductive hypothesis.

Proof (Lemma 3). By induction on the length of the schedule. The result is immediate for a schedule of length 0. Suppose the result is true for any schedule of length n, and consider any schedule of length $n+1$. Without loss of generality, assume transaction number 1 performs the additional operation t^{n+1}. Since transaction 1 is well-locked, we know $\Omega_1^n, \Gamma_1^n, L_1^n \vdash_\pi t; \Omega_1^{n+1}, \Gamma_1^{n+1}, L_1^{n+1}$ holds. There are several possible derivations; we include only the cases that differ significantly from the proof of Lemma 1:

- Rule (TNew): The result about lock sets is immediate. The concrete semantics of the new operator guarantee that the result x of allocation is fresh, and hence there are no pointers to x in the heap. It follows that all of the fields of x are stable and known only to the current transaction. The result about the observation sets then follows from the induction hypothesis. The result about local heaps and global heaps is a consequence of the freshness of x and the induction hypothesis.
- Rule (TUnlock): Disjointness of lock sets is immediate. Stability of the current transaction holds by definition, and for other transactions by the induction hypothesis; unlocking a lock cannot affect other transactions. Disjointness and subsetting hold by the induction hypothesis and the definition of stabilization. The forest condition ensures that a transaction can evict a pointer to a node x from its local heap only if $x \in \Gamma$, and hence the global forest condition is preserved given the induction hypothesis.

- Rule (TReadStable): The lock set result is immediate. Stability of each transaction's observation set holds since the rule requires that the transaction hold locks that ensure the newly read heap assertion is stable ($\mathsf{locked}_\pi(x.f, \Omega, \Gamma, L)$) and the induction hypothesis guarantees all other heap assertions are stable. The lockedness of $x.f$ ensures that the current transaction must hold the lock for $x.f$ on every path. No other transaction can have $x.f$ as part of its stable observation set because of the disjointness of observation sets from the induction hypothesis. The result about Γ and Ω follows from the forest condition on the heap; if we acquired a stable reference to a node y then it must have been the unique reference in the global heap, and hence no other transaction may have $y \in \Gamma$ by the induction hypothesis.
- Rule (TWrite): The lock set result is immediate. The newly written memory location must be stable since the transaction requires that $x.f$ be present in Ω and from the condition that ensures the transaction holds any locks reachable from $x.f$ that may change identity; the latter condition also ensures that the write cannot affect the stable observation sets of any other transaction. The disjointness and extension conditions hold by the induction hypothesis. The result about local and global heaps is immediate from the induction hypothesis; a transaction may freely update its local heap since the tree-structure condition need only be maintained at at unlock time.

B Concrete Semantics of Heap Operations

(CLock)
$$\frac{l \notin L}{h, L, \mathsf{lock}\ \ l \rightarrow h, L \cup \{l\}}$$

(CUnlock)
$$\frac{l \in L}{h, L, \mathsf{unlock}\ \ l \rightarrow h, L \setminus \{l\}}$$

(CRead)
$$\frac{(m \mapsto b) \in h}{h, L, \mathsf{read}(m) = b \rightarrow h, L}$$

(CWrite)
$$\frac{h' = h[m \mapsto b]}{h, L, \mathsf{write}(m, b) \rightarrow h', L}$$

(CObserve)
$$\frac{}{h, L, \mathsf{observe}(m) = b \rightarrow h, L}$$

Fig. 13. Concrete semantics of flat heap transactions. $h, L, t \rightarrow h', L'$ holds if executing operation t on heap h with locks held by any transaction L yields an updated heap h' with new locks held L'.

Figure 13 describes the concrete semantics of transaction operations over concrete heaps. The judgment $h, L, t \rightarrow h', L$ holds if executing operation t in global heap h and with global locks taken L yields an updated heap h' and update global lock set L'. Rule (CLock) states that a transaction may acquire any global lock l not already held by any transaction, and the lock l is marked as locked. Rule (CUnlock) allows any transaction to release any lock that is held. There is no requirement that a transaction release only the locks it acquired; such a requirement is enforced by the static well-lockedness rules, not by the concrete semantics.

Rule (CRead) allows any transaction to atomically read the state of any heap cell, whereas rule (CWrite) allows any transaction to atomically update any heap cell. Finally the (CObserve) rule states that every observe operation is physically a no-op; observe is a logical operation that identifies facts relevant to the serializability proof of a transaction; physically it does nothing.

The concrete semantics of tree and DAG operations are similar and we omit them.