Automatic Fine-Grain Locking using Shape Properties

Guy Golan-Gueta
Tel Aviv University
ggolan@tau.ac.il

Nathan Bronson
Stanford University
nbronson@cs.stanford.edu

Alex Aiken
Stanford University
aiken@cs.stanford.edu

G. Ramalingam
Microsoft Research
grama@microsoft.com

Mooly Sagiv
Tel Aviv University
msagiv@tau.ac.il

Eran Yahav *
Technion
yahave@cs.technion.ac.il

Abstract

We present a technique for automatically adding fine-grain locking to an abstract data type that is implemented using a dynamic forest —i.e., the data structures may be mutated, even to the point of violating forestness temporarily during the execution of a method of the ADT. Our automatic technique is based on *Domination Locking*, a novel locking protocol. Domination locking is designed specifically for software concurrency control, and in particular is designed for object-oriented software with destructive pointer updates. Domination locking is a strict generalization of existing locking protocols for dynamically changing graphs.

We show our technique can successfully add fine-grain locking to libraries where manually performing locking is extremely challenging. We show that automatic fine-grain locking is more efficient than coarse-grain locking, and obtains similar performance to hand-crafted fine-grain locking.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages; E.1 [*Data Structures*]: Trees

General Terms Theory, Algorithms, Languages, Performance

Keywords Concurrency, Locking Protocol, Synthesis, Serializability, Atomicity, Reduction

* Deloro Fellow

OOPSLA'11, October 22–27, 2011, Portland, Oregon, USA.

1. Introduction

The proliferation of multi-core processors and diminishing returns in single-threaded performance have increased the need for scalable multi-threading. Concurrent data structures are a key ingredient of parallel programs: these implement common data structures, such as search trees, but permit multiple threads to concurrently perform operations on a shared data structure instance. A commonly used correctness criterion for such data structures is that each operation should appear to execute atomically, which greatly simplifies reasoning about programs that use these data structures.

In this paper, we consider the problem of turning a sequential data structure into a concurrent data structure. One of the main challenges in this problem is to guarantee atomicity of concurrent operations in a scalable way, restricting parallelism only where necessary. Furthermore, we are interested in systematic techniques that are broadly applicable, rather than techniques specific to a single data structure.

Existing approaches to this problem are limited. Optimistic transactional memory [19, 22] provides limited benefit due to high overhead and its limited ability to handle irreversible operations (e.g., I/O). Automatic lock inference techniques (e.g., [11, 12, 15, 17, 24, 29]) are traditionally based on the two-phased locking protocol [16] that does not permit early lock release and therefore limits parallelism.

Fine-Grain Locking One way to achieve scalable multi-threading is to use fine-grain locking. In fine-grain locking, one associates, e.g., each object with its own lock, permitting multiple operations to simultaneously operate on different parts of a data-structure. Reasoning about fine-grain locking is challenging and error-prone. As a result, programmers often resort to coarse-grain locking, leading to limited scalability.

The Problem We address the problem of automatically adding fine-grain locking to a *module*. A module encapsulates shared data with a set of procedures, which may be invoked by concurrently executing threads. Given the code

of a module, our goal is to add correct locking that permits a high degree of parallelism. Specifically, we are interested in locking in which each shared object has its own lock, and locks may be released before the end of the computation.

Our main insight is that we can use the restricted topology of pointer data structures to simplify reasoning about fine-grain locking and automatically infer efficient and correct fine-grain locking.

Domination Locking We define a new locking protocol called *Domination Locking* (DL). Domination Locking is a set of conditions that guarantee atomicity and deadlock-freedom. Domination Locking is designed to handle dynamically manipulated recursive data-structures by leveraging natural domination properties of dynamic data structures. Domination locking is a strict generalization of several related fine-grain locking protocols such as hand-over-hand locking [4, 6], and dynamic DAG locking [4, 10].

Automatic Fine-Grain Locking We present an automatic technique to enforce the conditions of Domination Locking. The technique is applicable to modules where the shape of the shared memory is a forest. The technique allows the shape of the heap to *change dynamically* as long as the shape is a forest between invocations of module operations. In contrast to existing lock inference techniques, which are based on two-phased locking, our technique is able to release locks at early points of the computation.

Finally, as we demonstrate in Section 4 and Section 5, our technique adds effective fine-grain locking in several practical data-structures where it is extremely hard to manually produce similar locking. Our examples include balanced search-trees [3, 18], a self-adjusting heap [36] and specialized data-structure implementations [5, 31].

1.1 Motivating Example

Consider a module that implements the Treap data structure [3]. A Treap is a search tree that is simultaneously a binary search tree (on the `key` field) and a heap (on the `priority` field). If priorities are assigned randomly the resulting structure is equivalent to a random binary search tree, providing good asymptotic bounds for all operations. The Treap implementation consists of three procedures: `insert`, `remove` and `lookup`. Manually adding fine-grain locking to the Treap's code, is challenging since it requires considering many subtle details in the context of concurrency.

For example, consider the Treap's `remove` operation shown in Fig. 1. To achieve concurrent execution of its operations, we must release the lock on the root, while an operation is still in progress, once it is safe to do so. Either of the loops (starting at Lines 4 or 12) can move the current context to a subtree, after which the root (and, similarly, other nodes) should be unlocked. Several parts of this procedure implement tree rotations that change the order between the Treap's nodes, complicating any correctness reasoning that depends on the order between nodes. Fig. 2 shows an

```
1  boolean remove(Node par, int key) {
2   Node n = null;
3   n = par.right; // right child has root
4   while (n != null && key != n.key) {
5    par = n;
6    n = (key < n.key) ? n.left : n.right;
7   }
8   if (n == null)
9    return false; // search failed, no change
10  Node nL = n.left;
11  Node nR = n.right;
12  while (true) { // n is the node to be removed
13   Node bestChild = (nL == null ||
14      (nR != null && nR.prio > nL.prio)) ? nR : nL;
15   if (n == par.left)
16    par.left = bestChild;
17   else
18    par.right = bestChild;
19   if (bestChild == null)
20    break; // n was a leaf
21   if (bestChild == nL) {
22    n.left = nL.right; // rotate nL to n's spot
23    nL.right = n;
24    nL = n.left;
25   } else {
26    n.right = nR.left; // rotate nR to n's spot
27    nR.left = n;
28    nR = n.right;
29   }
30   par = bestChild;
31  }
32  return true;
33 }
```

Figure 1. Removing an element from a treap by locating it and then rotating it into a leaf position.

example of manual fine-grain locking of the Treap remove operation. Manually adding fine-grain locking to the code took an expert several hours and was an extremely error-prone process. In several cases, the expert locking released a lock too early, resulting in an incorrect concurrent algorithm (e.g., the release operation in Line 28).

Our technique is able to automatically produce fine-grain concurrency in the Treap's code, by relying on its tree shape. This is in contrast to existing alternatives, such as manually enforcing hand-over-hand locking, that require deep understanding of code details.

1.2 Overview of Our Approach

In this section, we present an informal brief description of our approach.

Domination Locking We define a new locking protocol, called *Domination Locking* (DL). DL is a set of conditions that are designed to guarantee atomicity and deadlock freedom for operations of a well-encapsulated module.

DL differentiates between a module's *exposed* and *hidden* objects: exposed objects (e.g., the Treap's root) act as the intermediary between the module and its clients, with pointers to such objects being passed back and forth between the module and its clients, while the clients are completely unaware of hidden objects (e.g., the Treap's intermediate nodes). The protocol exploits the fact that all operations must begin with one or more exposed objects and traverse the heap-graph to reach hidden objects. In essence, exposed

```
1  boolean remove(Node par, int key) {
2    Node n = null;
3    acquire(par);
4    n = par.right;
5    if(n != null) acquire(n);
6    while (n != null && key != n.key) {
7      release(par);
8      par = n;
9      n = (key < n.key) ? n.left : n.right;
10     if(n != null) acquire(n);
11   }
12   if (n == null){ release(par); return false; }
13   Node nL = n.left; if(nL != null) acquire(nL);
14   Node nR = n.right; if(nR != null) acquire(nR);
15   while (true) {
16     Node bestChild = (nL == null ||
17        (nR != null && nR.prio > nL.prio)) ? nR : nL;
18     if (n == par.left)
19       par.left = bestChild;
20     else
21       par.right = bestChild;
22     release(par);
23     if (bestChild == null)
24       break;
25     if (bestChild == nL) {
26       n.left = nL.right;
27       nL.right = n;
28       // release(nL); // an erroneous release statment
29       nL = n.left;
30       if(nL != null) acquire(nL);
31     } else {
32       n.right = nR.left;
33       nR.left = n;
34       nR = n.right;
35       if(nR != null) acquire(nR);
36     }
37     par = bestChild;
38   }
39   return true;
40 }
```

Figure 2. Treap's `remove` code with manual fine-grain locking.

objects own hidden objects. This implements an ownership scheme which permits ownership transfer.

The protocol requires the exposed objects passed as parameters to an operation to be locked in a fashion similar to two-phase-locking. However, hidden objects are handled differently. A thread is allowed to acquire a lock on a hidden object if the locks it holds *dominate* the hidden object. (A set S of objects is said to dominate an object u if all paths (in the heap-graph) from an exposed object to u contains some object in S.) In particular, hidden objects can be locked even after other locks have been released, thus enabling early release of other locked objects (hidden as well as exposed).

This simple protocol generalizes several fine-grain locking protocols defined for dynamically changing graphs [4, 6, 10] and is applicable in more cases (i.e., the conditions of DL are weaker). We use the DL's conditions as the basis for our automatic technique.

Automatic Locking of Forest-Based Modules Our technique is able to automatically enforce DL, in a way that releases locks at early points of the computation. Specifically, the technique is applicable for modules whose heap-graphs form a forest at the end of any complete sequential execution (of any sequence of operations).

Note that existing shape analyses, for sequential programs, can be used to automatically verify if a module satisfies this precondition (e.g., [34, 40]). In particular, we avoid the need for explicitly reasoning on concurrent executions.

For example, the Treap is a tree at the end of any of its operations, when executed sequentially. Note that, during some of its operation (`insert` and `remove`) its tree shape is violated by a node with multiple predecessors (caused by the tree rotations).

Our technique uses the following locking scheme: a procedure invocation maintains a lock on the set of objects directly pointed to by its local variables (called the immediate scope). When an object goes out of the immediate scope of the invocation (i.e., when the last variable pointing to that object is assigned some other value), the object is unlocked if it has (at most) one predecessor in the heap graph (i.e., if it does not violate the forest shape). If a locked object has multiple predecessors when it goes out of the immediate scope of the invocation, then it is unlocked eventually when the object has at most one predecessor. The forest-condition guarantees that every lock is eventually released.

To realize this scheme, we use a pair of reference counts to track incoming references from the heap and local variables of the current procedure. All the updates to the reference count can be done easily by instrumenting every assignment statement, allowing a relatively simple compile-time transformation. While we defer the details of the transformation to Section 4, Fig. 3 shows the transformed implementation of `remove` (from Fig. 1). `ASNL` and `ASNF` are macros that perform assignment to a local variable and a field, respectively, update reference counts, and conditionally acquire or release locks according to the above locking scheme.

1.3 Contributions

The contributions of this paper can be summarized as follows:

- We introduce a new locking protocol entitled *Domination Locking*. We show that domination locking can be enforced and verified by considering only sequential executions [4]: if domination locking is satisfied by all sequential executions, then atomicity and deadlock freedom are guaranteed in all executions, including non-sequential ones.
- We present an automatic technique to generate fine-grain locking by enforcing the domination locking protocol for modules where the heap graph is guaranteed to be a forest in between operations. Our technique can handle any temporary violation of the forest shape constraint, including temporary cycles.
- We present an initial performance evaluation of our technique on several examples, including balanced search-trees [3, 18], a self-adjusting heap [36] and specialized

```
1  boolean remove(Node par, int key) {
2    Node n = null;
3    Take(par);
4    ASNL(n, par.right);
5    while (n != null && key != n.key) {
6      ASNL(par, n);
7      ASNL(n, (key < n.key) ? n.left : n.right);
8    }
9    if (n == null) {
10     ASNL(par, null);
11     ASNL(n, null);
12     return false;
13   }
14   Node nL = null; ASNL(nL, n.left);
15   Node nR = null; ASNL(nR, n.right);
16   while (true) {
17     Node bestCh = null; ASNL(bestCh, (nL == null ||
18        (nR != null && nR.prio > nL.prio)) ? nR : nL);
19     if (n == par.left)
20       ASNF(par.left, bestCh);
21     else
22       ASNF(par.right, bestCh);
23     if (bestCh == null) {
24       ASNL(bestCh, null);
25       break;
26     }
27     if (bestCh == nL) {
28       ASNF(n.left, nL.right);
29       ASNF(nL.right, n);
30       ASNL(nL, n.left);
31     } else {
32       ASNF(n.right, nR.left);
33       ASNF(nR.left, n);
34       ASNL(nR, n.right);
35     }
36     ASNL(par, bestCh);
37     ASNL(bestCh, null);
38   }
39   ASNL(par, null); ASNL(n, null); ASNL(nL, null);
40   ASNL(nR, null);
41   return true;
42 }
```

Figure 3. Augmenting `remove` with macros to dynamically enforce domination locking.

data-structure implementations [5, 31]. The evaluation shows that our automatic locking provides good scalability and performance comparable to hand crafted locking (for the examples where hand crafted locking solutions were available).

- We discuss extensions and additional applications of our suggestions.

2. Preliminaries

Our goal is to augment a module with concurrency control that guarantees strict conflict-serializability [33]. In this section we formally define what a module is and the notion of strict conflict-serializability for modules.

Syntax and Informal Semantics A module defines a set of types and a set of procedures that may be invoked by clients of the module, potentially concurrently. A type consists of a set of fields of type *boolean*, *integer*, or *pointer* to a user-defined type. The types are private to the module: an object of a type T defined by a module M can be allocated or dereferenced only by procedures of module M. However, pointers to objects of type T can be passed back and forth between the clients of module M and the procedures of module M. Dually, types defined by clients are private to the client. Pointers to client-defined types may be passed back and forth between the clients and the module, but the module cannot dereference such pointers (or allocate objects of such type). Procedures have parameters and local variables, which are private to the invocation of the procedures. (Thus, these are thread-local variables.) There are no static or global variables shared by different invocations of procedures. (However, our results can be generalized to support them.)

```
stms = skip
     | x = e(y1,...,yk)
     | assume(b)
     | x = new R()
     | x = y.f | x.f = y
     | acquire(x) | release(x)
     | return(x)
```

Figure 4. Primitive instructions, b stands for a local boolean variable, e(y1,...,yk) stands for an expression over local variables.

We assume that body of a procedure is represented by a control-flow graph. We refer to the vertices of a control-flow graph as *program points*. The edges of a control-flow graph are annotated with primitive instructions, shown in Fig. 4. Conditionals are encoded by annotating control-flow edges with `assume` statements. Without loss of generality, we assume that a heap object can be dereferenced only in a load ("`x = y.f`") or store ("`x.f = y`") instruction. Operations to acquire or release a lock refer to a thread-local variable (that points to the heap object to be locked or unlocked). The other primitive instructions reference only thread-local variables.

We present a semantics for a module independent of any specific client. We define a notion of *execution* that covers all possible executions of the module that can arise with any possible client, but restricting attention to the part of the program state "owned" by the module. (In effect, our semantics models what is usually referred to as a "most-general-client" of the module.) For simplicity, we assume that each procedure invocation is executed by a different thread, which allows us to identify procedure invocations using a thread-id. We refer to each invocation of a procedure as a *transaction*. We model a procedure invocation as a creation of a new thread with an appropriate thread-local state. We describe the behavior of a module by the relation $\longrightarrow$. A transition $\sigma \longrightarrow \sigma'$ represents the fact that a state σ can be transformed into a state σ' by executing a single instruction.

Transactions share a heap consisting of an (unbounded) set of heap objects. Any object allocated during the execution of a module procedure is said to be a module (owned) object. In fact, our semantics models only module owned ob-

jects. Any module object that is returned by a module procedure is said to be an *exposed object*. Other module objects are *hidden objects*. Note that an exposed object remains exposed forever. A key idea encoded in the semantics is that at any point during execution a new procedure invocation may occur. The only assumption made is that any module object passed as a procedure argument is exposed; i.e., the object was returned by some earlier procedure invocation.

Each heap allocated object serves as a lock for itself. Locks are exclusive (i.e., a lock can be held by at most one transaction at a time). The execution of a transaction trying to acquire a lock (by an `acquire` statement) which is held by another transaction is blocked until a time when the lock is available (i.e., is not held by any transaction). Locks are reentrant; an `acquire` statement has no impact when it refers to a lock that is already held by the current transaction. A transaction cannot release a lock that it does not hold.

Whenever a new object is allocated, its boolean fields are initialized to `false`, its integer fields are initialized to `0`, and pointer fields are initialized to `null`. Local variables are initialized in the same manner.

A formal semantics for the language appears in Appendix A.

Running Transactions Each control-flow graph of a procedure has two distinguished control points: an *entry site* from which the transaction starts, and an *exit site* in which the transaction ends (if a CFG edge is annotated with a `return` statement, then this edge points to the exit site of the procedure). We say that a transaction t is *running* in a state σ, if t is not in its entry site or exit site. An *idle state*, is a state in which no transaction is running.

Executions The initial state σ_I has an empty heap and no transactions. A sequence of states $\pi = \sigma_0, \ldots, \sigma_k$ is an *execution* if the following hold: (i) σ_0 is the initial state, (ii) for $0 \leq i < k$, $\sigma_i \longrightarrow \sigma_{i+1}$.

An execution $\pi = \sigma_0, \ldots, \sigma_k$ is a *complete execution*, if σ_k is idle. An execution $\pi = \sigma_0, \ldots, \sigma_k$ is a *sequential execution*, if for each $0 \leq i \leq k$ at most one transaction in σ_i is running.

An execution is *non-interleaved* if transitions of different transactions are not interleaved (i.e., for every pair of transactions $t_i \neq t_j$ either all the transitions executed by t_i come before any transition executed by t_j , or vice versa). Note that, a sequential execution is a special case of a non-interleaved execution. In a sequential execution a new transaction starts executing only after all previous transactions have completed execution. In a non-interleaved execution, a new transaction can start executing before a previous transaction completes execution, but the execution is not permitted to include transitions by the previous transaction once the new transaction starts executing.

We say that a sequential execution is *completeable* if it is a prefix of a complete sequential execution.

Schedules The *schedule* of an execution $\pi = \sigma_0, \ldots, \sigma_k$ is a sequence $\langle t_0, e_0 \rangle, \ldots, \langle t_{k-1}, e_{k-1} \rangle$ such that for every $0 \leq i < k$, σ_i can be transformed into σ_i via transaction t_i executing the instruction annotating control-flow edge e_i.

Graph-Representation The heap (shared memory) of a state identifies a edge-labelled multidigraph (a directed graph in which multiple edges are allowed between the same pair of vertices), which we call the heap graph. Each heap-allocated object is represented by a vertex in the graph. A pointer field f in an object u that points to an object v is represented by an edge (u, v) labelled f. (Note that the heap graph represents only objects owned by the module. Objects owned by the client are not represented in the heap graph.) .

Strict Conflict-Serializability Given an execution, we say that two transitions conflict if: (i) they are executed by two different transactions, (ii) they access some common object (i.e., read or write fields of the same object).

Executions π and π' are said to be *conflict-equivalent* if they consist of the same set of transactions, and the schedule of every transaction t is the same in both executions, the executions agree on the order between conflicting transitions (i.e., the ith transition of a transaction t precedes and conflicts with the jth transition of a transaction t' in π, iff the former precedes and conflicts with the latter in π'). Conflict-equivalent executions produce the same state [38]. An execution is *conflict-serializable* if it is conflict-equivalent with a non-interleaved execution.

We say that an execution π is *strict conflict-serializable* if it is conflict-equivalent to a non-interleaved execution π' where a transaction t_1 completes execution before a transaction t_2 in π' if t_1 completes execution before a transaction t_2 in π.

Assume that all sequential executions of a module satisfy a given specification Φ. In this case, a strict conflict-serializable execution is also linearizable [23] with respect to specification Φ. Thus, correctness in sequential executions combined with strict conflict-serializability is sufficient to ensure linearizability.

3. Domination Locking

In this section we present the *Domination Locking Protocol* (abbreviated DL). We show that if every sequential execution of a module satisfies DL and is completable, then every concurrent execution of the module is strict conflict-serializable and completeable (i.e., atomicity and deadlock-freedom are guaranteed).

The locking protocol is parameterized by a total order $\leq$ on all heap objects, which remains fixed over the whole execution.

DEFINITION 3.1. *Let $\leq$ be a total order of heap objects. We say that an execution satisfies the* Domination Locking *protocol, with respect to $\leq$, if it satisfies the following conditions:*

1. *A transaction t can access a field of an object u, only if u is currently locked by t.*
2. *A transaction t can acquire an exposed object u, only if t has never acquired an exposed object v such that $u \leq v$.*
3. *A transaction t can acquire an exposed object, only if t has never released a lock.*
4. *A transaction t can acquire a hidden object u, only if every path between an exposed object to u includes an object which is locked by t.*

Intuitively, the protocol works as follows. Requirement (1) prevents race conditions where two transactions try to update an object neither has locked. Conditions (2) and (3) deal with exposed objects. Very little can be assumed about an object that has been exposed; references to it may reside anywhere and be used at any time by other transactions that know nothing about the invariants t is maintaining. Thus, as is standard, requirements (2) and (3) ensure all transactions acquire locks on exposed objects in a consistent order, preventing deadlocks. The situation with hidden objects is different, and we know more: other threads can only gain access to t's hidden objects through some chain of references starting at an exposed object, and so it suffices for t to guard each such potential access path with a lock. Another way of understanding the protocol is that previous proposals (e.g., [10, 25, 26, 35]) treat all objects as exposed, whereas domination locking also takes advantage of the information hiding of abstract data types to impose a different, and weaker, requirement on encapsulated data. In particular, no explicit order is imposed on the acquisition or release of locks on hidden objects, provided condition (4) is maintained.

THEOREM 3.1. *Let $\leq$ be a total order of heap objects. If every sequential execution of the module is completeable and satisfies Domination Locking with respect to $\leq$, then every execution of the module is strict conflict-serializable, and is a prefix of a complete-execution.*

This theorem implies that a concurrent execution cannot deadlock, since it is guaranteed to be the prefix of a complete-execution. The proof can be found in Appendix B.

Domination Locking generalizes previously proposed protocols such as Dynamic Tree Locking (DTL) protocol and Dynamic Dag Locking (DDL) protocol [4], which themselves subsume idioms such as hand-over-hand locking. The DTL and DDL protocols were inspired by database protocols for trees and DAGs ([10, 25, 26, 35]), but customized for use in programs where shape invariants may be temporarily violated.

In particular, any execution that satisfies DTL or DDL can be shown to satisfy DL. In comparing these protocols, it should be noted that DTL and DDL were described in a restricted setting where the exposed objects took the form of a statically fixed set of global variables. DL generalizes this by permitting a dynamic set of exposed objects (which can grow over time). More importantly, DL is a strict generalization of DTL and DDL: executions that satisfy DL might not satisfy either DTL or DDL. Among other things, DL does not require the heap graph to satisfy any shape invariants. Thus, the above theorem generalizes a similar theorem established for DDL and DTL in [4]. The above theorem, like those in [4], is important because it permits the use of sequential reasoning, e.g., to verify if a module guarantees strict conflict-serializability via DL. More interestingly, this reduction theorem also simplifies the job of automatically guaranteeing strict conflict-serializability via DL, as we illustrate in this paper.

One interesting aspect of the DL protocol is the following. Even if every sequential execution of a module is completable and satisfies DL, a concurrent execution of the module might not satisfy DL! This is in contrast to protocols such as DTL and DDL. This fact complicates the proof of the above theorem.

The requirement for a total order of exposed objects, does not restrict its applicability since in any conventional programming environment such order can be obtained (e.g., by using memory address of objects, or by using a simple mechanism that assigns unique identifiers to objects). Furthermore, no order is needed when each transaction accesses a single exposed object.

4. Enforcing DL in Forest-Based Modules

In this section, we describe our technique for automatically adding fine-grain locking to a module when the module operates on heaps of restricted shape. Specifically, the technique is applicable to modules that manipulate data structures with a forest shape, even with intra-transaction violations of forestness. For example, the Treap of Section 1 has a tree shape which is temporarily violated by tree-rotations (during tree-rotations a node may have two parents). Our technique has no limit on the number of violations or their effect on the data structures shape, as long as they are eliminated before the end of the transaction.

In Section 4.1, we describe the shape restrictions required by our technique, and present dynamic conditions that are enforced by our source transformation. We refer to these conditions as the *Eager Forest-Locking* protocol (EFL), and show that it ensures domination locking.

In Section 4.2, we show how to automatically enforce EFL by a source-to-source transformation of the original module code.

4.1 Eager Forest-Locking

When the shape of the heap manipulated by the module is known to be a forest (possibly with temporary violations), we can enforce domination locking by dynamically enforcing the conditions outlined below.

First, we define what it means for a module to be forest-based.

Forestness Condition We say that a hidden object u is consistent in a state σ, if u has at most one incoming edge in σ.[1] We say that an exposed object u is consistent in a state σ, if it does not have any incoming edges in σ.

DEFINITION 4.1. *A module M is a* forest-based module*, if in every sequential execution, all objects in idle states are consistent.*

For a forest-based module, we define the following Eager Forest-Locking conditions, and show that they guarantee that the module satisfies the domination locking conditions.

Eager Forest-Locking Requirements Given a transaction t, we define t's *immediate scope* as the set of objects which are directly pointed to by local variables of t. Intuitively, eager forest-locking is a simple protocol: a transaction should acquire a lock on an object whenever it enters its immediate scope and it should release a lock on an object whenever the object is out of its immediate scope and is consistent. The protocol description below is a bit complicated because the abovementioned invariant will be temporarily violated while an object is being locked or unlocked. (In particular, conditions 1, 2, and 4 restrict the extent to which the invariant can be violated.)

DEFINITION 4.2. *Let $\leq$ be a total order of heap objects. We say that an execution satisfies the Eager Forest-Locking (EFL) with respect to $\leq$, if it satisfies the following conditions:*

1. *A transaction t can access a field of an object, only if all objects in t's immediate scope are locked by t.*
2. *A transaction t can release an object, only if all objects in t's immediate scope are locked by t.*
3. *A transaction t can release a lock of an object u, only if u is consistent.*
4. *Immediately after a transaction t releases a lock of an object u, t removes u from its immediate scope (i.e., the next instruction of t removes u from immediate scope)*
5. *A transaction t can acquire an exposed object u, only if t has never acquired an exposed object v such that $u \leq v$.*

In contrast to the DL conditions, the EFL conditions can directly be enforced by instrumenting the code of a given module because all its dynamic conditions can be seen as conditions on its immediate scope and local memory. Such code instrumentation is allowed to only consider sequential executions, as stated by the following theorem and conclusion:

THEOREM 4.1. *Let $\leq$ be a total order of heap objects. Let π be a sequential execution of a forest-based module. If π satisfies EFL with respect to $\leq$, then π satisfies DL with respect to $\leq$.*

From Theorem 3.1 and Theorem 4.1 we conclude the following.

CONCLUSION 4.1. *Let $\leq$ be a total order of heap objects. If every sequential execution of a forest-based module is completeable and satisfies EFL with respect to $\leq$, then every execution of this module is strict conflict-serializable, and is a prefix of a complete-execution.*

4.2 Enforcing EFL

In this section, we present a source-to-source transformation that *enforces* EFL in a forest-based module. The idea is to instrument the module such that it counts stack and heap references to objects, and use these reference counts to determine when to acquire and release locks. Since the EFL conditions are defined over *sequential executions*, reasoning about the required instrumentation is fairly simple.

Run-Time Information The instrumented module tracks objects in the immediate scope of the current transaction[2] by using *stack-reference counters*; the stack-reference counter of an object u, tracks the number of references from local variables to u; hence u is in the immediate scope of current transaction whenever its stack-reference counter is greater than 0. To determine consistency of objects, it uses a *heap-reference counter*; the heap-reference counter of an object u, tracks the number of references in heap objects that point to u; a hidden object is consistent, whenever its heap-counter equals to 0 or 1; and an exposed object is consistent, whenever its heap-counter equals to 0. To determine whether an object has been exposed, it uses a boolean field; whenever an object is exposed (returned) by the module, this field is set to `true` (in that object).

Locking Strategy The instrumented code uses a strategy that follows EFL conditions. At the beginning of the procedure, the instrumented module acquires all objects that are pointed to by parameters (and are thus exposed objects). The order in which these objects are locked is determined by using a special function, `unique` that returns a unique identifier for each object[3]. After locking all exposed objects, the instrumented module acts as follows: (i) it acquires object u whenever its stack-reference-counter becomes 1; (ii) it releases object u whenever u is consistent, and its stack-reference-counter becomes 0.

This strategy releases all locks before completion of a transaction (since every object becomes consistent before that point), so it cannot create incompleteable sequential executions.

Source-to-Source Transformation Our transformation instruments each object with three additional fields: `stackRef`

[1] In the graph representation of the heap. Recall that the heap-graph contains only module owned objects. In particular, this definition does not consider pointers to exposed objects that may be stored in client objects

[2] Note that we consider sequential executions, so we can assume a single current transaction.

[3] Note that only exposed objects are pointed by the procedure parameters. And according to Definition 4.1 these are the only exposed objects the transaction will see.

Operation	Code

Take(x)	`if(x!=null) {`
	`  acquire(x);`
	`  x.stackRef++;`
	`}`

Drop(x)	`if(x!=null) {`
	`  x.stackRef-- ;`
	`  if(x.stackRef==0 && IsConsistent(x))`
	`  release(x);`
	`}`

IsConsistent(x)	`if(x.isExposed)`
	`  return (x.heapRef == 0);`
	`else`
	`  return (x.heapRef <= 1);`

MarkExposed(x)	`if(x!=null) x.isExposed=true;`

Table 1. Primitive operations used in the EFL transformation.

```
1  TakeArgs2(x,y) {
2   if(unique(x) < unique(y))
3    { Take(x); Take(y); }
4   else
5    { Take(y); Take(x); }
6  }
```

Figure 5. Acquiring two procedure arguments in a unique locking order.

```
1  void AddValues(Node x, Node y) {
2   while(x!=null && y!=null) {
3    x.value+=y.value;
4    x=x.next;
5    y=y.next;
6  }}
```

Figure 6. Example procedure adding the values from one linked-list into another.

x = ptrExp

```
ASNL(x,ptrExp) {
 temp=ptrExp;
 Take(temp);
 Drop(x);
 x=temp;
}
```

x.f = ptrExp

```
ASNF(x.f,ptrExp) {
 temp=x.f;
 Take(temp);
 if(temp!=null) temp.heapRef--;
 Drop(temp);
 temp=ptrExp;
 Take(temp);
 if(temp!=null) temp.heapRef++;
 Drop(temp);
 x.f = temp;
}
```

Table 2. The macros ASNL and ASNF for pointer assignments enforcing EFL.

and `heapRef` to maintain the stack and heap reference counts (respectively), and `isExposed` to indicate whether the object has been exposed. The transformation is based on the primitive operations of Table 1.

The procedures `Take` and `Drop` maintain stack reference counters and perform the actual locking. `Take(x)` locks the object referenced by x and increments the value of its stack reference counter. `Drop(x)` decreases the stack reference count of the object referenced by x, and releases its lock if it is safe to do so according to the EFL protocol, i.e., if the reference from x was the only reference to the object, and the object is consistent. `Drop` uses the function `IsConsistent` which indicates whether an object is consistent or not (according to its heap-counter and the `isExposed` field).

For each procedure, of the module, our transformation is performed as follows:

1. At the beginning of the procedure, add code that acquires all objects pointed to by arguments according to a fixed order; in a case of a single pointer argument l, this can be done by adding `Take(l)` (as in line 3 of Fig. 3); the code of Fig. 5 demonstrates the case of 2 pointer arguments; in the general case objects are sorted to obtain the proper order.
2. Replace every assignment of a pointer expression with the corresponding code macros in Table 2. The macro `ASNL(x,ptrExp)` replaces an assignment of a pointer expression `ptrExp` to a local pointer `x`, this macro performs this assignment, while maintaining stack-counters and following the required locking strategy. The macro `ASNF(x.f,ptrExp)` replaces an assignment of a pointer expression to a field of an object, this macro maintains the heap-counters in objects (its implementation follows the required locking strategy).
3. Whenever a local variable `l` reaches the end of its scope, add `ASNL(l,null)`; this releases the object pointed by l. If this is the end of the procedure, and l is about to be returned (i.e., by the statement `return(l)`), then instead of adding `ASNL(l,null)` add the block `{MarkExposed(l);Drop(l);}`.

Example The procedure of Fig. 6 takes a pair of pointers to singly-linked lists, and adds values of one list to the values of the other. Fig. 7 shows the code transformed to enforce EFL. The transformed procedure starts with an invocation of `TakeArgs2` (shown in Fig. 5) to lock exposed objects in

```
1  void AddValues(Node x, Node y) {
2   TakeArgs2(x,y);
3   while(x!=null && y!=null) {
4    x.value+=y.value;
5    ASNL(x,x.next);
6    ASNL(y,y.next);
7   }
8   ASNL(x,null); ASNL(y,null);
9  }
```

Figure 7. Transformed code enforcing EFL for the procedure `AddValues` of Fig. 6.

a fixed order. In the body of `AddValues`, the assignment `x=x.next` is replaced by the macro `ASNL(x,x.next)`, which assigns `x.next` to `x` while maintaining EFL requirements. The assignment `y=y.next` is handled in a similar way. At the end of `AddValues`, local variables go out of scope and locks are released by adding `ASNL(x,null)` and `ASNL(y,null)`.

Practical Consideration In some cases, some of our instrumentation code can be avoided. For example, instead of replacing `x=null` with `ASNL(x,null)`, we could just add `Drop(x)` before the assignment. Or whenever it is known that a variable will not have a `null` value, we could avoid the `if` statements in `Take` and `Drop`.

In modules where the forestness condition is not violated even temporarily, the heap reference counter is not needed (since all objects remain consistent during a sequential execution of this transaction).

In many cases, exposed objects can be identified by the types of objects (e.g., *List* is a type of exposed objects, and *Node* is a type of hidden object); in such cases type information can be used instead of using the `isExposed` field.

Using Static Analysis The shown instrumented code can be optimized by using various static techniques. It is sufficient for such static techniques to consider only sequential executions of the module.

A live-variables analysis [2] can detect local pointers with unused values. Assigning $null$ to such pointers will eliminate unused pointers, and as a result will release locks earlier.

Some static tools (e.g. [27]) can help avoid some of the instrumentation code. For example, if a tool can detect that a local variable l is always `null` at some point of the CFG, our instrumentation code can avoid calling `Take(l)` in this case.

4.3 Example for Dynamically Changing Forest

As an example for a dynamically changing forest, consider the procedure shown in Fig. 8. This procedure operates on two Skew-Heaps [36] (a self-adjusting minimum-heap implemented as a binary tree). The procedure moves the content of one Skew Heap (pointed by `src`) to another one (pointed by `dest`), by simultaneously traversing the heaps;

```
1  void move(SkewHeap src, SkewHeap dest) {
2   Node t1, t3, t2;
3   t1=dest.root;
4   t2=src.root;
5   if(t1.key > t2.key) { // assume both heaps are not empty
6    t3=t1; t1=t2; t2=t3;
7   }
8   dest.root=t1;
9   src.root=null;
10  t3=t1.right;
11  while(t3 != null && t2 != null) {
12   t1.right=t1.left;
13   if(t3.key < t2.key) {
14    t1.left=t3; t1=t3; t3=t3.right;
15   }
16   else {
17    t1.left=t2; t1=t2; t2=t2.right;
18   }
19  }
20  if(t3 == null) t1.right=t2;
21  else t1.right=t3;
22 }
```

Figure 8. Moving the content of one Skew-Heap to another Skew-Heap.

```
1  void move(SkewHeap src, SkewHeap dest) {
2   Node t1, t3, t2;
3   TakeArgs2(src,dest);
4   ASNL(t1, dest.root);
5   ASNL(t2, src.root);
6   if(t1.key > t2.key) {
7    ASNL(t3,t1); ASNL(t1,t2); ASNL(t2,t3);
8   }
9   ASNF(dest.root, t1);
10  ASNL(dest, null); // dest becomes dead
11  ASNF(src.root, null);
12  ASNL(src, null); // src becomes dead
13  ASNL(t3, t1.right);
14  while(t3 != null && t2 != null) {
15   ASNF(t1.right, t1.left);
16   if(t3.key < t2.key) {
17    ASNF(t1.left, t3); ASNL(t1, t3); ASNL(t3, t3.right);
18   }
19   else {
20    ASNF(t1.left, t2); ASNL(t1, t2); ASNL(t2, t2.right);
21   }
22  }
23  if(t3 == null) ASNF(t1.right, t2);
24  else ASNF(t1.right, t3);
25  ASNL(t1, null); ASNL(t2, null); ASNL(t3, null);
26 }
```

Figure 9. Moving Skew Heaps with automatic fine-grain locking.

during its operation, nodes are dynamically moved from one data-structure to another one. Fig. 9 show its code, after the source transformation.

5. Performance Evaluation

We evaluate the performance of our technique on several benchmarks. For each benchmark, we compare the performance of the benchmark using fine-grain locking automatically generated using our technique to the performance of the benchmark using a single coarse-grain lock. We also compare some of the benchmarks to versions with handcrafted fine-grain locking. For some benchmarks, manually

adding fine-grain locking turned out to be too difficult even for concurrency experts.

In our experiments, we consider 5 different benchmarks: two balanced search-tree data structures, a self-adjusting heap data structure, and two specialized tree-structures (which are tailored to their application).

The experiments were run on a machine with 8 hardware threads. Specifically, we used an Intel Core2 $i7$ processor with 4 cores that each multiplex 2 hardware threads.

5.1 General Purpose Data-Structures

5.1.1 Balanced Search-Trees

We consider two Java implementations of balanced search trees: a *Treap* [3], and a *Red-Black Tree* with a top-down balancing [8, 18]. For both balanced trees, we consider the common operations of *insert*, *remove* and *lookup*.

Methodology We follow the evaluation methodology of Herlihy et al. [20], and consider the data structures under a workload of 20% inserts, 10% removes, and 70% lookups. The keys are generated from a random uniform distribution between 1 and 2×10^6. To ensure consistent and accurate results, each experiment consists of five passes; the first pass warms up the VM and the four other passes are timed. Each experiment was run four times and the arithmetic average of the throughput is reported as the final result.

Every pass of the test program consists of each thread performing one million randomly chosen operations on a shared data-structure; a new data-structure is used for each pass.

Evaluation For both search trees, we compare the results of our automatic locking to a coarse-grain global lock. For the *Treap*, we also consider a version with manual hand-over-hand locking. Enforcing hand-over-hand locking for the *Treap* is challenging because after a rotation, the next thread to traverse a path will acquire a different sequence of locks. Assuring the absence of deadlock under different acquisition orders is challenging.

For the Red-Black Tree, the task of manually adding fine-grain locks proved to be too challenging and error prone. Rotations and deletions are much more complicated than in a *Treap*. Previous work on fine-grain locking for these trees alters the tree invariants and algorithm, as in [32]. Even after spending a whole day, we were unable to find or develop a correct manual locking strategy for true Red-Black Trees.

Fig. 10 shows results for the *Treap*. Our locking scales as well as manual hand-over-hand locking. They both outperform the single-lock as the number of threads is increased.

Fig. 11 shows results for the Red-Black Tree. Starting from 2 threads, our locking is faster than the single-lock.

5.1.2 Self-Adjusting Heap

We consider a Java implementation of a *Skew Heap* [8, 36], which is a self-adjusting heap data-structure. We consider the operations of *insert* and *removeMin*.

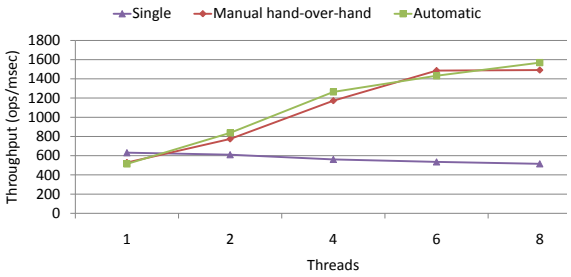

Figure 10. Throughput for a Treap with a single lock, manual hand-over-hand locking, and EFL-based automatic locking; with 70% lookups, 20% inserts and 10% removes.

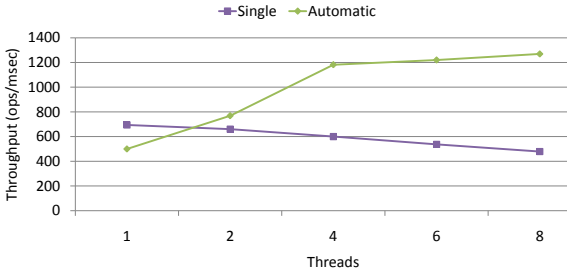

Figure 11. Throughput for a Red-Black Tree with a single lock, and EFL-based automatic locking; with 70% lookups, 20% inserts and 10% removes.

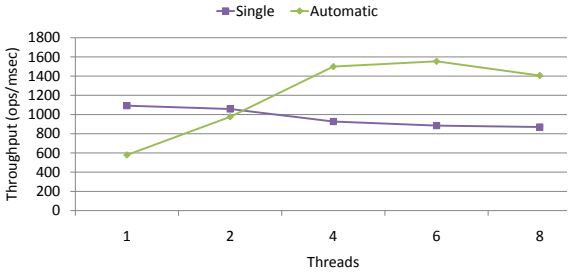

Figure 12. Throughput for a Skew Heap with a single lock, and EFL-based automatic locking; with 50% inserts and 50% removeMin.

We use the same evaluation methodology we used for the search trees. Here we consider a workload of 50% inserts and 50% removes on a heap initialized with one million elements. We compare the results of our automatic locking to a coarse-grain global lock. The results are shown in Fig. 12.

5.2 Specialized Implementations

To illustrate the applicability of our technique to specialized data-structures (which are tailored to their application), we consider Java implementation of Barnes-Hut algorithm [5], and a C++ implementation of the Apriori Data-Mining algorithm [1] from [31].

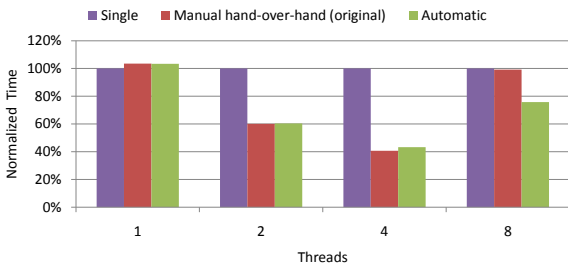

Figure 13. Apriori, Normalized Time of Hash-Tree Construction.

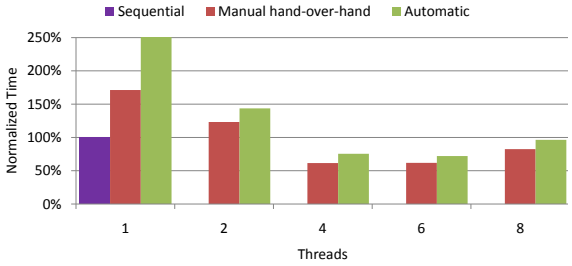

Figure 14. Barnes-Hut, Normalized Time of OCT-Tree Construction.

5.2.1 Apriori

In this application, a number of threads concurrently build a Hash-Tree data-structure (a tree data-structure in which each node is either a linked-list or a hash-table). The original application uses customized hand-over-hand locking tailored to this application. We evaluate the performance of our locking relative to this specialized manual locking and to a single global lock. We show that our locking performs as well as the specialized manual locking scheme in the original application.

In the experiments, we measured the time required for the threads to build the Hash-Tree. Fig. 13 shows the speedup of the original hand-crafted locking, and our locking over a single lock. For 2 and 4 threads, the speedup of our locking is almost as good as the original manual locking. In the case of 8 threads it performs better than the original locking (around 30% faster). Both have a small overhead in the case of a single thread (around 4% slower).

5.2.2 Barnes-Hut

The Barnes-Hut algorithm simulates the interaction of a system of bodies (such as galaxies or particles) and is built from several phases. Its main data-structure is an OCT-Tree. We parallelized the *Construction-Phase* in which the OCT-Tree is built, and used our technique for synchronization. We measured the benefit gained by our locking.

In the experiments, we measured the time required for threads to build the OCT-tree (i.e., the *Construction-Phase*). Fig. 14 shows the results. Our locking and manual hand-over-hand locking show high overhead for 1 and 2 threads (150% for our locking, and 70% for manual hand-over-hand). For a larger number of threads our locking and manual hand-over-hand are comparable, and they are both faster than the sequential version.

6. Discussion

In this section, we discuss extensions and additional potential applications of our approach.

Other Ways to Enforce Domination Locking We have presented a way to enforce domination locking on forest-based modules. Still, domination locking can be enforced in several other cases (both manually and automatically). In particular, it can be enforced in cases in which the shared heap is not a forest.

As an illustrative example, consider a non forest-based module M with a single exposed object e in which hidden objects never point to e. Assume that M uses the instrumentation presented in Section 4, and that each transaction releases all its locks before its completion (can be realized, for example, by adding a `ReleaseAll` statement at the end of transactions; this statement releases all locks that are owned by the transaction). It is easy to see that M follows domination locking in sequential executions. Hence, if M's sequential executions are completable, then its concurrent executions are strict conflict-serializable and completeable.

We believe that it is interesting to explore different ways for realizing the domination locking protocol, together with their practical implications.

Using Optimistic Synchronization for Read-Only Operations Domination locking can be combined with optimistic synchronization to improve scalability for read-only transactions by adding version information to objects. Read-write transactions would synchronize between themselves using DL, with no chance of rollback, while read-only transactions would use the version numbers to ensure consistent reads. Version numbers could either be managed locally, by incrementing them on each commit (e.g., as in [9]), or globally using a timestamp scheme (e.g., as in [14]). The local scheme would provide better scalability for writers, while the global scheme admits very efficient read-only transactions. Contention management in our system would be easier than in purely optimistic schemes such as STM, because read-only transactions can fall back to DL locking after experiencing too many rollbacks.

Verification Domination Locking protocol can provide a basis for verification. For example, [4] describes a verification technique based on special cases of domination locking (dynamic DAG and Tree locking). By using domination locking, their analysis can be simplified and extended because of the weaker conditions of domination locking.

7. Related Work

Locking Protocols Locking protocols are used in database and other software systems to guarantee correctness of concurrently executing transactions. A widely used protocol is the two-phase-locking (2PL) protocol [16] which guarantees conflict-serializability of transactions, but does not guarantee deadlock-freedom. In the 2PL protocol, locking is done in two phases, in the first phase locks are only allowed to be acquired (releasing locks is forbidden); in the second phase locks are only allowed to be released.

These restrictions require that locks are held until the final lock is obtained, thus preventing early release of a lock even when locking it is no longer required. This limits parallelism, especially in the presence of long transactions. (e.g., a tree traversal must hold the lock on the root until the final node is reached.)

Other locking protocols (non-2PL protocols) rely on the shape of the heap. Most of these protocols (e.g.[25, 35]) were designed for databases in which the shape of shared objects does not change during a transaction, and thus are not suitable for more general cases with dynamically changing heaps. [4, 10, 26] show protocols that can handle dynamically changing heap shapes.

Attiya et al. [4] present a dynamic tree-locking protocol and show that if it is satisfied by all sequential executions then it is also satisfied by all concurrent executions. In contrast, DL does not enforce any requirement on the heap. In fact, it is possible for a program that follows DL for all sequential executions to violate the DL conditions during a concurrent execution. Still, we show that if DL is satisfied by all sequential executions then all concurrent executions are correct, i.e., guarantee atomicity and deadlock freedom. This result simplifies the task of reasoning about programs using DL, since it allows both programmers and program analysis tools to ignore interleaved states.

Wang et al. [37] describe a static analysis and accompanying runtime instrumentation that eliminates the possibility of deadlock from multi-threaded programs using locks. Their tool adds additional locks that dominate any potential locking cycle, but it requires as a starting point a program that already has the locks necessary for atomicity.

Boyapati et al. [7] describe an ownership type system that guarantees data race freedom and deadlock freedom, but still not atomicity. Their approach can prevent deadlocks by relying on partial-order of objects, and also permit to dynamically change this partial-order. Interestingly, DL also relies on the intuition of dynamic ownership where exposed objects dominate hidden objects.

Lock inference There has been a lot of work on inferring locks for implementing atomic sections. Most of the algorithms in the literature infer locks for following the 2PL locking protocol [11, 12, 15, 17, 24, 29]. The algorithms in [15, 24, 29] employ a 2PL variant in which all locks are released at the end of a transaction. In these algorithms, deadlock is prevented by statically ordering locks and rejecting certain programs. The algorithms in [11, 17] use a 2PL variant in which all locks are acquired at the beginning of transactions and released at the end of transactions. In these algorithms, deadlock is prevented by using a customized locking protocol at the beginning of atomic sections. As described above, 2PL limits parallelism as all locks must be held until the final lock is acquired.

Transactional Memory Transactional memory approaches (TMs) dynamically resolve deadlocks by rolling back partially completed atomic regions.[4] The TM programming model can be implemented as an extension to the cache coherence protocol [22] or as a code transformation [21]. Preserving the ability to roll back requires that transactions be isolated from the rest of the system, which prohibits them from performing I/O. Software transactions are also prohibited from calling modules that have not been transformed by the TM. Ad-hoc proposals for specific forms of I/O are present in many TMs [30], but in the general case at most one transaction at a time can safely perform an irrevocable action [39]. Pessimistic automatic concurrency control schemes such as our technique, in contrast, do not limit concurrent I/O or calls to foreign modules.

GC Algorithms It is interesting to note that our locking using reference counters bears similarities with garbage collection algorithms based on reference counts (e.g., [13, 28]). In particular, it is beneficial to maintain separate reference counts from the stack and the heap.

Acknowledgments

We thank the anonymous referees for their useful comments. This research was partially supported by The Israeli Science Foundation (grant no. 965/10), the National Science Foundation (grant NSF CCF-0702681), and a gift from IBM.

References

[1] AGRAWAL, R., MANNILA, H., SRIKANT, R., TOIVONEN, H., AND VERKAMO, I. Advances in knowledge discovery and data mining. American Association for Artificial Intelligence, Menlo Park, CA, USA, 1996, ch. Fast discovery of association rules, pp. 307–328.

[2] AHO, A. V., LAM, M. S., SETHI, R., AND ULLMAN, J. D. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006.

[3] ARAGON, C., AND SEIDEL, R. Randomized search trees. *Foundations of Computer Science, Annual IEEE Symposium on 0* (1989), 540–545.

[4] ATTIYA, H., RAMALINGAM, G., AND RINETZKY, N. Sequential verification of serializability. In *POPL '10: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2010), ACM, pp. 31–42.

[4] Many TMs also roll back transactions in the case of incorrect speculation.

Instruction	Transition	Side Condition
skip	$\sigma \xrightarrow{\langle t,e \rangle} \langle h, r, \varrho[t \mapsto \langle k', \rho, L\rangle]\rangle$	
$x = e(y_1, \ldots, y_k)$	$\sigma \xrightarrow{\langle t,e \rangle} \langle h, r, \varrho[t \mapsto \langle k', \rho[x \mapsto [\![e]\!](\rho(y_1), \ldots, \rho(y_k))], L\rangle]\rangle$	
assume(b)	$\sigma \xrightarrow{\langle t,e \rangle} \langle h, r, \varrho[t \mapsto \langle k', \rho, L\rangle]\rangle$	$\rho(b) = \mathit{true}$
$x = \text{new}R()$	$\sigma \xrightarrow{\langle t,e \rangle} \langle h[a \mapsto o], r, \varrho[t \mapsto \langle k', \rho[x \mapsto a], L\rangle]\rangle$	$a \notin \text{dom(h)} \wedge \iota(R)() = o$
x = y.f	$\sigma \xrightarrow{\langle t,e \rangle} \langle h, r, \varrho[t \mapsto \langle k', \rho[x \mapsto h(\rho(y))(f)], L\rangle]\rangle$	$\rho(y) \in \text{dom(h)}$
x.f = y	$\sigma \xrightarrow{\langle t,e \rangle} \langle h[\rho(x) \mapsto (h(\rho(x))[f \mapsto \rho(y)])], r, \varrho[t \mapsto \langle k', \rho, L\rangle]\rangle$	$\rho(x) \in \text{dom(h)}$
acquire(x)	$\sigma \xrightarrow{\langle t,e \rangle} \langle h, r, \varrho[t \mapsto \langle k', \rho, L \cup \{\rho(x)\}\rangle]\rangle$	$\rho(x) \in L \vee \forall \langle k'', \rho', L'\rangle \in range(\varrho) : \rho(x) \notin L'$
release(x)	$\sigma \xrightarrow{\langle t,e \rangle} \langle h, r, \varrho[t \mapsto \langle k', \rho, L \setminus \{\rho(x)\}\rangle]\rangle$	$\rho(x) \in L$
return(x)	$\sigma \xrightarrow{\langle t,e \rangle} \langle h, r[\rho(x) \mapsto \mathit{true}], \varrho[t \mapsto \langle k', \rho, L\rangle]\rangle$	$\rho(x) \in \text{dom(h)}$
return(x)	$\sigma \xrightarrow{\langle t,e \rangle} \langle h, r, \varrho[t \mapsto \langle k', \rho, L\rangle]\rangle$	$\rho(x) \notin \text{dom(h)}$

Table 3. The semantics of primitive instructions. For brevity, we use the shorthands: $\sigma = \langle h, r, \varrho\rangle$ and $\varrho(t) = \langle k, \rho, L\rangle$, and omit $(k, k') = e \in \text{CFG}_t$ from all side conditions.

[5] BARNES, J., AND HUT, P. A hierarchical O(N log N) force-calculation algorithm. *Nature 324* (Dec. 1986), 446–449.

[6] BAYER, R., AND SCHKOLNICK, M. Concurrency of operations on B-Trees. *Acta Informatica 9* (1977), 1–21.

[7] BOYAPATI, C., LEE, R., AND RINARD, M. Ownership types for safe programming: preventing data races and deadlocks. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2002), OOPSLA '02, ACM, pp. 211–230.

[8] BRASS, P. *Advanced Data Structures*. Cambridge University Press, New York, NY, USA, 2008.

[9] BRONSON, N. G., CASPER, J., CHAFI, H., AND OLUKOTUN, K. A practical concurrent binary search tree. In *PPOPP* (2010), pp. 257–268.

[10] CHAUDHRI, V. K., AND HADZILACOS, V. Safe locking policies for dynamic databases. In *PODS '95: Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems* (New York, NY, USA, 1995), ACM, pp. 233–244.

[11] CHEREM, S., CHILIMBI, T., AND GULWANI, S. Inferring locks for atomic sections. In *PLDI* (2008), pp. 304–315.

[12] CUNNINGHAM, D., GUDKA, K., AND EISENBACH, S. Keep off the grass: Locking the right path for atomicity. In *Compiler Construction*, vol. 4959 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2008, pp. 276–290.

[13] DEUTSCH, L. P., AND BOBROW, D. G. An efficient, incremental, automatic garbage collector. *Commun. ACM 19*, 9 (1976), 522–526.

[14] DICE, D., SHALEV, O., AND SHAVIT, N. Transactional locking ii. In *DISC* (2006), pp. 194–208.

[15] EMMI, M., FISCHER, J. S., JHALA, R., AND MAJUMDAR, R. Lock allocation. In *POPL* (2007), pp. 291–296.

[16] ESWARAN, K. P., GRAY, J. N., LORIE, R. A., AND TRAIGER, I. L. The notions of consistency and predicate locks in a database system. *Commun. ACM 19* (November 1976), 624–633.

[17] GUDKA, K., EISENBACH, S., AND HARRIS, T. Lock Inference in the Presence of Large Libraries. Tech. rep., November 2010.

[18] GUIBAS, L. J., AND SEDGEWICK, R. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science* (Washington, DC, USA, 1978), IEEE Computer Society, pp. 8–21.

[19] HARRIS, T., LARUS, J., AND RAJWAR, R. Transactional memory, 2nd edition. *Synthesis Lectures on Computer Architecture 5*, 1 (2010), 1–263.

[20] HERLIHY, M., LEV, Y., LUCHANGCO, V., AND SHAVIT, N. A provably correct scalable concurrent skip list. In *OPODIS '06: Proceedings of the 10th International Conference On Principles Of Distributed Systems* (December 2006).

[21] HERLIHY, M., LUCHANGCO, V., MOIR, M., AND III, W. N. S. Software transactional memory for dynamic-sized data structures. In *PODC* (2003), pp. 92–101.

[22] HERLIHY, M., AND MOSS, J. E. B. Transactional memory: Architectural support for lock-free data structures. In *ISCA* (1993), pp. 289–300.

[23] HERLIHY, M. P., AND WING, J. M. Linearizability: a correctness condition for concurrent objects. *Proc. of ACM TOPLAS 12*, 3 (1990), 463–492.

[24] HICKS, M., FOSTER, J. S., AND PRATTIKAKIS, P. Lock inference for atomic sections. In *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing* (June 2006).

[25] KEDEM, Z. M., AND SILBERSCHATZ, A. A characterization of database graphs admitting a simple locking protocol. *Acta Inf. 16* (1981), 1–13.

[26] LANIN, V., AND SHASHA, D. Tree locking on changing trees. Tech. rep., 1990.

[27] LEV-AMI, T., AND SAGIV, M. TVLA: A framework for Kleene based static analysis. In *Saskatchewan* (2000), vol. 1824 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 280–301.

[28] LEVANONI, Y., AND PETRANK, E. An on-the-fly reference-counting garbage collector for Java. *ACM Trans. Program. Lang. Syst. 28*, 1 (2006), 1–69.

[29] MCCLOSKEY, B., ZHOU, F., GAY, D., AND BREWER, E. Autolocker: synchronization inference for atomic sections. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2006), ACM, pp. 346–358.

[30] MOSS, J. E. B. Open Nested Transactions: Semantics and Support. In *Poster at the 4th Workshop on Memory Performance Issues (WMPI-2006)*. February 2006.

[31] NARAYANAN, R., ÖZIS. IKYILMAZ, B., ZAMBRENO, J., MEMIK, G., AND CHOUDHARY, A. Minebench: A benchmark suite for data mining workloads. In *2006 IEEE International Symposium on Workload Characterization* (2006), pp. 182–188.

[32] NURMI, O., AND SOISALON-SOININEN, E. Uncoupling updating and rebalancing in chromatic binary search trees. In *Proceedings of the tenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems* (New York, NY, USA, 1991), PODS '91, ACM, pp. 192–198.

[33] PAPADIMITRIOU, C. H. The serializability of concurrent database updates. *J. ACM 26*, 4 (1979), 631–653.

[34] SAGIV, M., REPS, T., AND WILHELM, R. Parametric Shape Analysis via 3-valued Logic. *ACM Trans. on Prog. Lang. and Systems (TOPLAS) 24*, 3 (2002), 217–298.

[35] SILBERSCHATZ, A., AND KEDAM, Z. A family of locking protocols for database systems that are modeled by directed graphs. *Software Engineering, IEEE Transactions on SE-8*, 6 (Nov. 1982), 558 – 562.

[36] SLEATOR, D. D., AND TARJAN, R. E. Self adjusting heaps. *SIAM J. Comput. 15* (February 1986), 52–69.

[37] WANG, Y., LAFORTUNE, S., KELLY, T., KUDLUR, M., AND MAHLKE, S. A. The theory of deadlock avoidance via discrete control. In *POPL* (2009), pp. 252–263.

[38] WEIKUM, G., AND VOSSEN, G. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[39] WELC, A., SAHA, B., AND ADL-TABATABAI, A.-R. Irrevocable transactions and their applications. In *SPAA* (2008), pp. 285–296.

[40] YANG, H., LEE, O., BERDINE, J., CALCAGNO, C., COOK, B., AND DISTEFANO, D. Scalable shape analysis for systems code. In *In CAV* (2008).

$$
\begin{array}{lcll}
v & \in & Val & = & Loc \uplus \mathcal{Z} \uplus \{\mathit{true}, \mathit{false}, null\} \\
\rho & \in & \mathcal{E} & = & \mathcal{V} \hookrightarrow Val \\
h & \in & \mathcal{H} & = & Loc \hookrightarrow \mathcal{F} \hookrightarrow Val \\
l & \in & \mathcal{L} & = & Loc \\
s & \in & \mathcal{S} & = & \mathcal{K} \times \mathcal{E} \times 2^{\mathcal{L}} \\
\sigma & \in & \Sigma & = & \mathcal{H} \times (Loc \hookrightarrow \{\mathit{true}, \mathit{false}\}) \times (\mathcal{T} \hookrightarrow \mathcal{S})
\end{array}
$$

Figure 15. Semantic domains

A. Semantics

Fig. 15 defines the semantic domains of a state of a module, and meta-variables ranging over them. Let $t \in \mathcal{T}$ be the domain of transaction identifiers.

A *state* $\sigma = \langle h, r, \varrho \rangle \in \Sigma$ of a module is a triple: h assigns values to fields of dynamically allocated objects. A value $v \in$ Val can be either a location, an integer, a boolean value,or *null*. r maps exposed objects to *true*, and hidden objects to *false*. Finally, ϱ associates a transaction t with its transaction local state $\varrho(t)$. A *transaction-local state* $s = \langle k, \rho, L \rangle \in S$ is: k is the value of the transaction's program counter, ρ records the values of its local variables, and L is the transaction's *lock set* which records the locks that the transaction holds.

The behavior of a module is described by the relations $\longrightarrow$ and $\Rightarrow$. The relation $\longrightarrow$ is a subset of $\Sigma \times (\mathcal{T} \times (\mathcal{K} \times \mathcal{K})) \times \Sigma$, and is defined in Table 3.[5]

A transition $\sigma \xrightarrow{\langle t,e \rangle} \sigma'$ represents the fact that σ can be transformed into σ' via transaction t executing the instruction annotating control-flow edge e. Invocation of a new transaction is modeled by the relation $\Rightarrow \subseteq \Sigma \times \mathcal{T} \times \Sigma$; we say that $\langle h, r, \varrho \rangle \stackrel{t}{\Rightarrow} \sigma'$ if $\sigma' = \langle h, r, \varrho[t \mapsto s] \rangle$ where $t \notin \text{dom}(\varrho)$ and s is any valid initial local state: i.e., $s = \langle entry, \rho, \{\} \rangle$, where $entry$ is the entry vertex, and ρ maps local variables and parameters to appropriate initial values (based on their type). In particular, ρ must map any pointer parameter of a type defined by the module to an exposed object (i.e., an object u in h such that $r(u) = \mathit{true}$). We write $\sigma \longrightarrow \sigma'$, if there exists t such that $\sigma \stackrel{t}{\Rightarrow} \sigma'$ or there exists $\langle t, e \rangle$ such that $\sigma \xrightarrow{\langle t,e \rangle} \sigma'$.

The *schedule* of an execution $\pi = \sigma_0, \ldots, \sigma_k$ is a sequence $\langle t_0, e_0 \rangle, \ldots, \langle t_{k-1}, e_{k-1} \rangle$ such that for $0 \leq i < k$: $\sigma_i \xrightarrow{\langle t_i, e_i \rangle} \sigma_{i+1}$, or $\sigma_i \stackrel{t_i}{\Rightarrow} \sigma_{i+1}$ and $e_i = e_{init}$ (where e_{init} is disjoint with all edges in the CFG).

We say that a sequence $\xi = \langle t_0, e_0 \rangle, \ldots, \langle t_{k-1}, e_{k-1} \rangle$ is a *feasible schedule*, if ξ is a schedule of an execution. The *schedule of a transaction* t in an execution is the (possibly

[5] For simplicity of presentation, we use an idempotent variant of *acquire* (i.e., *acquire* has no impact when the lock has already owned by the current transaction). We note that this variant is permitted by the *Lock* interface from the *java.util.concurrent.locks* package, and can easily be implemented in languages such as *Java* and *C++*.

non-contiguous) subsequence of the execution's schedule consisting only of t's transitions.

We define the allocation id of an object in an execution to be the pair (t, i) if the object was allocated by the i-th transition executed by a transaction t. An object o_1 in an execution π_1 corresponds to an object o_2 in an execution π_2 iff their allocation ids are the same. In the sequel we will compare states and objects belonging to different executions modulo this correspondence relation.

B. Proofs

DEFINITION 1. *An execution is said to be* well-locked *if every transaction in the execution accesses a field of an object, only when it holds a lock on that object.*

DEFINITION 2. *We say that a set S of objects* dominates *an object u (in a given state) if every path from an exposed object to u contains some objects from S. We say that a transaction t* blocks *an object u (in a given state) if the set of objects locked by t dominates u.*

DEFINITION 3. *We say that a transaction t is in* phase-1 *if it is still running and has never released a lock. Otherwise, we say that t is in* phase-2 *(i.e., t is in* phase-2 *if it has already completed, or it has released at least one lock).*

LEMMA 1. *Let $\xi = \xi_p \xi_t \xi_s$ be any feasible well-locked schedule, where ξ_t is the schedule of a transaction t. If t is in phase-1 (after ξ), then there is no conflict between ξ_t and ξ_s (in ξ) .*

Proof Immediate from the definition of phase-1.

DEFINITION 4. *We say that an ni-execution (non-interleaved execution) is* phase-ordered *if all phase-2 transactions precede phase-1 transactions.*

LEMMA 2. *Any feasible well-locked ni-schedule $\xi_1 \xi_2 \cdots \xi_n$ is conflict-equivalent to a well-locked phase-ordered ni-schedule $\xi_{i_1} \cdots \xi_{i_n}$.*

Proof Because of Lemma 1, moving all phase-1 transactions to the end of the schedule does not affect any of the conflict-dependences.

In the following we assume that $\leq_h$ is a total order of all heap objects. We assume that $\leq_h$ has a minimal value $\perp$ (i.e., if u is an object then $\perp \leq_h u$). We say that $u <_h v$, if $u \neq v$ and $u \leq_h v$.

DEFINITION 5. *We say that $\max(\sigma, t) = u$, if u is the maximal exposed object that is locked by transaction t in state σ (i.e., u is locked by t in σ, and every exposed object v that is locked by t in σ satisfies $v \leq_h u$). If no exposed object is locked by t in σ, then $\max(\sigma, t) = \perp$.*

DEFINITION 6. *Let $\pi = \alpha_1 \cdots \alpha_k$ be a phase-ordered execution. Let s be the last state of π. We say that π is* fully-ordered*, if for every α_i and α_j that are in phase-1 the following holds: if $i < j$ then $\max(s, t_j) \leq_h \max(s, t_i)$.*

LEMMA 3. *Any feasible well-locked ni-schedule $\xi_1 \xi_2 \cdots \xi_n$ is conflict-equivalent to a well-locked fully-ordered ni-schedule $\xi_{i_1} \cdots \xi_{i_n}$.*

Proof Similar to Lemma 2. Here we also reorder the phase-1 transactions according to $\leq_h$.

LEMMA 4. *Consider any sequential-execution $\pi_1 \rightarrow \sigma \rightarrow \pi_2$ that follows domination locking. Assume that t is in phase 2 at the end of $\pi_1 \rightarrow \sigma$. For any object o in state σ, t can access o during the (remaining) execution $\sigma \rightarrow \pi_2$ only if t blocks o in σ.*

Proof Let u be an object in σ which is not blocked by t in σ. Hence σ contains a path P from an exposed object to u, such that none of the objects in P are locked by t. We can inductively show that none of the objects in P are locked, and hence not accessed or modified during the rest of the execution.

DEFINITION 7. *Let π_1 and π_2 be two executions such that for every transaction t the schedule of t in π_1 is a prefix of the schedule of t in π_2. π_2 is said to be a* conflict-equivalent extension *of π_1 if every step (t, e) in π_1 has the same conflict-predecessors as the corresponding step in π_2. π_2 is said to be an* equivalent completion *of π_1 if it is a complete execution and is a conflict-equivalent extension of π_1.*

Note that if an execution $\alpha_1 \beta_1 \cdots \alpha_n \beta_n$ is a conflict-equivalent extension of $\pi = \alpha_1 \cdots \alpha_n$, then the execution $\alpha_1 \cdots \alpha_n \beta_1 \cdots \beta_n$ is also a conflict-equivalent extension of π.

LEMMA 5. *Let π_{ni} be a well-locked ni-execution with a schedule $\alpha_1 \cdots \alpha_k$. Let π_e be a conflict-equivalent extension of π_{ni} with a schedule $\alpha_1 \beta_1 \cdots \alpha_k \beta_k$. Assume that t_i blocks an object u at the end of α_i in π_e. Then, the execution of $\alpha_{i+1} \cdots \alpha_k$ in π_{ni} does not access u* [6] .

Proof Let σ denote the state at the end of α_i in π_{ni}. For any object x in σ accessed by the execution of $\alpha_{i+1} \cdots \alpha_k$ in π_{ni} we define the path P_x inductively as follows. If x is an exposed object in σ, then P_x is defined to be the sequence $[x]$. If x is a hidden object in σ, then the execution of $\alpha_{i+1} \cdots \alpha_k$ must have dereferenced some field of some object that pointed to x. Consider the first field $y.f$ dereferenced by $\alpha_{i+1} \cdots \alpha_k$ that pointed to x, where y represents an object. We define P_x to consist of the sequence P_y followed by x.

[6] Note that in this case t_i might not actually block object u at the end of α_i in π_{ni}.

Assume that u is accessed during the execution of $\alpha_{i+1} \cdots \alpha_k$ in π_{ni}. Hence P_u exists at the end of α_i in π_{ni}. By the definition of a conflict-equivalent extension, P_u also exists at the end of execution of α_i in π_e. (in particular, for $1 \leq j \leq i$ the execution of β_j in π_e does not access any object in P_u). Hence, t_i must hold a lock on some object y in this path (at the end of α_i in both π_{ni} as well as π_e). Since π_{ni} is well-locked, the execution of $\alpha_{i+1} \cdots \alpha_k$ in π_{ni} could not have locked y which is a contradiction. Hence u is not accessed during the execution of $\alpha_{i+1} \cdots \alpha_k$ in π_{ni}

LEMMA 6. *Let $\pi = \alpha_1 \cdots \alpha_n$ be a well-locked fully-ordered execution with at least one incomplete transaction. Let t_k be the first incomplete transaction in π (i.e., k is the minimal number such that t_k is incomplete). If every sequential-execution of a module follows domination locking and is completable, then π has an equivalent extension $\alpha_1 \cdots \alpha_k \beta_k \alpha_{k+1} \cdots \alpha_n$ in which transaction t_k is completed.*

Proof Since $\alpha_1 \cdots \alpha_k$ represents a sequential-execution, it has a completion $\alpha_1 \cdots \alpha_k \beta_k$ that follows domination locking.
We consider the following cases.

Case 1: After π transaction t_k is in phase-2.
Let σ represent the state produced by the execution of $\alpha_1 \cdots \alpha_k$.
From Lemma 4, all objects in σ accessed during the execution of β_k (in $\alpha_1 \cdots \alpha_k \beta_k$) must be blocked by t_k in σ.
From Lemma 5 the execution of $\alpha_{k+1} \cdots \alpha_n$ (in $\alpha_1 \cdots \alpha_n$) cannot access any object blocked by t_k in σ.
Hence the schedule $\alpha_1 \cdots \alpha_k \beta_k \alpha_{k+1} \cdots \alpha_n$ is feasible and is a conflict-equivalent extension of $\alpha_1 \cdots \alpha_n$.

Case 2: $k = n$
Here $\alpha_1 \cdots \alpha_k \beta_k$ is the equivalent extension.

Case 3: $k < n$, and after π transaction t_k is in phase-1
Let σ represent the state produced by the execution of $\alpha_1 \cdots \alpha_{k-1}$.
Let $k < m \leq n$.
Because of Lemma 1, no conflict-dependence can exist between the running transactions (because they are all in phase-1), hence $\alpha_1 \cdots \alpha_{k-1} \alpha_m$ represents a feasible sequential execution that follows domination locking.

Let u be an exposed object in σ that is accessed by t_k in $\alpha_1 \cdots \alpha_{k-1} \alpha_k \beta_k$, we will show that u is not accessed by t_m in $\alpha_1 \cdots \alpha_{k-1} \alpha_m$.
If u is accessed or locked by α_k in $\alpha_1 \cdots \alpha_{k-1} \alpha_k \beta_k$, then u is not accessed or locked by α_m in $\alpha_1 \cdots \alpha_{k-1} \alpha_m$ (because t_k and t_m have no conflict in π).
Otherwise, u is locked by β_k in $\alpha_1 \cdots \alpha_{k-1} \alpha_k \beta_k$.
Let σ' denote the state produced by the execution of π.
$\max(\sigma', t_m) <_h \max(\sigma', t_k)$ (because after the fully-ordered execution π, t_k and t_m are in phase-1 and t_k precedes t_m).
$\max(\sigma', t_k) <_h u$ (because of condition 2).
Hence, $\max(\sigma', t_m) <_h u$
Hence, u is not accessed or locked by t_m in $\alpha_1 \cdots \alpha_{k-1} \alpha_m$.

Let v be a hidden object in σ that is accessed by t_k in $\alpha_1 \cdots \alpha_{k-1} \alpha_k \beta_k$. we will show that v is not accessed by t_m in $\alpha_1 \cdots \alpha_{k-1} \alpha_m$.
v is necessarily reachable from exposed objects in σ, hence there exists a path P (in σ) from an exposed object w to v, such that w is the only exposed object in P.
t_k accesses w in $\alpha_1 \cdots \alpha_{k-1} \alpha_k \beta_k$ (because of conditions 4 and 1).
Assume that v is accessed by t_m in $\alpha_1 \cdots \alpha_{k-1} \alpha_m$, then t_m accesses w in $\alpha_1 \cdots \alpha_{k-1} \alpha_m$ (conditions 4 and 1). But we have showed that this is not possible for exposed objects. Therefore v is not accessed by t_m in $\alpha_1 \cdots \alpha_{k-1} \alpha_m$.

We have showed, for every $k < m \leq n$, t_k does not access (in $\alpha_1 \cdots \alpha_{k-1} \alpha_k \beta_k$) any object that is accessed by t_m (in $\alpha_1 \cdots \alpha_{k-1} \alpha_m$). Hence, $\alpha_1 \cdots \alpha_k \beta_k \alpha_{k+1} \cdots \alpha_n$ is an equivalent extension of $\alpha_1 \cdots \alpha_n$.

LEMMA 7. *Let $\pi = \alpha_1 \cdots \alpha_n$ be a well-locked fully-ordered execution. If every sequential-execution of a module follows domination locking and is completable, then π has an equivalent completion $\alpha_1 \beta_1 \cdots \alpha_n \beta_n$.*

Proof If π is not a complete execution, we can construct an equivalent completion $\alpha_1 \beta_1 \cdots \alpha_n \beta_n$ by repeatedly applying Lemma 6.

LEMMA 8. *Let $\xi = \xi_p \xi_t \xi_s$ be any feasible well-locked nischedule, where ξ_t is the schedule of a transaction t. If $\xi \cdot (t, e)$ is feasible, then $\xi_p \xi_t \cdot (t, e)$ is also feasible.*

Proof Assume that $\xi \cdot (t, e)$ is feasible. We show that $\xi_p \xi_t \cdot (t, e)$ is feasible. The only sources of infeasibility are when the step (t, e) involves a conditional branch (i.e., an `assume` statement) or an attempt to acquire a lock. We make the simplifying assumption that an `assume` statement refers to only thread-local variables. (Note that there is no loss of generality here since any statement "`assume e`" can be rewritten as "`x = e; assume x`" where `x` is a thread-local variable.) As a result, $\xi_p \xi_t \cdot (t, e)$ must be feasible if (t, e) involves a conditional branch. Now, consider the case where (t, e) involves an "`acquire x`" instruction where `x` is a thread-local variable. If the object `x` points to is unlocked at the end of $\xi_p \xi_t \xi_s$, it must be unlocked at the end of $\xi_p \xi_t$ as well. Hence, feasibility follows in this case as well.

LEMMA 9. *If every sequential-execution of a module follows domination locking and is completable, then every ni-execution is well-locked.*

Proof We prove by induction on the length of the executions. Let ξ be a schedule of a well-locked ni-execution. We will prove that if $\xi \cdot (t, e)$ is feasible, then it is a schedule of a well-locked execution. Assume that after ξ, the step (t, e) accesses an object u. From Lemma 3, ξ is conflict-equivalent to a fully-ordered ni-schedule $\xi' = \alpha_1 \cdots \alpha_n$.
We consider the following cases.

Case 1: there exists i such that $t = t_i$ and $1 \leq i < n$.
From Lemma 8, $\alpha_1 \cdots \alpha_i \cdot (t_i, e)$ is a feasible schedule. From the induction hypothesis, t_i holds a lock on u after $\alpha_1 \cdots \alpha_i$. Hence, t_i holds a lock on u after $\xi' = \alpha_1 \cdots \alpha_n$. Hence, t_i holds a lock on u after ξ.

Case 2: $t = t_n$.
From Lemma 7, ξ' has an equivalent completion with the schedule $\alpha_1\beta_1 \cdots \alpha_n\beta_n$.
We define $\xi'' = \alpha_1\beta_1 \cdots \alpha_{n-1}\beta_{n-1}\alpha_n$ (this is a prefix of $\alpha_1\beta_1 \cdots \alpha_n\beta_n$).
The step (t_n, e) accesses u after ξ'' (because t_n has the same local state after ξ' and ξ''). Since $\xi'' \cdot (t_n, e)$ represents a sequential execution, u is locked by t_n after ξ''. Hence, t_n holds a lock on u after $\alpha_1 \cdots \alpha_n$. Hence, t_n holds a lock on u after ξ.

Case 3: t does not appear in ξ
According the definition of a schedule, the first step of a transaction does not access an object.

LEMMA 10. *If every sequential-execution of a module follows domination locking and is completable, then every execution π is conflict-equivalent to a fully-ordered execution π' such that a transaction t completes before a transaction t' begins in π' if t completes before t' begins in π.*

Proof We prove this by induction on the length of the execution. Consider any execution with a schedule $\xi \cdot (t_i, e)$. By the inductive hypothesis, the execution of ξ is conflict-equivalent to a fully-ordered execution with the schedule $\xi' = \alpha_1 \cdots \alpha_k$[7] such that a transaction t completes before a transaction t' begins in ξ' if t completes before t' begins in ξ.
From Lemma 9, ξ' is well-locked.
We consider the following cases:

Case 1: After $\alpha_1 \cdots \alpha_i$, transaction t_i is in phase 2, and (t_i, e) does not access an heap object.
In this case, $\alpha_1 \cdots \alpha_k \cdot (t_i, e)$ is conflict equivalent to $\alpha_1 \cdots \alpha_i \cdot (t_i, e) \cdot \alpha_{i+1} \cdots \alpha_k$

[7] Note that $1 \leq i \leq k$ and α_i may be empty

Case 2: After $\alpha_1 \cdots \alpha_i$, transaction t_i is in phase 2, and (t_i, e) accesses an heap object u.
According to Lemma 7, $\xi' = \alpha_1 \cdots \alpha_k$ has an equivalent completion $\xi'' = \alpha_1\beta_1 \cdots \alpha_k\beta_k$.
Let $\xi''' = \alpha_1\beta_1 \cdots \alpha_{i-1}\beta_{i-1}\alpha_i$ (ξ''' is a prefix of ξ'').
t_i has the same local state after $\alpha_1 \cdots \alpha_i$ and ξ''' (according the definition of conflict-equivalent extension).
According to Lemma 8, $\alpha_1 \cdots \alpha_i \cdot (t_i, e)$ is a feasible schedule, so $\xi''' \cdot (t_i, e)$ is also a feasible schedule.
Also $\xi''' \cdot (t_i, e)$ represents a sequential execution (which follows domination locking).
Hence according to Lemma 4, t_i blocks u after ξ'''.
Hence, t_i blocks u after α_i in ξ''.
Hence, according to Lemma 5, $\alpha_{i+1} \cdots \alpha_k$ does not access u in $\xi' = \alpha_1 \cdots \alpha_k$.
Therefore, $\alpha_1 \cdots \alpha_k \cdot (t_i, e)$ is conflict equivalent to $\alpha_1 \cdots \alpha_i \cdot (t_i, e) \cdot \alpha_{i+1} \cdots \alpha_k$

Case 3: Transaction t_i is in phase 1 after $\alpha_1 \cdots \alpha_i$.
Because of Lemma 1, we can reorder all phase-1 transactions and t_i (even if t_i is in phase-2 after $\alpha_1 \cdots \alpha_k \cdot (t_i, e)$).
If t_i is in phase-2 after $\alpha_1 \cdots \alpha_k \cdot (t_i, e)$, then we can construct the fully-ordered equivalent execution by moving $\alpha_i \cdot (t_i, e)$ just before all the phase-1 transactions.
Otherwise (t_i is still in phase-1 after $\alpha_1 \cdots \alpha_k \cdot (t_i, e)$), we can construct the fully-ordered equivalent execution by moving $\alpha_i \cdot (t_i, e)$ to the right place according to the $\max$ values (between the phase-1 transactions).

THEOREM B.1. *If every sequential-execution of a module follows domination locking and is completable, then every execution of the module is strict conflict-serializable.*

Proof Immediate from Lemma 10.

THEOREM B.2. *If every sequential-execution of a module follows domination locking and is completable, then every execution of the module is a prefix of a complete-execution.*

Proof Consider any execution π. According to Lemma 10, there exists a fully-ordered execution $\pi' = \alpha_1 \cdots \alpha_n$ which is conflict equivalent to π. According to Lemma 7, π' has an equivalent completion $\alpha_1\beta_1 \cdots \alpha_n\beta_n$. According the definition of conflict-equivalent extension, there exists execution $\alpha_1 \cdots \alpha_n\beta_1 \cdots \beta_n$. Hence, π' is a prefix of a complete execution. Since π and π' end with the same state, π is also a prefix of a complete execution.