Concurrent Libraries with Foresight

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

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

Linearizable libraries provide operations that appear to execute atomically. Clients, however, may need to execute a sequence of operations (a *composite operation*) atomically. We consider the problem of extending a linearizable library to support arbitrary atomic composite operations by clients. We introduce a novel approach in which the concurrent library ensures atomicity of composite operations by exploiting information (*foresight*) provided by its clients. We use a correctness condition, based on a notion of dynamic right-movers, that guarantees that composite operations execute atomically without deadlocks, and without using rollbacks.

We present a static analysis to infer the foresight information required by our approach, allowing a compiler to automatically insert the foresight information into the client. This relieves the client programmer of this burden and simplifies writing client code.

We present a generic technique for extending the library implementation to realize foresight-based synchronization. This technique is used to implement a general-purpose Java library for Map data structures — the library permits composite operations to simultaneously work with multiple instances of Map data structures.

We use the Maps library and the static analysis to enforce atomicity of a wide selection of real-life Java composite operations. Our experiments indicate that our approach enables realizing efficient and scalable synchronization for real-life composite operations.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.3 [*Programming Languages*]: Language Constructs and Features—Abstract data types, Concurrent programming structures, Data types and structures

Keywords Concurrency, Composition, Transactions, Data Structures, Automatic Synchronization

PLDI'13, June 16–19, 2013, Seattle, WA, USA.

1. Introduction

Writing concurrent software is hard and error prone. To ease the programmer's burden, programming languages such as Java, Scala, and C# provide libraries of efficient concurrent data structures. These libraries provide operations that are guaranteed to be *atomic*, while hiding the complexity of the implementation from clients. Unfortunately, clients often need to perform a *sequence of library operations that appears to execute atomically*, referred to hereafter as an *atomic composite operation*.

The problem of realizing atomic composite operations is an important and widespread one [9]. Atomic composite operations are a restricted form of *software transactions* [15]. However, general-purpose software transaction implementations have not gained acceptance [10] due to their high overhead as well as difficulties in using libraries that are incompatible with software transactions. Programmers typically realize such composite operations using ad-hoc synchronization leading to many concurrency bugs in practice [27].

Concurrency Control with Foresight In this paper, we address the problem of extending a linearizable library [19] to allow clients to execute an arbitrary composite operation atomically. Our basic methodology requires the client code to demarcate the sequence of operations for which atomicity is desired and provide declarative information to the library (*foresight*) about the library operations that the composite operation may invoke (as illustrated later). It is the library's responsibility to ensure the desired atomicity, exploiting the foresight information for effective synchronization.

Our first contribution is a formalization of this approach. We formalize the desired goals and present a sufficient correctness condition. As long as the clients and the library extension satisfy the correctness condition, all composite operations are guaranteed atomicity without deadlocks. Furthermore, our condition does not require the use of rollbacks. Our sufficiency condition is broad and permits a range of implementation options and fine-grained synchronization. It is based on a notion of *dynamic right-movers*, which generalizes traditional notions of static right-movers and commutativity [21, 23].

Our formulation decouples the implementation of the library from the client. Thus, the correctness of the client does not depend on the way the foresight information is used by library implementation. The client only needs to ensure the correctness of the foresight information.

Automatic Foresight for Clients We then present a simple static analysis to infer calls (in the client code) to the API used to pass the foresight information. Given a description of a library's API, our algorithm conservatively infers the required calls. This relieves

the client programmer of this burden and simplifies writing atomic composite operations.

Library Extension Realization Our approach permits the use of customized, hand-crafted, implementations of the library extension. However, we also present a generic technique for extending a linearizable library with foresight. The technique is based on a variant of the tree locking protocol in which the tree is designed according to semantic properties of the library's operations.

We used our technique to implement a general-purpose Java library for Map data structures. Our library permits composite operations to simultaneously work with multiple instances of Map data structures.

Experimental Evaluation We use the Maps library and the static analysis to enforce atomicity of a wide selection of real-life Java composite operations, including composite operations that manipulate multiple instances of Map data structures. Our experiments indicate that our approach enables realizing efficient and scalable synchronization for real-life composite operations.

Main Contributions We develop the concept of *concurrent libraries with foresight* along several dimensions, providing the theoretical foundations, an implementation methodology, and an empirical evaluation. Our main contributions are:

- We introduce the concept of concurrent libraries with foresight, in which the concurrent library ensures atomicity of composite operations by exploiting information (foresight) provided by its clients. The main idea is to shift the responsibility of synchronizing composite operations of the clients to the hands of the library, and have the client provide useful foresight information to make efficient library-side synchronization possible.
- We define a sufficient correctness condition for clients and the library extension. Satisfying this condition guarantees atomicity and deadlock-freedom of composite operations (Sec. 4).
- We show how to realize both the client-side (Sec. 5) and the library-side (Sec. 6) for leveraging foresight. Specifically, we present a static analysis algorithm that provides foresight information to the library (Sec. 5), and show a generic technique for implementing a family of libraries with foresight (Sec. 6).
- We realized our approach and evaluated it on a number of real-world composite operations. We show that our approach provides efficient synchronization (Sec. 7).

2. Overview

We now present an informal overview of our approach, for extending a linearizable library into a library with foresight-based synchronization, using a toy example. Fig. 1 presents the specification of a single Counter (library). The counter can be incremented (via the `Inc()` operation), decremented (via the `Dec()` operation), or read (via the `Get()` operation). The counter's value is always nonnegative: the execution of `Dec()` has an effect only when the counter's value is positive. All the counter's procedures are atomic.

Fig. 2 shows an example of two threads each executing a *composite operation*: a code fragment consisting of multiple counter operations. (The **mayUse** annotations will be explained later.) Our goal is to execute these composite operations atomically: a *serializable* execution of these two threads is one that is equivalent to either thread T_1 executing completely before T_2 executes or vice versa. Assume that the counter value is initially zero. If T_2 executes first, then neither decrement operation will change the counter value, and the subsequent execution of T_1 will produce a counter value of 2. If T_1 executes first and then T_2 executes, the final value of the counter will be 0. Fig. 3 shows a slightly more complex example.

```
int value = I;
void Inc() { atomic { value=value+1; } }
void Dec() { atomic { if (value > 0) then value=value-1; } }
int Get() { atomic { return value; } }
```

Figure 1. Specification of the Counter library. ***I*** denotes the initial value of the counter.

```
1  /* Thread T1 */
2  /* @atomic */ {
3    @mayUseInc()
4    Inc();
5    Inc();
6    @mayUseNone()
7  }
```

```
1  /* Thread T2 */
2  /* @atomic */ {
3    @mayUseDec()
4    Dec();
5    Dec();
6    @mayUseNone()
7  }
```

Figure 2. Simple compositions of counter operations.

```
1  /* Thread T1 */
2  /* @atomic */ {
3    @mayUseAll()
4    c = Get();
5    @mayUseInc()
6    while (c > 0) {
7      c = c-1;
8      Inc();
9    }
10   @mayUseNone()
11 }
```

```
1  /* Thread T2 */
2  /* @atomic */ {
3    @mayUseDec()
4    Dec();
5    Dec();
6    @mayUseNone()
7  }
```

Figure 3. Compositions of counter dependent operations.

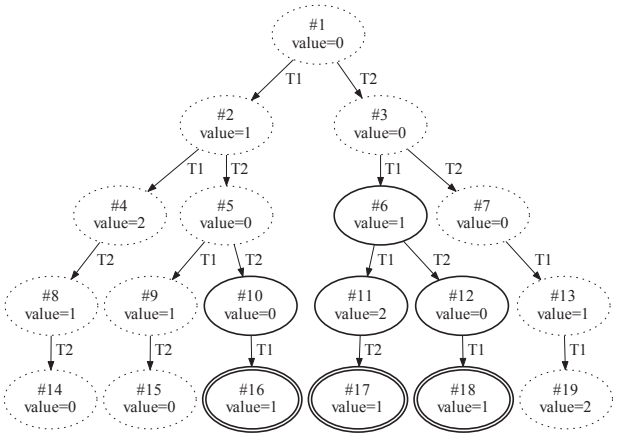

Figure 4. Execution prefixes of the code shown in Fig. 2, for a counter with $I = 0$. Each node represents a prefix of an execution; a leaf node represents a complete execution.

2.1 Serializable and Serializably-Completable Executions

Fig. 4 shows prefixes of various interleaved executions of the code shown in Fig. 2 for an initial counter value of 0. Nodes are annotated with the values of the counter. Bold double circles depict *non-serializable* nodes: these nodes denote execution prefixes that are not serializable. E.g., node $\#18$ is a non-serializable node since it represents the non-serializable execution $T_2.Dec(); T_1.Inc(); T_2.Dec(); T_1.Inc()$ (which produces a final counter value of 1).

Bold single circles depict *doomed* nodes: once we reach a doomed node, there is no way to order the remaining operations in a way that achieves serializability. E.g., node #6 is a doomed node since it only leads to non-serializable complete executions (represented by nodes #17 and #18). Finally, dashed circles depict *safe* nodes, which represent *serializably-completable* executions. We formalize this notion later, but safe nodes guarantee that the execution can make progress, while ensuring serializability.

Our goal is to ensure that execution stays within safe nodes. Even in this simple example, the set of safe nodes and, hence, the potential for parallelism depends on the initial value I of the counter. For $I \geq 1$ all nodes are safe and thus no further synchronization is necessary. Using our approach enables realizing all available parallelism in this example (for every $I \geq 0$), while avoiding the need for any backtracking (i.e., rollbacks).

2.2 Serializably-Completable Execution: A Characterization

We now present a characterization of serializably-completable executions based on a generalization of the notion of *static right movers* [23]. We restrict ourselves to executions of two threads here, but our later formalization considers the general case.

We define an operation o by a thread T to be a *dynamic right-mover with respect to thread T' after an execution p*, iff for any sequence of operations s executed by T', if $p; T.o; s$ is feasible, then $p; s; T.o$ is feasible and equivalent to the first execution. Given an execution ξ, we define a relation $\sqsubset$ on the threads as follows: $T \sqsubset T'$ if ξ contains a prefix $p; T.o$ such that o is not a dynamic right-mover with respect to T' after p. As shown later, if $\sqsubset$ is acyclic, then ξ is a serializably-completable execution (as long as every sequential execution of the threads terminates).

In Fig. 4, node #2 represents the execution prefix $T_1.Inc()$ for which $T_1 \sqsubset T_2$.This is because $T_2.Dec()$ is a possible suffix executed by T_2, and $T_1.Inc(); T_2.Dec()$ is not equivalent to $T_2.Dec(); T_1.Inc()$. On the other hand, node #5 represents the execution prefix $T_1.Inc(); T_2.Dec()$ for which $T_2 \not\sqsubset T_1$.This execution has one possible suffix executed by T_1 (i.e., $T_1.Inc()$), and the execution $T_1.Inc(); T_2.Dec(); T_1.Inc()$ is equivalent to the execution $T_1.Inc(); T_1.Inc(); T_2.Dec()$.

Observe that the relation $\sqsubset$ corresponding to any non-serializable or doomed node has a cycle, while it is acyclic for all safe nodes.

Note that the use of a *dynamic* (i.e., *state-dependent*) right-mover relation is critical to a precise characterization above. E.g., Inc and Dec are not static right-movers with respect to each other.

2.3 Synchronization Using Foresight

We now show how we exploit the above characterization to ensure that an interleaved execution stays within safe nodes.

Foresight. A key aspect of our approach is to exploit knowledge about the possible future behavior of composite operations for more effective concurrency control. We enrich the interface of the library with operations that allow the composite operations to assert temporal properties of their future behavior. In the Counter example, assume that we add the following operations:

- **mayUseAll()**: indicates transaction may execute arbitrary operations in the future.
- **mayUseNone()**: indicates transaction will execute no more operation.
- **mayUseDec()**: indicates transaction will invoke only `Dec` operations in the future.
- **mayUseInc()**: indicates transaction will invoke only `Inc` operations in the future.

The code in Fig. 2 is annotated with calls to these operations in a straightforward manner. The code shown in Fig. 3, is conservatively annotated with a call to mayUseAll() since the interface does not provide a way to indicate that the transaction will only invoke `Get` and `Inc` operations.

Utilizing Foresight. We utilize a suitably modified definition of the dynamic right mover relation, where we check for the right mover condition only with respect to the set of all sequences of operations the other threads are allowed to invoke (as per their foresight assertions). To utilize foresight information, a library implementation maintains a conservative over-approximation $\sqsubset'$ of the $\sqsubset$ relation. The implementation permits an operation to proceed iff it will not cause the relation $\sqsubset'$ to become cyclic (and blocks the operation otherwise until it is safe to execute it). This is sufficient to guarantee that the composite operations appear to execute atomically, without any deadlocks.

We have created an implementation of the counter that (implicitly) maintains a conservative over-approximation $\sqsubset'$ (see [14]). Our implementation permits all serializably-completable execution prefixes for the example shown in Fig. 2 (for every $I \geq 0$). Our implementation also provides high degree of parallelism for the example shown in Fig. 3 — for this example the loop of T_1 can be executed in parallel to the execution of T_2.

Fine-grained Foresight. We define a library operation to be a tuple that identifies a procedure as well as the values of the procedure's arguments. For example, `removeKey(1)` and `removeKey(2)` are two **different** operations of a library with the procedure `removeKey(int k)`. In order to distinguish between different operations which are invoked using the same procedure, a *mayUse* procedure (which is used to pass the foresight information) can have parameters. For example, a library that represents a single Map data structure can have a *mayUse* procedure `mayUseKey(int k)`, where `mayUseKey(1)` is defined to refer to all operations on key 1 (including, for example, `removeKey(1)`), and `mayUseKey(2)` is defined to refer to all operations on key 2 (including, for example, `removeKey(2)`).

Special Cases. Our approach generalizes several ideas that have been proposed before. One example, from databases, is locking that is based on operations-commutativity (e.g., see [8, chapter 3.8]). Such locking provides several lock modes where each mode corresponds to a set of operations; two threads are allowed to execute in parallel as long as they do not hold lock modes that correspond to non-commutative operations. A simple common instance is a read-write lock [12], in which threads are allowed to simultaneously hold locks in a read-mode (which corresponds with read-only operations that are commutative with each other). Interestingly, the common `lock-acquire` and `lock-release` operations used for locking, can be seen as special cases of the procedures used to pass the foresight information.

2.4 Realizing Foresight Based Synchronization

What we have described so far is a methodology and formalism for foresight-based concurrency control. This prescribes the conditions that must be satisfied by the clients and library implementations to ensure atomicity for composite operations.

Automating Foresight For Clients. One can argue that adding calls to *mayUse* operations is an error prone process. Therefore, in Section 5 we show a simple static analysis algorithm which conservatively infers calls to *mayUse* operations (given a description of the *mayUse* operations supported by the library). Our experience indicates that our simple algorithm can handle real-life programs.

Library Implementation. We permit creating customized, handcrafted, implementations of the library extension (a simple example is demonstrated in [14]). However, in order to simplify creating such libraries, we present a generic technique for implementing a family of libraries with foresight (Section 6). The technique is based on a variant of the tree locking protocol in which the

tree is designed according to semantic properties of the library's operations. We have utilized the technique to implement a general purpose Java library for Map data structures. Our library permits a composite operation to simultaneously work with multiple maps.

3. Preliminaries

3.1 Libraries

A library $\mathcal{A}$ exposes a set of procedures $\text{PROCS}_{\mathcal{A}}$. We define a library *operation* to be a tuple $(p, v_1, \cdots, v_k)$ consisting of a procedure name p and a sequence of values (representing actual values of the procedure arguments). The set of operations of a library $\mathcal{A}$ is denoted by $\text{OP}_{\mathcal{A}}$. Library operations are invoked by client threads (defined later). Let $\mathcal{T}$ denote the set of all thread identifiers. An *event* is a tuple (t, m, r), where t is a thread identifier, m is a library operation, and r is a return value. An event captures both an operation invocation as well as its return value.

A *history* is defined to be a finite sequence of events. The semantics of a library $\mathcal{A}$ is captured by a set of histories $H_{\mathcal{A}}$. If $h \in H_{\mathcal{A}}$, then we say that h is *feasible* for $\mathcal{A}$. Histories capture the interaction between a library and its client (a set of threads). Though multiple threads may concurrently invoke operations, this simple formalism suffices in our setting, since we assume the library to be linearizable. An *empty history* is an empty sequence of events.

Let $h \circ h'$ denote the concatenation of history h' to the end of history h. Note that the set $H_{\mathcal{A}}$ captures multiple aspects of the library's specification. If h is feasible, but $h \circ (t, m, r)$ is not, this could mean one of three different things: r may not be a valid return value in this context, or t is not allowed to invoke m is this context, or t is allowed to invoke m in this context, but the library will block and not return until some other event has happened.

A library $\mathcal{A}$ is said to be *total* if for any thread t, operation $m \in \text{OP}_{\mathcal{A}}$ and $h \in H_{\mathcal{A}}$, there exists r such that $h \circ (t, m, r) \in H_{\mathcal{A}}$.

3.2 Clients

We now briefly describe clients and their semantics. (A more complete definition appears in [14].) A client $t_1 \mid\mid t_2 \mid\mid \cdots \mid\mid t_n$ consists of the parallel composition of a set of client programs t_i (also referred to as threads). Each t_i is a sequential program built out of two types of statements: statements that change only the thread's local-state, and statements that invoke a library operation. Threads have no shared state except the (internal) state of the library, which is accessed or modified only via library operations.

The semantics $[\![t_i]\!]$ of a single thread t_i is defined to be a labelled transition system $(\Sigma_i, \Rightarrow_i)$ over a set of thread-local states Σ_i, with some states designated as initial and final states. The execution of any instruction other than a library operation invocation is represented by a (thread-local) transition $\sigma \overset{\epsilon}{\Rightarrow}_i \sigma'$. The execution of a library operation invocation is represented by a transition of the form $\sigma \overset{e}{\Rightarrow}_i \sigma'$, where event e captures both the invocation as well as the return value. This semantics captures the semantics of the "open" program t_i. When t_i is "closed" by combining it with a library $\mathcal{A}$, the semantics of the resulting closed program is obtained by combining $[\![t_i]\!]$ with the semantics of $\mathcal{A}$, as illustrated later.

A t_i-execution is defined to be a sequence of t_i-transitions $s_0 \overset{a_1}{\Rightarrow}_i s_1, s_1 \overset{a_2}{\Rightarrow}_i s_2, \cdots, s_{k-1} \overset{a_k}{\Rightarrow}_i s_k$ such that s_0 is an initial state of t_i and every a_j is either ϵ or an event. Such an execution is said to be *complete* if s_k is a final state of t_i.

The semantics of a client $\mathcal{C} = t_1 \mid\mid \cdots \mid\mid t_n$ is obtained by composing the semantics of the individual threads, permitting any arbitrary interleaving of the executions of the threads. We define the set of transitions of $\mathcal{C}$ to be the disjoint union of the set of transitions of the individual threads. A $\mathcal{C}$-execution is defined to be a sequence ξ of $\mathcal{C}$-transitions such that each $\xi \mid t_i$ is a t_i-execution, where $\xi \mid t_i$ is the subsequence of ξ consisting of all t_i-transitions.

We now define the semantics of the composition of a client $\mathcal{C}$ with a library $\mathcal{A}$. Given a $\mathcal{C}$-execution ξ, we define $\phi(\xi)$ to be the sequence of event labels in ξ. The set of $(\mathcal{C}, \mathcal{A})$-executions is defined to be the set of all $\mathcal{C}$-executions ξ such that $\phi(\xi) \in H_{\mathcal{A}}$. We abbreviate "$(\mathcal{C}, \mathcal{A})$-execution" to *execution* if no confusion is likely.

Threads as Transactions Our goal is to enable threads to execute code fragments containing multiple library operations as atomic transactions (i.e., in *isolation*). For notational simplicity, we assume that we wish to execute each thread as a single transaction. (Our results can be generalized to the case where each thread may wish to perform a sequence of transactions.) In the sequel, we may think of threads and transactions interchangeably. This motivates the following definitions.

Non-Interleaved and Sequential Executions An execution ξ is said to be a *non-interleaved execution* if for every thread t all t-transitions in ξ appear contiguously. Thus, a non-interleaved execution ξ is of the form $\xi_1, \cdots, \xi_k$, where each ξ_i represents a different thread's (possibly incomplete) execution. Such a non-interleaved execution is said to be a *sequential execution* if for each $1 \leq i < k$, ξ_i represents a complete thread execution.

Serializability Two executions ξ and ξ' are said to be *equivalent* iff for every thread t, $\xi \mid t = \xi' \mid t$. An execution ξ is said to be *serializable* iff it is equivalent to some non-interleaved execution.

Serializably Completable Executions For any execution ξ, let $W(\xi)$ denote the set of all threads that have at least one transition in ξ. An execution ξ is said to be a *complete execution* iff $\xi \mid t$ is complete for every thread $t \in W(\xi)$. A client execution ξ is *completable* if ξ is a prefix of a complete execution ξ_c such that $W(\xi) = W(\xi_c)$. An execution ξ is said to be *serializably completable* iff ξ is a prefix of a complete serializable execution ξ_c such that $W(\xi) = W(\xi_c)$. Otherwise, we say that ξ is a *doomed* execution.

An execution may be incompletable due to problems in a client thread (e.g., a non-terminating loop) or due to problems in the library (e.g., *blocking* by a library procedure leading to deadlocks).

4. Foresight-Based Synchronization

We now formalize our goal of extending a base library $\mathcal{B}$ into a foresight-based library $\mathcal{E}$ that permits clients to execute arbitrary composite operations atomically.

4.1 The Problem

Let $\mathcal{B}$ be a given total library. (Note that $\mathcal{B}$ can also be considered to be a specification.) We say that a library $\mathcal{E}$ is a *restrictive extension* of $\mathcal{B}$ if (i) $\text{PROCS}_{\mathcal{E}} \supset \text{PROCS}_{\mathcal{B}}$, (ii) $\{h \downarrow \mathcal{B} \mid h \in H_{\mathcal{E}}\} \subseteq H_{\mathcal{B}}$, where $h \downarrow \mathcal{B}$ is the subsequence of events in h that represent calls of operations in $\text{OP}_{\mathcal{B}}$, and (iii) $\text{PROCS}_{\mathcal{E}} \setminus \text{PROCS}_{\mathcal{B}}$ do not have a return value. We are interested in extensions where the extension procedures ($\text{PROCS}_{\mathcal{E}} \setminus \text{PROCS}_{\mathcal{B}}$) are used for synchronization to ensure that each thread appears to execute in isolation.

Given a client $\mathcal{C}$ of the extended library $\mathcal{E}$, let $\mathcal{C} \downarrow \mathcal{B}$ denote the program obtained by replacing every extension procedure invocation in $\mathcal{C}$ by the skip statement. Similarly, for any execution ξ of $(\mathcal{C}, \mathcal{E})$, we define $\xi \downarrow \mathcal{B}$ to be the sequence obtained from ξ by omitting transitions representing extension procedures. We say that an execution ξ of $(\mathcal{C}, \mathcal{E})$ is $\mathcal{B}$-serializable if $\xi \downarrow \mathcal{B}$ is a serializable execution of $(\mathcal{C} \downarrow \mathcal{B}, \mathcal{B})$. We say that ξ is $\mathcal{B}$-serializably-completable if $\xi \downarrow \mathcal{B}$ is a serializably completable execution of $(\mathcal{C} \downarrow \mathcal{B}, \mathcal{B})$. We say that $\mathcal{E}$ is a *transactional extension* of $\mathcal{B}$ if for any (correct) client $\mathcal{C}$ of $\mathcal{E}$, every $(\mathcal{C}, \mathcal{E})$-execution is $\mathcal{B}$-serializably-completable. Our goal is to build transactional extensions of a given library.

4.2 The Client Protocol

In our approach, the extension procedures are used by transactions (threads) to provide information to the library about the future operations they may perform. We refer to procedures in $\text{PROCS}_{\mathcal{E}} \setminus \text{PROCS}_{\mathcal{B}}$ as *mayUse* procedures, and to operations in $MU_{\mathcal{E}} = \text{OP}_{\mathcal{E}} \setminus \text{OP}_{\mathcal{B}}$ as *mayUse* operations. We now formalize the *client protocol*, which captures the preconditions the client must satisfy, namely that the foresight information provided via the *mayUse* operations must be correct.

The semantics of *mayUse* operations is specified by a function $may_{\mathcal{E}} : MU_{\mathcal{E}} \mapsto \mathcal{P}(\text{OP}_{\mathcal{B}})$ that maps every *mayUse* operation to a set of base library operations[1]. In Section 5 we show simple procedure annotations that can be used to define the set $MU_{\mathcal{E}}$ and the function $may_{\mathcal{E}}$.

The *mayUse* operations define an *intention-function* $I_{\mathcal{E}} : H_{\mathcal{E}} \times \mathcal{T} \mapsto \mathcal{P}(\text{OP}_{\mathcal{B}})$ where $I_{\mathcal{E}}(h,t)$ represents the set of (base library) operations thread t is allowed to invoke after the execution of h. For every thread $t \in \mathcal{T}$ and a history $h \in H_{\mathcal{E}}$, the value of $I_{\mathcal{E}}(h,t)$ is defined as follows. Let M denote the set of all *mayUse* operations invoked by t in h. (I) IF M IS NON-EMPTY, THEN $I_{\mathcal{E}}(h,t) = \bigcap_{m \in M} may_{\mathcal{E}}(m)$. (II) IF M IS EMPTY, THEN $I_{\mathcal{E}}(h,t) = \{\}$. We extend the notation and define $I_{\mathcal{E}}(h,T)$, for any set of threads T, to be $\bigcup_{t \in T} I_{\mathcal{E}}(h,t)$.

Once a thread executes its first *mayUse* operation, the intention set $I_{\mathcal{E}}(h,t)$ can only shrink as the execution proceeds. Subsequent *mayUse* operations cannot be used to increase the intention set.

DEFINITION 1 (Client Protocol). *Let h be a history of library $\mathcal{E}$. We say that h follows the client protocol if for any prefix $h' \circ (t,m,r)$ of h, we have $m \in I_{\mathcal{E}}(h',t) \cup MU_{\mathcal{E}}$.*

We say that an execution ξ follows the client protocol, if $\phi(\xi)$ follows the client protocol.

4.3 Dynamic Right Movers

We now consider how the library extension can exploit the foresight information provided by the client to ensure that the interleaved execution of multiple threads is restricted to safe nodes (as described in Section 2). First, we formalize the notion of a *dynamic right mover*.

Given a history h of a library $\mathcal{A}$, we define the set $E_{\mathcal{A}}[h]$ to be $\{h' \mid h \circ h' \in H_{\mathcal{A}}\}$. (Note that if h is not feasible for $\mathcal{A}$, then $E_{\mathcal{A}}[h] = \emptyset$.) Note that if $E_{\mathcal{A}}[h_1] = E_{\mathcal{A}}[h_2]$, then the concrete library states produced by h_1 and h_2 cannot be distinguished by any client (using any sequence of operations). Dually, if the concrete states produced by histories h_1 and h_2 are equal, then $E_{\mathcal{A}}[h_1] = E_{\mathcal{A}}[h_2]$.

DEFINITION 2 (Dynamic Right Movers). *Given a library $\mathcal{A}$, a history h_1 is said to be a dynamic right mover with respect to a history h_2 in the context of a history h, denoted $h : h_1 \triangleright_{\mathcal{A}} h_2$, iff*

$$E_{\mathcal{A}}[h \circ h_1 \circ h_2] \subseteq E_{\mathcal{A}}[h \circ h_2 \circ h_1].$$

An operation m is said to be a dynamic right mover with respect to a set of operations M_s in the context of a history h, denoted $h : m \triangleright_{\mathcal{A}} M_s$, iff for any event (t,m,r) and any history h_s consisting of operations in M_s, we have $h : (t,m,r) \triangleright_{\mathcal{A}} h_s$.

The following example shows that an operation m can be a dynamic right mover with respect to a set M after some histories but not after some other histories.

[1] Our approach can be extended to use a more precise semantic function $may_{\mathcal{E}} : MU_{\mathcal{E}} \mapsto \mathcal{P}(\text{OP}^*_{\mathcal{B}})$ that maps each *mayUse* operation to a set of sequence of operations, enabling client transactions to more precisely describe their future behavior.

EXAMPLE 4.1. *Consider the Counter described in Section 2. Let h_p be a history that ends with a counter value of $p > 0$. The operation `Dec` is a dynamic right mover with respect to the set $\{$`Inc`$\}$ in the context of h_p since for every n the histories $h_p \circ (t, Dec, r) \circ (t_1, Inc, r_1), \ldots, (t_n, Inc, r_n)$ and $h_p \circ (t_1, Inc, r_1), \ldots, (t_n, Inc, r_n) \circ (t, Dec, r)$ have the same set of suffixes (since the counter value is $p - 1 + n$ after both histories).*

Let h_0 be a history that ends with a counter value of 0. The operation `Dec` is not *a dynamic right mover with respect to the set $\{$`Inc`$\}$ in the context of h_0 since after a history $h_0 \circ (t, Dec, r) \circ (t', Inc, r')$ the counter's value is 1, and after $h_0 \circ (t', Inc, r') \circ (t, Dec, r)$ the counter's value is 0. Thus, $(t, Get, 1)$ is a feasible suffix after the first history but not the second.*

The following example shows that the dynamic right mover is not a symmetric property.

EXAMPLE 4.2. *Let h_i be a history that ends with a counter value of $i > 0$. The operation `Inc` is* not *a dynamic right mover with respect to the set $\{$`Dec`$\}$ in the context of h_i since after a history $h_i \circ (t, Inc, r) \circ (t_1, Dec, r_1), \ldots, (t_{i+1}, Dec, r_{i+1})$ the Counter's value is 0, and after $h_i \circ (t_1, Dec, r_1), \ldots, (t_{i+1}, Dec, r_{i+1}) \circ (t, Inc, r)$ the Counter's value is 1.*

One important aspect of the definition of dynamic right movers is the following: it is possible to have $h : m \triangleright_{\mathcal{A}} \{m_1\}$ and $h : m \triangleright_{\mathcal{A}} \{m_2\}$ but not $h : m \triangleright_{\mathcal{A}} \{m_1, m_2\}$.

Static Movers. We say that an operation m is a *static right mover* with respect to operation m', if every feasible history h satisfies $h : m \triangleright_{\mathcal{A}} \{m'\}$. We say that m and m' are *statically-commutative*, if m is a *static right mover* with respect to m' and vice versa.

4.4 Serializability

It follows from the preceding discussion that an incomplete history h may already reflect some execution-order constraints among the threads that must be satisfied by any other history that is equivalent to h. These execution-order constraints can be captured as a partial-ordering on thread-ids.

DEFINITION 3 (Safe Ordering). *Given a history h of $\mathcal{E}$, a partial ordering $\sqsubseteq \subseteq \mathcal{T} \times \mathcal{T}$, is said to be safe for h iff for any prefix $h' \circ (t,m,r)$ of h, where $m \in \text{OP}_{\mathcal{B}}$, we have $h' \downarrow \mathcal{B} : m \triangleright_{\mathcal{B}} I(h', T)$, where $T = \{t' \in \mathcal{T} \mid t \not\sqsubseteq t'\}$.*

A safe ordering represents a conservative over-approximation of the execution-order constraints among thread-ids (required for serializability). Note that in the above definition, the right-mover property is checked only with respect to the *base library* $\mathcal{B}$.

EXAMPLE 4.3. *Assume that the Counter is initialized with a value $I > 0$. Consider the history (return values omitted for brevity):*

$$h = (t, mayUseDec), (t', mayUseInc), (t, Dec), (t', Inc).$$

If $\sqsubseteq$ is a safe partial order for h, then $t' \sqsubseteq t$ because after the third event `Inc` is not *a dynamic right mover with respect to the operations allowed for t (i.e., $\{$`Dec`$\}$). Dually, the total order defined by $t' \sqsubseteq' t$ is safe for h since after the second event, the operation `Dec` is a dynamic right mover with respect to the operations allowed for t' (i.e., $\{$`Inc`$\}$) because the Counter's value is larger than 0.*

DEFINITION 4 (Safe Extension). *We say that library $\mathcal{E}$ is safe extension of $\mathcal{B}$, if for every $h \in H_{\mathcal{E}}$ that follows the client protocol there exists a partial ordering $\sqsubseteq_h$ on threads that is safe for h.*

The above definition prescribes the synchronization (specifically, *blocking*) that a safe extension must enforce. In particular,

assume that h is feasible history allowed by the library. If the history $h \circ (t, m, r)$ has no safe partial ordering, then the library must block the call to m by t rather than return the value r.

THEOREM 4.1 (Serializability). *Let $\mathcal{E}$ be a safe extension of a library $\mathcal{B}$. Let $\mathcal{C}$ be a client of $\mathcal{E}$. Any execution ξ of $(\mathcal{C}, \mathcal{E})$ that follows the client protocol is $\mathcal{B}$-serializable.*

The proofs appear in [14].

4.5 $\mathcal{B}$-Serializable-Completability

We saw in Section 2 and Fig. 4 that some serializable (incomplete) executions may be doomed: i.e., there may be no way of *completing* the execution in a serializable way. Safe extensions, however, ensure that all executions avoid doomed nodes and are serializably completable. However, we cannot guarantee completability if a client thread contains a non-terminating loop or violates the client protocol. This leads us to the following conditional theorem.

THEOREM 4.2 ($\mathcal{B}$-Serializable-Completability). *Let $\mathcal{E}$ be a safe extension of a total library $\mathcal{B}$. Let $\mathcal{C}$ be a client of $\mathcal{E}$. If every sequential execution of $(\mathcal{C}, \mathcal{E})$ follows the client protocol and is completable, then every execution of $(\mathcal{C}, \mathcal{E})$ is $\mathcal{B}$-serializably-completable.*

The precondition in the above theorem is worth noting. We require client threads to follow the client protocol and terminate. However, it is sufficient to check that clients satisfy these requirements in *sequential executions*. This simplifies reasoning about the clients.

4.6 $\mathcal{E}$-Completability

The preceding theorem about $\mathcal{B}$-Serializable-Completability has a subtle point: it indicates that *it is possible* to complete any execution of $(\mathcal{C}, \mathcal{E})$ in a serializable fashion *in* $\mathcal{B}$. The extended library $\mathcal{E}$, however, could choose to *block* operations unnecessarily and prevent progress. This is undesirable. We now formulate a desirable progress condition that the extended library must satisfy.

In the sequel we assume that every thread (transaction) always executes a *mayUse* operation m such that $may_{\mathcal{E}}(m) = \{\}$ before it terminates. (Essentially, this is an *end-transaction* operation.) Given a history h and a thread t, we say that t is *incomplete* after h iff $I_{\mathcal{E}}(h, t) \neq \emptyset$. We say that history h is *incomplete* if there exists some incomplete thread after h.

We say that a thread t is *enabled* after history h, if for all events (t, m, r) such that $h \circ (t, m, r)$ satisfies the client protocol and $h \circ (t, m, r) \downarrow \mathcal{B} \in H_{\mathcal{B}}$, we have $h \circ (t, m, r) \in H_{\mathcal{E}}$. Note that this essentially means that $\mathcal{E}$ will not block t from performing any legal operation.

DEFINITION 5 (Progress Condition). *We say that a library $\mathcal{E}$ satisfies the* progress condition *iff for every history $h \in H_{\mathcal{E}}$ that follows the client protocol the following holds:*

- *If h is incomplete, then at least one of the incomplete threads t is enabled after h.*
- *If h is complete, then every thread t that does not appear in h is enabled after h.*

THEOREM 4.3 ($\mathcal{E}$-Completability). *Let $\mathcal{E}$ be a safe extension of a total library $\mathcal{B}$ that satisfies the progress condition. Let $\mathcal{C}$ be a client of $\mathcal{E}$. If every sequential execution of $(\mathcal{C}, \mathcal{E})$ follows the client protocol and is completable, then every execution of $(\mathcal{C}, \mathcal{E})$ is completable and $\mathcal{B}$-serializable.*

4.7 Special Cases

In this subsection we describe two special cases of safe extension.

```
int createNewMap();
int put(int mapId,int k,int v);
int get(int mapId,int k);
int remove(int mapId,int k);
bool isEmpty(int mapId);
int size(int mapId);
```

Figure 5. Base procedures of the example Maps library.

Eager-Ordering Library Our notion of safe-ordering permits $\sqsubseteq$ to be a partial order. In effect, this allows the system to determine the execution-ordering between transactions lazily, only when forced to do so (e.g., when one of the transactions executes a non-right-mover operation). One special case of this approach is to use a total order on threads, eagerly ordering threads in the order in which they execute their first operations. The idea of shared-ordered locking [7] in databases is similar to this. Using such approach guarantees *strict-serializability* [25] which preserves the runtime order of the threads.

DEFINITION 6. *Given a history h we define an order $\leq_h$ of the threads in h such that: $t \leq_h t'$ iff $t = t'$ or the first event of t precedes the the first event of t' (in h).*

DEFINITION 7 (Eager-Ordering Library). *We say that library $\mathcal{E}$ is eager-ordering if for every $h \in H_{\mathcal{E}}$ that follows the client protocol, $\leq_h$ is safe for h.*

Commutative-Blocking Library A special case of eager-ordering library is *commutative-blocking library*. (This special case is common in the database literature, e.g., see [8, chapter 3.8]). The idea here is to ensure that two threads are allowed to execute concurrently only if any operations they can invoke commute with each other. This is achieved by treating each *mayUse* operation as a lock acquisition (on the set of operations it denotes). A *mayUse* operation m by any thread t, after a history h, will be blocked if there exists a thread $t' \neq t$ such that some operation in $may_{\mathcal{E}}(m)$ does not statically commute with some operation in $I_{\mathcal{E}}(h, t')$.

DEFINITION 8 (Commutative-Blocking Library). *We say that library $\mathcal{E}$ is commutative-blocking, if for every $h \in H_{\mathcal{E}}$ that follows the client protocol:*
if $t \neq t'$, $m \in I_{\mathcal{E}}(h, t)$ and $m' \in I_{\mathcal{E}}(h, t')$, then m and m' are statically-commutative.

Note that, for the examples shown in Section 2 such library will not allow the threads to run concurrently. This is because the operations `Inc` and `Dec` are not statically-commutative.

5. Automatic Foresight for Clients

In this section, we present our static analysis to infer calls (in the client code) to the API used to pass the foresight information. The static analysis works for the general case covered by our formalism, and does not depend on the specific implementation of the extended library.

We assume that we are given the interface of a library $\mathcal{E}$ that extends a base library $\mathcal{B}$, along with a specification of the semantic function $may_{\mathcal{E}}$ using a simple annotation language. We use a static algorithm for analyzing a client $\mathcal{C}$ of $\mathcal{B}$ and instrumenting it by inserting calls to *mayUse* operations that guarantee that (all sequential executions of) $\mathcal{C}$ correctly follows the client protocol.

Example Library. In this section, we use a library of Maps as an example. The base procedures of the library are shown in Fig. 5 (their semantics will be described later). The *mayUse* procedures are shown in Fig. 6 — their semantic function is specified using the annotations that are shown in this figure.

```
void mayUseAll();@{(createNewMap),(put,*,*,*),(get,*,*),
(remove,*,*),(isEmpty,*),(size,*)}
void mayUseMap(int m);@{(put,m,*,*),(get,m,*),(remove,m,*),
(isEmpty,m),(size,m)}
void mayUseKey(int m,int k);@{(put,m,k,*),(get,m,k),
(remove,m,k)}
void mayUseNone();@{}
```

Figure 6. Annotated *mayUse* procedures of the example library.

```
mayUseMap(m);
if (get(m,x) == get(m,y)) {
  mayUseKey(m,x); remove(m,x); mayUseNone();
} else {
  remove(m,x); mayUseKey(m,y); remove(m,y); mayUseNone(); }
```

Figure 7. Code section with inferred calls to *mayUse* procedures.

Fig. 7 shows an example of a code section with calls to the base library procedures. The calls to *mayUse* procedures shown in bold are inferred by our algorithm.

5.1 Annotation Language

The semantic function is specified using annotations. These annotations are described by *symbolic operations* and *symbolic sets*.

Let *PVar* be a set of variables, and $*$ be a symbol such that $* \notin$ *PVar*. A *symbolic operation* (over *PVar*) is a tuple of the form $(p, a_1, \cdots, a_n)$, where p is a base library procedure name, and each $a_i \in$ *PVar* $\cup \{*\}$. A *symbolic set* is a set of symbolic operations.

EXAMPLE 5.1. *Here are four symbolic sets for the example library (we assume that $m, k \in$ PVar):*

$SY_1 = \{(createNewMap), (put, *, *, *), (get, *, *), (remove, *, *), (isEmpty, *), (size, *)\}$
$SY_2 = \{(put, m, *, *), (get, m, *), (remove, m, *), (isEmpty, m), (size, m)\}$
$SY_3 = \{(put, m, k, *), (get, m, k), (remove, m, k)\}.$
$SY_4 = \{\}$

Let *Value* be the set of possible values (of parameters of base library procedures). Given a function *asn* : $PVar \mapsto$ *Value* and a symbolic set SY, we define the set of operations SY(*asn*) to be

$$\bigcup_{(p,a_1,\ldots,a_n)\in \mathrm{SY}} \{(p, v_1, \ldots, v_n) \mid \forall i.(a_i \neq *) \Rightarrow (v_i = asn(a_i))\}.$$

EXAMPLE 5.2. *Consider the symbolic sets from Example 5.1. The set $SY_3(asn)$ contains all operations with the procedures* `put`, `get`, *and* `remove` *in which the first parameter is equal to $asn(m)$ and the second parameter is equal to $asn(k)$. The sets $SY_1(asn)$ and $SY_4(asn)$ are not dependent on asn. The set $SY_1(asn)$ contains all operations with the procedures* `createNewMap`, `put`, `get`, `remove`, `isEmpty` *and* `size`. *The set $SY_4(asn)$ is empty.*

The Annotations Every *mayUse* procedure p is annotated with a symbolic set over the the set of formal parameters of p.

In Fig. 6, the procedure *mayUseAll* is annotated with SY_1, *mayUseMap* is annotated with SY_2, *mayUseKey* is annotated with SY_3, and *mayUseNone* is annotated with SY_4.

Let p be a *mayUse* procedure with parameters $x_1, \ldots, x_n$ which is annotated with SY_p. An invocation of p with the values $v_1, \ldots, v_n$ is a *mayUse* operation that refers to the set defined by SY_p and a function that maps x_i to v_i (for every $1 \leq i \leq n$).

EXAMPLE 5.3. *In Fig. 6, the procedure* `mayUseAll()` *is annotated with SY_1, hence its invocation is a* mayUse *operation that refers to all the base library operations . The procedure* `mayUseKey(int m, int k)` *is annotated with SY_3, hence* `mayUseKey(0,7)` *refers to all operations with the procedures* `put`, `get`, *and* `remove` *in which the first parameter is 0 and the the second parameter is 7.*

5.2 Inferring Calls to *mayUse* Procedures

We use a simple abstract interpretation algorithm to infer calls to *mayUse* procedures. Given a client $\mathcal{C}$ of $\mathcal{B}$ and annotated *mayUse* procedures, our algorithm conservatively infers calls to the *mayUse* procedures such that the client protocol is satisfied in all sequential executions of $\mathcal{C}$. We have implemented the algorithm for Java programs in which the relevant code sections are annotated as *atomic*. More details are described in [14].

Assumptions. The algorithm assumes that there exists a *mayUse* procedure (with no parameters) that refers to the set of all base library operations (the client protocol can always be enforced by adding a call to this procedure at the beginning of each code section). It also assumes that there exists a *mayUse* procedure (with no parameters) that refers to an empty set, the algorithm adds a call to this procedure at the end of each code section.

Limitations. Our implementation may fail to enforce atomicity of the code sections because: (i) Java code can access shared-memory which is not part of the extended library (e.g., by accessing a global variable); (ii) our simple implementation does not analyze the procedures which are invoked by the annotated code sections. The implementation reports (warnings) about suspected accesses (to shared-memory) and about invocations of procedures that do not belong to the extended library. These reports should be handled by a programmer or by a static algorithm (e.g., purity analysis [28]) that verifies that they will not be used for inter-thread communication (in our formal model, they can be seen as thread-local operations).

6. Implementing Libraries with Foresight

In this section we present a generic technique for realizing an eager-ordering safe extension (see Definition 7) of a given base library $\mathcal{B}$. Our approach exploits a variant of the tree locking protocol over a tree that is designed according to semantic properties of the library's operations.

In Section 6.1 we describe the basic idea of our technique which is based on locking and static commutativity. In Section 6.2 we show how to utilize dynamic properties like dynamic right-movers. In Section 6.3 we show an optimistic synchronization that enables mitigating *lock contention* that may be caused by the locking in our approach. Further extensions of our technique are described in [14].

6.1 The Basic Approach

Example Library. Here, we use the example from Section 5. The procedures of the base library are shown in Fig. 5. The procedure `createNewMap` creates a new Map and returns a unique identifier corresponding to this Map. The other procedures have the standard meaning (e.g., as in *java.util.Map*), and identify the Map to be operated on using the unique `mapId` identifier. In all procedures, `k` is a key, `v` is a value.

We now describe the *mayUse* procedures we use to extend the library interface (formally defined in Fig. 6): (1) `mayUseAll()`: indicates that the transaction may invoke any library operations. (2) `mayUseMap(int mapId)`: indicates that the transaction will invoke operations only on Map `mapId`; (3) `mayUseKey(int mapId, int k)` : indicates that the transaction will invoke operations only on Map `mapId` and key `k`; (4) `mayUseNone()`: indicates the end of transaction (it will invoke no more operations).

In the following, we write $may_{\mathcal{E}}(m)$ to denote the the set of operations associated with the *mayUse* operation m.

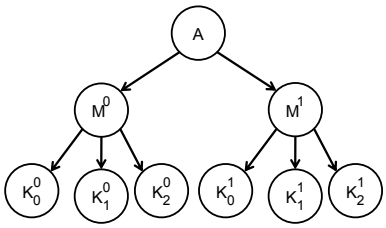

Figure 8. A *locking-tree* used in the example.

Implementation Parameters Our technique is parameterized and permits the creation of different instantiations offering tradeoffs between concurrency granularity and overheads. The parameters to our extension are a *locking-tree* and a *lock-mapping*.

A *locking-tree* is a directed static tree where each node n represents a (potentially unbounded) set of library operations O_n, and satisfies the following requirements: (i) the root of the locking-tree represents all operations of the base library; (ii) if n' is a child of n, then $O_{n'} \subseteq O_n$; (iii) if n and n' are roots of disjoint sub-trees, then every $m \in O_n$ and $m' \in O_{n'}$ are statically-commutative.

EXAMPLE 6.1. *Fig. 8 shows a possible locking-tree for the Map library. The root A represents all (library) operations. Each M^i ($i = 0, 1$) represents all operations with argument* `mapId` *that satisfies*[2] *: $i =$ mapId $\% 2$. Each K^i_j ($i = 0, 1$ and $j = 0, 1, 2$) represents all operations with arguments* `mapId` *and k that satisfy: $i =$ mapId $\% 2 \wedge j = k \% 3$.*

The *lock-mapping* is a function P from *mayUse* operations to tree nodes and a special value $\perp$. For a *mayUse* operation m, $P(m)$ is the node which is associated with m. For each *mayUse* operation m, P should satisfy: if $may_{\mathcal{E}}(m) \neq \emptyset$, then $may_{\mathcal{E}}(m) \subseteq O_{P(m)}$, otherwise $P(m) = \perp$.

EXAMPLE 6.2. *Here is a possible lock-mapping for our example.* `mayUseAll()` *is mapped to the root A.* `mayUseMap(mapId)` *is mapped to M^i where $i =$ mapId $\% 2$.* `mayUseKey(mapId,k)` *is mapped to K^i_j where $i =$ mapId $\% 2 \wedge j = k \% 3$.* `mayUseNone()` *is mapped to $\perp$.*

Implementation We associate a lock with each node of the locking-tree. The *mayUse* operations are implemented as follows:

- The first invocation of a *mayUse* operation m by a thread or transaction (that has not previously invoked any *mayUse* operation) acquires the lock on $P(m)$ as follows. The thread follows the path in the tree from the root to $P(m)$, locking each node n in the path before accessing n's child. Once $P(m)$ has been locked, the locks on all nodes except $P(m)$ are released.[3]
- An invocation of a *mayUse* operation m' by a thread that holds the lock on $P(m)$, locks all nodes in the path from $P(m)$ to $P(m')$ (in the same tree order), and then releases all locks except $P(m')$. If $P(m') = P(m)$ or $P(m')$ is not reachable from $P(m)$,[4] then the execution of m' has no impact.
- If a *mayUse* operation m is invoked by t and $P(m) = \perp$, then t releases all its owned locks .

Furthermore, our extension adds a wrapper around every base library procedure, which works as follows. When a *non-mayUse* operation m is invoked, the current thread t must hold a lock on some node n (otherwise, the client protocol is violated). Conceptually, this operation performs the following steps: (1) wait until all the nodes reachable from n are unlocked; (2) invoke m of the base library and return its return value. Here is a possible pseudo-code for `isEmpty`:

```
bool isEmpty(int mapId) {
  n := the node locked by the current thread
  if(n is not defined) error // optional
  wait until all nodes reachable from n are unlocked
  return baseLibrary.isEmpty(mapId);
}
```

Correctness The implementation satisfies the **progress condition** because: if there exist threads that hold locks, then at least one of them will never wait for other threads (because of the tree structure, and because the base library is total).

We say that t is smaller than t', if the lock held by t is reachable from the lock held by t'. The following property is guaranteed: if $t \leq_h t'$ (see Definition 6) then either t is smaller than t' or all operations allowed for t and t' are statically-commutative. In an implementation, a *non-mayUse* operation waits until all smaller threads are completed, hence the extended library is a **safe extension**.

Further Extensions In [14] we show extensions of the basic approach. We show how to associate several nodes with the same *mayUse* operation — this enables, for example, *mayUse* operation that is associated with operations on several different keys. We also show how to utilize read-write locks — this enables situations in which several threads hold the same lock (node).

6.2 Using Dynamic Information

The dynamic information utilized by the basic approach is limited. In this section we show two ways that enable (in some cases) to avoid blocking by utilizing dynamic information.

6.2.1 Utilizing the State of the Locks

In the basic approach, a *non-mayUse* operation, invoked by thread t, waits until all the reachable nodes (i.e., reachable from the node which is locked by t) are unlocked — this ensures that the operation is a right-mover with respect to the preceding threads. In some cases this is too conservative; for example:

EXAMPLE 6.3. *Consider the example from Section 6.1, and a case in which thread t holds a lock on M^0 (assume t is allowed to use all operations of a single Map). If t invokes* `remove(0,6)` *it will wait until K^0_0, K^0_1 and K^0_2 are unlocked. But, waiting for K^0_1 and K^0_2 is not needed, because threads that hold locks on these nodes are only allowed to invoke operations that are commutative with* `remove(0,6)`*. In this case it is sufficient to wait until K^0_0 is unlocked.*

So, if a *non-mayUse* operation m is invoked, then it is sufficient to wait until all reachable nodes in the following set are unlocked:

$$\{n \mid \exists m' \in O_n : m \text{ is not static-right-mover with } m'\}$$

6.2.2 Utilizing the State of the Base Library

In some cases, the state of the base library can be used to avoid waiting. For example:

EXAMPLE 6.4. *Consider the example from Section 6.1, and a case in which thread t holds a lock on M^0 (assume t is allowed to use all operations of a single Map), and other threads hold locks on K^0_0, K^0_1 and K^0_2. If t invokes* `isEmpty`*, it will have to wait until all the other threads unlock K^0_0, K^0_1 and K^0_2. This is not always needed, for example, if the Map manipulated by t has 4 elements, then the other threads will never be able to make the Map empty (because,*

[2] We write $\%$ to denote the *modulus* operator. Note that we can use a hash function (before applying the modulus operator).

[3] This is simplified version. Other variants, such as hand-over-hand locking, will work as well.

[4] This may happen, for example, when $O_{P(m')} \supset O_{P(m)}$.

according to the Map semantics, they can only affect 3 keys, so they cannot remove more than 3 elements). Hence, the execution of `isEmpty` *by t is a dynamic-right-mover.*

A library designer can add code that observes the library's state and checks that the operation is a dynamic-right-mover; in such a case, it executes the operation of the base library (without waiting). For example, the following code lines can be added to the beginning of `isEmpty(int mapId)`:

```
bool c1 = M^0 or M^1 are held by the current thread ;
bool c2 = baseLibrary.size(mapId) > 3 ;
if(c1 and c2) return baseLibrary.isEmpty(mapId);
... // the remaining code of isEmpty
```

This code verifies that the actual Map cannot become empty by the preceding threads; in such a case we know that the operation is a dynamic-right-mover. Note that writing code that dynamically verifies right-moverness may be challenging, because it may observe inconsistent state of the library (i.e., the library may be concurrently changed by the other threads).

6.3 Optimistic Locking

In the approach, the threads are required to lock the root of the locking-tree. This may create *contention* (because of several threads trying to lock the root at the same time) and potentially degrade performance [18].

To avoid contention, we use the following technique. For each lock we add a counter — the counter is incremented whenever the lock is acquired. When a *mayUse* operation m is invoked (by a thread that has not invoked a *mayUse* operation) it performs the following steps: **(1)** go over all nodes from the root to $P(m)$ and read the counter values; **(2)** lock $P(m)$; **(3)** go over all nodes from the root to $P(m)$ (again), if one node is locked or its counter has been modified then unlock $P(m)$ and restart (i.e., go to step 1).

The idea is to simulate hand-over-hand locking by avoid writing to shared memory. This is done by only locking the node $P(m)$ (and only read the state of the other nodes). When we do not restart in step **3**, we know that the execution is equivalent to one in which the thread performs hand-over-hand locking from the root to $P(m)$.

7. Experimental Evaluation

In this section we present an experimental evaluation of our approach. The goals of the evaluation are: (i) to measure the precision and applicability of our simple static analysis algorithm (ii) to compare the performance of our approach to a synchronization implemented by experts, and (iii) to determine if our approach can be used to perform synchronization in realistic software with reasonable performance.

Towards these goals, we implemented a general purpose Java library for Map data structures using the technique presented in Section 6 (we used the extensions from Sections 6.2 and 6.3). Implementation details can be found in [14]. In all cases, in which our library is used, the calls to the *mayUse* operations have been automatically inferred by our static algorithm.

Evaluation Methodology For all performance benchmarks (except the *GossipRouter*), we followed the evaluation methodology of [17]. We used a Sun SPARC enterprise T5140 server machine running Solaris 10 — this is a 2-chip Niagara system in which each chip has 8 cores (the machine's hyperthreading was disabled). More details about the evaluation methodology can be found in [14].

7.1 Applicability and Precision Of The Static Analysis

We applied our static analysis to 58 Java code sections (composite operations) from [27] that manipulate Maps (taken from open-source projects). We have found that for all composite operations, our simple algorithm infers calls to *mayUse* operations which cannot be further improved manually. For 18 composite operations, our implementation reported (warnings) about procedure invocations that did not belong to the library — we manually verified that these invocations are *pure*[5] (so they can be seen as thread-local operations). These results are summarized in [14].

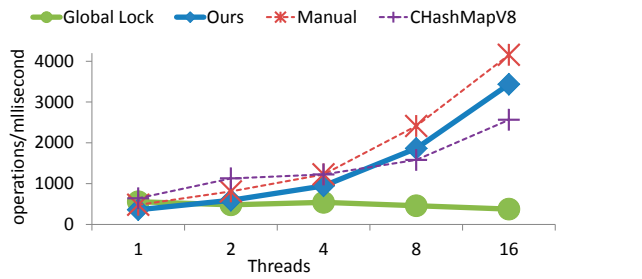

Figure 9. Throughput of *ComputeIfAbsent*.

7.2 Comparison To Hand-Crafted Implementations

We selected several composite operations over a single Map: the *computeIfAbsent* pattern [1], and a few other common composite Map operations (that are supported by [2]). For these composite operations, we compare the performance of our approach to a synchronization implemented by experts in the field. The results of the *computeIfAbsent* pattern are reported here, and the rest of the results are reported in [14].

ComputeIfAbsent The *ComputeIfAbsent* pattern appears in many Java applications. Many bugs in Java programs are caused by non-atomic realizations of this simple pattern (see [27]). It can be described with the following pseudo-code:

```
if(!map.containsKey(key)) {
  value = ... // pure computation
  map.put(key, value);
}
```

The idea is to compute a value and store it in a Map, if and only if, the given key is not already present in the Map. We chose this benchmark because there exists a new version of Java Map, called *ConcurrentHashMapV8*, with a procedure that gets the computation as a parameter (i.e., a function is passed as a parameter), and atomically executes the pattern's code [1].

We compare four implementations of this pattern: (i) an implementation which is based on a global lock; (ii) an implementation which is based on our approach; (iii) an implementation which is based on *ConcurrentHashMapV8*; (iv) an implementation which is based on hand-crafted fine-grained locking (we used *lock stripping*, similar to [16], with 32 locks; this is an attempt to estimate the benefits of manual hand-crafted synchronization w/o changing the underlying library). The computation was emulated by allocating a relatively-large Java object ($\sim$ 128 bytes).

The results are shown in Fig. 9. We are encouraged by the fact that our approach provides better performance than *ConcurrentHashMapV8* for at least 8 threads. Also, it is (at most) 25% slower than the hand-crafted fine-grained locking.

7.3 Evaluating The Approach On Realistic Software

We applied our approach to three benchmarks with multiple Maps — in these benchmarks, several Maps are simultaneously manipulated by the composite operations. We used the *Graph* benchmark [16], Tomcat's *Cache* [3], and a multi-threaded application

[5] We have found that the purity of the invoked procedures is obvious, and can be verified by existing static algorithms such as [28].

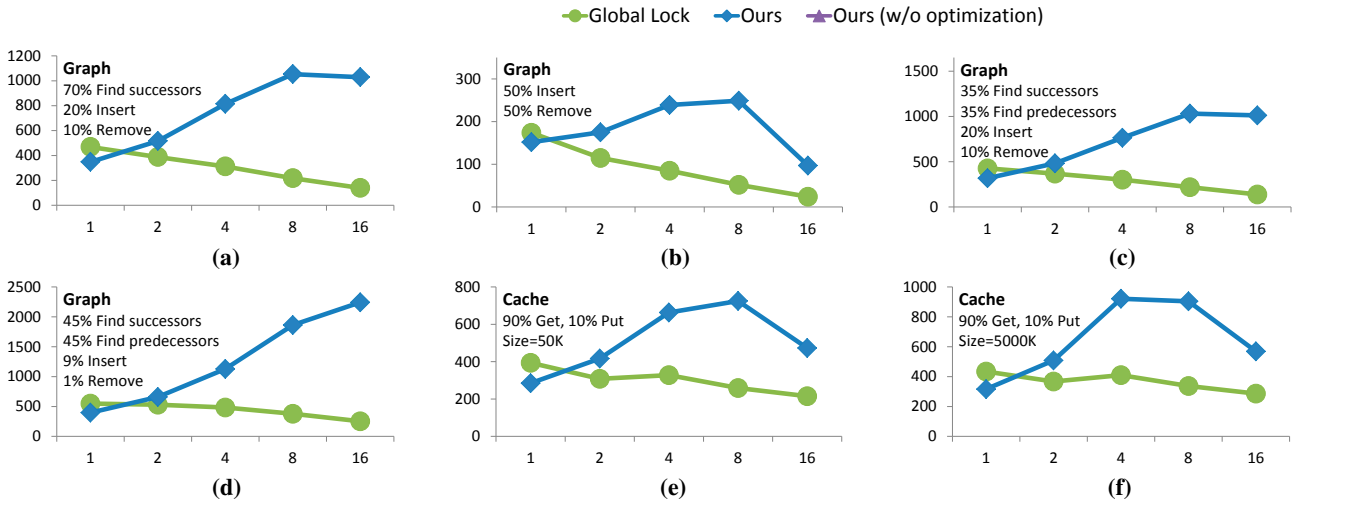

Figure 10. Throughput (operations/millisecond) as a function of the number of threads (1-16).

GossipRouter [6]. In these benchmarks, we compare the performance to coarse grained locking.

Graph This benchmark is based on a Java implementation of the Graph that has been used for the evaluation of [16]. The Graph consists of four composite operations: *find successors*, *find predecessors*, *insert edge*, and *remove edge*. Its implementation uses two Map data structures in which several different values can be associated with the same key (such type of Maps is supported by our library; also [5] contains an example for such type of Maps).

We compare a synchronization which is based on a global lock, and a synchronization which is based on our approach. We use the workloads from [16]. The results are shown in Fig. 10(a)–(d).

For some of the workloads, we see that there is a drop of performance between 8 and 16 threads. This can be explained by the fact that each chip of the machine has 8 cores, so using 16 threads requires using both chips (this creates more overhead).

Tomcat's Cache This benchmark is based on a Java implementation of Tomcat's cache [3]. This cache uses two types of Maps which are supported by our library: a standard Map, and a *weak Map* (see [4]). The cache consists of two composite operations `Put` and `Get` which manipulate the internal Maps. In this cache, the `Get` is not a read-only operation (in some cases, it copies an element from one Map to another). The cache gets a parameter (*size*) which is used by its algorithm. Fig. 10(e) and Fig. 10(f) show results for two workloads.

GossipRouter The *GossipRouter* is a Java multi-threaded routing service from [6]. Its main state is a *routing table* which consists of several Map data structures. (The exact number of Maps is dynamically determined).

We use a version of the router (*"3.1.0.Alpha3"*) with several bugs that are caused by an inadequate synchronization in the code that access the *routing table*. We have manually identified all code sections that access the *routing table* as atomic sections; and verified (manually) that: whenever these code sections are executed atomically, the known bugs are not occurred.

We compare two ways to enforce atomicity of the code sections: a synchronization which is based on a global lock, and a synchronization which is based on our approach. We used a performance tester from [6] (called *MPerf*) to simulate 16 clients where each client sends 5000 messages. In this experiment the number of threads cannot be controlled from the outside (because the threads are autonomously managed by the router). Therefore, instead of changing the number of threads, we changed the number of active cores. The results are shown in Fig. 11.

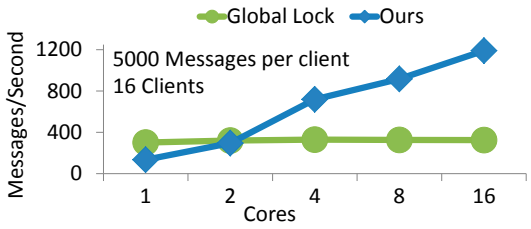

Figure 11. Throughput of *GossipRouter*.

For the router's code, our static analysis has reported (warnings) about invocations of procedures that do not belong to the Maps library. Interestingly, these procedures perform I/O and do not violate atomicity of the code sections. Specifically, they perform two types of I/O operations: logging operation (used to print debug messages), and operations that send network messages (to the router's clients). They do not violate the atomicity of the atomic sections, because they are not used to communicate between the threads (from the perspective of our formal model, they can be seen as thread-local operations).

8. Related Work

8.1 Concurrent Data Structures

Many sophisticated concurrent data structures (e.g., [2, 18, 24]) were developed and integrated into modern software libraries. These data structures ensure atomicity of their basic operations, while hiding the complexity of synchronization inside their libraries. Unfortunately as shown in [27] employing concurrent data structures in client code is error prone. The problem stems from the inability of concurrent data structures to ensure atomicity of client operations composed from several data structure operations.

In this paper we focus on enabling efficient atomicity of client operations composed from several data structure operations. The foresight information enables the library to provide concurrency without violating atomicity of composite client operations. This prevents the errors reported in [27] without the need for the library to directly support composite operations as suggested in [1].

8.2 Synchronization by Utilizing Semantics Properties

Many synchronization approaches aim to utilize semantics properties of concurrent operations for the sake of concurrency and efficiency. (e.g., by utilizing commutativity of read-only operations.) These approaches are developed for databases (e.g.,[26, 29]) and for general programming models (e.g.,[21, 22]).

Rollbacks A notable property of most approaches (e.g., all the approaches discussed in [21, 22, 26, 29]) is that they require a rollback mechanism (otherwise atomicity and deadlock-freedom are not guaranteed). In contrast, in this paper we show an approach that ensures atomicity and deadlock-freedom without using a rollback mechanism. This may be an advantage in some cases (e.g., when a rollback mechanism has a high runtime and memory overhead [10], or when I/O operations are involved).

Using foresight may also be beneficial for approaches with a rollback mechanism. Though, in this work we focus on synchronization that does not require rollback mechanisms.

Commutativity and Movers We base our approach on right-movers whereas most approaches are based on commutativity. Indeed, [21] shows that many synchronization schemes can be based on either right-movers or left-movers. In [21], they use a variant of a "static" right-mover which is a special case of our definition for dynamic-right-mover.

Locking Mechanisms Locking mechanisms are widely used for synchronization, some of them utilize semantics properties of shared operations (e.g., [12, 20]). Usually these mechanisms do not allow several threads to hold locks which correspond to non-commutative operations. An interesting locking mechanism is shared-ordered locking [7] which allow several threads to hold locks which correspond to non-commutative operations. Such locking mechanisms can be seen as special cases of libraries with foresight-based synchronization.

8.3 Synchronization via Locking

Locking is a widely used approach for software synchronization. Writing software with locking that permits concurrency, such as fine-grain locking, is considered hard and error prone.

In order to mitigate this problem several algorithms for automatically inferring locks using static analysis were recently suggested (e.g. [11, 13]). Our algorithm for inferring *mayUse* operations is similar to these algorithms; still with the following differences: (i) we deal *mayUse* operations which can be seen as generalizations of *lock-acquire* and *lock-release* operations — this enables our approach to utilize non-trivial semantic properties of shared operations; (ii) lock inference algorithms usually need to consider the structure of a dynamically manipulated state, we avoid this by considering a single shared library that can be statistically identified.

Acknowledgments

We thank Yehuda Afek, Adam Morrison, Noam Rinetzky, Omer Tripp and the anonymous referees for their insightful feedbacks. This work was partially supported by Microsoft Research through its PhD Scholarship Programme, and by The Israeli Science Foundation (grant no. 965/10).

References

[1] gee.cs.oswego.edu/dl/jsr166/dist/jsr166edocs/jsr166e/ConcurrentHashMapV8.html.

[2] docs.oracle.com/javase/6/docs/api/java/util/concurrent/ConcurrentHashMap.html.

[3] www.devdaily.com/java/jwarehouse/apache-tomcat-6.0.16/java/org/apache/el/util/ConcurrentCache.java.shtml.

[4] docs.oracle.com/javase/6/docs/api/java/util/WeakHashMap.html.

[5] guava-libraries. code.google.com/p/guava-libraries/.

[6] Jgroups toolkit. www.jgroups.org/index.html.

[7] AGRAWAL, D., AND EL ABBADI, A. Constrained shared locks for increasing concurrency in databases. In *Selected papers of the ACM SIGMOD symposium on Principles of database systems* (1995).

[8] BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[9] BRONSON, N. *Composable Operations on High-Performance Concurrent Collections*. PhD thesis, Stanford University, Dec. 2011.

[10] CASCAVAL, C., BLUNDELL, C., MICHAEL, M., CAIN, H. W., WU, P., CHIRAS, S., AND CHATTERJEE, S. Software transactional memory: Why is it only a research toy? *Queue 6*, 5 (Sept. 2008), 46–58.

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

[12] COURTOIS, P. J., HEYMANS, F., AND PARNAS, D. L. Concurrent control with *readers* and *writers*. *Commun. ACM 14*, 10 (Oct. 1971).

[13] GOLAN-GUETA, G., BRONSON, N., AIKEN, A., RAMALINGAM, G., SAGIV, M., AND YAHAV, E. Automatic fine-grain locking using shape properties. In *OOPSLA* (2011).

[14] GOLAN-GUETA, G., RAMALINGAM, G., SAGIV, M., AND YAHAV, E. Concurrent libraries with foresight. Tech. Rep. TR-2012-89, Tel Aviv University, 2012.

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

[16] HAWKINS, P., AIKEN, A., FISHER, K., RINARD, M., AND SAGIV, M. Concurrent data representation synthesis. In *PLDI* (2012).

[17] HERLIHY, M., LEV, Y., LUCHANGCO, V., AND SHAVIT, N. A provably correct scalable concurrent skip list. In *OPODIS* (2006).

[18] HERLIHY, M., AND SHAVIT, N. *The Art of Multiprocessor Programming*. Morgan Kauffman, Feb. 2008.

[19] HERLIHY, M. P., AND WING, J. M. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12* (July 1990).

[20] KORTH, H. F. Locking primitives in a database system. *J. ACM 30* (January 1983), 55–79.

[21] KOSKINEN, E., PARKINSON, M., AND HERLIHY, M. Coarse-grained transactions. In *POPL* (2010), pp. 19–30.

[22] KULKARNI, M., PINGALI, K., WALTER, B., RAMANARAYANAN, G., BALA, K., AND CHEW, L. P. Optimistic parallelism requires abstractions. In *PLDI* (2007).

[23] LIPTON, R. J. Reduction: a method of proving properties of parallel programs. *Commun. ACM 18*, 12 (Dec. 1975), 717–721.

[24] MICHAEL, M. M., AND SCOTT, M. L. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC* (1996), pp. 267–275.

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

[26] SCHWARZ, P. M., AND SPECTOR, A. Z. Synchronizing shared abstract types. *ACM Trans. Comput. Syst. 2*, 3 (Aug. 1984), 223–250.

[27] SHACHAM, O., BRONSON, N., AIKEN, A., SAGIV, M., VECHEV, M., AND YAHAV, E. Testing atomicity of composed concurrent operations. In *OOPSLA* (2011).

[28] SĂLCIANU, A., AND RINARD, M. Purity and side effect analysis for Java programs. In *VMCAI* (2005), pp. 199–215.

[29] WEIHL, W. E. Commutativity-based concurrency control for abstract data types. *IEEE Trans. Comput. 37* (December 1988), 1488–1505.