Concurrent Data Representation Synthesis

Peter Hawkins
Computer Science Department, Stanford University
hawkinsp@cs.stanford.edu

Alex Aiken *
Computer Science Department, Stanford University
aiken@cs.stanford.edu

Kathleen Fisher †
Computer Science Department, Tufts University
kfisher@eecs.tufts.edu

Martin Rinard
MIT Computer Science and Artificial Intelligence Laboratory
rinard@csail.mit.edu

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

Abstract

We describe an approach for synthesizing data representations for concurrent programs. Our compiler takes as input a program written using *concurrent relations* and synthesizes a representation of the relations as sets of cooperating data structures as well as the placement and acquisition of locks to synchronize concurrent access to those data structures. The resulting code is correct by construction: individual relational operations are implemented correctly and the aggregate set of operations is serializable and deadlock free. The relational specification also permits a high-level optimizer to choose the best performing of many possible legal data representations and locking strategies, which we demonstrate with an experiment autotuning a graph benchmark.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Abstract data types, Concurrent programming structures, Data types and structures; E.2 [*Data Storage Representations*]

Keywords Synthesis, Lock Placement

1. Introduction

Consider the problem of implementing concurrent operations on a directed graph. We must decide how to represent the graph as a collection of data structures, perhaps using a lookup table mapping each node to the set of its adjacent nodes. We will need to pick concrete representations for both the lookup table (e.g., a concurrent hashmap) and the adjacency sets (e.g., linked lists). We must also decide how concurrency will be realized. We could add our own synchronization using locks and/or we could use a concurrent container data structure to implement the lookup table, the sets of adjacent nodes, or both.

Assume for the moment that we decide both containers will be concurrent. We must of course ensure there is enough synchronization to avoid harmful races, but not so much that we either limit scalability or introduce deadlocks. Using off-the-shelf concurrent containers can simplify this task, but even using concurrent containers for both data structures does not automatically imply that high-level graph operations that touch both structures (such as inserting or removing an edge from the graph) are correct. In fact, recent work in bug detection for concurrent programs has shown that programmers frequently fail to use standard concurrent containers correctly, especially when they must compose multiple concurrent operations [20].

On the other hand, it may be more efficient to have only the top-level lookup table be concurrent and use non-concurrent data structures for the sets of adjacent nodes—if it is very infrequent that threads try to access the same node simultaneously the extra overhead of a concurrent data structure for the adjacency sets won't be worthwhile. This design has different correctness requirements and would likely result in a different choice of where to place any needed synchronization to guarantee correctness. The right answer to the decision of whether to use a concurrent or non-concurrent data structure for the adjacency sets likely depends on the typical workload and it will be difficult to modify the interlinked synchronization and data structures if we decide later that the graph should be implemented differently.

In this paper we present an approach to synthesizing concurrent data representations, meaning that from a high-level specification of data we produce both the concrete data structures and the corresponding synchronization to implement the specification. In our approach, programs are written using *concurrent relations* (Section 2), a generalization of standard concurrent collections to relations with a concurrent interface to perform insertions, deletions, and lookups of tuples. Our compiler automatically synthesizes all aspects of the data representation, including the choice of data structures and how they interact, the number and placement of locks to guard access to those data structures (including, for example, whether locking should be fine-grain or coarse-grain), an order in which locks can be acquired to guarantee deadlock freedom, and all of the code to correctly manage the interplay of the data structures and synchronization.

* This work was supported by NSF Grant CCF-0702681 and Stanford's Army High Performance Research Center.

† The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense or U.S. Government. Distribution Statement A (Approved for Public Release, Distribution Unlimited)

PLDI'12, June 11–16, 2012, Beijing, China.

By specifying a program's access to data using concurrent relations and avoiding a premature commitment to a particular representation, much of the low-level detail of programs is removed, making them easier to read and maintain, while simultaneously making it much easier to change the implementation if desired. Furthermore, concurrent relations give a high-level and pointer-free specification of data, which is good for compilers, because the compiler is now free to choose the concrete representation of the data without the usual requirement that it first perform a complex and usually brittle pointer analysis. Programs written using relational data specifications are simpler, correct by construction, and can be automatically optimized in ways that are out of reach of compilers for languages with traditional data structure definitions.

Beginning with Cohen and Campbell [5] researchers have investigated how to compile programs written using relations as the main (and sometimes only) aggregate form of data into low-level data representations. Our method builds on two recent results (Section 4): we use the *decompositions* of Hawkins et al. [12] to describe how relations can be decomposed into a set of cooperating data structures, and we use the theory of *lock placements* [13] to describe the space of possible locking strategies. Our specific contributions are:

- We introduce *concurrent relations*, a generalization of standard concurrent container data structures to containers of tuples, with concurrent operations to insert, remove, and query relations (Section 2).
- The selection of data structures is subtler than in the non-concurrent case, because there is the added dimension of using concurrent container structures, which may or may not require additional synchronization depending on the relational specification, and, in addition, different concurrent containers provide varying guarantees about the safety of concurrent access. We give a taxonomy of containers and their properties relevant to concurrent data representation synthesis (Section 3).
- We extend the relational decomposition language [12] to support concurrent relations. Just as the original decomposition language describes how to assemble a representation of a relation from a library of container data structures, concurrent decompositions describe how to compose concurrent and non-concurrent data structures together with locks to implement a concurrent relation primitive (Section 4.1).
- We show how to integrate *lock placements* [13], which describe a space of possible locking strategies on data structures, with the problem of selecting the data structures themselves. The choice of data structures and lock placements is done in such a way that the resulting code is guaranteed to ensure the serializability of relational operations (Section 4.2).
- We adapt and generalize the problem of selecting a good implementation of the relational primitives, called *query planning*, to concurrent relations (Section 5). One of the major issues is ensuring deadlock freedom, which we accomplish by selecting a global lock ordering that all relational operations obey by construction. Deadlock is not addressed in the previous work on lock placements [13].
- The optimal decomposition depends on the usage patterns of the data structure and the target machine. We present results from a full implementation, which includes an autotuner that allows us to discover a good combination of both locks and container data structures automatically for a training workload. We perform an evaluation of a concurrent graph benchmark, showing that the best data representation varies with the workload, and thus it is important to have the flexibility to easily alter the representation of concurrent data (Section 6).

2. Concurrent Relations

We advocate a view in which programmers write programs that operate on relations and a compiler selects the concrete representation of the relations. A *relational specification* is a set of column names C together with a set of *functional dependencies* Δ. Functional dependencies (FDs) specify which columns are uniquely determined by other columns. For example, we can represent the edges of a directed graph as a relation with three columns: src (source node), dst (destination node), and weight, together with a functional dependency $\mathrm{src}, \mathrm{dst} \to \mathrm{weight}$, which specifies that every edge of the graph has a unique weight. The relational specification is a contract between the client of our compiler and the generated code: If the client obeys the functional dependencies, then the compiler guarantees that the generated code preserves the semantics of the relational operations.

Values, Tuples, Relations We assume a set of untyped values v drawn from a universe $\mathbb{V}$ that includes the integers ($\mathbb{Z} \subseteq \mathbb{V}$). A *tuple* $t = \langle c_1 \colon v_1, c_2 \colon v_2, \dots \rangle$ maps a set of *columns* $\{c_1, c_2, \dots\}$ to values drawn from $\mathbb{V}$. We write $\operatorname{dom} t$ for the columns of t. A tuple t is a *valuation* for a set of columns C if $\operatorname{dom} t = C$. A *relation* r is a set of tuples $\{t_1, t_2, \dots\}$ over identical columns C. We write $t(c)$ for the value of column c in tuple t. We write $t \supseteq s$ if the tuple t *extends* tuple s, that is $t(c) = s(c)$ for all c in $\operatorname{dom} s$. We say tuple t *matches* tuple s, written $t \sim s$, if the tuples are equal on all common columns. Tuple t matches a relation r, written $t \sim r$, if t matches every tuple in r. A tuple t is a *key* for a relation r if the columns $\operatorname{dom} t$ functionally determine all columns of r.

We use the standard notation of relational algebra. Union ($\cup$), intersection ($\cap$), set difference ($\setminus$) have their usual meanings. The operator $\pi_C\, r$ projects relation r onto a set of columns C. A relation r has a *functional dependency* (FD) $C_1 \to C_2$ if any pair of tuples in r that are equal on columns C_1 are also equal on columns C_2.

Relational Operations We provide four atomic operations for creating and manipulating *concurrent relations*. In the following specification we represent relations as ML-style references to a set of tuples: $\mathsf{ref}\ x$ creates a new reference to x, $!x$ retrieves the current value of x, and $x \leftarrow y$ sets the current value of x to y.

$$\begin{aligned}
\mathsf{empty}\ () &= \mathsf{ref}\ \emptyset \\
\mathsf{remove}\ r\ s &= r \leftarrow\ !r \setminus \{t \in\ !r \mid t \supseteq s\} \\
\mathsf{query}\ r\ s\ C &= \pi_C\{t \in\ !r \mid t \supseteq s\} \\
\mathsf{insert}\ r\ s\ t &= \mathsf{if}\ \nexists u.\, u \in\ !r \wedge s \subseteq u\ \mathsf{then}\ r \leftarrow\ !r \cup \{s \cup t\}
\end{aligned}$$

Informally, $\mathsf{empty}\ ()$ creates a new empty relation. Operation $\mathsf{remove}\ r\ s$ removes tuples matching s; in practice, our implementation requires that s is a key for the relation. Operation $\mathsf{query}\ r\ s\ C$ returns columns C of all tuples in r matching tuple s.

The most interesting operation is $\mathsf{insert}\ r\ s\ t$, which inserts a new tuple x, where x is the union of the columns of tuples s and t, into a relation r, provided there is no existing tuple in r matching s. We require that s and t have disjoint domains. Insert generalizes the put-if-absent operation provided by standard concurrent key-value maps: *put-if-absent(k,v)* inserts value v into the map if no other value is already associated with key k, and would be written

$$\mathsf{insert}\ r\ \langle \mathrm{key} \colon k \rangle\ \langle \mathrm{value} \colon v \rangle$$

Insert operations may violate functional dependencies, and it is the client's obligation to ensure functional dependencies are observed. The form of the insert operation allows clients to test whether functional dependencies will be satisfied by a new tuple even in the presence of concurrent updates.

The relational compiler ensures that the implementations of the relational operations are *linearizable* [15] (equivalently *serializable*, since the relational operations are single operation transactions on a single object). Operations on an object are *linearizable* if every

Data Structure	Concurrency-safety			
	L/L L/S S/S	L/W	S/W	W/W
`HashMap`	yes	no	no	no
`TreeMap`	yes	no	no	no
`ConcurrentHashMap`	yes	yes	weak	yes
`ConcurrentSkipListMap`	yes	yes	weak	yes
`CopyOnWriteArrayList`	yes	yes	yes	yes

Figure 1. Concurrency safety properties of selected containers from the JDK. Possible operations are lookup (L), scan (S), or write (W). For an operation pair α/β, concurrently executing operations α and β on a container is either unsafe ("no"), safe but only weakly consistent ("weak"), or both safe and linearizable ("yes").

operation appears to take place atomically at a single point in time in between its invocation and response.

Continuing with our graph example, we create a new, empty graph relation r_0 using the empty () operation. Inserting an edge

$$\mathsf{insert}\ r_0\ \langle \mathrm{src}\colon 1,\ \mathrm{dst}\colon 2\rangle\ \langle \mathrm{weight}\colon 42\rangle$$

results in a new relation $r_1 = \{\langle \mathrm{src}\colon 1,\ \mathrm{dst}\colon 2,\ \mathrm{weight}\colon 42\rangle\}$. A subsequent insertion

$$\mathsf{insert}\ r_1\ \langle \mathrm{src}\colon 1,\ \mathrm{dst}\colon 2\rangle\ \langle \mathrm{weight}\colon 101\rangle$$

leaves the relation unchanged, because relation r_1 already contains an edge with the same src and dst fields.

We can retrieve the dst and weight fields corresponding to the successors of node 1 in a relation r using the operation query r $\langle \mathrm{src}\colon 1\rangle$ $\{\mathrm{dst}, \mathrm{weight}\}$, and finally we can delete edges with a dst of 2 using the operation remove r $\langle \mathrm{dst}\colon 2\rangle$.

3. A Taxonomy of Concurrent Containers

Decompositions describe how to implement concurrent relations as a combination of both concurrent and non-concurrent data structures. Before diving into the details of the decomposition language (Section 4), we first describe the concurrency properties of the container data structures found in the wild, which form the building blocks of concurrent decompositions.

Container Interface A *container* is a data structure that implements an associative key-value map interface consisting of *read operations* lookup(k) and scan(f), and a *write operation* write(k, v).

- The *lookup* operation lookup(k) returns the value associated with a key k, if any.
- The *scan* operation scan(f) iterates over the map, and invokes the function $f(k, v)$ once for each key k and its associated value v in the map. A scan may or may not return the entries of the map in sorted order.
- The *write* operation write(k, v) sets the value associated with a key k to v. Here v is an optional value, in the style of ML. If v is Some w, then the operation updates the value associated with key k to w, whereas if v is None, representing the absence of a value, then any existing value associated with k is removed. The write operation subsumes operations to insert, update, and remove entries from a map.

3.1 Concurrency Safety and Consistency

We next discuss two related properties of containers, *concurrency safety*, which describes whether it is safe for two operations to occur in parallel, and *consistency*, which characterizes what a container guarantees about the possible orders of events in a concurrent execution. Figure 1 lists the concurrency safety and consistency properties of a selection of Java containers from the JDK; as we show, containers differ greatly in their support for concurrency.

Concurrency Safety For a given data structure, we say a pair of operations α/β is *concurrency-safe* if two threads may safely execute operations α and β in parallel with no external synchronization. A container is *concurrency-safe* if all pairs of operations are concurrency-safe. Concurrency safety is strictly a statement about the correct usage of the interface of a data structure; it is irrelevant how the data structure guarantees safety internally, whether by locks, atomic instructions, or by some other means.

Consider the data structures described in Figure 1. Almost all data structures support parallel read operations; for example concurrent threads may safely read or iterate over a Java `HashMap` in parallel without synchronization. Exceptions exist; for example, it would not be safe for threads to perform concurrent reads of a splay tree because splay tree read operations rebalance the tree.

Only a few containers permit write operations in parallel with other operations. For example, it is unsafe to read from or write to a `HashMap` object while another thread is writing to the same `HashMap`. By contrast a `ConcurrentHashMap` or a `CopyOnWriteArrayList` allow concurrent lookup and write operations, or pairs of concurrent write operations. On a concurrency-safe container, such as a `ConcurrentHashMap`, the lookup and write operations are linearizable even in the absence of any external concurrency control. For concurrency-unsafe operations, such as reading a splay tree, linearizability is the responsibility of an external concurrency control primitive, such as a lock.

The scan operation, however, behaves differently. Even many containers that allow iteration in parallel with mutation do not guarantee that iteration is linearizable. We identify two different possibilities. Some containers, such as `ConcurrentHashMap` provide *weakly consistent* concurrent iteration; that is, concurrent iteration is safe, but may or may not reflect inserts and removals that occur in parallel with the iteration. Iteration over a weakly-consistent container may not be linearizable, that is, the result of the iteration may not correspond to the set of entries present in the container at any instant in time. Conversely for containers that provide *snapshot iteration*, such as a `CopyOnWriteArrayList`, iteration behaves as if it operated over a linearizable snapshot of the container.

4. Concurrent Decompositions

The concurrent decomposition language describes how to assemble container data structures into representations of relations that support concurrent serializable transactions that implement the various relational operations. By combining concurrent and non-concurrent data structures with locks we can build a representation of a relation with strong concurrency guarantees, even if the constituent data structures themselves have limited support for concurrency.

We extend the relational decomposition language [12] to support concurrent operations from multiple threads. Two key ideas underlie safe and scalable concurrent decompositions: leveraging existing concurrent containers (Section 4.1) to the full extent possible, and supplementing containers with locks as necessary to ensure the safety and serializability of concurrent transactions over the complete decomposition (Section 4.2).

As a focus of this paper is extending decompositions to support concurrency, we first review the definition of decompositions.

4.1 Decompositions

A *decomposition* describes how to represent a relation as a combination of primitive container data structures. A decomposition is a static description of the heap, similar to a type. We use a graphical

notation for decompositions isomorphic to the let-binding notation in the literature [12]. Figure 2(a) shows a decomposition of a filesystem directory tree relation, based on the directory entry cache in the Linux kernel. The relation has three columns parent, name, and child and obeys a functional dependency $\mathrm{parent}, \mathrm{name} \to \mathrm{child}$. Each 'parent' directory entry has zero or more 'child' directory entries, each with a distinct file 'name'.

Formally, a decomposition $\hat{d}$ is a rooted, directed acyclic graph, consisting of a set of vertices $V = \{u, v, \dots\}$ and a set of edges drawn from $V \times V$. A decomposition has a unique *source* vertex ρ with no incoming edges. All vertices must be reachable from the source vertex.

Each node v of a decomposition has an associated "type" $A \triangleright B$, written $v\colon A \triangleright B$. Intuitively, A is the set of columns whose representation is specified by paths from the root node to v, and B is the residual set of columns represented by the subgraph reachable from v. Each edge uv of the decomposition has an associated set of columns $\mathsf{cols}(uv)$, and a choice of container $\mathsf{ds}(uv)$ the compiler should use to implement the edge.

In Figure 2(a), the edge ρx from the root indicates that the relation is implemented by a `TreeMap` from each parent value to the residual relation of all $(\mathrm{name}, \mathrm{child})$ pairs for that parent. Recursively, this subrelation is implemented by another `TreeMap` from name to the child directory (edge xy). Finally, the functional dependency guarantees that the child directory is a singleton tuple and is implemented by its single value (edge yz). This structure (a map from parents to the set of child directory names) enables efficient iteration over the children of a directory, which is useful when, for example, unmounting a filesystem. To enable efficient directory lookup the decomposition also includes a global hashtable mapping $(\mathrm{parent}, \mathrm{name})$ pairs to child objects (edge ρy).

We assume that decompositions are *adequate* [12], that is decompositions are capable of representing all relations satisfying the relational specification. The adequacy conditions imply that for every edge uv where $u\colon A \triangleright B$ and $v\colon C \triangleright D$ we have $C \supseteq A \cup \mathsf{cols}(uv)$.

Decomposition Instances The run-time (dynamic) counterpart of a decomposition $\hat{d}$ is a *decomposition instance* d, which represents a particular concrete relation. Each node $v\colon A \triangleright B$ in a decomposition $\hat{d}$ has a set of instances $\{v_t\}$ where $\mathrm{dom}\, t = A$, each corresponding to an object in memory. Each edge uv of a decomposition has a set of edge instances $\{uv_t\}$, where if $u\colon A \triangleright B$ then $\mathrm{dom}\, t = A \cup \mathsf{cols}(uv)$. If $u\colon A \triangleright B$ and $\mathrm{dom}\, t \supset A$, we write u_t to denote the node instance $u_{\pi_A\, t}$; similarly if $\mathrm{dom}\, t \supset A \cup \mathsf{cols}(uv)$ we write uv_t to denote $uv_{\pi_{A \cup \mathsf{cols}(uv)}\, t}$. A formal characterization of *well-formed* decomposition instances and an *abstraction function* that maps well-formed decomposition instances back to the relations they represent can be found in the literature [12].

Figure 2(b) depicts an instance of the decomposition of Figure 2(a) representing the relation containing 3 directory entries:

$$\{\, \langle \mathrm{parent}\colon 1,\ \mathrm{name}\colon \text{'a'},\ \mathrm{child}\colon 2\rangle,\\ \langle \mathrm{parent}\colon 2,\ \mathrm{name}\colon \text{'b'},\ \mathrm{child}\colon 3\rangle,\\ \langle \mathrm{parent}\colon 2,\ \mathrm{name}\colon \text{'c'},\ \mathrm{child}\colon 4\rangle \}.$$

4.2 Logical Locks, Transactions, and Serializability

Locks Given a decomposition d, we compile each relational operation into a transaction tailored to d. For safety and consistency transactions must acquire locks that protect the invariants upon which a transaction relies. By "lock" we mean a class of pessimistic synchronization primitives that may be held by a transaction in either of two different *modes*, namely *shared* or *exclusive*. A lock may, but need not, permit multiple transactions to hold shared access simultaneously; however if a transaction holds exclusive access to a lock the lock must not allow any other transaction to hold either shared or exclusive access.

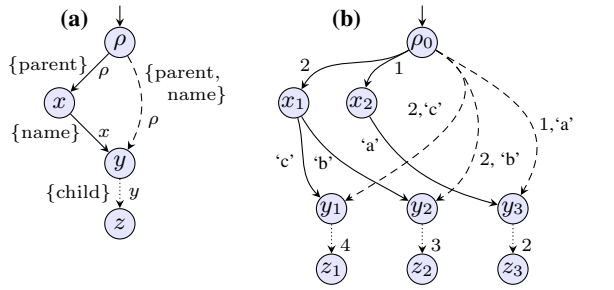

Figure 2. (a): A decomposition representing a directory tree relation with three columns $\{\mathrm{parent}, \mathrm{name}, \mathrm{child}\}$ and **(b)**: an instance of decomposition (a). Each edge of the decomposition is labeled with a set of columns $\{\cdots\}$, together with the label of the node whose lock protects instances of that edge (Section 4.3). Each edge of the instance is labeled with a valuation for the corresponding decomposition edge's columns. Solid edges indicate a `TreeMap`, dashed edges represent a `ConcurrentHashMap`, and dotted edges represent singleton tuples.

Logical Locks To ensure that transactions are serializable, the data in decomposition instances are protected by *logical locks*. We associate a distinct logical lock with every edge uv_t of a decomposition instance. Logical locks protect the state, either presence or absence, of an edge instance. If a transaction observes the presence or absence of an edge it must hold shared access to the corresponding logical lock, and if a transaction adds or removes an edge it must hold exclusive access to the corresponding logical lock. Logical locks are defined for every possible edge instance, irrespective of whether the edge is actually present in a particular decomposition instance or not.

For now, we leave the implementation of each logical lock uv_t abstract. In Section 4.3, we implement logical locks using a smaller set of *physical locks* attached to the nodes of a decomposition instance. By placing restrictions on the possible mappings from logical locks to physical locks we can ensure that containers (concurrent or not) and compositions of containers are used safely.

Two-Phase Locking Protocol Each transaction consists of a sequence of locks, reads, writes, and unlocks of the edges of a decomposition instance. The purpose of introducing logical locks is that by having a distinct lock for every edge (including edges that are absent) we are able to state a very simple and obviously correct protocol for transactions on decomposition instances. To ensure consistency of transactions, we use a standard two-phase locking protocol on logical locks:

- Transaction operations must be *logically well-locked*; that is, if a transaction observes the state (either present or absent) of an edge instance uv_t via a lookup or scan operation then it must hold shared access to the logical lock of uv_t, and if a transaction adds, removes, or updates an instance edge uv_t via a write operation then it must hold exclusive access to the logical lock of uv_t.
- Transactions must be *logically two-phase*, that is, transactions must be divided into a *growing* phase during which logical locks are acquired and a *shrinking* phase during which logical locks are released. All lock acquisitions must precede all lock releases.

It is a classic result that well-locked and two-phase transactions are serializable [10].

4.3 Physical Locks and Lock Placements

By associating a unique logical lock with every edge instance we can use a simple two-phase locking protocol to ensure that transactions are serializable. Such an approach would be impractical, however.

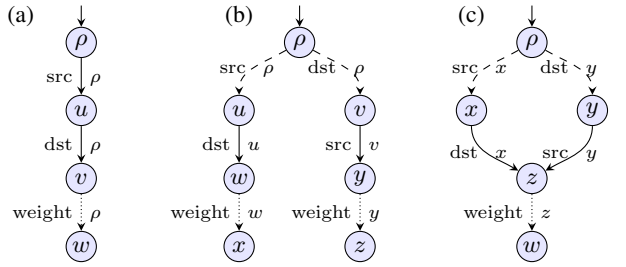

Figure 3. Three concurrent decompositions for a directed graph relation: (a) a "stick", with a single coarse lock around non-concurrent data structures, (b) a "split" decomposition, with locks at different granularities, and (c) a "diamond", with a mixture of speculatively-locked concurrent data structures and non-concurrent data structures. Solid edges indicate `TreeMap` containers, dashed edges represent `ConcurrentHashMap` containers, and dotted edges represent singleton tuples. Each edge is labeled with a set of columns on the left and the associated lock placement on the right.

Each edge instance corresponds to a entry in a container in the heap, and it would often be too slow to actually use locks at such a fine granularity, not to mention the practical problem that there are infinitely many logical locks defined for container entries that are absent. Further, as shown in Figure 1, in practice different containers have different levels of support for safe concurrency, and so while we must use locks to protect some containers from all concurrent accesses, in other cases we can rely on the container to mediate concurrent access. Finally, since we treat container implementations as black boxes, we have no way to attach locks to the edge instances directly.

Instead of literally maintaining one lock for every possible edge instance, we implement logical locks using a smaller set of *physical locks* attached to instances of nodes in a decomposition. We describe the correspondence between logical and physical locks using a *lock placement* [13], which is a mapping from the set of logical locks onto the set of physical locks. Many logical locks may map onto the same physical lock, and acquiring a physical lock corresponds to taking all of the corresponding logical locks. By choosing different lock placements we can describe different granularities of locking. Since physical locks are attached to node instances, in general there may be an unbounded number of physical locks.

Physical Locks To each node v in a decomposition we attach a set of physical locks $\{v^0, v^1, \dots\}$. If there is only a single physical lock attached to a node we simply write v for both the node and its unique physical lock.

Lock Placements A *lock placement* ψ is a function mapping the logical lock associated with each edge onto a physical lock on a node that implements it. We define lock placements on the (static) decomposition which we extend to (dynamic) decomposition instances in the obvious way; if a lock placement ψ maps the logical lock on edge e to the physical lock on node v, then at runtime the compiler maps the logical lock on edge instance e_t onto the physical lock on node instance v_t.

Recall the directed graph example from Section 2. The relation in question has three columns $\{\mathrm{src}, \mathrm{dst}, \mathrm{weight}\}$ related via the functional dependency $\mathrm{src}, \mathrm{dst} \rightarrow \mathrm{weight}$. Figure 3 shows three possible decompositions, each with a different choice of data structures and lock placements.

Figure 3(a) uses a coarse-grain lock placement

$$\psi_1(e) = \rho \text{ for all edges } e,$$

which protects all edges of the decomposition using a single lock at the root ρ. Since there is only one instance of the root node ρ in any decomposition instance, the same lock is used to protect everything. Further, since the logical locks of all edge instances are mapped to the same physical lock ρ, the lock serializes access to the entire decomposition data structure, ensuring that each (non-concurrent) `TreeMap` is only accessed by one transaction at a time.

Figure 3(b) depicts a fine-grain locking strategy decomposition in which each edge is protected by a lock at its head (i.e., objects in a container are protected by a single lock on the container itself), using the lock placement

$$\psi_2(\alpha\beta) = \alpha \text{ for all edges } \alpha\beta.$$

Edges ρu and ρv are protected by a lock at ρ, whereas edges uw, vy, wx and yz are protected by locks at u, v, w and y, respectively.

Both of the example lock placements described so far use a single lock to protect all the entries in each container. Figure 3(c) makes use of *speculative locking* (Section 4.5), one of two extensions which allow different dynamic instances of an edge in the same container to be protected by different locks. We defer further discussion of this example to Section 4.5.

Well-Formed Lock Placements We require that all lock placements satisfy the following conditions:

- The lock placement $\psi(uv)$ of each edge uv either must dominate the edge's source vertex u or be equal to v. (The latter case occurs in the speculatively-placed locks of Section 4.5.) By definition this condition ensures that the lock placement for an edge lies on every path from the root of the decomposition including that edge. This condition ensures the instance of the node $\psi(uv)$ named by the lock placement is unique for each edge instance uv_t. The domination requirement also simplifies query planning (Section 5), since it ensures that a query plan will always encounter the necessary locks for each edge no matter how the edge is reached.
- All edges between an edge and its lock placement share the same placement. That is, fix any edge uv and take any edge xy in a path in the decomposition from $\psi(uv)$ to u. Then we have $\psi(xy) = \psi(uv)$. This requirement ensures that if a lock protects an edge, then the lock also protects the path from the lock to that edge, thereby ensuring that if we hold a lock then the set of edges protected by that lock cannot change.

Logical Lock Implication Since we implement logical locks by mapping them onto a smaller set of physical locks, a transaction cannot acquire logical locks directly. Instead, a transaction must acquire physical locks that *imply* access to the logical locks that the transaction requires.

We say that a set of physical locks P held by a transaction *imply* exclusive or shared access, respectively, to the logical lock of edge instance uv_t under lock placement ψ if:

- the transaction holds exclusive or shared access, respectively, to the corresponding physical lock, that is, $\psi(uv)_t \in P$, and
- the mapping between the logical lock to the corresponding physical lock is stable, that is, there exists a path $\mathbf{w}_t$ from the root of the decomposition instance to u_t such that the transaction holds shared access to every edge in $\mathbf{w}_t$.

The stability criterion means that a physical lock only covers a logical lock if the transaction also holds locks that guarantee that the logical lock does in fact correspond to that physical lock; if not, a concurrent transaction might alter the heap and change the association between logical and physical locks. For example, consider a concurrent hashtable where the elements of each hash bucket are guarded by a lock. If a transaction moves an element v from bucket b_1 to bucket b_2 the lock guarding access to v changes, and any transaction that was concurrently accessing v by acquiring

the lock on b_1 no longer holds the correct lock for v for the lock placement. Thus, in the presence of updates that can change the structure of the heap, it is not sufficient to just hold the locks L guarding access to the particular data, but it is also necessary to hold locks on whatever portion of the heap structure guarantees that L remains the correct set of locks to hold!

Since lock placements are defined using a decomposition structure, for locking using a placement to be well-defined we must ensure that transactions always yield heap states that are valid instances of the corresponding decomposition. One of the benefits of data representation synthesis is that we are guaranteed that the operations emitted by the compiler preserve the decomposition structure by construction.

4.4 Lock Striping

Lock striping is a technique for boosting the throughput of a transaction by using a set of locks instead of a single lock. Consider again the decomposition of Figure 3(b), in which the lock placement maps the logical lock on each edge to a physical lock at the source of the edge. While this lock placement ensures safe and consistent transactions, by protecting each container with a single lock we serialize access to containers, and hence we cannot make effective use of concurrent containers such as `ConcurrentHashMap`. To leverage concurrent containers we can partition the elements of the container into a number of *stripes*, each with its own lock.

For example, in Figure 3(b), rather than mapping all instances of edges ρu and ρv to a single physical lock at node ρ, we can use k physical locks $\rho^0, \ldots \rho^{k-1}$. We then use a lock placement that stripes the logical locks attached to instances of ρu and ρv across the k physical locks:

$$\psi_3(e,t) = \begin{cases} \rho^i & \text{if } e = \rho u,\ i = t(\text{src}) \bmod k \\ \rho^i & \text{if } e = \rho v,\ i = t(\text{dst}) \bmod k \\ \alpha_t & \text{otherwise, where } e = \alpha\beta \end{cases} \quad (1)$$

The lock placement takes as input an edge e annotated with a tuple t; the fields of tuple t are used to select one of the k physical locks at ρ. If we do not know the relevant tuple fields in advance, for example if we want to iterate over the container, we can always conservatively take all k locks.

Lock striping is only applicable for containers that are concurrency-safe. For a concurrency-unsafe container, such as a `TreeMap`, we are limited to at most one lock for the entire container to ensure that no two threads access the container concurrently.

By increasing the value k we can reduce lock contention to arbitrarily low levels, at the cost of making operations such as iteration that access the entire container more expensive.

4.5 Speculative Lock Placements

When striping logical locks across physical locks, as the number of physical locks k increases in the limit each container entry has its own individual lock. Rather than preallocating locks for an unbounded number of objects, we can achieve this limiting case more efficiently by using a technique called *speculative locking* [13], motivated by transactional predication [3]. Speculative locking lazily constructs a unique physical lock for each logical lock.

The key to speculative locking is the identity of the lock that protects an edge instance depends on the state of the edge instance itself. We map the logical lock to a distinct physical lock for each edge instance present in a container by placing the lock in the node that is the target of the edge instance. For serializability the lock placement must also be defined for edge instances that are absent from the decomposition, not just those edges that are present. Since we cannot place locks for non-existent edge instances at the target of the edge, instead we map the logical locks for absent edges onto physical locks at the edge's source.

Up to this point, we have required that the lock guarding an edge e appear on all paths from the root before e is reached. For speculative locks, this invariant does not hold—we do not know what lock to acquire until we have reached the object we wish to protect. The key is that it is safe to perform unlocked reads of a concurrency-safe container to guess the identity of the lock that we should acquire. Since the container is concurrency-safe, reading without holding a lock is safe, however we have no guarantees that any information that we read will remain stable. Once we have guessed and acquired a lock, we can check to see if our guess was correct. There are two possibilities — either we guessed correctly, in which case we already held the lock that protects the edge and our read was stable, or we guessed incorrectly, in which case the edge must point somewhere else. In the latter case we can release the lock we guessed and try again. While speculatively acquiring a lock is not physically two-phase, a transaction can be viewed as acquiring logical locks in a two phase manner [13].

Speculative lock acquisition differs from the well-known but broken double-checked locking idiom in two key ways—firstly, we always acquire a lock and recheck reads under that lock, and secondly we require that concurrent containers are linearizable, that is, with semantics analogous to a Java volatile field.

For example, the decomposition depicted in Figure 3(c) uses a mixture of both speculative and non-speculative locking — in particular, the locks that protect edges ρx and ρy are placed at the target of each edge on nodes x and y respectively. To take a lock on an edge instance ρx_t a transaction must first speculatively lookup entry t in the map without locking, acquire the lock on ρ or x_t if the edge instance is absent or present, respectively, and then verify that the correct lock was taken. The data structure implementing edge ρx is a `ConcurrentHashMap`, which is concurrency-safe, so it is safe to speculatively read an edge without holding its lock.

$$\psi_4(e,t) = \begin{cases} u_t & \text{if } e = \rho u,\ e_t \text{ is present} \\ \rho^i & \text{if } e = \rho u,\ e_t \text{ is not present},\ i = t(\text{src}) \bmod k \\ v_t & \text{if } e = \rho v,\ e_t \text{ is present} \\ \rho^i & \text{if } e = \rho v,\ e_t \text{ is not present},\ i = t(\text{dst}) \bmod k \\ \alpha_t & \text{otherwise, where } e = \alpha\beta \end{cases}$$

5. Query Planning and Lock Ordering

In Section 4 we introduced concurrent decompositions, which describe a relational specification using both concurrent and non-concurrent containers in combination with locks. In this section we show how to compile the relational operations of Section 2 into code tailored to a particular concurrent decomposition.

Existing work [12] described how to compile relational operations in a non-concurrent context. There are two additional complications we must deal with when generating concurrent implementations of relational operations—we must ensure that a transaction takes the locks that protect the decomposition edges it touches, and we must ensure that transactions are deadlock-free.

5.1 Deadlock-Freedom and Lock Ordering

A common strategy for ensuring that a set of concurrent transactions is deadlock-free is to impose a total order on locks. If all transactions acquire locks in ascending lock order, then we are guaranteed that concurrent transactions are deadlock-free.

We ensure deadlock-freedom for concurrent decomposition operations by imposing a total lock order on the physical locks of a decomposition; it is the responsibility of the query planner to generate code that respects this order.

$$q ::= x \mid \mathsf{let}\ x = q_1\ \mathsf{in}\ q_2 \mid \mathsf{lock}(q, v) \mid \mathsf{unlock}(q, v) \mid \mathsf{scan}(q, uv) \mid \mathsf{lookup}(q, uv) \quad \text{expressions}$$

Figure 4. Concurrent query language. We only show the fragment necessary for implementing query operations.

All query plans must obey a single static order on all possible physical locks of a decomposition. The precise set of physical locks in existence may change as we allocate and deallocate node instances, but the relative order of any given pair of physical locks never changes during runtime. We order physical locks firstly on a topological sort of the decomposition nodes to which they belong. We order different instances of the same node lexicographically on the values of the key columns. Finally, we order the physical locks attached to each node instance by number.

For example, consider the decomposition of Figure 3(c). We fix a topological order of the nodes, say

$$\rho < x < y < z < w;$$

meaning that all locks attached to ρ are ordered before all locks attached to instances of x, and so on. We lift the topological order on nodes to a total order on node instances

$$\rho < x_{s_0} < x_{s_1} < \cdots < y_{t_0} < y_{t_1} < \cdots,$$

where the tuple sequences (s_i) and (t_i) are in lexicographic order. Finally, since there may be more than one physical lock per node due to lock striping, we lift the total order on node instances to a total order on physical locks:

$$\rho^0 < \rho^1 < \cdots < x^0_{s_0} < x^1_{s_0} < \cdots < x^0_{s_1} < x^1_{s_1} \cdots.$$

As an aside, it is necessary that we totally order the physical locks of a decomposition, not the logical locks; a query only acquires physical locks directly and it is the order of those physical locks that is pertinent to deadlock.

5.2 Query Language

Once we have fixed a total order on the physical locks of a decomposition, the query planner must generate well-locked, two-phase code that respects the lock order for each possible query.

A key requirement of a query plan is that it must make explicit which locks to acquire and in which order. The query trees in the literature [12] are not suitable for reasoning about locks since they have no notion of sequencing of expressions.

In the concurrent setting, we extend query trees to a fragment of a strict, impure functional language, shown in Figure 4. The let-binding construct of the concurrent query language can describe the order of execution of operations with side-effects, in particular lock and unlock operations. A query expression q is one of: a variable reference x, let-binding $\mathsf{let}\ x = q_1\ \mathsf{in}\ q_2$, a lock acquisition $\mathsf{lock}(q, v)$, a lock release $\mathsf{unlock}(q, v)$, an edge lookup $\mathsf{lookup}(q, v)$, or an edge iteration $\mathsf{scan}(q, v)$. We discuss the semantics of expressions shortly.

Query States Evaluating any expression in the query language yields a set of *query states*. A *query state* is a pair (t, m) of a tuple t containing a subset of the relation's columns, together with a mapping m from decomposition nodes v to the corresponding node instance v_t. If a vertex v with type $v\colon A \triangleright B$ appears in the domain of m, tuple t must contain sufficient columns such that v_t is uniquely defined, that is, $A \subseteq \operatorname{dom} t$.

Query Expressions We now describe the semantics of each query expression. Variable lookup and variable binding are standard. Let bindings also allow us to sequence operations with side effects, such as locks; we use a don't-care variable $\mathsf{let}\ \_ = q_1\ \mathsf{in}\ q_2$ to denote executing q_1 just for its side effects, discarding its return value, and then executing q_2.

A lock acquisition $\mathsf{lock}(q, v)$ acquires the physical locks associated with the instance of node v in each query state in set q. Like all expressions in the query language, lock acts on a set of query states q, locking the instance of physical lock v in each element of the set. The lock operation must acquire locks in accordance with the lock order. While the query planner always produces the query plans with lock expressions in correct node order, the lock operator must sort node instances into the correct lexicographic order before acquiring locks. The counterpart $\mathsf{unlock}(q, v)$ unlocks the instances of node v in the set q; unlike the lock operation the unlock operation does not need to enforce sorted order on its arguments.

Recall that for each node instance u_t an edge uv in a decomposition corresponds to a container data structure that maps a set of key columns $\mathsf{cols}(uv)$ to a set of node instances of $v_{t'}$. The operation $\mathsf{scan}(q, uv)$ iterates over the contents of the container, returning the natural join of the input query states q together with the entries of the map. If the query states in q contain a superset of the key columns $\mathsf{cols}(uv)$, we can instead use the more efficient operation $\mathsf{lookup}(q, uv)$, which looks up the particular entry v_t in the container. Both the lookup and scan operations require that the input query states contain an instance of the source vertex u.

For example, suppose we wanted to iterate over all of the tuples of a directory entry relation represented using the decomposition of Figure 2(a) under a coarse lock placement which places all locks at the root node ($\psi(e) = \rho$ for all e). One possible query plan is:

$$\begin{array}{l} 1\text{: } \mathsf{let}\ \_ = \mathsf{lock}(a, \rho)\ \mathsf{in} \\ 2\text{: } \mathsf{let}\ b = \mathsf{scan}(\mathsf{scan}(a, \rho y), yz)\ \mathsf{in} \\ 3\text{: } \mathsf{let}\ \_ = \mathsf{unlock}(a, \rho)\ \mathsf{in} \\ 4\text{: } b \end{array} \tag{2}$$

The query plan takes as input a variable a, consisting of a singleton query state containing the location of the decomposition root ρ. The query plan first locks the unique instance of the root vertex ρ in set a (line 1), and then iterates over instances of the edge ρy from the root vertex in set a (line 2, $\mathsf{scan}(a, \rho y)$); the iteration yields a set of query states that contain instances of node y together with valuations of the parent and name fields. For each such query state, we then iterate over the singleton instances of edge yz (line 2, $\mathsf{scan}(\cdots, yz)$), yielding a valuation for the child field; we store the resulting set of query states as a set b. We release the acquired locks (line 3), and return our final query states b (line 4).

To make the execution concrete, suppose we execute the query plan (2) on the decomposition instance of Figure 2(b). The query plan receives the input query state $a = \{(\langle\rangle, \{\rho \mapsto \rho_0\})\}$ as input, which contains the location of the decomposition root but no valuations for relation columns. The lock statement acquires the lock attached to ρ_0, which is the unique instance of ρ in set a. Evaluation of the expression $\mathsf{scan}(a, \rho y)$ in line 2 yields states

$$\begin{array}{l} \{(\langle \text{parent: } 1,\ \text{name: `a'} \rangle, \{\rho \mapsto \rho_0,\ y \mapsto y_3\}) \\ \ (\langle \text{parent: } 2,\ \text{name: `b'} \rangle, \{\rho \mapsto \rho_0,\ y \mapsto y_2\}) \\ \ (\langle \text{parent: } 2,\ \text{name: `c'} \rangle, \{\rho \mapsto \rho_0,\ y \mapsto y_1\})\}. \end{array}$$

Applying the expression $\mathsf{scan}(\cdots, yz)$ in line 2 yields the states

$$\begin{array}{l} \{(\langle \text{parent: } 1,\ \text{name: `a'},\ \text{child: } 2 \rangle, \{\rho \mapsto \rho_0,\ y \mapsto y_3\ z \mapsto z_3\}) \\ \ (\langle \text{parent: } 2,\ \text{name: `b'},\ \text{child: } 3 \rangle, \{\rho \mapsto \rho_0,\ y \mapsto y_2\ z \mapsto z_2\}) \\ \ (\langle \text{parent: } 2,\ \text{name: `c'},\ \text{child: } 4 \rangle, \{\rho \mapsto \rho_0,\ y \mapsto y_1\ z \mapsto z_1\})\} \end{array}$$

which we store as set b. Finally, we unlock the lock at ρ_0 and return the entries of b as the query result.

Query plan (2) was not the only possible query plan, even under the same decomposition and lock placement. Another possible query

plan uses edges ρx and xy instead of the edge ρy.

$$\begin{array}{l}
1\text{: let } _ = \mathsf{lock}(a, \rho) \text{ in} \\
2\text{: let } b = \mathsf{scan}(\mathsf{scan}(\mathsf{scan}(a, \rho x), xy), yz) \text{ in} \\
3\text{: let } _ = \mathsf{unlock}(a, \rho) \text{ in} \\
4\text{: } b
\end{array} \tag{3}$$

Now suppose we want to make the same query on the same decomposition, under the lock placement shown in Figure 2(a), in which a lock on every node protects the edges with their source at that node. The equivalent of query plan (3) under the new finer-grained lock placement is:

$$\begin{array}{l}
1\text{: let } _ = \mathsf{lock}(a, \rho) \text{ in} \\
2\text{: let } b = \mathsf{scan}(a, \rho x) \text{ in} \\
3\text{: let } _ = \mathsf{lock}(b, x) \text{ in} \\
4\text{: let } c = \mathsf{scan}(b, xy) \text{ in} \\
5\text{: let } _ = \mathsf{lock}(c, y) \text{ in} \\
6\text{: let } d = \mathsf{scan}(c, yz) \text{ in} \\
7\text{: let } _ = \mathsf{unlock}(c, y) \text{ in} \\
8\text{: let } _ = \mathsf{unlock}(b, x) \text{ in} \\
9\text{: let } _ = \mathsf{unlock}(a, \rho) \text{ in} \\
10\text{: } d
\end{array} \tag{4}$$

Query Planner To pick a good implementation for each query, the compiler uses a query planner that finds the query plan with the lowest cost as measured by a heuristic cost estimation function. The concurrent query planner is based on the non-concurrent query planner described in the literature [12]; like the non-concurrent query planner, the concurrent query planner enumerates valid query plans and chooses the plan with the lowest cost estimate.

The main extension for concurrency is the query planner must only permit queries that acquire and hold the right locks in the right order. Internally the query planner only considers plans with two phases, a growing phase consisting of a sequence of lock, scan, and lookup statements, and a shrinking phase containing a matching sequence of unlock statements in reverse order; such plans are trivially two-phase. To ensure that queries acquire the correct locks in the correct order, we extend the definition of query validity to require that lock statements in a query plan appear in the decomposition node lock order, and that lookup and scan operations must be preceded by a lock of the corresponding physical lock.

As in the non-concurrent case, we reuse the query planning infrastructure to compile mutation operations. Code for mutations is generated by first constructing a concurrent query plan that locates and locks all of the edges that require updating; the code generator then emits code that uses the query results to perform the required updates, just as in the non-concurrent case, sandwiched between the growing and shrinking phases of the query plan.

Query Expression Compilation Each query expression evaluates to a set of query states. Internally we compile each query expression into an iterator over query states. We compile let-bindings by evaluating the right-hand side of the binding and storing the results into a temporary set of query states; subsequent references to a bound variable compile to iterations over the stored state set.

In general the lock statement must sort the locks that it acquires. However, in some cases the set of locks may already be in the correct order, so it is superfluous to sort them. For example, consider the acquisition of the locks on node b in query 4 (line 3). Edge ρx is represented by a `TreeMap` in the decomposition, which stores its entries in sorted order; a scan over the edge will therefore yield entries in sorted order, which coincides with the correct lock order. Conversely, if edge was represented using a `HashMap` then iteration would return entries in an unpredictable order, so the code would have to sort the locks before acquiring them. The compiler uses a simple static analysis to detect lock statements where it can avoid sorting.

6. Experimental Evaluation

We have developed a prototype implementation of concurrent data representation synthesis, targeted at the Java virtual machine. The prototype is implemented as a Scala [18] compiler plugin; relations and relational operations are translated into Scala ASTs, which the Scala compiler backend converts to JVM bytecode. In this section we evaluate the performance of the resulting implementation.

6.1 Autotuner

A programmer may not know the best possible representation for a concurrent relation. To help find an optimal decomposition for a particular relational specification, we have implemented an autotuner which, given a concurrent benchmark, automatically discovers the best combination of decomposition structure, container data structures, and choice of lock placement.

Existing work [12] described an autotuner capable of identifying a good decomposition in the absence of concurrency. We extend the idea of autotuning to a concurrent setting.

To enumerate decompositions, the autotuner first chooses an adequate decomposition structure, exactly as for the non-concurrent case [12]. Next, the autotuner chooses a well-formed lock placement; every edge of a decomposition needs a corresponding physical lock. Finally the autotuner chooses a data structure implementation for each edge. If the chosen lock placement serializes access to an edge, the autotuner picks a non-concurrent container, whereas if concurrent access to a container is permitted by the lock placement then the autotuner chooses a concurrency-safe container.

6.2 Evaluation

We evaluate the generated code using a synthetic benchmark modeled after the methodology of Herlihy et. al [14] for comparing concurrent map implementations, extended to the more general context of a relation. We fix a particular relational specification, together with a set of relational operations. For any given choice of decomposition, the benchmark uses k identical threads that operate on a single shared relation. Starting from an initially empty relation, each thread executes 5×10^5 randomly chosen operations. We plot the total throughput of all threads in operations per second against the number of threads to obtain a throughput-scalability curve. By varying the distribution of relational operations we can evaluate the performance of the relation under different workloads.

For our benchmarks we use the directed graph relation described in Section 4.3, together with four relational operations, namely find successors, find predecessors, insert edge, and remove edge. The find successor operation chooses a random src value and queries the relation for the set of all $\mathrm{dst}, \mathrm{weight}$ pairs corresponding to that src. The find predecessor operation is similar but chooses a random dst and queries for $\mathrm{src}, \mathrm{weight}$ pairs. The insert edge operation chooses a random $\mathrm{src}, \mathrm{dst}, \mathrm{weight}$ triple to insert into the relation; to ensure that the relation's functional dependency is not violated we use the compare-and-set functionality of the insert operation to check that no existing edge shares the same $\mathrm{src}, \mathrm{dst}$ parameters. Finally the remove operation chooses a random $(\mathrm{src}, \mathrm{dst})$ tuple and removes the corresponding edge, if present.

We performed our experiments on a machine with two six-core 3.33Ghz Intel X5680 Xeon CPUs, each with 12Mb of L3 cache, and 48Gb memory in total. Hyperthreading was enabled for a total of 24 hardware thread contexts. All benchmarks were run on a OpenJDK 6b20 Java virtual machine in server mode, with a 4Gb initial and maximum heap size. We repeated each experiment 8 times within the same process, with a full garbage collection between runs. We discarded the results of the first 3 runs to allow the JIT compiler time to warm up; the reported values are the average of the last 5 runs.

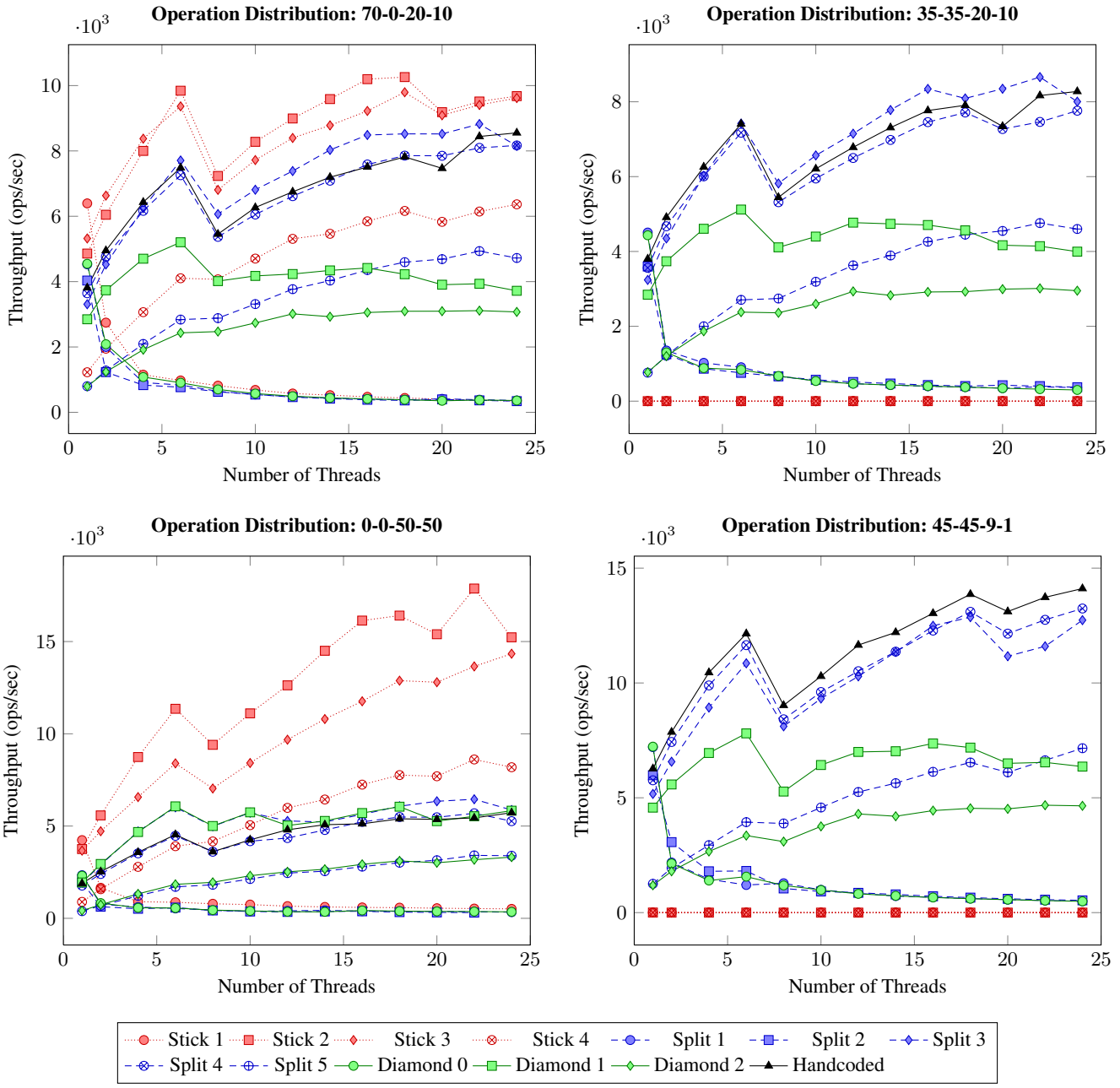

Figure 5. Throughput/scalability curves for a selection of decompositions. Each thread performs 5×10^5 random graph operations. Each graph is labeled x-y-z-w, denoting a distribution of $x\%$ successors, $y\%$ predecessors, $z\%$ inserts, and $w\%$ removes. "Stick" decompositions are structurally isomorphic to Figure 3(a) but have different choices of data structures and lock placements, similarly "split" to Figure 3(b), and "diamond" to Figure 3(c).

Figure 5 presents throughput-scalability curves for a selection of decompositions. We generated 448 variants of the three decomposition structures shown in Figure 3 using the autotuner, varying the choice of lock placement, lock striping factor (chosen for simplicity to be either 1 or 1024), and selection of containers from the options `ConcurrentHashMap`, `ConcurrentSkipListMap`, `HashMap`, and `TreeMap`. For clarity of presentation we selected 12 representative decompositions that cover a spectrum of different performance levels across the 4 benchmarks; we compare the performance of both the automatically generated implementations and a hand-written implementation.

One obvious feature of the results is that the "stick" decompositions, which are variants of the decomposition shown in Figure 3(a), perform relatively well for the two workloads (70-0-20-10 and 0-0-50-50) that consist only of successor, insert, and remove operations. For the workloads that include finding predecessors (35-35-20-10 and 45-45-9-1), "split" (Figure 3(b)) and "diamond" (Figure 3(c)) perform far better. Finding successors in a stick decomposition is much more efficient than finding predecessors, which requires iterating over all edges in the graph.

Coarsely-locked data structures scale poorly; three of the decompositions shown in the graph (Stick 1, Split 1, Diamond 1) use a single coarse lock to protect the entire decomposition; each con-

tainer uses a coarsely locked HashMap to represent the top level of edges in the decomposition, and a TreeMap to represent the second level of edges. Another decomposition (Split 2) uses striped locks and concurrent maps on the left side of the decomposition (ρu, uw, wx), but uses a single coarse lock to protect the other edges of the graph, leading to similarly poor performance.

Sticks 2, 3, 4 use a striped lock at the root to protect a ConcurrentHashMap of HashMap containers, a ConcurrentHashMap of TreeMap containers, and a ConcurrentSkipListMap of HashMap containers, respectively; all scale much better than the coarsely-locked data structures.

Decompositions which do not share nodes between the two sides of the decomposition outperform decompositions that do. For example, Split 3 and Diamond 1 both use ConcurrentHashMap containers to represent the top-level edges and HashMap containers to represent the second level edges, differing only in the sharing structure of the decomposition; the split decomposition performs better in most cases. Split 4 is a variant of Split 3 with TreeMap containers in place of the HashMap containers. Interestingly, there is a small but consistent effect where Split 3 is the best choice for the 35-35-20-10 workload and Split 4 is better for the 45-45-9-1 workload. Split 5 and Diamond 2 are also similar to Split 3 and Diamond 2, except with ConcurrentSkipListMap containers in place of ConcurrentHashMap containers; once again, the split decomposition outperforms the diamond decomposition.

The handcoded implementation (which was written before the automated experiments) is essentially Split 4, and produces almost identical results; the difference in performance between the two is probably due to extra boxing in the generated code that could be eliminated with improvements to the code generator. But clearly the automatically generated code is competitive with the hand-written code but requires much less programmer effort, and unless one was willing to write many different hand-coded versions, the autotuner will be able to find variants that outperform any single hand-written code for particular workload characteristics.

It is interesting to note that diamond decompositions outperformed split decompositions in the non-concurrent case [12]; the result here is reversed for two reasons. The split decomposition produces less lock contention, since a pair of transactions may query for successors and predecessors in parallel without interfering with one another. Much of the benefit for sharing in the non-concurrent case came from the fact that it is possible to remove an object from the middle of an intrusive doubly-linked list in constant time. Since it is impossible to write such intrusive containers in a generic fashion in the Java type system, we do not gain the advantage of more efficient removals from shared nodes.

The prominent decrease in throughput evident in Figure 5 when increasing from 6 to 8 threads is an artifact of the thread scheduler and the memory hierarchy of the test machine. The test machine has two six-core CPUs, each with two hardware contexts per core. The benchmark harness schedules up to the first six threads on different cores of the same CPU, sharing a common on-chip L3 cache. The harness schedules the next six threads on the second CPU; when threads are split across two CPUs they must communicate via the processor interconnect, rather than via a shared on-chip cache. Communication off-chip is substantially slower than on-chip communication, producing the "notch" in the graph.

Overall the experiments show the benefits of automatic synthesis of both data structures and synchronization: sophisticated implementations competitive with hand written code can be produced at much lower cost in programmer effort, while at the same time providing guarantees about the correctness of the implementation of the high-level concurrent relational program.

7. Discussion and Related Work

Our results touch upon a great deal of previous work in several distinct domains. For space reasons our survey is necessarily brief.

We build upon previous work in data representation synthesis for sequential data structures [12] and the general theory of lock placements [13]. Our contributions beyond previous works include the extension of the programming interface to concurrent relations, the integration of different kinds of concurrent and non-concurrent data structures as building blocks, the extensions needed to integrate decompositions and lock placements, the redesign of query planning in the concurrent setting including guaranteeing deadlock freedom, and a complete implementation and experiments.

As mentioned in Section 1, the idea of compiling programs that work on relations into specialized data structures originated with [5], and its various developments [1, 2, 22]. Earlier work explored data structure selection for sets in SETL [7, 19]. Neither line of work addresses the synthesis of concurrent data representations.

The closest to our work in spirit is Paraglider [23], which provides semi-automatic assistance in synthesizing low-level concurrent algorithms and data structures. Paraglider focuses on the correct implementation of a single concurrent data structure, while our work is about assembling multiple concurrent and non-concurrent data structures into more complex abstractions. Thus, Paraglider is complementary to our approach, and we could extend our menu of concurrent building blocks with Paraglider-generated components.

Various authors have investigated techniques for inferring locks to implement atomic sections [4, 6, 9, 11, 16, 17, 24]. A related problem is automatically optimizing programs with explicit locking by combining multiple locks into one [8]. A key part of this class of work is constructing a mapping from program objects to the locks that protect them, which is similar to, but more specialized than, lock placements. This body of work also takes the data structures as fixed and attempts to infer the needed locks, while we consider the possible data representations and lock placements simultaneously.

Our system can be viewed as implementing a pessimistic software transactional memory [21]. Future extensions of our work could synthesize optimistic concurrency control primitives in addition to pessimistic locks. Unlike traditional software transactional memory systems, which perform on-line dynamic analysis to determine the read and write sets of transactions, our system performs much of the same analysis statically, resulting in run-time code with considerably lower overhead. Furthermore, our approach is able to automatically change the data structures and granularity of locking used to improve overall performance. It is also worth noting that speculative locking was first introduced in the context of advanced software transactional memory systems [3].

Finally the original paper on two-phase locking made explicit the idea of locking not just objects, but program predicates [10]. Logical locks are such predicates, which we realize in practice by mapping logical locks onto physical locks using a lock placement.

8. Conclusion

We have described an approach for synthesizing data representations for concurrent programs. Beginning with a program written using high-level concurrent relations, our system automatically selects the decomposition of the relation into a set of subrelations, chooses concrete data structures for the sub-relations, and selects a locking strategy that guarantees all relational operations are both serializable and deadlock free for the chosen representation. Because the high-level description admits multiple choices for each of these dimensions (subject to correctness constraints that rule out some possibilities), programs written in this style describe a space of possible implementations, a fact that we are able to exploit by using a combination of static and dynamic techniques to search this space to

find a high-performance implementation. We have described an extensive experiment on a concurrent graph benchmark, demonstrating the wide range of possible implementations and their trade-offs.

References

[1] Don Batory and Jeff Thomas. P2: A lightweight DBMS generator. *Journal of Intelligent Information Systems*, 9:107–123, 1997. ISSN 0925-9902. doi: 10.1023/A:1008617930959.

[2] Don Batory, Gang Chen, Eric Robertson, and Tao Wang. Design wizards and visual programming environments for GenVoca generators. *IEEE Transactions on Software Engineering*, 26(5):441–452, May 2000. ISSN 0098-5589. doi: 10.1109/32.846301.

[3] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. Transactional predication: high-performance concurrent sets and maps for stm. In *Proceeding of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, pages 6–15, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-888-9. doi: 10.1145/1835698.1835703.

[4] Sigmund Cherem, Trishul Chilimbi, and Sumit Gulwani. Inferring locks for atomic sections. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 304–315, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375619.

[5] Donald Cohen and Neil Campbell. Automating relational operations on data structures. *IEEE Software*, 10(3):53–60, May 1993. doi: 10.1109/52.210604.

[6] Dave Cunningham, Khilan Gudka, and Susan Eisenbach. Keep off the grass: Locking the right path for atomicity. In Laurie Hendren, editor, *Compiler Construction*, volume 4959 of *Lecture Notes in Computer Science*, pages 276–290. Springer Berlin / Heidelberg, 2008. ISBN 978-3-540-78790-7. doi: 10.1007/978-3-540-78791-4_19.

[7] Robert B. K. Dewar, Arthur Grand, Ssu-Cheng Liu, Jacob T. Schwartz, and Edmond Schonberg. Programming by refinement, as exemplified by the SETL representation sublanguage. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(1):27–49, January 1979. ISSN 0164-0925. doi: 10.1145/357062.357064.

[8] Pedro C. Diniz and Martin C. Rinard. Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs. *Journal of Parallel and Distributed Computing*, 49(2):218–244, 1998. ISSN 0743-7315. doi: 10.1006/jpdc.1998.1441.

[9] Michael Emmi, Jeffrey S. Fischer, Ranjit Jhala, and Rupak Majumdar. Lock allocation. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*, pages 291–296, New York, NY, USA, 2007. ACM. ISBN 1-59593-575-4. doi: 10.1145/1190216.1190260.

[10] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19:624–633, November 1976. ISSN 0001-0782. doi: 10.1145/360363.360369.

[11] Richard L. Halpert, Christopher J. F. Pickett, and Clark Verbrugge. Component-based lock allocation. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 353–364, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2944-5. doi: 10.1109/PACT.2007.23.

[12] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. Data representation synthesis. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 38–49, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993504.

[13] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. Reasoning about lock placements. In *Proceedings of the European Symposium on Programming (ESOP)*, LNCS. Springer Berlin / Heidelberg, 2012. To appear.

[14] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A provably correct scalable concurrent skip list. In *Conference On Principles of Distributed Systems (OPODIS)*, 2006.

[15] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12:463–492, July 1990. ISSN 0164-0925. doi: 10.1145/78969.78972.

[16] Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Lock inference for atomic sections. In *Workshop on Languages, Compilers and Hardware Support for Transactional Computing*, 2006.

[17] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker: Synchronization inference for atomic sections. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 346–358, New York, NY, USA, 2006. ACM. ISBN 1-59593-027-2. doi: 10.1145/1111037.1111068.

[18] Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean McDirmid, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Lex Spoon, and Matthias Zenger. An Overview of the Scala Programming Language, second edition. Technical Report LAMP-REPORT-2006-001, École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland, 2006.

[19] Edmond Schonberg, Jacob T. Schwartz, and Micha Sharir. Automatic data structure selection in SETL. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 197–210, New York, NY, USA, 1979. ACM. doi: 10.1145/567752.567771.

[20] Ohad Shacham, Nathan Bronson, Alex Aiken, Mooly Sagiv, Martin Vechev, and Eran Yahav. Testing atomicity of composed concurrent operations. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 51–64, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0940-0. doi: 10.1145/2048066.2048073.

[21] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10:99–116, 1997. ISSN 0178-2770. doi: 10.1007/s004460050028.

[22] Yannis Smaragdakis and Don Batory. DiSTiL: A transformation library for data structures. In *Conference on Domain-Specific Languages*, pages 257–271, October 1997.

[23] Martin Vechev and Eran Yahav. Deriving linearizable fine-grained concurrent objects. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 125–135, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375598.

[24] Yuan Zhang, Vugranam Sreedhar, Weirong Zhu, Vivek Sarkar, and Guang Gao. Minimum lock assignment: A method for exploiting concurrency among critical sections. In *Languages and Compilers for Parallel Computing*, volume 5335 of *Lecture Notes in Computer Science*, pages 141–155. Springer Berlin / Heidelberg, 2008. ISBN 978-3-540-89739-2. doi: 10.1007/978-3-540-89740-8_10.