Learning Quantitative Representation Synthesis

Mayur Patil Farzin Houshmand Mohsen Lesani
{mpati005, fhous001, lesani}@ucr.edu
Univ. of California, Riverside, USA

Abstract

Software systems often use specialized combinations of data structures to store and retrieve data. Designing and maintaining custom data structures particularly concurrent ones is time-consuming and error-prone. We let the user declare the required data as a high-level specification of a relation and method interface, and automatically synthesize correct and efficient concurrent data representations. We present provably sound syntactic derivations to synthesize structures that efficiently support the interface. We then synthesize synchronization to support concurrent execution on the structures. Multiple candidate representations may satisfy the same specification and we aim at quantitative selection of the most efficient candidate. Previous works have either used dynamic auto-tuners to execute and measure the performance of the candidates or used static cost functions to estimate their performance. However, repeating the execution for many candidates is time-consuming and a single performance model cannot be an effective predictor of all workloads across all platforms. We present a novel approach to quantitative synthesis that learns the performance model. We developed a synthesis tool called Leqsy that trains an artificial neural network to statically predict the performance of candidate representations. Experimental evaluations demonstrate that Leqsy can synthesize near-optimum representations.

CCS Concepts• **Software and its engineering → Concurrent programming structures**;
Keywords: Synthesis, Data Structures, Code generation

ACM Reference Format:
Mayur Patil, Farzin Houshmand, and Mohsen Lesani. 2020. Learning Quantitative Representation Synthesis . In *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL '20), June 15, 2020, London, UK*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3394450.3397467

MAPL '20, June 15, 2020, London, UK

ACM ISBN 978-1-4503-7996-0/20/06…$15.00
https://doi.org/10.1145/3394450.3397467

1 Introduction

From the outset, system development involves the choice and aggregation of the data structures that store and retrieve data. Designing and tuning data structures particularly those that are safe and efficient on multi-core processors is difficult and error-prone. Mainstream programming languages offer libraries of concurrent data structures that are atomic (linearizable) [14], deadlock-free and efficient. However, applications often require specialized data structures that are not immediately provided by standard libraries.

Programmers usually commit to a particular assembly of data structures to represent the application data. Manually implementing elaborate data structures can be time-consuming. Enforcing invariants on multiple overlapping structures is error-prone. Further, composing multiple operations on concurrent data structures is not necessarily atomic and has led to numerous errors [16, 24]. More importantly, system requirements evolve. Modifying the assembly and synchronization can introduce inadvertent bugs.

We let the user declare her required data as a short high-level relational specification. She specifies a relation and the required interface. We automatically synthesize a concrete representation for the specification. This approach offers advantages in programmer productivity, correctness and performance. Although representations may be complicated, they usually have simple specifications. As the low-level implementation details are abstracted, programmer's time and effort is saved for both creation and maintenance of the structure. Changes in the requirements lead to only small changes in the specification. Synthesis produces correct-by-construction representations that faithfully implement the specification. Thus, the risk of introducing defects during maintenance is reduced as well.

High-level specifications give the synthesizer the freedom to choose from a space of solutions. Multiple representations may exist for the same specification. Different representations exhibit different performance characteristics and the most efficient representation varies with the workload [12]. This variance highlights the importance of the flexibility that synthesis offers to easily switch between representations. Previous works [11, 12, 18] have used auto-tuners that given a sample workload, execute and measure the performance of the synthesized candidates to choose one. However, repeating the execution for many candidates is time-consuming.

Quantitative synthesis [1, 3–5] aims to synthesize programs that are not only correct but optimum in terms of a

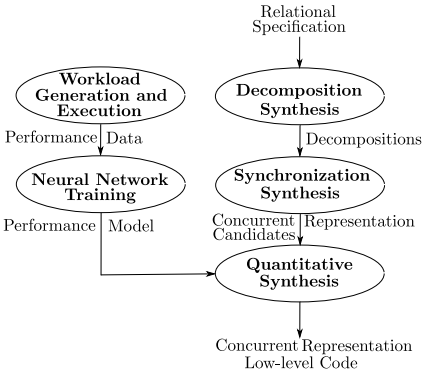


Figure 1. Overview of Leqsy

quantitative metric. Previous works used static cost functions to estimate the cost of context switching [3] and the performance of representations [17]. However, a performance model depends on the usage patterns and the target platform. Therefore, a single performance model may not be a good predictor of all use-cases across all platforms. Further, due to complicated architectural behaviors, a realistic performance model may not match the intuition. For instance, previous work [3] reported that coarse-grained outperformed fine-grained locking for certain workloads. We observe that performance models are insights that can be only learned from experimental data. We present a novel platform-independent approach to quantitative synthesis that learns the performance model from training workloads.

Given high-level relational specifications, we benefit from the trained model to synthesize efficient concurrent representations. A specification declares the relation and its functional dependencies. In addition, it captures the method interface on the relation together with method call frequencies. We check that the interface is well-formed i.e. it complies with the functional dependencies. We then use the interface to construct map structures called decompositions that support the interface efficiently. We present novel syntactic derivations that given an interface, synthesize decompositions that support the interface. We formalize the decomposition language, present a type system that associates union types with decompositions, and then define entailment of an interface by a decomposition and its type. We prove that the synthesis derivations are sound i.e. every derived decomposition entails the given interface. We use the derivation rules to enumerate [30] candidate decompositions.

The synthesized data representations may be accessed from multiple threads concurrently. Synchronization synthesis involves non-trivial choices for the number and placement of locks, and the order and level of their acquisition and release with implications for correctness and efficiency. We couple each map of a decomposition with a read-write lock array and synthesize candidates with different array sizes per map. We present locking protocols that provide linearizability and deadlock-freedom.

To choose the most efficient candidate representation, we need a model that can estimate the performance of candidates. We train a multi-layer perceptron [22, 32] artificial neural network to represent the performance model. We generate ample training datapoints by enumerating different representation structures and call frequencies. We execute each generated representation by a workload with the corresponding call frequencies and measure its performance. Datapoints are identified by a set of features including the number of branches and maps, the number of locks at each map and the frequency of lookup and iteration on each map. For each datapoint, the value of features are inputs and performance is the output to train the neural network. Training is a preprocess that is done only once for a platform.

We have implemented this approach in a tool called Leqsy (for Learning Quantitative Synthesis). An overview of Leqsy's structure is presented in Figure 1. Given a relational specification, it outputs a concurrent data representation as a Java source code class that developers can integrate to their codebase. The synthesized data structures can also replace existing data structures in legacy codebases. We empirically evaluated Leqsy on benchmarks that we adopted from previous work: Graph, Process scheduler and File System [11, 12] use-cases. The results show that Leqsy can successfully synthesize a concurrent representation whose performance matches or is close to the performance of the optimal representation. In summary, the contributions are the following:

- A high-level specification language that captures the method interface and the call frequencies in addition to the relation and its functional dependencies. (§ 2)
- A formal model of decompositions, and their type system. A formalization of entailment of an interface by a decomposition and its type. Syntactic derivations to synthesize decompositions for a given interface and the proof of soundness of synthesis. Further, synchronization synthesis for decompositions. (§ 3 and § 4)
- A novel approach to quantitative synthesis that learns the performance model. Feature engineering and training a multi-layer perceptron that can predict the performance of candidate concurrent representations. (§ 5)
- A synthesis tool called Leqsy that given user specifications generates Java source code of concurrent representations and its experimental evaluation. § 6.

2 Specification

We now present the high-level specifications. The user simply specifies her desired data structure as a relation with a set of attributes, their functional dependencies and an access interface. Specifying relations is more high-level that defining decompositions that previous works [11, 12] require.

$\langle \mathcal{A} := \{s, d, w\},$
$\mathcal{F} := \{s, d \to w\},$
$\mathcal{I} := \{\langle s \to [d],\ 40\%\rangle,$
$\langle \langle s, d\rangle \to w,\ 40\%\rangle,$
$\langle s \to [\langle d, w\rangle],\ 10\%\rangle,$
$\langle w \to [\langle s, d\rangle],\ 5\%\rangle\},$
$\mathcal{P} := 5\%\rangle$

(a) Graph use-case

$\langle \mathcal{A} := \{pid, ns, state, cpu\},$
$\mathcal{F} := \{ns, pid \to state, cpu\},$
$\mathcal{I} := \{\langle ns \to [pid],\ 45\%\rangle,$
$\langle \langle ns, pid\rangle \to cpu,\ 45\%\rangle,$
$\langle state \to [cpu],\ 5\%\rangle\},$
$\mathcal{P} := 5\%\rangle$

(b) Process Scheduler use-case

Figure 2. Relational specifications

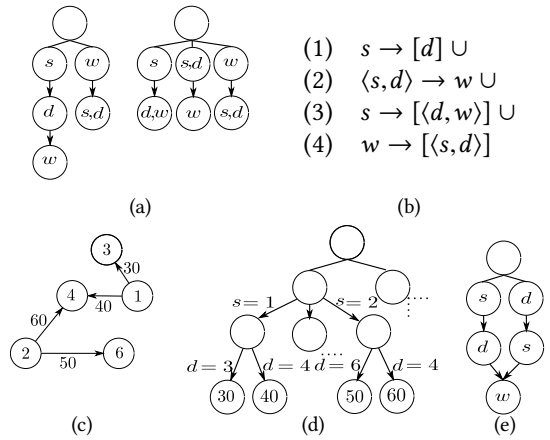
$$I \quad ::= \quad [A] \mid A \to A \mid A \to [A] \mid I \cup I \qquad \text{Interface}$$

$$\text{F-Att}\ \frac{}{\mathcal{F} \vdash [A]} \qquad \text{F-Map}\ \frac{A \to A' \in \text{closure}(\mathcal{F})}{\mathcal{F} \vdash A \to A'} \qquad \text{F-MMap}\ \frac{}{\mathcal{F} \vdash A \to [A']} \qquad \text{F-Uni}\ \frac{\mathcal{F} \vdash I_1 \qquad \mathcal{F} \vdash I_2}{\mathcal{F} \vdash I_1 \cup I_2}$$

Figure 3. $\mathcal{F} \vdash I$.

A user specification is a record $\langle \mathcal{A}, \mathcal{F}, \mathcal{I}, \mathcal{P}\rangle$. For instance, Figure 2.(a) shows the user specification of the directed graph use-case (adopted from [11]). The set $\mathcal{A}$ specifies the attributes A of the relation. In the example, the set of attributes $\mathcal{A}$ are $\{s, d, w\}$ for the source, destination and weight of edges between them. The set $\mathcal{F}$ specifies the functional dependencies between the attributes in $\mathcal{A}$. A functional dependency $\overline{A} \to \overline{A'}$ (where the overline notation denotes multiple attributes) states that every record of values for the attributes $\overline{A}$ in the relation is associated with a unique record of values for the attributes $\overline{A'}$. In the example, the set of functional dependency $\mathcal{F}$ is the single dependency $s, d \to w$ that states that given a source s and a destination d in the relation, there is a unique weight w associated with them.

The interface I represents the set of pairs of the type and the call ratio of the methods that access the relation. In the example, the interface $\mathcal{I}$ is the set of four access methods. The tuple of attributes A_1 to A_n is denoted by $\langle A_1, .., A_n\rangle$. We also use the notation $[A]$ to represent sets of values in contrast to a single value for the attribute A. The type $s \to [d]$ describes a method that given a source s returns a set of destinations d. In a map, this query returns all the cities that are directly reachable from the given city. The calls on this method is specified to be 40% of all calls on the relation. Call ratios can be obtained from legacy workloads or simple counting of calls in a typical run. The synthesizer accelerates experimenting with different ratios. The call ratios will help us quantify the performance of synthesis candidates. Similarly, the type $\langle s, d\rangle \to w$ describes a method that given a pair of source s and destination d, returns the weight w between them. The user may want to get all the routes from a city which is the list of all the destinations and the associated distances. This query is represented as $s \to [\langle d, w\rangle]$. The number $\mathcal{P}$ represents the call ratio of put operations. In the example, it is 5%. Given a tuple of values for the attributes $\mathcal{A}$, a put operation adds or updates the tuple in the relation. As another example, Figure 2.(b) shows the specification for the data structure of a process scheduler (adopted from [11]).



$$d \quad ::= \quad \bot \mid \langle \overline{A} \rangle \mid x \mid \langle \overline{A} \rangle \mapsto d \qquad \text{Decomposition}$$
$$\mid \quad \text{let } x := d \text{ in } d \mid d_1 \bowtie d_2$$

(f)

Figure 4. (a) Two decompositions for the graph use-case. (b) The supported interface. (c) A graph example. (d) A decomposition instance that represents the graph. (e) A sharing decomposition. (f) The decomposition grammar.

Not all the user-specified interfaces are well-formed. In particular, a method type $A \to A'$ is well-formed only if A' is functionally dependent on A. Otherwise, multiple values of A' might be associated with a value of A. Then, the appropriate method type is $A \to [A']$. We check that the user-specified interface complies with the functional dependencies. Figure 3 presents the checking rules. For brevity, we present the rules for a core interface language I with single attributes as the input and output. An interface is either (1) $[A]$ which returns a set of values of the attribute A, (2) $A \to A'$ which given a value of A returns a value of A', (3) $A \to [A']$ which given a value of A returns a set of values of A' or (4) $I \cup I'$ the union of a pair of interfaces. The inference rules derive judgments of the form $\mathcal{F} \vdash I$ which states that the interface I complies with the functional dependencies $\mathcal{F}$. Importantly, the rule F-Map checks that for every interface $A \to A'$, there is a functional dependency from A' to A in the closure of the given set of functional dependencies.

3 Decompositions

In this section, we show how relational specifications can be represented as map structures.

To process user method calls efficiently, we represent relations as map structures called decompositions. Given a relation and an interface on it, multiple decompositions can

serve the interface. For instance, Figure 4.(a) shows two decompositions for the graph use-case specified in Figure 2.(a). The left decomposition consists of two branches. The left branch is a map from sources to a map from destinations to weights. The right branch is a map from weights to sets of pairs of source and destination. Figure 4.(c) shows a graph and Figure 4.(d) shows the representation of the graph as an instance of the decomposition on the left of Figure 4.(a).

Both of the decompositions shown in Figure 4.(a) can serve the user-specified interface I that we restate in Figure 4.(b) as a union type. The left decomposition can serve the method (1) $s \to [d]$ in the left branch by a lookup in the first map and iterating the key set of the second map. It can similarly serve the method (2) $\langle s, d\rangle \to w$ by lookups in the two maps of the left branch. The methods (1) and (2) share the same branch. The method (3) $s \to [\langle d, w\rangle]$ is also served in the left branch by a lookup in the first map and then an iteration of the keys and the values of the second. Finally, the method (4) $w \to [\langle s, d\rangle]$ is simply served by the right branch. In contrast, in the right decomposition, the two methods (1) and (2) do not share the same branch and are served in the left and middle branches respectively.

A branch that serves multiple methods bears more contention. On the other hand, adding a tuple to a decomposition with more branches involves more updates. Given the call frequencies of the methods, which one of these two decompositions is more efficient? The answers to such questions are moot, platform-dependent and can be confirmed only by experiments. Is it possible to statically determine and synthesize the most efficient decomposition? In the following, we answer this question by decomposition and synchronization synthesis and training a performance model.

We presented decompositions graphically in Figure 4.(a). They can be equivalently captured as programs of the grammar d in Figure 4.(d). A decomposition d is either empty $\perp$, a tuple of attributes $\langle\overline{A}\rangle$, a variable x, a mapping from a tuple of attributes $\langle\overline{A}\rangle$ to a decomposition d, a let statement that binds the variable x to a decomposition in the definition of another decomposition, or the join $\bowtie$ of two decompositions. We note that a single attribute A is the special case of a unary tuple of attributes $\langle\overline{A}\rangle$. As an example, the left decomposition in Figure 4.(a) can be represented as the program $[s \mapsto (d \mapsto w)] \bowtie [w \mapsto \langle s, d\rangle]$. The let statement is particularly used to represent sharing of leaf attributes. For example, Figure 4.(e) shows a decomposition where the two branches share the weight values. This decomposition can be represented as the program $\mathsf{let}\ x := w\ \mathsf{in}\ [s \mapsto (d \mapsto x)] \bowtie [d \mapsto (s \mapsto x)]$.

4 Representation Synthesis

In this section, we present how decomposition candidates are synthesized for a given specification and show the soundness

$$
\begin{array}{lcll}
d & ::= & \perp \mid A \mid x \mid A \mapsto d & \text{Decomposition} \\
 & \mid & \mathsf{let}\ x := d\ \mathsf{in}\ d \mid d_1 \bowtie d_2 & \\
T & ::= & \mathsf{Unit} \mid A \mid A \to T \mid T_1 \cup T_2 & \text{Type} \\
I & ::= & [A] \mid A \to A \mid A \to [A] \mid I \cup I & \text{Interface}
\end{array}
$$

(a)

$$
\frac{}{\Gamma \vdash \perp : \mathsf{Unit}}\ \text{T-Unit} \qquad \frac{}{\Gamma \vdash A : A}\ \text{T-ID} \qquad \frac{\Gamma \vdash d : T}{\Gamma \vdash A \mapsto d : A \to T}\ \text{T-Map} \qquad \frac{x \in \mathsf{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)}\ \text{T-Var}
$$

$$
\frac{\Gamma \vdash d : T \qquad \Gamma[x \mapsto T] \vdash d' : T'}{\Gamma \vdash \mathsf{let}\ x := d\ \mathsf{in}\ d' : T'}\ \text{T-Let} \qquad \frac{\Gamma \vdash d_1 : T_1 \qquad \Gamma \vdash d_2 : T_2}{\Gamma \vdash d_1 \bowtie d_2 : T_1 \cup T_2}\ \text{T-Join}
$$

(b)

Figure 5. (a) Syntax. (b) Type System. $\Gamma \vdash d : T$.

of the synthesis. Then, we present synchronization synthesis for decompositions. We start with an example.

Consider the interface presented in Figure 4.(b). We incrementally build the left decomposition of Figure 4.(a) that supports the interface. In the beginning, the decomposition is empty. To serve the method $s \to [d]$, a map from s to (set of) d is added. The second method is $\langle s, d\rangle \to w$. A new branch can be added; however, the existing branch can be extended as well. Thus, a nested map from d to w is installed as the new value of the first map. We note that the support for the first method is preserved. It can be served by a lookup with s and an iteration on the key set d of the obtained map. The next method is $s \to [\langle d, w\rangle]$. It can be served by the existing map structure by a lookup followed by an iteration on the key and value. Thus, the decomposition stays unchanged with no branch. The final method is $w \to [\langle s, d\rangle]$. It is not supported by the current decomposition as there is no map with the key w. Thus, a new branch is added.

We first present a core language for decompositions, and a type system that assigns a union type to a decomposition. We then define entailment of an interface by a decompositions and its type. We finally define syntactic derivations to synthesize decompositions that entail an interface.

Type System. The core language of decompositions d is shown in Figure 5.(a). For brevity, this core language models the keys as single attributes. We saw decompositions d in § 3. A decomposition can be represented as a directed acyclic graph. The types T are defined in Figure 5.(a). A type is either the Unit type, an attribute A, a function type from an attribute A to another type T, or the union of two types. In contrast to decompositions, types are always trees.

Figure 5.(b) shows the type system with the judgement $\Gamma \vdash d : T$ which states that under the typing context Γ, the decomposition d has type T. The rules T-Unit and T-ID type the basic decompositions, empty and attribute respectively. The rule T-Map types a map decomposition as a map type. The rule T-Var types variables using the context. The rule T-Let types a let statement by first typing the bound variable

$$\frac{}{A \to T \vDash [A]}\ \text{I-Map-Key} \qquad \frac{}{A \to A' \vDash [A']}\ \text{I-Map-Val} \qquad \frac{T \vDash [A']}{A \to T \vDash [A']}\ \text{I-Map-Val}'$$

$$\frac{}{A \to A' \vDash A \to A'}\ \text{I-Map-Map} \qquad \frac{T \vDash [A']}{A \to T \vDash A \to [A']}\ \text{I-Map-Map}' \qquad \frac{T \vDash I_1 \qquad T \vDash I_2}{T \vDash I_1 \cup I_2}\ \text{I-Uni}$$

$$\frac{T_1 \vDash I}{T_1 \cup T_2 \vDash I}\ \text{I-UniL} \qquad \frac{T_2 \vDash I}{T_1 \cup T_2 \vDash I}\ \text{I-UniR} \qquad \frac{\emptyset \vdash d : T \qquad T \vDash I}{d \vDash I}\ \text{I-D}$$

Figure 6. Interface Entailment. $T \vDash I$ and $d \vDash I$.

and extending the typing context with the found typing to type the following decomposition. The rule T-Join types the join of two decompositions as a union type.

Interface Entailment. We now define whether a decomposition type T entails an interface I. The interface language I is presented in Figure 5.(a). We saw I in § 2.

Figure 6 presents the entailment inference rules. The judgment $T \vDash I$ states that the type T entails the interface I. The rule I-Map-Key states that the map type $A \to T$ entails the interface $[A]$ that returns a set of values of the attribute A. Intuitively, the interface is served by the key set of the map. The rule I-Map-Val states that the map type $A \to A'$ entails the interface $[A']$. Intuitively, the interface is served by the value set of the map. The rule I-Map-Val′ states that if the range type T of a map type $A \to T$ can serve the interface $[A']$, then the map type itself can serve the interface as well. The interface can be served by iterating the value set of the map and recursively calling the interface $[A']$ on the iterated values. The rule I-Map-Map states that the map type $A \to A'$ entails the interface $A \to A'$. Intuitively, the interface can be simply served by lookup in the map. The rule I-Map-Map′ states that assuming that the range type T of a map type $A \to T$ can serve the interface $[A']$, then the map type itself can serve the interface $A \to [A']$ that given a value of A, returns a set of values of A'. The interface can be served by a lookup with A in the map and then recursively calling the interface $[A']$ on the value. The rule I-Uni states that a type entails the union of two interfaces if it entails each. The rules I-UniL and I-UniR state that the union of two types entails an interface if either of the two entails the interface. The rules I-D states that a decomposition d entails an interface I, written as $d \vDash I$, if d is of type T and T entails I.

Decomposition Synthesis. Given an interface I, what are the decompositions d that entail I? Figure 7 presents derivation rules for the judgement $d, I \triangleright d'$ that states that given a decomposition d and interface I, the decomposition d can be transformed to d' which entails I. The rule S-ID states that if d already entails I, the transformation leaves d unchanged. The rule S-Uni states that the support for the union of two interfaces can be added for the two interfaces in sequence. The rules S-JoinL and S-JoinR state that either

$$\frac{}{d, I \triangleright d}\ \text{S-ID} \qquad \frac{d, I_1 \triangleright d' \qquad d', I_2 \triangleright d''}{d, (I_1 \cup I_2) \triangleright d''}\ \text{S-Uni} \qquad \frac{d_1, I \triangleright d_1'}{d_1 \bowtie d_2, I \triangleright d_1' \bowtie d_2}\ \text{S-JoinL}$$

$$\frac{d_2, I \triangleright d_2'}{d_1 \bowtie d_2, I \triangleright d_1 \bowtie d_2'}\ \text{S-JoinR} \qquad \frac{d \nvDash [A]}{d, [A] \triangleright d \bowtie (A \mapsto \bot)}\ \text{S-Add-1}$$

$$\frac{d \nvDash A \to A'}{(A \mapsto \bot), (A \to A') \triangleright (A \mapsto A')}\ \text{S-Ext-2} \qquad \frac{d \nvDash A \to A'}{d, (A \to A') \triangleright d \bowtie (A \mapsto A')}\ \text{S-Add-2}$$

$$\frac{d \nvDash A \to [A']}{(A \mapsto \bot), (A \to [A']) \triangleright (A \mapsto (A' \mapsto \bot))}\ \text{S-Ext-3} \qquad \frac{d \nvDash A \to [A']}{d, (A \to [A']) \triangleright d \bowtie (A \mapsto (A' \mapsto \bot))}\ \text{S-Add-3}$$

Figure 7. Synthesis. $d, I \triangleright d'$.

of the two sides of a join can be extended to support the interface. The rules S-Add-1, S-Add-2 and S-Add-3 state that if the given interface type is not supported, the output decomposition is the input decomposition joined with a map structure that corresponds to the interface. For example, the rule S-Add-3, supports the interface $A \to [A']$ by joining the map structure $(A \mapsto (A' \mapsto \bot))$. However, the existing map structures can sometimes be extended to support the interface. The rules S-Ext-2 and S-Ext-3 state that if part of the needed map structure is already in the input decomposition, the map structure is extended without affecting the already supported interfaces. The rule S-Ext-2 extends $A \mapsto \bot$ to $A \mapsto A'$ and the rule S-Ext-3 extends $A \mapsto \bot$ to $A \mapsto (A' \mapsto \bot)$.

The following theorem states the soundness of synthesis. Every decomposition synthesized for an interface provides that interface. More precisely, given an interface I, if a derivation transforms the empty decomposition $\bot$ to a decomposition d, then d entails I.

Theorem 4.1 (Soundness). $\forall I, d.\ (\bot, I \triangleright d) \to (d \vDash I)$

The proof of the theorem is available in the appendix.

As we saw in Figure 4.(a), an interface can be supported by multiple decompositions. The non-determinism of the inference rules can derive different decompositions. In particular, the methods in the interface can be reordered before being passed to the rule S-Uni and the Add and Ext rules can either add or extend map structures. We use the enumerative synthesis technique [30] to generate the decompositions. In addition, if two branches of a decomposition have the same tuple of attributes as the leaf node, the two branches can share the leaf. Figure 4.(e) shows an example where the two branches share the weight attribute w.

Synchronization Synthesis. Multiple threads can concurrently access the synthesized data structures. To maintain the consistency and availability of the structures, we synthesize adequate synchronization to ensure the linearizability

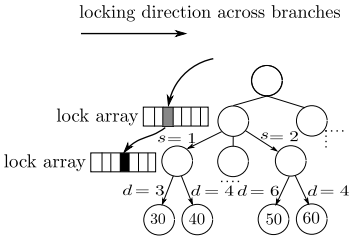


Figure 8. For put operations, locks are acquired in shared mode (grey) until a missing key is reached and it is locked in exclusive mode (black).

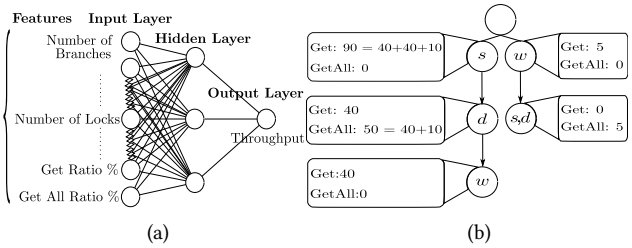


Figure 9. (a) Multi-Layer Perceptron as the Performance Model. (b) Get and GetAll features for Figure 2.(a) and 4.(a)

and deadlock-freedom of the calls. We briefly present the synthesized synchronization. As Figure 8 shows, we associate an array of read-write locks with each hash-map. Hashing associates each lock to a separate set of buckets. The protocol follows two-phase locking; thus, it maintains linearizability. To prevent deadlocks, locks are acquired in a total order from top to bottom in the tree and from left to right in the arrays.

Get operations. In the traversal down the tree, before each lookup, we access the the lock array of the map and lock (in shared mode) the lock at the index corresponding to the hash of the key. Before iterating a key set, we acquire all the locks of the corresponding lock array in shared mode.

Put operations. To maintain the consistency of the replicated data across the branches of a decomposition, we do not release the acquired locks of a branch until the insertion is completed on all branches. We traverse branches in sequence from left to right. We acquire all the locks in the shared mode during the traversal until we reach a missing key. Then the lock for that key is reacquired in the exclusive mode. Then, no other lock is needed to be acquired in the current branch. Doing so prevents other readers as well as writers from accessing buckets that are being updated.

5 Learning

This section presents the training of a performance model that predicts the throughput of candidate representations.

Multi-Layer Perceptron. We build a multi-layer perceptron [22, 32] as shown in Figure 9.(a) which is an artificial neural network with multiple layers of neurons. The first and last layers are the input and output neurons respectively and the hidden layers come in between. The input layer has a neuron per input feature and the output layer has a neuron per output. The output of each input and hidden neuron is an input to every neuron of the next layer. A weight w_{ji} is associated with the input from a neuron j to another neuron i. Given values for the input features, the network calculates output values of neurons layer by layer from the input layer forward to the output layer. Input neurons simply output their input features. The output o_i of each hidden and output neuron i is calculated as the weighted sum of the previous layer outputs o_j and then applying a sigmoid function Φ $(o_i = \Phi\left(\Sigma_{j=1}^{n} w_{ji} \times o_j\right))$

A datapoint is a pair of feature values and the desired output values. A training set is a set of datapoints. Given a datapoint, the difference between the output that the network calculates and the desired output is the error. Given a training set, the goal is to learn the weights of the network that minimize the error E across the training set. Multilayer perceptron is trained by a supervised learning technique called back-propagation that uses the gradient descent optimization algorithm. Gradient descent moves towards the minimum of a function by iteratively shifting the input point in the opposite direction of the derivative. Back-propagation calculates the derivative of the error over the neuron outputs and connection weights layer by layer from the output layer backwards to the input layer. Given the derivative of the error, $\frac{\partial E}{\partial w_{ji}}$ for a weight w_{ji}, the term $\Delta(wji) = -\alpha \times \frac{\partial E}{\partial w_{ji}} + \mu \times \Delta'(wji)$ is the update applied to w_{ji} where $\Delta'(wji)$ is the update to w_{ji} in the previous iteration, α is the learning rate and μ is the momentum.

We repeat the training over the training set by the k-fold cross-validation technique to get an out-of-sample estimate of the model. We do 10-fold cross validation. This allows us to avoid overfitting. Using this technique, we split the test data set into 10 equal subsets. In each iteration of the learning, we train on one subset and use the other 9 subsets to test the model. In the end, we get the average of the 10 trained models to compute the final model. We started with small instances of networks and increased the complexity when needed. Our network has 53 input neurons, one hidden layer with 6 neurons and an output neuron for the throughput.

Feature Selection. Selection of features that can predict the value of the output is crucial to the success of learning. For a pair of a decomposition and an interface (including the call frequencies), we choose features that are correlated to the throughput of a workload with those call frequencies on the decomposition. The features capture the decomposition structure including the number of its branches, the number of locks on a node, the cumulative ratio of operations (lookup on the map of a node, iteration of the map of a node), the put ratio, and whether the branches share leaves. We use a bounded number of nodes and uniquely number them

B*	S	A	MT	ST	SP	AT	AP	SU	AC
Graph	7	3	4	14	5	14323	5	1023	100%
PS	6	4	3	11	5.2	9621	5.7	874	91%
FS	6	4	4	13	5.5	12975	5.5	998	100%

* B: Benchmark name; (PS: Process Scheduler, FS: File System) S: Specification lines of code; A: Number of attributes; MT: Number of methods; ST: Synthesis time (second); SP: Synthesized representation throughput (million ops / sec); AT: Auto-tuner time (second); AP: Auto-tuned representation throughput (million ops / sec); SU: Speed-up = AT / ST; AC: Accuracy = SP / AP. Workload: 99% Query, 1% Update. Platform: P_1.

Table 1. Quantitative Synthesis vs. Auto-tuning.

according to their position in branches. If a decomposition does not have a node at a position, the values of features for that node are simply set to zero.

We have separate features for lookup and iteration as they exhibit different performance characteristics because the former acquires a single lock and the latter acquires all the locks in the array. As Figure 9.(b) shows, we have two features Get and GetAll for each node that represent the cumulative ratio of lookup and iteration. Figure 9.(b) shows the values of these features for the specification of Figure 2.(a). For example, the method call $\langle s \rightarrow [d],\ 40\% \rangle$ executes a lookup on the node s and an iteration on the node d. Therefore, it adds 40% to the Get feature of node s and 40% to the GetAll feature of node d. We sum the lookup and iteration ratios by all methods at each node to calculate the Get and GetAll features of that node.

We tried decision trees as well with no better results. It remains open to study the effectiveness of random forests and graph convolutional neural networks on this domain.

6 Experimental Results

Implementation. We developed a tool called LEQSY that synthesizes concurrent data representations. LEQSY is implemented in Scala [19] and Java. Its input specification language was described in § 2. It synthesizes decompositions by enumerating over the derivation choices of the synthesis rules presented in § 4. It includes synchronization templates for decomposition structures that implement the protocols presented in § 4 and synthesizes synchronization by instantiating the templates. It uses Weka libraries [31] to train the performance model and generates Java source code classes.

Platform Setup. We performed our experiments on two platforms P_1 and P_2. The platform P_1 is a quad core 3.60GHz Intel® Core™ i7-7700 CPUs(8) and 16Gb memory with ubuntu 16.04 LTS. The platform P_2 has 2 AMD Opteron 6272 CPUs with a total of 8 cores with 64GB ECC protected memory of RAM with CentOS 7.4 Linux x86_64 V 3.10.0 All benchmarks were run on an OpenJDK V-1.8.0_171 64bit Server VM mode, with a 12Gb initial and maximum heap size.

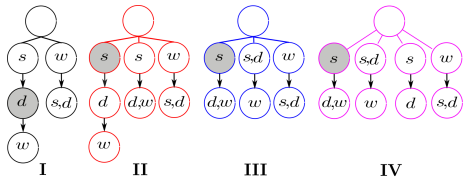
Learning. To prepare the training dataset, we generated decompositions up to width and depth of 4. We generated synchronization with permutations of 16, 32, 64, 128, 256 and 512 for lock array sizes. We generated different interfaces with different call ratios. These representations and interfaces are independent of any concrete benchmark. For each decomposition and interface, we executed a workload that corresponds to the call ratios on the decomposition 5 times and captured the throughput (that is the number of processed operations per second). We used the gathered datapoints to train a multi-layer perceptron by 10-fold cross-validation with 500 epochs, the learning rate of 0.3, and the momentum of 0.2. The training for P_1 with 30000 data-points took 240 minutes and for P_2 with 2000 data-points took 28 minutes.

Benchmarks. We run our experiments on three benchmarks: graph (Figure 2.(a)), process scheduler (Figure 2.(b)) and file system that we have adopted from previous work [11, 12]. We present detailed results for the Graph benchmark in the main body. In the interest of space, the results for the other benchmarks are available in the appendix.

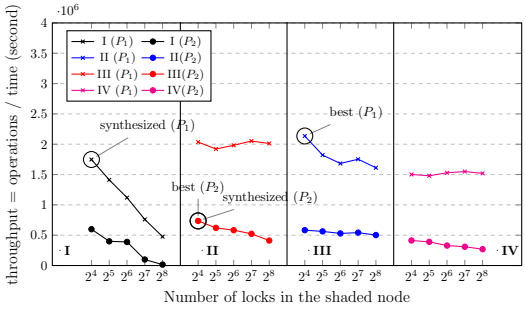
Measurements. We adopted the road network of the Northwestern USA graph from RelC [11]. We initialized each file system and process scheduler with 10000 and 15000 random tuples respectively. For each specification, we generated a random workload of 10^6 operations per thread that match the call frequencies of the specification and execute candidate representations with that workload. We measured the throughput that is the number of processed operations per second. We repeated each experiment 8 times within the same process with 8 threads, with a garbage collection in between. We discarded the results of the first 3 runs to warm up the JIT compiler. Each reported value is the average of the last 5 runs. The auto-tuner executes all candidates with the same sample workload, measures their throughputs and picks the candidate with the highest throughput.

Assessment. We first measure the speed-up and the accuracy of synthesis compared to the auto-tuning baseline. We then consider the accuracy of the predicted throughput versus the actual throughput for the synthesis candidates. Finally, we analyze the actual throughput for various candidates and the synthesized representation.

Table 1 shows the speed-up and accuracy of static quantitative synthesis versus dynamic auto-tuning. For each benchmark, in addition to information about the specification, it presents the time to produce the output representation and its throughput for both techniques. It reports the speed-up (SU) gained by the quantitative synthesis that is the processing time of auto-tuning (AT) divided by the processing time of quantitative synthesis (ST), and the accuracy (AC) of quantitative synthesis that is the throughput of its output (SP) divided by the throughput of auto-tuning output (AT). Quantitative synthesis uses the trained model to predict throughput and avoids execution of the candidates. It achieves more than two orders of magnitude speed-up and



(a) Decomposition candidates for the interface of Figure 4.(b)



(b) Benchmark: Graph. Workload: 90% Query, 10% Update.

Figure 10. Actual throughput for candidate representations and the synthesized representation. The optimum representation for platform P_1 is III with 16 locks. The trained model picks I with 16 locks whose throughput is 83% of the optimum. For the platform P_2, the two are the same.

more than 90% accuracy across the benchmarks. For two of the benchmarks, the synthesized representation is the most efficient candidate. With larger benchmarks, the runtime increases but the static prediction time stays the same. Therefore, for larger benchmarks, the speed-up is expected to be even higher. The synthesizer supports quick reconfiguration from a representation to another for varying workloads or new user interfaces during system maintenance.

Let us consider the average error of the predicted throughput for all the representations that were considered in Table 1 with respect to their actual throughput. The average error is 17%, 14%, and 21%, for the Graph, File System, and Process scheduler benchmarks respectively. A near-optimum representation can be synthesized even if the network predicts throughput with an error but keeps the relative order of representations. The relatively low prediction error suggests that the selected features are correlated with the throughput and the training has been able to learn the correlation.

Now, we closely look at the throughput of candidate representations for the Graph benchmark and compare the throughputs of the synthesized representation and the most efficient candidate. We have already reported the overall results for this use-case in Table 1. The purpose of this experiment is illustration and comparison of different decomposition and synchronization choices. Given the specification, the synthesis process results in the four decompositions I, II, III, and IV presented in Figure 10.(a). We increase the size of the lock array for the nodes that are shaded in Figure 10.(a) from 16 to 256. The size of the lock array for the other nodes is constant, 256 in Figure 10.(b).

For the decomposition I, we increase the number of locks at the shaded map d. The two methods $s \rightarrow [d]$ and $s \rightarrow [\langle d, w\rangle]$ iterate the map d. An iteration acquires all the locks of the array. As the left-most part in Figure 10.(b) shows, increasing the number of locks aggravates the throughput. The decomposition II reduces iterations. It serves the method $s \rightarrow [\langle d, w\rangle]$ by a lookup on the second branch; thus, in contrast to the decomposition I, only the method $s \rightarrow [d]$ performs an iteration in the first branch. The decomposition III reduces iteration further. It serves the above methods by lookups on the first branch. Yet, the two methods $s \rightarrow [\langle d, w\rangle]$ and $s \rightarrow [d]$ share a branch. The decomposition IV introduces a separate branch for the latter to reduce contention.

Therefore, on an experiment with a query-dominated workload (with 1% updates presented in the appendix), we see a trend of throughput increase for the decompositions from I to IV. In contrast, in Figure 10.(b) (the workload with more update operations), we see a trend of throughput decrease from the decomposition I to IV. In decompositions with more branches, put operations have to acquire more locks in exclusive mode and perform more mutations. Thus, the decomposition I with two branches outperforms the others. These observations suggest that wider decompositions deliver higher performance for query-dominated workloads and narrower decompositions deliver higher performance for update-dominated workloads. In general, increasing the number of locks is believed to increase the throughput; however, there are exceptions that do not match the intuition. These observations further highlight the unpredictable nature of performance and the importance of learning.

7 Related Work

Synthesizing data structures. The importance of data structure synthesis has been recognized since 70s [6, 7, 23, 25]. RelC [11, 12] synthesizes data representations for given decompositions based on auto-tuning. Cozy and its follow-up work [17, 18] synthesizes efficient sequential data structures using a static cost model. Sketching [26, 27], Boosting [13], Semantic Locking [10], Predication [2], and Transactional Libraries [28] synthesize and compose concurrent data structures. Users can use these techniques to compose atomic structures manually. In contrast to these works, LEQSY supports more general relational specifications, learns the performance model and synthesizes concurrent data structures.

Learning and concurrent data structures. Machine learning has been used to predict the number of concurrent threads for optimum performance [20, 21, 29]. Smartlocks [9] use learned models to adapt spin-locks and Smart Data Structures [8] use online learning to adapt data structures to varying workloads. Data Calculator [15] synthesizes read-only data structures and predicts performance based on

user-defined layout specifications and target architectures. In contrast to the above, LEQSY does not require layouts, learns the performance model, and synthesizes the most efficient data structure for a target workload.

8 Conclusion

We presented a new approach to quantitative synthesis that trains a performance model to predict the efficiency of concurrent representations for relational specifications. The adaptability and platform-independence of learning the performance model can carry over to other synthesis domains.

References

[1] Roderick Bloem, Krishnendu Chatterjee, Thomas A Henzinger, and Barbara Jobstmann. 2009. Better quality in synthesis through quantitative objectives. In *International Conference on Computer Aided Verification*. Springer, 140–156.

[2] Nathan G Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. Transactional predication: high-performance concurrent sets and maps for stm. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*. ACM, 6–15.

[3] Pavol Černỳ, Krishnendu Chatterjee, Thomas A Henzinger, Arjun Radhakrishna, and Rohit Singh. 2011. Quantitative synthesis for concurrent programs. In *International Conference on Computer Aided Verification*. Springer, 243–259.

[4] Krishnendu Chatterjee, Thomas A Henzinger, Barbara Jobstmann, and Rohit Singh. 2010. Measuring and synthesizing systems in probabilistic environments. In *International Conference on Computer Aided Verification*. Springer, 380–395.

[5] Swarat Chaudhuri, Martin Clochard, and Armando Solar-Lezama. 2014. Bridging boolean and quantitative synthesis using smoothed proof search. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 207–220.

[6] Donald Cohen and Neil Campbell. 1993. Automating relational operations on data structures. *IEEE Software* 10, 3 (1993), 53–60.

[7] Jay Earley. 1975. High level iterators and a method for automatically designing data structure representation. *Computer Languages* 1, 4 (1975), 321–342.

[8] Jonathan Eastep, David Wingate, and Anant Agarwal. 2011. Smart data structures: an online machine learning approach to multicore data structures. In *Proceedings of the 8th ACM international conference on Autonomic computing*. ACM, 11–20.

[9] Jonathan Eastep, David Wingate, Marco D Santambrogio, and Anant Agarwal. 2010. Smartlocks: lock acquisition scheduling for self-aware synchronization. In *Proceedings of the 7th international conference on Autonomic computing*. ACM, 215–224.

[10] Guy Golan-Gueta, G Ramalingam, Mooly Sagiv, and Eran Yahav. 2015. Automatic scalable atomicity via semantic locking. *ACM SIGPLAN Notices* 50, 8 (2015), 31–41.

[11] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. 2011. Data Representation Synthesis. *SIGPLAN Not.* 46, 6 (June 2011), 38–49. https://doi.org/10.1145/1993316.1993504

[12] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. 2012. Concurrent Data Representation Synthesis. *SIGPLAN Not.* 47, 6 (June 2012), 417–428. https://doi.org/10.1145/2345156.2254114

[13] Maurice Herlihy and Eric Koskinen. 2008. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. ACM, 207–216.

[14] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492. https://doi.org/10.1145/78969.78972

[15] Stratos Idreos, Kostas Zoumpatianos, Brian Hentschel, Michael S. Kester, and Demi Guo. 2018. The Data Calculator: Data Structure Design and Cost Synthesis from First Principles and Learned Cost Models. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. ACM, New York, NY, USA, 535–550. https://doi.org/10.1145/3183713.3199671

[16] Mohsen Lesani, Todd Millstein, and Jens Palsberg. 2014. Automatic Atomicity Verification for Clients of Concurrent Data Structures. In *International Conference on Computer Aided Verification*. Springer, 550–567.

[17] Calvin Loncaric, Michael D Ernst, and Emina Torlak. 2018. Generalized data structure synthesis. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 958–968.

[18] Calvin Loncaric, Emina Torlak, and Michael D Ernst. 2016. Fast synthesis of fast collections. *ACM SIGPLAN Notices* 51, 6 (2016), 355–368.

[19] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. 2004. *An overview of the Scala programming language*. Technical Report.

[20] Diego Rughetti, Pierangelo Di Sanzo, Bruno Ciciani, and Francesco Quaglia. 2012. Machine Learning-Based Self-Adjusting Concurrency in Software Transactional Memory Systems. In *Proceedings of the 2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '12)*. IEEE Computer Society, Washington, DC, USA, 278–285. https://doi.org/10.1109/MASCOTS.2012.40

[21] Diego Rughetti, Pierangelo Di Sanzo, Alessandro Pellegrini, Bruno Ciciani, and Francesco Quaglia. 2015. *Tuning the Level of Concurrency in Software Transactional Memory: An Overview of Recent Analytical, Machine Learning and Mixed Approaches*. Springer International Publishing, Cham, 395–417. https://doi.org/10.1007/978-3-319-14720-8_18

[22] David E Rumelhart, Geoffrey E Hinton, James L McClelland, et al. 1986. A general framework for parallel distributed processing. *Parallel distributed processing: Explorations in the microstructure of cognition* 1, 45-76 (1986), 26.

[23] Edmond Schonberg, Jacob T Schwartz, and Micha Sharir. 1979. Automatic data structure selection in SETL. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 197–210.

[24] Ohad Shacham, Nathan Bronson, Alex Aiken, Mooly Sagiv, Martin Vechev, and Eran Yahav. 2011. Testing atomicity of composed concurrent operations. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 51–64.

[25] Yannis Smaragdakis and Don Batory. 1997. DiSTiL: A Transformation Library for Data Structures. In *Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997 (DSL'97)*. USENIX Association, Berkeley, CA, USA, 20–20. http://dl.acm.org/citation.cfm?id=1267950.1267970

[26] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. 2007. Sketching stencils. In *ACM SIGPLAN Notices*, Vol. 42. ACM, 167–178.

[27] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. 2008. Sketching concurrent data structures. In *ACM SIGPLAN Notices*, Vol. 43. ACM, 136–148.

[28] Alexander Spiegelman, Guy Golan-Gueta, and Idit Keidar. 2016. Transactional data structure libraries. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 682–696.

[29] Omer Tripp and Noam Rinetzky. 2013. Tightfit: Adaptive parallelization with foresight. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 169–179.

[30] Abhishek Udupa, Arun Raghavan, Jyotirmoy V Deshmukh, Sela Mador-Haim, Milo MK Martin, and Rajeev Alur. 2013. TRANSIT: specifying protocols with concolic snippets. *ACM SIGPLAN Notices* 48, 6 (2013), 287–296.

[31] Web. 2018. Weka 3: Data Mining Software in Java. https://www.cs.waikato.ac.nz/ml/weka/. (2018). Accessed: 2018-08-01.

[32] Paul Werbos. 1974. Beyond Regression:" New Tools for Prediction and Analysis in the Behavioral Sciences. *Ph. D. dissertation, Harvard University* (1974).