Learning Quantitative Representation Synthesis

Mayur Patil
University of California, Riverside
mpati005@ucr.edu

Farzin Houshmand
University of California, Riverside
fhous001@ucr.edu

Mohsen Lesani
University of California, Riverside
lesani@cs.ucr.edu

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.

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) [30], deadlock-free and efficient. However, applications often require specialized data structures that are not immediately provided by standard libraries.

MAPL '20, June 15, 2020, London, UK
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7996-0/20/06…$15.00
https://doi.org/10.1145/3394450.3397467

Programmers usually choose and 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. Previous research [35, 49] shows that programmers often fail to atomically compose the standard collection libraries. More importantly, system requirements evolve. Modifying the connections and the protecting synchronization of data structures can easily 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. Programmers no longer prematurely commit to a particular representation; thus, 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 [27]. This variance highlights the importance of the flexibility that synthesis offers to easily switch between representations. Previous works [26, 27, 37] 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 [5, 9, 14, 15] aims to synthesize programs that are not only correct but optimum in terms of a quantitative metric. Previous works used static cost functions to estimate the cost of locking and context switching [9] and the performance of candidate representations [36]. 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.

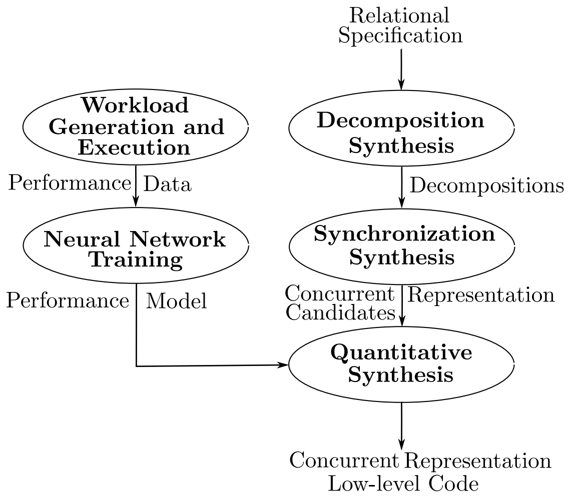


Figure 1. Overview of Leqsy

For instance, previous work [9] reported that coarse-grained out-performed 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 performance model to synthesize efficient concurrent representations. The representations include both the concrete structures and the protecting synchronization.

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 we 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 [57] candidate decompositions.

The synthesized data representations may be accessed from multiple threads concurrently. To preserve the consistency of data, we want the representations to be linearizable and equip them with synchronization. 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 for get and put operations on decompositions. The query and update planning guarantees linearizability and deadlock-freedom.

To choose the most efficient candidate representation, we need a performance model that can estimate the performance of candidates. We train a multi-layer perceptron [47, 62] artificial neural network to learn 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 these features are the inputs and the resulting performance is the output to train the neural network. After training iterations, the neural network learns the performance model. Given values for the features, it can predict the performance. We use the learned model to compare the performance of candidate representations.

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. Leqsy generates training workloads, executes them and gathers training performance data. It then trains a neural network to learn a performance model of the underlying platform. Training is a pre-process that needs to be done only once for a platform. Given a relational specification, Leqsy automatically synthesizes decompositions. It subsequently synthesizes synchronization to generate candidate concurrent representations. It then uses the learned performance model to quantitatively choose the most efficient candidate. 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 [26, 27] use-cases. The results show that Leqsy can successfully synthesize a concurrent data 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 frequency of method calls in addition to the relation and its functional dependencies. A checker that ensures that the interface is well-formed with respect to the 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. (§ 3 and § 4)

$$
\begin{aligned}
\langle \mathcal{A} &:= \{s, d, w\}, \\
\mathcal{F} &:= \{s, d \rightarrow w\}, \\
\mathcal{I} &:= \{\langle s \rightarrow [d],\ 40\%\rangle, \\
&\quad \langle\langle s, d\rangle \rightarrow w,\ 40\%\rangle, \\
&\quad \langle s \rightarrow [\langle d, w\rangle],\ 10\%\rangle, \\
&\quad \langle w \rightarrow [\langle s, d\rangle],\ 5\%\rangle\}, \\
\mathcal{P} &:= 5\%\rangle
\end{aligned}
$$

(a)

$$
\begin{aligned}
\langle \mathcal{A} &:= \{pid, ns, state, cpu\}, \\
\mathcal{F} &:= \{ns, pid \rightarrow state, cpu\}, \\
\mathcal{I} &:= \{\langle ns \rightarrow [pid],\ 45\%\rangle, \\
&\quad \langle\langle ns, pid\rangle \rightarrow cpu,\ 45\%\rangle, \\
&\quad \langle state \rightarrow [cpu],\ 5\%\rangle\}, \\
\mathcal{P} &:= 5\%\rangle
\end{aligned}
$$

(b)

$$
\begin{aligned}
\langle \mathcal{A} &:= \{parent, name, child\}, \\
\mathcal{F} &:= \{parent, name \rightarrow child\}, \\
\mathcal{I} &:= \{\langle parent \rightarrow [name],\ 40\%\rangle, \\
&\quad \langle\langle parent, name\rangle \rightarrow child,\ 40\%\rangle, \\
&\quad \langle child \rightarrow [\langle parent, name\rangle],\ 15\%\rangle, \}, \\
\mathcal{P} &:= 5\%\rangle
\end{aligned}
$$

(c)

Figure 2. Relational specifications of (a) the Graph use-case, (b) the Process Scheduler use-case, (c) the File System use-case

- 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. (§ 6)
- A synthesis tool called Leqsy that given user specifications generates Java source code of concurrent representations. (§ 5 and § 7)
- Experimental evaluation of the approach that showcases its effectiveness to synthesize near-optimum representations in the space of map structures. (§ 7)

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, a set of functional dependencies between the attributes and an access interface.

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 [26]). 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

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

$$
\textsc{F-Att}\ \frac{}{\mathcal{F} \vdash [A]} \qquad \textsc{F-Map}\ \frac{A \rightarrow A' \in \text{closure}(\mathcal{F})}{\mathcal{F} \vdash A \rightarrow A'} \qquad \textsc{F-MMap}\ \frac{}{\mathcal{F} \vdash A \rightarrow [A']}
$$

$$
\textsc{F-Uni}\ \frac{\mathcal{F} \vdash I_1 \qquad \mathcal{F} \vdash I_2}{\mathcal{F} \vdash I_1 \cup I_2}
$$

Figure 3. Compliance of the interface with functional dependencies. $\mathcal{F} \vdash I$.

$\overline{A} \rightarrow \overline{A'}$ (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 $s, d \rightarrow w$ 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. In our example, $[d]$ denotes multiple destinations. The type $s \rightarrow [d]$ 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. For example, finding the destinations of a source may occur more frequently than getting the source and destination pairs for a particular weight. The synthesizer accelerates experimenting with different ratios. The call ratios will help us quantify the performance of synthesis candidates. Similarly, the type $\langle s, d\rangle \rightarrow w$ 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 $s \rightarrow [\langle d, w\rangle]$. The user may also want to find the cities that are apart by a particular distance which is to get the source and destination pairs with a given weight. This query is represented as $w \rightarrow [\langle s, d\rangle]$. The number $\mathcal{P}$ represents the call ratio of put operations. In the example, the put ratio is 5%. Given a tuple of values for the attributes $\mathcal{A}$, a put operation adds or updates the tuple in the relation.

Figure 2.(b) shows the specification for the data structure used by a process scheduler (adopted from [26]). The relation represents processes. The set of attributes are the process identifier *pid*, its namespace *ns*, its state *state* (that is running or idle) and its assigned processor *cpu*. The functional dependency *ns*, *pid* → *state*, *cpu* states that given a namespace and a process identifier, there are unique values for the state and the processor. For example, the query *state* → [*cpu*] returns all the CPUs that are assigned any process with the given state.

As another example, Figure 2.(c) presents the File System benchmark adopted from [26]. The set of attributes $\mathcal{A}$ is $\{parent, name, child\}$. Each *parent* directory entry has zero or more *child* directory entries, each with a distinct file *name*. The set of functional dependency $\mathcal{F}$ is the single dependency *parent*, *name* → *child*. The interface $\mathcal{I}$ is the set of three access methods. The first one gets all the names in a directory. The second one gets a file given its parent directory and name. The third one gets the access paths of a file.

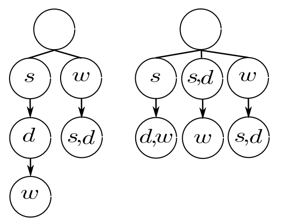
Not all the user-specified interfaces are well-formed. In particular, a method type $A \rightarrow A'$ is well-formed only if A' is functionally dependent on A. Otherwise, multiple values of A' might be associated with a value of A. Then, the appropriate method type is $A \rightarrow [A']$. We check 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) $A \rightarrow A'$ which given a value of A returns a value of A', (3) $A \rightarrow [A']$ 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 $A \rightarrow A'$, 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.

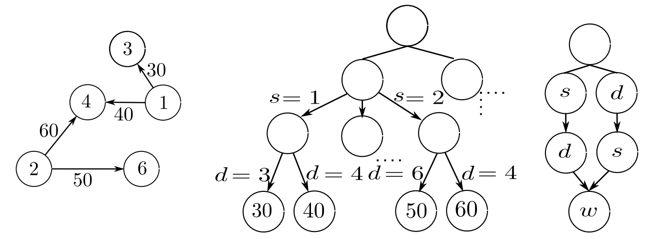
To process user method calls efficiently, we represent relations as map structures called decompositions. Given a relation and an interface on it, multiple decompositions can serve the interface. For instance, Figure 4.(a) shows two decompositions for the graph use-case specified in Figure 2.(a). The 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) $s \rightarrow [d]$ in the left branch by a lookup in the first map and iterating the key set of the second map. It can similarly



(1) $s \rightarrow [d] \cup$
(2) $\langle s, d\rangle \rightarrow w \cup$
(3) $s \rightarrow [\langle d, w\rangle] \cup$
(4) $w \rightarrow [\langle s, d\rangle]$

(a) (b)



(c) (d) (e)

$$
\begin{array}{rcll}
d & ::= & \perp \mid \langle \overline{A} \rangle \mid x \mid \langle \overline{A} \rangle \mapsto d & \text{Decomposition} \\
 & \mid & \text{let } x := d \text{ in } d \mid d_1 \bowtie d_2 &
\end{array}
$$

(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.

serve the method (2) $\langle s, d\rangle \rightarrow w$ by lookups in the two maps of the left branch. The methods (1) and (2) share the same branch. The method (3) $s \rightarrow [\langle d, w\rangle]$ 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) $w \rightarrow [\langle s, d\rangle]$ 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 frequency distribution 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. We will consider synchronization choices for the decompositions in § 5. These choices only make the decision harder by extending the possible candidates. Is it possible to statically determine and synthesize the most efficient decomposition? In the following sections, 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 as shown in § 4. A decomposition d is either empty $\perp$, a tuple of attributes $\langle \overline{A} \rangle$, a variable x, a mapping from

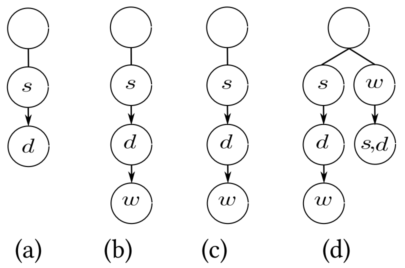


Figure 5. Decomposition synthesis for Figure 4.(b).

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 $\text{let } x := w \text{ in } [s \mapsto (d \mapsto x)] \bowtie [d \mapsto (s \mapsto x)]$.

4 Decomposition Synthesis

In this section, we present how decomposition candidates are synthesized for a given specification and show the soundness of the synthesis. We start with an example.

Consider the interface presented in Figure 4.(b). In Figure 5, 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 in Figure 5.(a). 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. In Figure 5.(b), 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 in Figure 5.(c). 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 in Figure 5.(d).

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 6.(a). For brevity, this core language models

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

(a)

$$
\textsc{T-Unit}\ \frac{}{\Gamma \vdash \bot \colon \text{Unit}}
\qquad
\textsc{T-ID}\ \frac{}{\Gamma \vdash A \colon A}
\qquad
\textsc{T-Map}\ \frac{\Gamma \vdash d \colon T}{\Gamma \vdash A \mapsto d \colon A \to T}
$$

$$
\textsc{T-Var}\ \frac{x \in \text{dom}(\Gamma)}{\Gamma \vdash x \colon \Gamma(x)}
\qquad
\textsc{T-Let}\ \frac{\Gamma \vdash d \colon T \qquad \Gamma[x \mapsto T] \vdash d' \colon T'}{\Gamma \vdash \text{let } x := d \text{ in } d' \colon T'}
$$

$$
\textsc{T-Join}\ \frac{\Gamma \vdash d_1 \colon T_1 \qquad \Gamma \vdash d_2 \colon T_2}{\Gamma \vdash d_1 \bowtie d_2 \colon T_1 \cup T_2}
$$

(b)

Figure 6. (a) Syntax. (b) Type System. $\Gamma \vdash d \colon T$.

the keys as single attributes. As we saw in § 3, a decomposition d is either empty $\bot$, an attribute A, a variable x, a mapping from an attribute A 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. A decomposition can be represented as a directed acyclic graph. The types T are defined in Figure 6.(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 6.(b) shows the type system with the judgement $\Gamma \vdash d \colon T$ which states that under the typing context Γ, the decomposition d has type T. The typing context Γ is a mapping from variables to types. 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 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 6.(a). (We saw I in § 2.) The interface $[A]$ returns a set of values of the attribute A. Given a value of A, the two interfaces $A \to A'$ and $A \to [A']$ return a value and a set of values of A' respectively. An interface can be the union of two interfaces.

Figure 7 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

$$\textsc{I-Map-Key}\quad \frac{}{A \rightarrow T \vDash [A]} \qquad \textsc{I-Map-Val}\quad \frac{}{A \rightarrow A' \vDash [A']} \qquad \textsc{I-Map-Val}'\quad \frac{T \vDash [A']}{A \rightarrow T \vDash [A']}$$

$$\textsc{I-Map-Map}\quad \frac{}{A \rightarrow A' \vDash A \rightarrow A'} \qquad \textsc{I-Map-Map}'\quad \frac{T \vDash [A']}{A \rightarrow T \vDash A \rightarrow [A']}$$

$$\textsc{I-Uni}\quad \frac{T \vDash I_1 \qquad T \vDash I_2}{T \vDash I_1 \cup I_2}$$

$$\textsc{I-UniL}\quad \frac{T_1 \vDash I}{T_1 \cup T_2 \vDash I} \qquad \textsc{I-UniR}\quad \frac{T_2 \vDash I}{T_1 \cup T_2 \vDash I}$$

$$\textsc{I-D}\quad \frac{\emptyset \vdash d \colon T \qquad T \vDash I}{d \vDash I}$$

Figure 7. Interface Entailment. $T \vDash I$ and $d \vDash I$.

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 $A \rightarrow A'$ entails the interface $[A']$ that returns a set of values of the attribute 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 $A \rightarrow T$ can serve the interface $[A']$, then the map type itself can serve the interface as well. Intuitively, 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 $A \rightarrow A'$ entails the interface $A \rightarrow A'$ that given a value of A, returns a value of 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 $A \rightarrow T$ can serve the interface $[A']$, then the map type itself can serve the interface $A \rightarrow [A']$ that given a value of A, returns a set of values of A'. Intuitively, 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.

Synthesis. Given an interface I, what are the decompositions d that entail I? Figure 8 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

$$\textsc{S-ID}\quad \frac{d \vDash I}{d, I \triangleright d} \qquad \textsc{S-Uni}\quad \frac{d, I_1 \triangleright d' \qquad d', I_2 \triangleright d''}{d, (I_1 \cup I_2) \triangleright d''}$$

$$\textsc{S-JoinL}\quad \frac{d_1, I \triangleright d_1'}{d_1 \bowtie d_2, I \triangleright d_1' \bowtie d_2} \qquad \textsc{S-JoinR}\quad \frac{d_2, I \triangleright d_2'}{d_1 \bowtie d_2, I \triangleright d_1 \bowtie d_2'}$$

$$\textsc{S-Add-1}\quad \frac{d \nvDash [A]}{d, [A] \triangleright d \bowtie (A \mapsto \bot)}$$

$$\textsc{S-Ext-2}\quad \frac{d \nvDash A \rightarrow A'}{(A \mapsto \bot), (A \rightarrow A') \triangleright (A \mapsto A')}$$

$$\textsc{S-Add-2}\quad \frac{d \nvDash A \rightarrow A'}{d, (A \rightarrow A') \triangleright d \bowtie (A \mapsto A')}$$

$$\textsc{S-Ext-3}\quad \frac{d \nvDash A \rightarrow [A']}{(A \mapsto \bot), (A \rightarrow [A'])\ \triangleright (A \mapsto (A' \mapsto \bot))}$$

$$\textsc{S-Add-3}\quad \frac{d \nvDash A \rightarrow [A']}{d, (A \rightarrow [A'])\ \triangleright d \bowtie (A \mapsto (A' \mapsto \bot))}$$

Figure 8. Synthesis. $d, I \triangleright d'$.

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 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 the input decomposition joined with a map structure that corresponds to the interface. For example, the rule S-Add-3, supports the interface $A \rightarrow [A']$ 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.

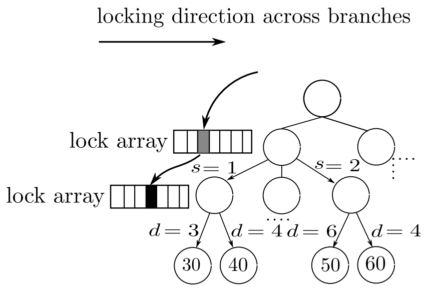


Figure 9. Locking protocol for put. Locks in grey, black and white colors are locked in shared mode, locked in exclusive mode and unlocked respectively.

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

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 [57] 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.(f) shows an example where the two branches share the weight attribute w.

5 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 and deadlock-freedom of the calls. In this section, we describe the synchronization protocols.

As Figure 9 shows, we associate an array of read-write locks with each hash-map. Hashing associates each lock to a separate set of buckets. The user can call both get and put operations on the structure. We consider each in turn.

Get operations. Get operations such as $\langle s, d\rangle \rightarrow w$ retrieve a single value while others such as $s \rightarrow [d]$ retrieve a set of values. Both involve a sequence of lookups in nested maps. The latter, in addition, ends in an iteration over the key set of the final map. For example, in the decomposition of Figure 5.(d), the method $\langle s, d\rangle \rightarrow w$ is translated to a lookup with s in the outer map and a lookup with d on the obtained inner map. The method $s \rightarrow [d]$ is translated to a lookup with the key s and an iteration on the key set d of the obtained map. In the traversal down the tree, before each lookup, we access the lock array of the map and lock (in the shared mode) the lock at the index corresponding to the hash of the key. For example, an execution of the method $\langle s, d\rangle \rightarrow w$ acquires the lock for the input s and then the lock for the input d. Before iterating a key set, we acquire all the locks of the corresponding lock array in the shared mode. For example, an execution of the method $s \rightarrow [d]$ acquires the lock for the input s and then the lock array of the obtained map d. Acquiring locks in the shared mode prevents racing concurrent put operations from mutating the buckets that are accessed but allows concurrent get operations. After the return value is retrieved, all the acquired locks are released. As the protocol follows two-phase locking and acquires the lock corresponding to each data that it accesses, 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.

Put operations. Put operations are more complicated. We must maintain the consistency of the replicated data across the branches of a decomposition. Thus, we do not release the acquired locks of a branch until the insertion is completed on all branches. The argument of a put operation is a tuple of values for all the attributes. We traverse branches in sequence from left to right. In each map $A \mapsto d$, we first check if the value that is passed to be inserted for the attribute A is already present as a key. To perform the lookup, we first acquire the lock of the key in the shared mode. If the lookup results in a value, we continue the traversal on that value. Otherwise, if the lookup results in a missing key, we elevate the access privilege of the lock to exclusive mode and insert the key and the corresponding value (or nested maps). After this privilege elevation, no other lock is needed to be acquired in the current branch. In general, 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. Doing so prevents other readers as well as writers from accessing buckets that are being updated. As the protocol follows the two-phase locking strategy and acquires the locks corresponding to the data that it reads in the shared mode and the data that it writes in the exclusive mode, it maintains linearizability.

For example, consider a put operation with arguments $\langle s = 1, d = 6, w = 15\rangle$ on the decomposition instance presented in Figure 9. In the first level, the lock for $s = 1$ is acquired in the shared mode. As $s = 1$ is present, the traversal continues to the second level. The lock for $d = 6$ is acquired in the shared mode. The key $d = 6$ is missing; thus, the lock for $d = 6$ is reacquired in the exclusive mode. Then, $d = 6$ as the key and $w = 15$ as the value are inserted.

The structure of the delete protocol is similar to the put protocol. We traverse branches from left to right and top to bottom. We use the given attributes as keys to traverse down each branch and acquire locks in the shared mode. The difference with the put protocol is that lock acquisition in the exclusive mode may happen earlier: if we reach a map whose size is 1, we reacquire its lock in its parent in exclusive mode. After this lock is acquired in exclusive mode, no other lock is acquired in the current branch. This is because the deletion of the lower maps may make this map empty and it should be deleted as well. Once, we reach a map whose key is not in the given inputs, we delete the map and its parent maps that become empty.

Concurrent accesses to decompositions can be synchronized using other optimistic mechanisms such as transactional memory or hybrid mechanisms. The employed synchronization mechanism is orthogonal to learning the quantitative performance model for representations.

6 Learning

In this section, we present how we train a performance model that can predict the throughput of candidate representations.

Multi-Layer Perceptron. We build a multi-layer perceptron [47, 62] as shown in Figure 10.(a). A multi-layer perceptron 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 data point, the difference between the output that the network calculates and the desired output is called 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. Multi-layer perceptron is trained by a supervised learning technique called back-propagation. Back-propagation 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 at that point. 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 $\frac{\partial E}{\partial w_{ji}}$ of the error for a weight w_{ji}, gradient descent updates w_{ji} by

$$\Delta(wji) = -\alpha \times \frac{\partial E}{\partial w_{ji}} + \mu \times \Delta'(wji)$$

where $\Delta'(wji)$ is the update to w_{ji} in the previous iteration. The learning rate α adjusts the move to a fraction of the derivative, and the momentum μ smooths the move by adding a fraction of the previous move.

We repeat the training over the training set by the k-fold cross-validation technique. This allows us to avoid overfitting by getting an out-of-sample estimate of the fit. This technique splits the data set into k equal subsets. In each iteration of the learning, it trains using one subset and tests the model using the other k-1 subsets.

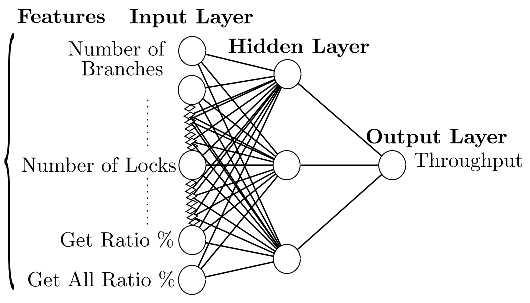
We started with small instances of networks and increased the complexity when needed. Our network has 53 input neurons, one hidden layer with 6 neurons and an output neuron for the throughput.

Feature Selection. Selection of features that can predict the value of the output is crucial to the success of learning. We choose features for representations that can predict their performance. More precisely, 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 that lookup the map of a node, the cumulative ratio of operations that iterate the map of a node, the put ratio, and whether the branches share leaves. We use a bounded number of nodes and uniquely identify them 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.

Lookup and iteration exhibit different performance characteristics as the former acquires a single lock and the latter acquires all the locks in the array. Therefore, we have separate features for them. As Figure 10.(b) shows, we have two features Get and GetAll for each node that represent the cumulative ratio of lookup and iteration that calls on the interface execute on that node. Figure 10.(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.

7 Experimental Results

Implementation. We developed a tool called Leqsy that synthesizes concurrent data representations. Leqsy is implemented using Scala [41] 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 § 5 and synthesizes synchronization by instantiating the templates. It uses Weka libraries [61] to train the performance model. Leqsy generates Java source code classes that can be easily integrated into the client codebase.



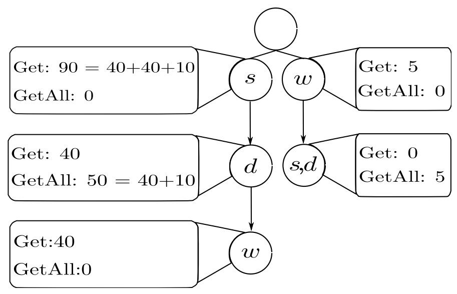


Figure 10. (a) Multi-Layer Perceptron Trained as the Performance Model. (b) Get and GetAll features for the specification of Figure 2.(a) and the decomposition of Figure 5.(d).

Benchmark*	S	A	MT	ST	SP	AT	AP	SU	AC
Graph	7	3	4	14	5	14323	5	1023	100%
Process Scheduler	6	4	3	11	5.2	9621	5.7	874	91%
File System	6	4	4	13	5.5	12975	5.5	998	100%

* 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 (sec); AP: Auto-tuned representation throughput (million ops / sec); SU: Speed-up = AT / ST; AC: Accuracy = SP / AP. Workload: 99% Query, 1% Update. (The query ratio is evenly distributed between get methods.) Platform: P_1.

Table 1. Quantitative Synthesis vs. Auto-tuning.

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), each with 8Mb of L3 cache, 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.

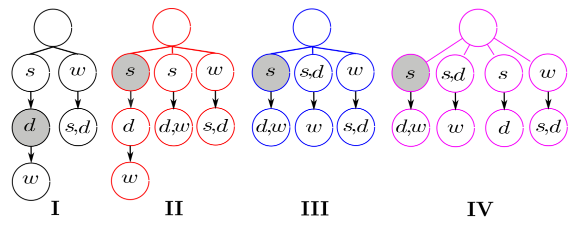
Learning. To prepare the training dataset, we generated decompositions up to width and depth of 4. We generated synchronization with permutations of 16, 32, 64, 128, 256 and 512 for lock array sizes. We generated different interfaces with different call ratios. These representations and interfaces are 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 can similarly capture and train for other performance metrics. 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 datapoints 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 [26, 27]. 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.

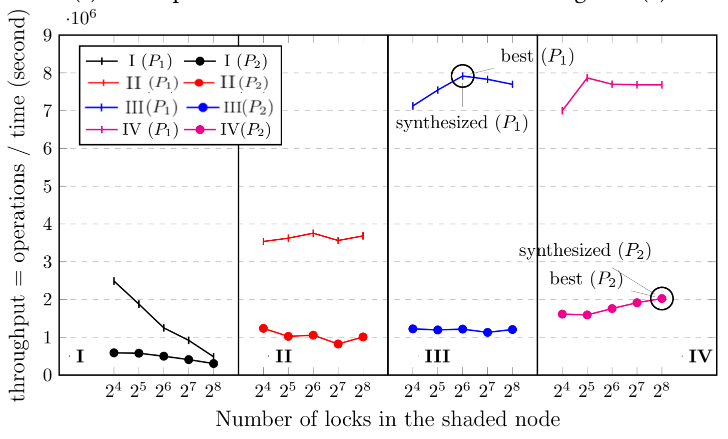
Measurements. We adopted the road network of the Northwestern USA graph from RelC [26]. We initialized each file system and process scheduler with 10000 and 15000 random tuples respectively. For each specification, we generated a random workload of 10^6 operations per thread that match the call frequencies 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 full garbage collection between runs. 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 candidate representations with the same sample workload and measure their throughputs. It finally picks the candidate with the highest throughput.

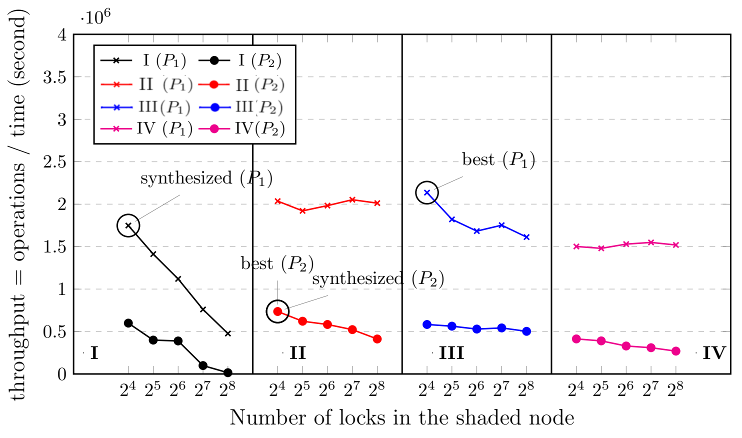
Assessment. We first measure the speed-up and the accuracy of synthesis compared to the auto-tuning baseline. We then measure the accuracy of the predicted throughput



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



(b) Benchmark: Graph. Workload: 99% Query 1% Update.



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

Figure 12. Actual throughput for candidate representations and the synthesized representation.

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 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 and evolution.

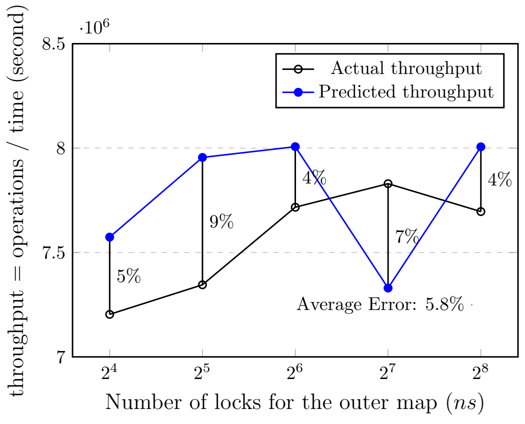


Figure 11. Prediction Accuracy: predicted vs. actual throughput. Benchmark: Process Scheduler. Decomposition: ($ns \mapsto \langle pid, cpu\rangle$) $\bowtie$ ($state \mapsto cpu$). The number of locks for maps except ns is constant 128. Workload: 90% Query, 10% Update. Platform: P_1.

Figure 11 shows the prediction accuracy for a few representation candidates of the Process Scheduler benchmark. It presents the actual versus predicted throughput for each representation. The average error for these representations is below 6%. The average error for all the representations that were considered in Table 1 was 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 throughput order of representations. The relatively low prediction error suggests that the selected features are correlated with the throughput and the training process 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 comparison of quantitative synthesis and auto-tuning 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 of the Graph benchmark, the synthesis process presented in § 4 results in the four decompositions I, II, III, and IV presented in Figure 12.(a). We study the effect of increasing the number of locks on the throughput with a query-dominated workload with 1% updates and also a workload with 10% updates in Figure 12.(b) and (c) respectively. The query ratio is evenly distributed between the get methods. We increase the size of the lock array for the nodes that are shaded in Figure 12.(a) from 16 to 256. The size of the lock array for the other nodes is constant, 128 in Figure 12.(b) and 256 in Figure 12.(c). We experiment on both platforms P_1 and P_2.

For the decomposition I, we increase the number of locks at the shaded map d. The two methods $s \to [d]$ and $s \to [\langle d, w\rangle]$ iterate the map d. An iteration acquires all the locks of the array. As the left-most parts in Figure 12.(b) and (c) (for the decomposition I) show, increasing the number of locks aggravates the throughput. The decomposition II reduces iterations. It serves the method $s \to [\langle d, w\rangle]$ by a lookup on the second branch; thus, in contrast to the decomposition I, only the method $s \to [d]$ performs an iteration in the first branch. The decomposition III reduces iteration even further. It serves both of the above methods by lookups on the first branch. Yet, the two methods $s \to [\langle d, w\rangle]$ and $s \to [d]$ share a branch. The decomposition IV introduces a separate branch for the latter to reduce contention. Thus, in Figure 12.(b) (the query-dominated workload), we see a general trend of throughput increase for the decompositions from I to IV.

In contrast, in Figure 12.(c) (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 Figure 12.(c), the most efficient representation for platform P_1 is the decomposition III with 16 locks. The trained model picks the decomposition I with 16 locks whose throughput is close to (83% of) the optimum. For the platform P_2, the optimum and synthesized representations are both from the decomposition II with 16 locks. Similarly, in Figure 12.(b), the optimum and synthesized candidates for both platforms P_1 and P_2 are the same.

We observe different throughputs across different decompositions, sizes of lock arrays and the underlying platforms. In general, increasing the number of locks increases the throughput; however, there are exceptions that do not match the intuition. For example, in Figure 12.(c) for the decomposition III, increasing the number of locks exhibits a decrease in throughput. These observations further highlight the unpredictable nature of performance and the importance of learning rather than predefining performance models.

8 Related Work

Synthesizing data structures. The importance of data structure synthesis has been recognized since 70s. Iterator inversion [19] automatically constructs iterators over data representations based on iteration specifications. SETL [48] dynamically maps abstract set and map types to concrete implementations. PReps [17] inputs relational specifications and annotations that aid compilation to efficient data structures. Similarly, DiSTiL [52] presents a declarative language for data structure specifications and replaces specifications with low-level implementations. In contrast to these pioneering works, Leqsy supports more general relational specifications, learns the performance model and statically synthesizes more elaborate and concurrent data structures.

RelC [26, 27] synthesizes both sequential and concurrent data representations. It requires the user to specify a decomposition in addition to the relation and uses an auto-tuner to execute and compare candidates. RelC considers more data structure primitives than Leqsy. On the other hand, Leqsy automatically synthesizes decompositions and learns a performance model to statically compare candidates. Cozy [37] synthesizes efficient sequential data structures. It enumerates candidate representations based on implementation outlines and uses a static cost model to compare candidates. In a follow-up work [36], Cozy supports operations across data structures and automatically synthesizes updates in addition to queries. In contrast, Leqsy learns a performance model instead of using a predefined model and synthesizes concurrent data structures.

Sketching [53, 54] automatically completes a sketch implementation of a concurrent data structure by an iterative counter-example-guided synthesis process. In contrast, Leqsy requires only a high-level relational specification. Storyboards [51] synthesizes sequential data structures from high-level structures and example operations. CnC [13] captures a graphical specification of computation based on data and control dependencies. In contrast to both Storyboards and CnC, Leqsy synthesizes concurrent structures without additional information about the control-flow structure of the solution. NR [8] transforms a sequential data structure into a NUMA-aware concurrent data structure [12, 23]. Similar to Leqsy, NR automatically synthesizes synchronization. On the other hand, Leqsy accepts a high-level specification and searches for the optimum structure. Chameleon [50] dynamically profiles client calls on data structures to suggest more efficient data structure implementations. GLS [2] and Fable [42] dynamically adapt locks to the contention level. Similarly, Fast-path-slow-path [33] switches between lock-free and wait-free versions based on the contention. In contrast, Leqsy is a static tool.

Boosting [28] and Semantic Locking [24] use commutative pessimistic synchronization and Predication [7] and Transactional Libraries [55] use optimistic synchronization [29] to implement composable concurrent data structures. While users can use these techniques to compose atomic structures, they can use Leqsy to automatically synthesize them.

Learning and concurrent data structures. Machine learning has aided concurrent systems. It has been used to predict the number of concurrent threads for optimum performance [45, 46, 56]. Smartlocks [21] uses machine learning and heuristic functions to adapt spin-locks to dynamic workloads. Smart Data Structures [20] use online learning to dynamically adapt the data structure to varying workloads. In contrast, Leqsy learns a performance model to statically synthesize the most efficient data structure for a target workload. Data Calculator [32] synthesizes data structures and predicts their performance based on user-defined layout specifications and target architectures. It learns and uses performance models for its primitives. In contrast, Leqsy does not require layouts, supports put in addition to query operations, learns the performance model of layouts, and automatically explores the space of layouts.

Synthesizing concurrent programs. Various projects [6, 16, 18, 25, 31, 39, 63] infer atomic sections and insert locks to implement them. Others [3, 10, 11, 22] apply semantic-preserving transformations to efficiently synchronize, repair or verify synchronization. Similarly, Paraglider [59] and AGS [60] explore the execution space to iteratively add synchronization that removes violating interleavings. Other works [1, 4, 34] automatically insert fence instructions to data structures. RCU [40] automatically reduces latency of read accesses to the data structure. RLU [38] improves RCU by allowing concurrent writers. Elixir [43, 44] automatically synthesizes synchronization specifically for parallel graph processing. These works synthesize synchronization with no quantitative [5, 9, 14, 15] measure of optimality. Quantitative synchronization synthesis [58], however, uses analytical performance models to synthesize concurrent programs. However, it uses a static model. Leqsy learns and uses the performance model for quantitative comparisons.

9 Conclusion and Future Works

We presented a new approach to quantitative synthesis that trains a performance model to predict the efficiency of candidates. We showed that performance is irregular for the space of candidate data representations with different workloads and platforms; therefore, one predefined performance model cannot be universally predictive of performance. We showed that a performance model can be learned and effectively used to select efficient concurrent representations for relational specifications. The benefits of learning the performance model such as adaptability and platform-independence can carry over to other synthesis domains.

Feature engineering is an experimental, repetitive and time-consuming process. The deciding features may be unintuitive and easily missed, and irrelevant features may unnecessarily slow down the learning process. Deep learning has been successful in automatic feature extraction for image processing and natural language processing. In addition, deep learning is known to steadily increase performance with an increase in the training datapoints. We will apply deep learning to automatically extract and validate the features that can predict throughput.

References

[1] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. 2014. Don't sit on the fence. In *International Conference on Computer Aided Verification*. Springer, 508–524.

[2] Jelena Antić, Georgios Chatzopoulos, Rachid Guerraoui, and Vasileios Trigonakis. 2016. Locking Made Easy. In *Proceedings of the 17th International Middleware Conference (Middleware '16)*. ACM, New York, NY, USA, Article 20, 14 pages. https://doi.org/10.1145/2988336.2988357

[3] Maya Arbel, Guy Golan-Gueta, Eshcar Hillel, and Idit Keidar. 2015. Towards automatic lock removal for scalable synchronization. In *International Symposium on Distributed Computing*. Springer, 170–184.

[4] John Bender, Mohsen Lesani, and Jens Palsberg. 2015. Declarative Fence Insertion. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 367–385. https://doi.org/10.1145/2814270.2814318

[5] 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.

[6] Roderick Bloem, Georg Hofferek, Bettina Könighofer, Robert Könighofer, Simon Ausserlechner, and Raphael Spörk. 2014. Synthesis of Synchronization Using Uninterpreted Functions. In *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design (FMCAD '14)*. FMCAD Inc, Austin, TX, Article 11, 8 pages. http://dl.acm.org/citation.cfm?id=2682923.2682937

[7] 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.

[8] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. 2017. Black-box Concurrent Data Structures for NUMA Architectures. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 207–221. https://doi.org/10.1145/3037697.3037721

[9] 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.

[10] Pavol Černý, Thomas A. Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, and Thorsten Tarrach. 2013. Efficient Synthesis for Concurrency by Semantics-Preserving Transformations. In *Computer Aided Verification*, Natasha Sharygina and Helmut Veith (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 951–967.

[11] Pavol Černý, Thomas A. Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, and Thorsten Tarrach. 2014. Regression-Free Synthesis for Concurrency. In *Computer Aided Verification*, Armin Biere and Roderick Bloem (Eds.). Springer International Publishing, Cham, 568–584.

[12] Milind Chabbi, Michael Fagan, and John Mellor-Crummey. 2015. High performance locks for multi-level NUMA systems. *ACM SIGPLAN Notices* 50, 8 (2015), 215–226.

[13] Aparna Chandramowlishwaran, Kathleen Knobe, and Richard Vuduc. 2010. Performance evaluation of concurrent collections on high-performance multicore computing systems. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 1–12.

[14] 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.

[15] 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.

[16] Sigmund Cherem, Trishul Chilimbi, and Sumit Gulwani. 2008. Inferring locks for atomic sections. *ACM SIGPLAN Notices* 43, 6 (2008), 304–315.

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

[18] Dave Cunningham, Khilan Gudka, and Susan Eisenbach. 2008. Keep off the grass: Locking the right path for atomicity. In *International Conference on Compiler Construction*. Springer, 276–290.

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

[20] 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.

[21] 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.

[22] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. 2009. A calculus of atomic actions. In *ACM SIGPLAN Notices*, Vol. 44. ACM, 2–15.

[23] Elad Gidron, Idit Keidar, Dmitri Perelman, and Yonathan Perez. 2012. SALSA: Scalable and Low Synchronization NUMA-aware Algorithm for Producer-consumer Pools. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '12)*. ACM, New York, NY, USA, 151–160. https://doi.org/10.1145/2312005.2312035

[24] 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.

[25] Richard L Halpert, Christopher JF Pickett, and Clark Verbrugge. 2007. Component-based lock allocation. In *Parallel Architecture and Compilation Techniques, 2007. PACT 2007. 16th International Conference on*. IEEE, 353–364.

[26] 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

[27] 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

[28] 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.

[29] Maurice Herlihy and J Eliot B Moss. 1993. *Transactional memory: Architectural support for lock-free data structures*. Vol. 21. ACM.

[30] 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

[31] Michael Hicks, Jeffrey S Foster, and Polyvios Pratikakis. 2006. Lock inference for atomic sections. In *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*.

[32] 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

[33] Alex Kogan and Erez Petrank. 2012. A methodology for creating fast wait-free data structures. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 141–150.

[34] Michael Kuperstein, Martin Vechev, and Eran Yahav. 2012. Automatic Inference of Memory Fences. *SIGACT News* 43, 2 (June 2012), 108–123.

https://doi.org/10.1145/2261417.2261438

[35] 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.

[36] 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.

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

[38] Alexander Matveev, Nir Shavit, Pascal Felber, and Patrick Marlier. 2015. Read-log-update: A Lightweight Synchronization Mechanism for Concurrent Programming. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 168–183. https://doi.org/10.1145/2815400.2815406

[39] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. 2006. Autolocker: synchronization inference for atomic sections. In *ACM Sigplan Notices*, Vol. 41. ACM, 346–358.

[40] Paul E McKenney and John D Slingwine. 1998. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*. 509–518.

[41] 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.

[42] Filip Pizlo, Daniel Frampton, and Antony L Hosking. 2011. Fine-grained adaptive biased locking. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*. ACM, 171–181.

[43] Dimitrios Prountzos, Roman Manevich, and Keshav Pingali. 2012. Elixir: A system for synthesizing concurrent graph programs. *ACM SIGPLAN Notices* 47, 10 (2012), 375–394.

[44] Dimitrios Prountzos, Roman Manevich, and Keshav Pingali. 2015. Synthesizing Parallel Graph Programs via Automated Planning. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 533–544. https://doi.org/10.1145/2737924.2737953

[45] 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

[46] 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

[47] 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.

[48] 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.

[49] 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.

[50] Ohad Shacham, Martin Vechev, and Eran Yahav. 2009. Chameleon: Adaptive Selection of Collections. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 408–418. https://doi.org/10.1145/1542476.1542522

[51] Rishabh Singh and Armando Solar-Lezama. 2011. Synthesizing data structure manipulations from storyboards. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 289–299.

[52] 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

[53] 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.

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

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

[56] 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.

[57] 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.

[58] Pavol Černý, Krishnendu Chatterjee, Thomas A. Henzinger, Arjun Radhakrishna, and Rohit Singh. 2011. Quantitative Synthesis for Concurrent Programs. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 243–259. http://dl.acm.org/citation.cfm?id=2032305.2032325

[59] Martin Vechev and Eran Yahav. 2008. Deriving linearizable fine-grained concurrent objects. *ACM SIGPLAN Notices* 43, 6 (2008), 125–135.

[60] Martin Vechev, Eran Yahav, and Greta Yorsh. 2010. Abstraction-guided synthesis of synchronization. In *ACM Sigplan Notices*, Vol. 45. ACM, 327–338.

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

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

[63] Yuan Zhang, Vugranam C Sreedhar, Weirong Zhu, Vivek Sarkar, and Guang R Gao. 2008. Minimum lock assignment: A method for exploiting concurrency among critical sections. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 141–155.