Hiding the Complexity of Concurrent Indexes

Pedro da Rocha Pinto
Imperial College London
pmd09@doc.ic.ac.uk

Thomas Dinsdale-Young
Imperial College London
td202@doc.ic.ac.uk

Mike Dodds
University of Cambridge
md466@cl.cam.ac.uk

Philippa Gardner
Imperial College London
pg@doc.ic.ac.uk

Mark Wheelhouse
Imperial College London
mjw03@doc.ic.ac.uk

ABSTRACT

Index data structures, such as B^{Link} trees and hash tables, are commonly used in database implementations. A database implementor typically works with a high-level view of indexes as mappings from keys to values. Using this high-level view, he is able to build up complex abstractions such as transactions, without needing to know exactly how indexes work 'under the hood'.

This high-level intuition breaks down when the index is accessed concurrently. If two threads share access to a key, neither can be sure whether the other has removed its value. A specification of a concurrent index must account for this sharing. Also, the algorithms that implement concurrent indexes are highly complex and subtle. Justifying the use of a high-level specification requires hiding the complexity of an implementation and is thus a major challenge.

In this paper, we give an abstract specification for concurrent indexes, and prove that it is satisfied by concrete implementations. Our specification captures high-level sharing of index elements, allowing collaboration between threads. We show that a widely-used implementation based on B^{Link} trees is correct with respect to our abstract specification. Finally, we demonstrate that our specification is useful by verifying several client algorithms.

General Terms

Algorithms, Concurrency, Theory, Verification.

Keywords

B^{Link} Tree, Concurrent Abstract Predicates, Separation Logic.

1. INTRODUCTION

Index data structures are commonly used in database implementations. The standard abstract view of an index is appealingly simple: an index is a mapping from keys to values where the values can be read, added or removed. However, in the pursuit of improved performance, implementations of indexes have become increasingly complex. Practical algorithms such as Sagiv's B^{Link} tree [13] employ ferociously complex patterns of locking that allow many threads to write to the same index. There is a substantial gulf between the naïve abstract view of an index and the practical reality; in particular the abstract view of an index does not account for concurrency.

We have developed a simple abstract specification for concurrent indexes, and a way of showing that complex implementations such as Sagiv's B^{Link} tree algorithm satisfy this specification. Such a specification is not obvious, as many threads may share keys in the index. For example, if two threads running in parallel are trying to remove the value from the same key, then neither thread can be sure when it gets scheduled if there is a mapping for the key or not. However, once both threads have terminated we know that the key will be in the index. An abstract specification must, therefore, allow high-level reasoning about both exclusive and non-exclusive access to keys.

Our specification is based on separation logic [12] with concurrent abstract predicates [6]. Concurrent abstract predicates allow distinct keys to be reasoned about separately, even though these keys may participate in the same underlying shared structure. For example, we provide a predicate $\mathsf{in}(h, k, v)$ that states that there is a mapping from key k to value v in the index at address h. We can use this predicate to reason about many different keys used at the same time by separate threads: for example, multiple keys can have their values concurrently removed from the index.

We extend concurrent abstract predictes to provide a high-level specification that captures not only facts about the contents of the index, but also the changes allowed to it. For example, we provide a predicate $\mathsf{in}_{\mathsf{rem}}(h, k, v)_i$ which states that there was a mapping from key k to value v in the index stored at h in the past, but that both the current thread and other threads in the environment are allowed to remove this key's mapping in the index. The fractional permission i allows this predicate to be shared amongst several threads, but only a thread which owns some part of this predicate may remove the mapping from the key in question. The permissions must ensure that a predicate is self-stable: that is, immune from interference from the surrounding environment. Our predicates are thus able to specify independent properties about the data, even though this data is shared.

We demonstrate the utility of our specification by verifying some simple algorithms. We look at a set range function that sets a range of keys to a particular value and makes use of disjoint concurrency to perform this repetitive task

in an efficient way. We then look at two examples which demonstrate sharing between threads. The first is a range search algorithm which makes use of the search operation from our concurrent index specification. This example also shows how our abstract specification can be linked to SQL's transaction isolation levels. In particular, our $\mathsf{in}_{\mathsf{def}}(h, k, v)_1$ predicate (analogous to the in predicate described above) illustrates the behaviour of the SQL *Serializable* isolation level. Our second example is the Sieve of Eratosthenes [2, 9] which filters the keys of an index, pruning out non-prime keys. This example demonstrates that our specification is applicable to real-world code, but is also representative of database operations which involve sharing on keys.

Our approach allows clients to work with our abstract specification of concurrent indexes, without worrying about the underlying implementation. One of the principle advantages of using concurrent abstract predicates [6] is that they allow us to verify that a concrete implementation is correct with respect to our abstract specification. We choose to take an implementation of Sagiv's B^{Link} tree algorithms [13] and show that it does indeed adhere to the behaviour of our high-level index specification. Not only is this a highly relevant real-world example of an index implementation, but this is also probably one of the most complex examples to date of concurrent reasoning using separation logic.

Contributions

The research contributions of this paper are:

1. An abstract specification for concurrent indexes that captures the addition and removal of shared values.

2. Proofs of algorithms for set range, range search and the Sieve of Eratosthenes using our specification, showing that the sharing in the specification can be used to reason about high-level coordination between threads.

3. A proof that a concurrent B^{Link} tree algorithm is correct with respect to our abstract specification, which allows us to hide the complexity of the implementation details. This demonstrates that our abstract specification is respected by real-world implementations of concurrent indexes.

Related Work

There has been some work in the past on analysing the performance of various B-tree algorithms [10], but very little work on the formal verification of such algorithms. The search and insert algorithms have been verified to be fault free in a simplified sequential setting [14], but this does not handle the significantly more complex problem of verifying that the concurrent versions of these algorithms are correct. A sequential B-tree implementation has been included as part of a fully verified relational database management system in Coq [11], but the authors highlight that their B-tree invariants and proofs were highly complex and lacked abstraction. Rather than attempting to prove such invariants, we instead choose to show that this highly-complex algorithm implements the simple behaviour required by clients.

2. SEPARATION LOGIC & ABSTRACTION

The work in this paper is based on separation logic [12], a program logic for reasoning *locally* (or modularly) about programs that manipulate resource: for example, C programs that manipulate the heap. If we establish that the following specification holds for a command $\mathbb{C}$ (here P and Q are assertions):

$$\{P\}\ \mathbb{C}\ \{Q\}$$

then this specification says (1) executing $\mathbb{C}$ in a state satisfying assertion P will result in a state satisfying assertion Q, if the command terminates; and (2) the resources represented by P are the only resources needed for $\mathbb{C}$ to execute successfully. Other resources can be conjoined with such a specification without affecting its validity. This is expressed by the following proof rule,

$$\textsc{Frame}\quad \frac{\{P\}\ \mathbb{C}\ \{Q\}}{\{P * F\}\ \mathbb{C}\ \{Q * F\}}\quad \text{side-condition}$$

which allows us to extend a specification on a small resource with an unmodified *frame assertion* F, giving a larger resource. Here, '$*$' is the so-called *separating conjunction*. Combining two assertions P and F into a separating conjunction $P * F$ asserts that both resources are independent of each other. In the proof rule, the side-condition simply states that no variable occurring free in the frame F is modified by the program $\mathbb{C}$.

Separation logic provides straightforward reasoning about sequential programs. It also handles concurrency, using the parallel rule:

$$\textsc{Par}\quad \frac{\{P_1\}\ \mathbb{C}_1\ \{Q_1\} \qquad \{P_2\}\ \mathbb{C}_2\ \{Q_2\}}{\{P_1 * P_2\}\ \ \mathbb{C}_1 \parallel \mathbb{C}_2\ \ \{Q_1 * Q_2\}}$$

Reasoning using this rule is *thread-local*: that is, each thread reasons purely about the resources that are mentioned in its precondition, without requiring global reasoning about interleaving. As with sequential reasoning, locality is the key to compositional reasoning about threads.

Abstract specifications are a mechanism for specifying the external behaviour of a module's functions, while hiding their implementation details from clients. In an abstract specification, resources are represented by *abstract predicates*. Clients do not need to know the concrete definitions of these predicates; they can reason purely in terms of the module's operations. For example, insertion in a set module can be specified by:

$$\{\,\mathsf{set}(x, S)\,\}\quad \texttt{insert}(x, v)\quad \{\,\mathsf{set}(x, S \cup \{v\})\,\}$$

The insert operation requires the set S at address x and returns the updated set $S \cup \{v\}$ at the same address. A client can reason about the high level behaviour of `insert` without knowing about the concrete definition of this predicate.

Abstract predicates provide reasoning about complete sets. *Concurrent abstract predicates* provide reasoning about shared elements of a concurrent set. We will use the same approach to reason about concurrent indexes.

3. INDEX SPECIFICATION: DISJOINTNESS

We start by providing a simple specification for concurrent indexes. Our specification divides an index up into its constituent keys and ensures that each key is accessed disjointly by at most one thread. Using concurrent abstract predicates, we hide the fact that each key is part of an underlying shared data structure, allowing straightforward high-level reasoning about keys.

The specification given in this section is a restricted version of our full specification, given in §4, which permits the sharing of keys.

An index is a partial function mapping keys to values:

$$H : \mathsf{Keys} \rightharpoonup \mathsf{Vals}$$

In a *concurrent index*, this function can be accessed and modified by many threads at the same time. There are three operations on a concurrent index; `search`, `insert` and `remove`: (here we assume that h is a pointer to the underlying shared index H).

- `search`(h, k) looks for the key k in the index. It returns $H(k)$ if it is defined, and nil otherwise.
- `insert`(h, k, v) tries to modify H to associate the key k with value v. If $k \in \mathrm{dom}(H)$ then `insert` returns false and does nothing. Otherwise it modifies the shared index to $H[k \mapsto v]$ and returns true.
- `remove`(h, k) tries to remove the value of the key k from the index. If the $k \notin \mathrm{dom}(H)$ then `remove` returns false. Otherwise it rewrites the index to $H \setminus \{k\}$ and returns true.

This view of operations on the index is appealingly simple, but cannot be used for practical concurrent reasoning. This is because it depends on *global* knowledge of the underlying index H. To reason in this way, a thread would require perfect knowledge of the behaviour of other threads.

To avoid this, we give a specification that breaks the index up by key value. Our specification allow threads to hold the exclusive ownership of an individual key. (In §4 we will extend this approach to allow reasoning about keys that are shared). Each key in the index is represented by a predicate, either in or out depending on whether the key is defined or not. The predicates have the following intuitive interpretation:

$\mathsf{in}(h, k, v)$: there is a mapping in the index h from k to v.

$\mathsf{out}(h, k)$: there is no mapping in the index h from k.

These predicates not only capture knowledge about a key, but also exclusive permission to alter that key. A thread holding the predicate for a key can be sure that no other thread will modify the value associated with that key.

The operations on an index have the following specifications with respect to these predicates:

$$\begin{array}{lll}
\{\mathsf{in}(h,k,v)\} & r\texttt{:=search}(h,k) & \{\mathsf{in}(h,k,v) \wedge r = v\} \\
\{\mathsf{out}(h,k)\} & r\texttt{:=search}(h,k) & \{\mathsf{out}(h,k) \wedge r = \mathsf{nil}\} \\
\{\mathsf{in}(h,k,v')\} & r\texttt{:=insert}(h,k,v) & \{\mathsf{in}(h,k,v') \wedge r = \mathsf{ff}\} \\
\{\mathsf{out}(h,k)\} & r\texttt{:=insert}(h,k,v) & \{\mathsf{in}(h,k,v) \wedge r = \mathsf{tt}\} \\
\{\mathsf{in}(h,k,v)\} & r\texttt{:=remove}(h,k) & \{\mathsf{out}(h,k) \wedge r = \mathsf{tt}\} \\
\{\mathsf{out}(h,k)\} & r\texttt{:=remove}(h,k) & \{\mathsf{out}(h,k) \wedge r = \mathsf{ff}\}
\end{array}$$

Note that our specification allows us to reason about the index as a collection of disjoint, independent elements, despite the fact that they are often implemented as a single shared data structure. Predicates can be combined using the separating conjunction $*$, indicating that they hold independently of each other.

Each predicate represents exclusive ownership of a particular key. Our specification represents this fact by exposing the following axiom:

$$(\mathsf{in}(h,k,v) \vee \mathsf{out}(h,k)) * (\mathsf{in}(h,k,v') \vee \mathsf{out}(h,k)) \implies \mathsf{false}$$

To justify this abstract specification, we use concurrent abstract predicates [6]. In this approach, predicates hide sharing in their implementation, presenting a high-level *fiction of disjointness* [7]. A thread can reason independently about individual predicates, even though they may be implemented by a single underlying shared structure. In §5 we show how to use concurrent abstract predicates to prove a generalised version of this specification against an implementation based on B^{Link} trees.

Consider the following simple program:

$$C \triangleq r\texttt{:=search}(h,k_2)\texttt{; (insert}(h,k_1,r) \parallel \texttt{remove}(h,k_2)\texttt{)}$$

This program retrieves the value v associated with the key k_2. It then concurrently associates v with the key k_1 and removes the key k_2. When the program completes, k_1 will be associated with v, and k_2 will have been removed from the index. This program satisfies the following specification:

$$\{\mathsf{out}(h,k_1) * \mathsf{in}(h,k_2,v)\}\ C\ \{\mathsf{in}(h,k_1,v) * \mathsf{out}(h,k_2)\}$$

Using our specifications we can prove that this program satisfies its specification, as follows:

$$\begin{array}{c}
\{\mathsf{out}(h,k_1) * \mathsf{in}(h,k_2,v)\} \\
r\texttt{:=search}(h,k_2); \\
\{\mathsf{out}(h,k_1) * \mathsf{in}(h,k_2,v) \wedge r = v\} \\
\begin{array}{ll}
\{\mathsf{out}(h,k_1) \wedge r = v\} & \{\mathsf{in}(h,k_2,v)\} \\
r_1\texttt{:=insert}(h,k_1,r) & r_2\texttt{:=remove}(h,k_2) \\
\{\mathsf{in}(h,k_1,v) \wedge r_1 = \mathsf{tt}\} & \{\mathsf{out}(h,k_2) \wedge r_2 = \mathsf{tt}\}
\end{array} \\
\{\mathsf{in}(h,k_1,v) * \mathsf{out}(h,k_2)\}
\end{array}$$

In this proof, the `search` operation first uses the predicate $\mathsf{in}(h,k_2,v)$ to retrieve the value v. Then the parallel rule hands `insert` and `remove` the $\mathsf{out}(h,k_1)$ and $\mathsf{in}(h,k_2,v)$ predicates respectively. The postcondition of the program consists of the separating conjunction of the two thread postconditions where we also forget the return values r_1 and r_2.

3.1 Example: Set Range

Consider a function `setRange` which sets all keys in a specified interval to a particular value, inserting keys into the index if necessary. We implement `setRange` recursively by repeatedly splitting the interval and starting threads in parallel for each sub-interval. The value is inserted by calling a helper function `set`.

```
setRange(h, k1, k2, v) {
 if (k1 = k2) {
   r := search(h, k1);
   set(h, k1, r, v);
 } else {
    setRange(h,k1,k1+((k2-k1)/2),v)
  || setRange(h,k1+((k2-k1)/2)+1,k2,v)
}}
```

```
set(h, k, r, v) {
 if (r = nil) {
   insert(h,k,v);
 } else {
   remove(h,k);
   insert(h,k,v);
}}
```

We specify `setRange` as follows:

$$\begin{array}{c}
\{\circledast_{k_1 \le i \le k_2}.\ \mathsf{out}(h,i) \vee \mathsf{in}(h,i,-)\} \\
\texttt{setRange}(h,k_1,k_2,v) \\
\{\circledast_{k_1 \le i \le k_2}.\ \mathsf{in}(h,i,v)\}
\end{array}$$

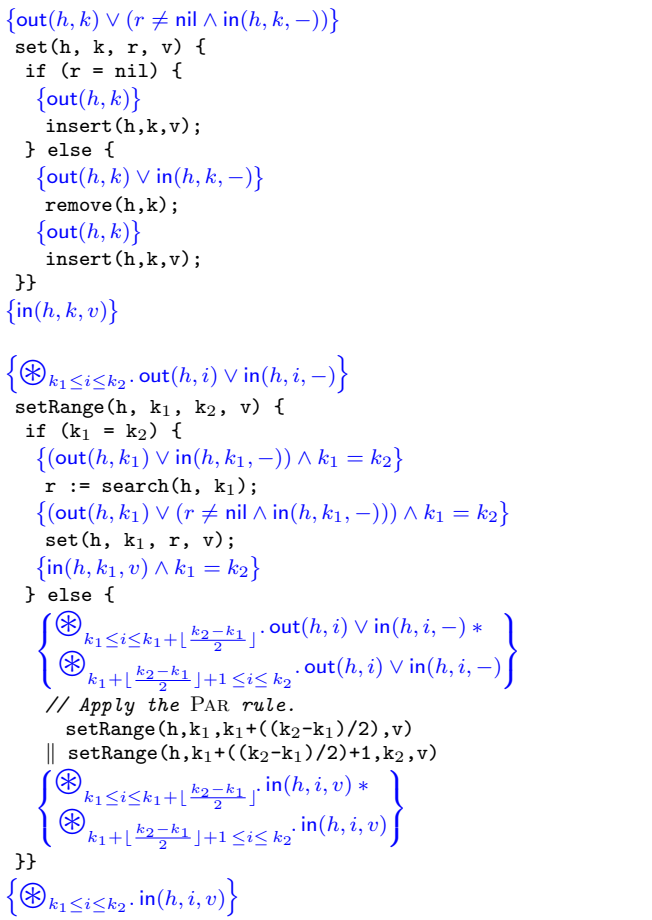

Figure 1: Proofs for `setRange` and `set`.

(Here $\circledast$ is the iterated separating conjunction. That is, $\circledast_{x\in\{1,2,3\}}.\, P$ is equivalent to $P[1/x] * P[2/x] * P[3/x]$.)

We also specify `set` as follows:

$$\{\mathsf{out}(h,k) \vee (r \neq \mathsf{nil} \wedge \mathsf{in}(h,k,-))\}\ \ \mathtt{set}(h,k,r,v)\ \ \{\mathsf{in}(h,k,v)\}$$

We prove that the specifications for `setRange` and `set` are correct. Proof-sketches are given in Fig 1. We write $-$ to indicate an unknown, existentially quantified value.

This kind of reasoning extends to any similar program that employs disjoint parallel computation on a shared concurrent index. However, it assumes that threads access entirely disjoint sets of keys. In the next section, we present a specification that allows sharing between threads.

4. INDEX SPECIFICATION: SHARING

Disjoint ownership of keys makes for simple reasoning, but in some cases, several threads access and modify a single key. For example, we should be able to reason about programs such as

$$C \quad \triangleq \quad r_1 \texttt{:=search}(h,k) \ \parallel\ r_2 \texttt{:=insert}(h,k,v)$$

If we know that at the start of the program there is no mapping from key k, we should be able to establish that there will be a mapping to the value v at the end. However, we will not know the value of r_1, because we do not know at which point during the `insert` operation the `search` will read the value of key k. We want a specification for the concurrent index that permits the sharing of keys between threads.

Implementations have many different ways of handling the sharing of keys (for example using mutual exclusion locks or transactions), but at the abstract level they all behave in the same way. If a thread reads a key multiple times, the reads all return the same result, unless another thread also writes to that key.

We define a revised set of abstract predicates that express *three* facts about a given key:

1. whether there is a mapping from the key to a particular value;
2. whether the thread holding the predicate can add or remove the value of the key in the index;
3. whether any other concurrently running threads (the *environment*) can add or remove the value of the key in the index.

These facts are related. If a key maps to a value in the index, but other threads are allowed to remove the value of the key, the current thread cannot assume the value will remain in the index. Our predicates therefore reflect the uncertainty generated by sharing in a local way.

We define the following set of predicates, parametric on key k and index h. The fractional component $i \in (0,1]$ and the subscripts def, ins and rem record the behaviours allowed by the current thread and its environment.

$\mathsf{in}_{\mathsf{def}}(h,k,v)_i$:	there is definitely a mapping from key k to value v.
$\mathsf{out}_{\mathsf{def}}(h,k)_i$:	there is definitely no mapping from key k.
$\mathsf{in}_{\mathsf{ins}}(h,k,v)_i$:	there is a mapping from key k to value v and other threads can only insert.
$\mathsf{out}_{\mathsf{ins}}(h,k)_i$:	there may be a mapping from key k and other threads can only insert.
$\mathsf{in}_{\mathsf{rem}}(h,k,v)_i$:	there may be a mapping from key k and other threads can only remove.
$\mathsf{out}_{\mathsf{rem}}(h,k)_i$:	there is no mapping from key k and other threads can only remove.
$\mathsf{read}(h,k)$:	the thread can only search for key k.
$\mathsf{unk}(h,k)_i$:	the thread can search, remove and insert for key k, but so can other threads.

The def subscript specifies that only one thread can write (insert or remove) to a key at a time. The ins subscript specifies that both thread and environment can insert and search on the key, while rem specifies that the thread and environment can search on and remove the key. As in Boyland [3], fractional permissions are used to record predicate ownership. We allow permissions to be split / joined. A value $i \in (0,1)$ records that the predicate is shared with other threads, while $i = 1$ records it is held exclusively.

The choice of predicates may, at first, seem somewhat arbitrary, but each represents a stable combination of facts about the key and the behaviours permitted by the thread and the environment. Table 1 shows how various combinations of fractional permissions and subscripts correspond to various behaviours. As can be seen, our predicates give almost a complete coverage of all possible combinations. There are 16 possible permission combinations, but the 8

		Thread		Env.	
Predicate	Perm.	Ins.	Del.	Ins.	Del.
$\mathsf{in}_{\mathsf{def}}$ / $\mathsf{out}_{\mathsf{def}}$	1	Yes	Yes	No	No
$\mathsf{in}_{\mathsf{def}}$ / $\mathsf{out}_{\mathsf{def}}$	i	No	No	No	No
$\mathsf{in}_{\mathsf{ins}}$ / $\mathsf{out}_{\mathsf{ins}}$	1	Yes	No	No	No
$\mathsf{in}_{\mathsf{ins}}$ / $\mathsf{out}_{\mathsf{ins}}$	i	Yes	No	Yes	No
$\mathsf{in}_{\mathsf{rem}}$ / $\mathsf{out}_{\mathsf{rem}}$	1	No	Yes	No	No
$\mathsf{in}_{\mathsf{rem}}$ / $\mathsf{out}_{\mathsf{rem}}$	i	No	Yes	No	Yes
unk	i	Yes	Yes	Yes	Yes
read	-	No	No	Yes	Yes

Table 1: Predicates and their interference.

missing combinations are either cases where the current thread has no access to a key, or it is only safe to conclude that a key has an unknown value. For example, the case where the current thread is only allowed to insert and the environment is only allowed to remove leads to an unknown value of that key as neither is limited to when, or how many times its action may be called.

The full specification for `insert`, `remove` and `search` and axioms controlling the splitting of predicates are given in Fig 2. In the definition of the axioms, X is used to stand for $\mathsf{in}_{\mathsf{def}}$, $\mathsf{out}_{\mathsf{def}}$, $\mathsf{in}_{\mathsf{ins}}$, etc.

Predicate splitting is used to allow the sharing of keys between threads. Our specification includes a set of *axioms* defining the ways that predicates can be split safely. For example, if there is definitely a mapping from key k to a value v, this fact can be shared between two threads.

$$\mathsf{in}_{\mathsf{def}}(h,k,v)_{i+j} \iff \mathsf{in}_{\mathsf{def}}(h,k,v)_i * \mathsf{in}_{\mathsf{def}}(h,k,v)_j \quad \text{if } i+j \leq 1$$

Fractions are used as a record of splitting so that the original predicate can be recovered. All of the basic in, out and unk predicates allow this kind of splitting.

We use splitting and joining to resolve uncertainty in the predicates. For example, the predicate $\mathsf{out}_{\mathsf{ins}}(h,k)_i$ records that the key k started with no value associated, but that it may have been added by another thread. Another thread may hold $\mathsf{in}_{\mathsf{ins}}(h,k,v)_j$ for some v, recording that v has been inserted. When we combine the two predicates the uncertainty is resolved; k must be associated with v.

$$\mathsf{out}_{\mathsf{ins}}(h,k)_i * \mathsf{in}_{\mathsf{ins}}(h,k,v)_j \implies \mathsf{in}_{\mathsf{ins}}(h,k,v)_{i+j} \quad \text{if } i+j \leq 1$$

Recall now the program C with which we began this section. We would like C to satisfy the following specifications:

$$\{\mathsf{out}_{\mathsf{def}}(h,k)_1\} \;\; C \;\; \{\mathsf{in}_{\mathsf{def}}(h,k,v)_1\}$$
$$\{\mathsf{in}_{\mathsf{def}}(h,k,v')_1\} \;\; C \;\; \{\mathsf{in}_{\mathsf{def}}(h,k,v')_1\}$$

Using our axioms and the abstract specifications for `search` and `insert`, along with separation logic's standard inference rules [12], we can prove the first of these specifications as follows:

$$\{\mathsf{out}_{\mathsf{def}}(h,k)_1\}$$
$$\{\mathsf{read}(h,k) * \mathsf{out}_{\mathsf{def}}(h,k)_1\}$$
$$\begin{array}{rcl} \{\mathsf{read}(h,k)\} & \| & \{\mathsf{out}_{\mathsf{def}}(h,k)_1\} \\ r_1\texttt{:=search}(h,k) & \| & r_2\texttt{:=insert}(h,k,v) \\ \{\mathsf{read}(h,k)\} & \| & \{\mathsf{in}_{\mathsf{def}}(h,k,v)_1 \wedge r_2 = \mathsf{tt}\} \end{array}$$
$$\{\mathsf{read}(h,k) * \mathsf{in}_{\mathsf{def}}(h,k,v)_1 \wedge r_2 = \mathsf{tt}\}$$
$$\{\mathsf{in}_{\mathsf{def}}(h,k,v)_1\}$$

SPECIFICATIONS:

$$\begin{array}{rll}
\{\mathsf{in}_{\mathsf{def}}(h,k,v)_i\} & r\texttt{:=search}(h,k) & \{\mathsf{in}_{\mathsf{def}}(h,k,v)_i \wedge r = v\} \\
\{\mathsf{out}_{\mathsf{def}}(h,k)_i\} & r\texttt{:=search}(h,k) & \{\mathsf{out}_{\mathsf{def}}(h,k)_i \wedge r = \mathsf{nil}\} \\
\{\mathsf{in}_{\mathsf{ins}}(h,k,v)_i\} & r\texttt{:=search}(h,k) & \{\mathsf{in}_{\mathsf{ins}}(h,k,v)_i \wedge r \neq \mathsf{nil}\} \\
\{\mathsf{out}_{\mathsf{rem}}(h,k)_i\} & r\texttt{:=search}(h,k) & \{\mathsf{out}_{\mathsf{rem}}(h,k)_i \wedge r = \mathsf{nil}\} \\
\{\mathsf{read}(h,k)\} & r\texttt{:=search}(h,k) & \{\mathsf{read}(h,k)\} \\
\\
\{\mathsf{in}_{\mathsf{def}}(h,k,v)_i\} & r\texttt{:=insert}(h,k,v) & \{\mathsf{in}_{\mathsf{def}}(h,k,v)_i \wedge r = \mathsf{ff}\} \\
\{\mathsf{out}_{\mathsf{def}}(h,k)_1\} & r\texttt{:=insert}(h,k,v) & \{\mathsf{in}_{\mathsf{def}}(h,k,v)_1 \wedge r = \mathsf{tt}\} \\
\{\mathsf{in}_{\mathsf{ins}}(h,k,v')_i\} & r\texttt{:=insert}(h,k,v) & \{\mathsf{in}_{\mathsf{ins}}(h,k,v') \wedge r = \mathsf{ff}\} \\
\{\mathsf{out}_{\mathsf{ins}}(h,k)_i\} & r\texttt{:=insert}(h,k,v) & \{\mathsf{in}_{\mathsf{ins}}(h,k,v)_i\} \\
\{\mathsf{unk}(h,k)_i\} & r\texttt{:=insert}(h,k,v) & \{\mathsf{unk}(h,k)_i\} \\
\\
\{\mathsf{in}_{\mathsf{def}}(h,k,v)_1\} & r\texttt{:=remove}(h,k) & \{\mathsf{out}_{\mathsf{def}}(h,k)_1 \wedge r = \mathsf{tt}\} \\
\{\mathsf{out}_{\mathsf{def}}(h,k)_i\} & r\texttt{:=remove}(h,k) & \{\mathsf{out}_{\mathsf{def}}(h,k)_i \wedge r = \mathsf{ff}\} \\
\{\mathsf{in}_{\mathsf{rem}}(h,k,v)_i\} & r\texttt{:=remove}(h,k) & \{\mathsf{out}_{\mathsf{rem}}(h,k)_i\} \\
\{\mathsf{out}_{\mathsf{rem}}(h,k)_i\} & r\texttt{:=remove}(h,k) & \{\mathsf{out}_{\mathsf{rem}}(h,k)_i \wedge r = \mathsf{ff}\} \\
\{\mathsf{unk}(h,k)_i\} & r\texttt{:=remove}(h,k) & \{\mathsf{unk}(h,k)_i\}
\end{array}$$

AXIOMS:

$$\begin{array}{rcll}
X(h,k)_i * X(h,k)_j & \Leftrightarrow & X(h,k)_{i+j} & \text{if } i+j \leq 1 \\
X(h,k)_i * \mathsf{read}(h,k) & \Leftrightarrow & X(h,k)_i & \\
X_{\mathsf{def}}(h,k)_1 & \Leftrightarrow & X_{\mathsf{ins}}(h,k)_1 & \\
X_{\mathsf{ins}}(h,k)_1 & \Leftrightarrow & X_{\mathsf{rem}}(h,k)_1 & \\
X_{\mathsf{rem}}(h,k)_1 & \Rightarrow & \mathsf{unk}(h,k)_1 & \\
\mathsf{in}_{\mathsf{ins}}(h,k,v)_i & \Rightarrow & \mathsf{in}_{\mathsf{ins}}(h,k,-)_i & \\
\mathsf{in}_{\mathsf{ins}}(h,k,v)_i * \mathsf{out}_{\mathsf{ins}}(h,k)_j & \Rightarrow & \mathsf{in}_{\mathsf{ins}}(h,k,v)_{i+j} & \text{if } i+j \leq 1 \\
\mathsf{in}_{\mathsf{ins}}(h,k,v_1)_i * \mathsf{in}_{\mathsf{ins}}(h,k,v_2)_j & \Rightarrow & \mathsf{in}_{\mathsf{ins}}(h,k,-)_{i+j} & \text{if } i+j \leq 1 \\
\mathsf{in}_{\mathsf{rem}}(h,k,v)_i * \mathsf{out}_{\mathsf{rem}}(h,k)_j & \Rightarrow & \mathsf{out}_{\mathsf{rem}}(h,k)_{i+j} & \text{if } i+j \leq 1 \\
X(h,k)_i * X(h,k)_j & \Rightarrow & \mathsf{false} & \text{if } i+j > 1 \\
X_{tag}(h,k)_i * Y_{tag'}(h,k)_j & \Rightarrow & \mathsf{false} & \text{if } tag \neq tag' \\
\mathsf{unk}(h,k)_i * X_{tag}(h,k)_j & \Rightarrow & \mathsf{false} &
\end{array}$$

Figure 2: Full specification for concurrent indexes.

The proof starts with the predicate $\mathsf{out}_{\mathsf{def}}(h,k)_1$, which specifies that there is no mapping from key k in the index. The def subscript asserts that no other thread can modify the value mapped by this key. We use the following axiom to create a $\mathsf{read}(h,k)$ predicate:

$$X(h,k)_i \iff X(h,k)_i * \mathsf{read}(h,k)$$

This allows the left-hand thread to perform a simple `search` operation, although the postcondition establishes nothing about the result. This captures the fact that we do not know at which point during the `insert` operation the `search` operation will read the key's value. The $\mathsf{out}_{\mathsf{def}}(h,k)_1$ allows the right-hand thread to insert its value successfully, as we know that it is the only thread changing the shared state for the key k. When both threads finish their execution, we use the same axiom to merge the $\mathsf{read}(h,k)$ predicate back into the $\mathsf{in}_{\mathsf{def}}(h,k,v)_1$ predicate.

We can prove the second specification as follows:

$$\begin{array}{c}
\{\mathsf{in}_{\mathsf{def}}(h,k,v')_1\} \\
\{\mathsf{in}_{\mathsf{def}}(h,k,v')_{\frac{1}{2}} * \mathsf{in}_{\mathsf{def}}(h,k,v')_{\frac{1}{2}}\} \\
\{\mathsf{in}_{\mathsf{def}}(h,k,v')_{\frac{1}{2}}\} \quad \| \quad \{\mathsf{in}_{\mathsf{def}}(h,k,v')_{\frac{1}{2}}\} \\
r_1\texttt{:=search}(h,k) \quad \| \quad r_2\texttt{:=insert}(h,k,v) \\
\{\mathsf{in}_{\mathsf{def}}(h,k,v')_{\frac{1}{2}} \wedge r_1 = v'\} \quad \| \quad \{\mathsf{in}_{\mathsf{def}}(h,k,v')_{\frac{1}{2}} \wedge r_2 = \mathsf{ff}\} \\
\{\mathsf{in}_{\mathsf{def}}(h,k,v')_1 \wedge r_1 = v' \wedge r_2 = \mathsf{tt}\} \\
\{\mathsf{in}_{\mathsf{def}}(h,k,v')_1\}
\end{array}$$

Here we use the splitting axiom discussed on the previous page. Unlike the previous proof-sketch, the `insert` fails to insert the value, and instead returns false.

With the current work we can specify quite complex combinations of `insert`, `remove` and `search`. For example, consider the parallel composition of two removes:

$$r_1\texttt{:=remove}(h,k) \;\|\; r_2\texttt{:=remove}(h,k)$$

In this program, we do not know which deletion will succeed and which will fail, but we do know that there will definitely not be a mapping from key k afterwards. By splitting the predicates, we can communicate this knowledge between threads.

$$\begin{array}{c}
\{\mathsf{in}_{\mathsf{def}}(h,k,v)_1\} \\
\{\mathsf{in}_{\mathsf{rem}}(h,k,v)_{\frac{1}{2}} * \mathsf{in}_{\mathsf{rem}}(h,k,v)_{\frac{1}{2}}\} \\
\{\mathsf{in}_{\mathsf{rem}}(h,k,v)_{\frac{1}{2}}\} \quad \| \quad \{\mathsf{in}_{\mathsf{rem}}(h,k,v)_{\frac{1}{2}}\} \\
r_1\texttt{:=remove}(h,k) \quad \| \quad r_2\texttt{:=remove}(h,k) \\
\{\mathsf{out}_{\mathsf{rem}}(h,k)_{\frac{1}{2}}\} \quad \| \quad \{\mathsf{out}_{\mathsf{rem}}(h,k)_{\frac{1}{2}}\} \\
\{\mathsf{out}_{\mathsf{rem}}(h,k)_{\frac{1}{2}} * \mathsf{out}_{\mathsf{rem}}(h,k)_{\frac{1}{2}}\} \\
\{\mathsf{out}_{\mathsf{def}}(h,k)_1\}
\end{array}$$

We sometimes cannot establish the exact state of an index after a program has run. For example, consider the following program:

$$r_1\texttt{:=remove}(h,k) \;\|\; r_2\texttt{:=insert}(h,k,v)$$

Following the execution of the program will not know if there is a mapping from key k. However, using our specification, we can still establish that the program does not fault.

$$\begin{array}{c}
\{\mathsf{in}_{\mathsf{def}}(h,k,v')_1\} \\
\{\mathsf{unk}(h,k)_1\} \\
\{\mathsf{unk}(h,k)_{\frac{1}{2}} * \mathsf{unk}(h,k)_{\frac{1}{2}}\} \\
\{\mathsf{unk}(h,k)_{\frac{1}{2}}\} \quad \| \quad \{\mathsf{unk}(h,k)_{\frac{1}{2}}\} \\
r_1\texttt{:=remove}(h,k) \quad \| \quad r_2\texttt{:=insert}(h,k,v) \\
\{\mathsf{unk}(h,k)_{\frac{1}{2}}\} \quad \| \quad \{\mathsf{unk}(h,k)_{\frac{1}{2}}\} \\
\{\mathsf{unk}(h,k)_{\frac{1}{2}} * \mathsf{unk}(h,k)_{\frac{1}{2}}\} \\
\{\mathsf{unk}(h,k)_1\}
\end{array}$$

It also scales to more complex programs. For example, consider the following program containing parallel composition:

$$r\texttt{:=search}(h,k) \;\Big\|\; \begin{array}{l} \texttt{insert}(h,k,v)\,; \\ (\,\texttt{search}(h,k) \;\|\; \texttt{search}(h,k)\,)\,; \\ \texttt{remove}(h,k) \end{array}$$

We can verify this program as follows:

$$\begin{array}{c}
\{\mathsf{out}_{\mathsf{def}}(h,k)_1\} \\
\{\mathsf{read}(h,k) * \mathsf{out}_{\mathsf{def}}(h,k)_1\} \\
\{\mathsf{read}(h,k)\} \\
r\texttt{:=search}(h,k) \\
\{\mathsf{read}(h,k)\}
\end{array} \;\Big\|\; \begin{array}{c}
\{\mathsf{out}_{\mathsf{def}}(h,k)_1\} \\
\texttt{insert}(h,k,v) \\
\{\mathsf{in}_{\mathsf{def}}(h,k,v)_1\} \\
\{\mathsf{in}_{\mathsf{def}}(h,k,v)_{\frac{1}{2}} * \mathsf{in}_{\mathsf{def}}(h,k,v)_{\frac{1}{2}}\} \\
\{\mathsf{in}_{\mathsf{def}}(h,k,v)_{\frac{1}{2}}\} \quad \| \quad \{\mathsf{in}_{\mathsf{def}}(h,k,v)_{\frac{1}{2}}\} \\
\texttt{search}(h,k) \quad \| \quad \texttt{search}(h,k) \\
\{\mathsf{in}_{\mathsf{def}}(h,k,v)_{\frac{1}{2}}\} \quad \| \quad \{\mathsf{in}_{\mathsf{def}}(h,k,v)_{\frac{1}{2}}\} \\
\{\mathsf{in}_{\mathsf{def}}(h,k,v)_{\frac{1}{2}} * \mathsf{in}_{\mathsf{def}}(h,k,v)_{\frac{1}{2}}\} \\
\{\mathsf{in}_{\mathsf{def}}(h,k,v)_1\} \\
\texttt{remove}(h,k) \\
\{\mathsf{out}_{\mathsf{def}}(h,k)_1\}
\end{array}$$

$$\begin{array}{c}
\{\mathsf{read}(h,k) * \mathsf{out}_{\mathsf{def}}(h,k)_1\} \\
\{\mathsf{out}_{\mathsf{def}}(h,k)_1\}
\end{array}$$

Using our specification, we establish that there is definitely no mapping from key k at the end of the program. Here we have dropped return values to concentrate on the key k. If we tracked them we would also show that, whilst the leftmost search has an unknown return value, both of the other two searches return the same value v. If there was a mapping from key k at the start of the program a similar proof would show that there would still definitely not be a mapping from key k at the end of the program.

4.1 Example: Range Search

A database transaction manager will use concurrent indexes at various points in its code. We can use our specification to ensure that the index is used correctly in the transaction manager. In particular, we can ensure that a transaction manager complies with the standard levels of isolation (such as serialisability) when using the index. If the transaction manager can acquire an appropriate set of predicates, then we can then verify that no other thread can break the level of isolation.

We will consider a simple example, rather than the code of an actual implementation. Consider the function `rangeSearch`:

```
r := rangeSearch(h, v1, v2) {
  r := listNew();
  for(i := v1; i ≤ v2; i++) {
    temp := search(h, i);
    listAdd(r, temp);
} }
```

`rangeSearch` returns a list of the values held in the keys in a the range from k_1 to k_2. It represents a small part of the internal behaviour of a transaction, triggered by an SQL SELECT query for example. Here we assume that `rangeSearch` has already acquired the appropriate level of access to the index. A full transaction manager would require a locking mechanism for controlling and distributing access to keys, but we leave specifying this to future work.

Reasoning about transactions that operate over disjoint sets of keys is simple in our system. Each transaction takes full permission on all of the keys it requires to operate. The more interesting case is where keys may be accessed concurrently by multiple transactions. We assume that `range-`

`Search` is a part of a database management system supporting a variety of isolation levels. There are four major isolation levels defined by ANSI/ISO SQL [1]:

1. *Serialisable* transactions occur in complete isolation. The transaction locks all the keys it reads and writes, and also all locks all absent keys that *would* have satisfied one of its conditionals. This corresponds to full permission ($i = 1$) on all keys used in the transaction.
2. *Repeatable Read* transactions lock all keys that are read and written, but not absent keys. This means that new keys matching a conditional can appear in the index after the start of a transaction. Consequently, threads are not isolated from one another; *phantom reads* may occur. We can capture this behaviour using the ins predicate. This specifies that other threads can insert a value after a thread has read the key.
3. *Read Committed* transactions lock writes to the end of the transaction, but release locks on reads immediately. Consequently a transaction may read several different values for the same key during a transaction. We can capture this behaviour using the unk predicate.
4. *Read Uncommitted* transactions release both read and write locks immediately, allowing dirty reads. A transaction may see uncommitted changes by other transactions. We can capture this behaviour using the read predicate, which allows a thread to read but not modify a key.

Recall that our predicates capture the level of interference allowed by other threads. This notion of permitted interference is exactly the idea behind the isolation levels of a transaction.

To ensure a transaction is *Serialisable*, the transaction must hold def predicates for all the keys in the range of the search. We have the following specification for the range search:

$$\left\{\circledast_{v_1 \le k \le v_2}.\, \mathsf{out}_{\mathsf{def}}(h,k)_i \lor \exists p.\, \mathsf{in}_{\mathsf{def}}(h,k,p)_i\right\}$$
$$r := \texttt{rangeSearch}(h, v_1, v_2)$$
$$\left\{\begin{array}{r}\circledast_{v_1 \le k \le v_2}.\, \mathsf{out}_{\mathsf{def}}(h,k)_i \land r[k-v_1+1] = \mathsf{nil} \\ \lor \exists p.\, \mathsf{in}_{\mathsf{def}}(h,k,p)_i \land r[k-v_1+1] = p\end{array}\right\}$$

(For simplicity, in this this specification we have chosen to existentially quantify the actual values associated with the keys in the range.)

If we instead assume the *Read Uncommitted* isolation level, then we have a different specification:

$$\left\{\circledast_{v_1 \le k \le v_2}.\, \mathsf{read}(h,k)_i\right\}$$
$$r := \texttt{rangeSearch}(h, v_1, v_2)$$
$$\left\{\circledast_{v_1 \le k \le v_2}.\, \mathsf{read}(h,k)_i\right\}$$

Here, can only acquire a read permission on the keys, as other threads are allowed to modify the concurrent index while the range search is taking place. Obviously this level of isolation leads to a much weaker postcondition where we only know that the operation did not fault.

In a similar way we can give specifications corresponding to the other isolation levels. The level of isolation affects the properties we can establish about a program. In the worst case, we can establish only that an operation will not fault.

$\{\mathsf{emp}\}$
```
sieve(int max) = {
  tid *thr;
  thr := malloc(sizeof(tid) * (sqrt(max)-2)) - 2;
  idx := idxrange(1,max);
```
$\left\{\circledast_{2 \le i \le \sqrt{\texttt{max}}}.\, \texttt{thr}[i] \mapsto - * \circledast_{j \in [1\,..\,\texttt{max}]}.\, \mathsf{in}_{\mathsf{rem}}(\texttt{idx}, j, 0)_1\right\}$
```
  for(tc := 2; tc ≤ sqrt(max); tc++) {
    thr[tc] := fork(worker(tc,max,*idx));
```
$\left\{\begin{array}{r}\circledast_{2 \le i \le \texttt{tc}}.\, \texttt{thr}[i] \mapsto t * \texttt{thr}\left(t,\, \mathsf{wpst}(i, \texttt{max}, \texttt{idx})_{\frac{1}{\sqrt{\texttt{max}}-1}}\right) \\ * \circledast_{j \in [1\,..\,\texttt{max}]}.\, \mathsf{in}_{\mathsf{rem}}(\texttt{idx}, j, 0)_{\frac{\sqrt{\texttt{max}}-\texttt{tc}}{\sqrt{\texttt{max}}-1}} \\ * \circledast_{\texttt{tc} < k \le \sqrt{\texttt{max}}}.\, \texttt{thr}[k] \mapsto \_\end{array}\right\}$
```
    tc := tc+1;
  }
  for(i := 0; i < length(thr); i++) join(thr[i]);
```
$\left\{\circledast_{2 \le i \le \sqrt{\texttt{max}}}.\, \mathsf{wpst}(i, \texttt{max}, \texttt{idx})_{\frac{1}{\sqrt{\texttt{max}}-1}} * \texttt{thr}[i] \mapsto \_\right\}$
```
  free(thr+2, sqrt(max)-2);
```
$\left\{\circledast_{2 \le i \le \sqrt{\texttt{max}}}.\, \mathsf{wpst}(i, \texttt{max}, \texttt{idx})_{\frac{1}{\sqrt{\texttt{max}}-1}}\right\}$
```
  // By module axioms and CONSEQ - see Lemma 1 below.
```
$\left\{\begin{array}{r}\circledast_{i \le \texttt{max}}.\, \mathsf{isPrime}(i) \Rightarrow \mathsf{in}_{\mathsf{rem}}(\texttt{idx}, i, 0)_1 \\ \land\, \neg\mathsf{isPrime}(i) \Rightarrow \mathsf{out}_{\mathsf{def}}(\texttt{idx}, i)_1\end{array}\right\}$
```
  return *idx;
}
```
$\left\{\begin{array}{l}\texttt{ret} = \texttt{idx} \land \circledast_{i \in [1\,..\,\texttt{max}]}.\, \mathsf{isPrime}(i) \Rightarrow \mathsf{in}_{\mathsf{rem}}(\texttt{idx}, i, 0)_1 \\ \qquad\qquad\qquad\qquad \land\, \neg\mathsf{isPrime}(i) \Rightarrow \mathsf{out}_{\mathsf{def}}(\texttt{idx}, i)_1\end{array}\right\}$

$\left\{\circledast_{i \in [1\,..\,\texttt{max}]}.\, \mathsf{in}_{\mathsf{rem}}(\texttt{idx}, i, 0)_t\right\}$
```
worker(int v, int max, index *idx) = {
 c := v;
 do {
```
$\left\{\begin{array}{r}\circledast_{i \in [1\,..\,(\texttt{c}-1)]}.\, (i \ne v \land (i \bmod v) = 0) \Rightarrow \mathsf{out}_{\mathsf{def}}(\texttt{idx}, i)_t \land \\ (i = v \lor (i \bmod v) \ne 0) \Rightarrow \mathsf{in}_{\mathsf{rem}}(\texttt{idx}, i, 0)_t \\ * \circledast_{j \in [\texttt{c}\,..\,\texttt{max}]}.\, \mathsf{in}_{\mathsf{rem}}(\texttt{idx}, j, 0)_t\end{array}\right\}$
```
   c := c + v;
   remove(idx,c);
 } while(c < max)
}
```
$\left\{\begin{array}{r}\circledast_{i \in [1\,..\,\texttt{max}]}.\, (i \ne v \land (i \bmod v) = 0) \Rightarrow \mathsf{out}_{\mathsf{def}}(\texttt{idx}, i)_t \land \\ (i = v \lor (i \bmod v) \ne 0) \Rightarrow \mathsf{in}_{\mathsf{rem}}(\texttt{idx}, i, 0)_t\end{array}\right\}$

Figure 3: Proofs for the `sieve` and `worker` programs.

4.2 Example: The Sieve of Eratosthenes

Let us consider an example where many threads require write access to the same shared value in a concurrent index. We choose the Sieve of Eratosthenes [2, 9], an algorithm for generating all of the prime numbers up to a given maximum value `max`, which is representative of a database operation that involves a high level of sharing on keys.

The algorithm works as follows. A shared index is constructed with mappings from all of the keys 1..`max` to an arbitrary value (we choose zero). Then for every i in the interval 2..$\sqrt{\texttt{max}}$ a worker thread is created. Each worker thread removes the mappings from the index from all the keys with multiples of its i. When all worker threads finish, the concurrent index contains only mappings from the keys which are prime. The correctness of the algorithm results from the fact that any non-prime in 1..`max` must have a factor in 2..$\sqrt{\texttt{max}}$.

The main function `sieve` is defined as follows. Here `tc`

counts the number of worker threads, and the array `thr` is used to keep track of thread identifiers for forked worker threads.

```
sieve(int max) = {
  tid *thr;
  thr := malloc(sizeof(tid) * (sqrt(max)-2)) - 2;
  idx := idxrange(1,max);
  for(tc := 2; tc ≤ sqrt(max); tc++) {
    thr[tc] := fork(worker(tc,max,*idx));
    tc := tc+1;
  }
  for(i := 0; i < length(thr); i++) join(thr[i]);
  free(thr+2, sqrt(max)-2);
  return *idx;
}
```

Intuitively, we first generate a set of integers from 1 to the maximum value required. We generate worker threads for value in the range $2..\sqrt{\texttt{max}}$, which filters out non-primes. Finally the sieve joins on each of the threads, and returns the resulting list of primes.

Worker threads are defined as follows:

```
worker(int v, int max, index *idx) = {
  c := v;
  do {
    c := c + v;
    remove(idx,c);
  } while(c < max)
}
```

We first give a specification for the worker thread. Each worker thread starts with an $\mathsf{in}_{\mathsf{rem}}$ predicate for keys in the range $1..\mathrm{max}$, and calls remove on just the keys that correspond to multiples of its first argument. The result is the following specification:

$$\left\{\circledast_{i\in[1\,..\,m]}.\ \mathsf{in}_{\mathsf{rem}}(idx,i,0)_t\right\}$$
$$\texttt{worker}(v,m,idx)$$
$$\left\{\begin{array}{r}\circledast_{i\in[1\,..\,m]}.\ (i\neq v\wedge(i\bmod v)=0)\Rightarrow \mathsf{out}_{\mathsf{rem}}(idx,i)_t\ \wedge\\ (i=v\vee(i\bmod v)\neq 0)\Rightarrow \mathsf{in}_{\mathsf{rem}}(idx,i,0)_t\end{array}\right\}$$

We summarise the worker thread postcondition as a predicate wpst in order to make the proof clearer:

$$\mathsf{wpst}(v,m,idx)_t \triangleq$$
$$\circledast_{i\in[1\,..\,m]}.\ (i\neq v\wedge(i\bmod v)=0)\Rightarrow \mathsf{out}_{\mathsf{rem}}(idx,i)_t\ \wedge$$
$$(i=v\vee(i\bmod v)\neq 0)\Rightarrow \mathsf{in}_{\mathsf{rem}}(idx,i,0)_t$$

Proofs of `worker` and `sieve` are given in Fig 3. The proof of `worker` is relatively straightforward. The mappings are removed if the keys are multiples of the input value.

The proof of `sieve` is a little more subtle. The main sieve algorithm has the following specification:

$$\{\mathsf{emp}\}$$
$$x\texttt{:= sieve(max)}$$
$$\left\{\begin{array}{r}\circledast_{i\leq\texttt{max}}.\ \mathsf{isPrime}(i)\Rightarrow \mathsf{in}_{\mathsf{def}}(idx,i,0)_1\\ \wedge\neg\mathsf{isPrime}(i)\Rightarrow \mathsf{out}_{\mathsf{def}}(idx,i)_1\end{array}\right\}$$

(Here 'emp' asserts that the thread holds no resources.)

To reason about `fork` and `join` we use the following proof rules, taken from [8]:

$$\frac{\{P\}\ \mathbb{C}\ \{Q\}}{\{P\}\ \ t\ \texttt{:=fork}(\mathbb{C})\ \ \{\mathsf{thr}(t,Q)\}}\qquad\frac{}{\{\mathsf{thr}(t,Q)\}\ \ \texttt{join}(t)\ \ \{Q\}}$$

Intuitively, calling `fork` on a command $\mathbb{C}$ requires that we prove a specification $\{P\}\,\mathbb{C}\,\{Q\}$, and that the thread holds the resource P. After calling `fork` the thread loses P, but gains a predicate $\mathsf{thr}(t,Q)$ representing the forked thread's postcondition. Calling `join` removes the thr predicate but give back the postcondition Q.

The function begins by constructing a shared index holding values for the keys in the range from 1 to `max`. It therefore holds a predicate $\mathsf{in}_{\mathsf{def}}(idx,v,0)_1$ for each value in the range. It gives each of the worker threads an $\mathsf{in}_{\mathsf{rem}}$ predicate with a fractional value $\frac{1}{\sqrt{\texttt{max}}-1}$. Each of the worker threads can consequently remove any value from the shared index. Once all of the workers have returned, the predicates are recombined. To establish the postcondition we use the following lemma.

LEMMA 1. $\quad\circledast_{2\leq j\leq\sqrt{m}}.\ \mathsf{wpst}(j,m,idx)_{\frac{1}{\sqrt{m}-1}}$

implies $\quad\circledast_{i\in[1\ldots m]}.\ \mathsf{isPrime}(i)\Rightarrow \mathsf{in}_{\mathsf{def}}(idx,i,0)_1$
$\wedge\ \neg\mathsf{isPrime}(i)\Rightarrow \mathsf{out}_{\mathsf{def}}(idx,i)_1.$

PROOF. Consequence of the fact that every non-prime i has a factor in $2..\sqrt{i}$. See Appendix A for details. It is unavoidable that verifying this algorithm depends on the arithmetic properties of prime numbers. □

4.3 Extension: Handling the Unknown

With the specifications presented above, once we get to an unknown state, as in the parallel `remove` and `insert` example, we are stuck. This means that if there is a race condition, where we cannot determine the contents of the index for some key, from this point onward we can only prove safety with respect to this key. We cannot establish what the value associated with the key might be.

However, we are sometimes able to recover from an unknown state because we can work out what the contents are. For example, if all the threads of a program end deleting a mapping from a certain key, then we can be sure that there is no mapping from this key in the index at the end of the program. We can extend the unknown predicate $\mathsf{unk}(h,k)_i$ of our system by adding tags which record the last write operation carried out by the thread.

$\mathsf{unk}_{\mathsf{ins}}(h,k)_i$:	nothing is known about the key k except the last write operation carried out by the thread on that key was an insert
$\mathsf{unk}_{\mathsf{rem}}(h,k)_i$:	nothing is known about the key k except the last write operation carried out by the thread on that key was a remove

We then use these predicates when an operation is called in an unknown state. In particular, we extend our specifications as following:

$$\{\mathsf{unk}_X(h,k)_i\}\ \ r\texttt{:=search}(h,k)\ \ \{\mathsf{unk}_X(h,k)_i\}$$
$$\{\mathsf{unk}_X(h,k)_i\}\ \ r\texttt{:=insert}(h,k,v)\ \ \{\mathsf{unk}_{\mathsf{ins}}(h,k)_i\}$$
$$\{\mathsf{unk}_X(h,k)_i\}\ \ r\texttt{:=remove}(h,k)\ \ \{\mathsf{unk}_{\mathsf{rem}}(h,k)_i\}$$

where unk_X represents any one of unk, $\mathsf{unk}_{\mathsf{ins}}$ or $\mathsf{unk}_{\mathsf{rem}}$. Note that the specification for search simply maintains the tag on the unk predicate. We can use these new predicates to recover from an unknown state to a definite one, once we have full permission on that key again. In particular, we

add the following axioms to our reasoning system.

$$\begin{aligned}
\mathsf{unk}_X(h,k)_i &\implies \mathsf{unk}(h,k)_i \\
\mathsf{unk}_{\mathsf{ins}}(h,k)_1 &\implies \mathsf{in}_{\mathsf{def}}(h,k,-)_1 \\
\mathsf{unk}_{\mathsf{rem}}(h,k)_1 &\implies \mathsf{out}_{\mathsf{def}}(h,k)_1 \\
\mathsf{unk}_{\mathsf{ins}}(h,k)_i * \mathsf{unk}_{\mathsf{rem}}(h,k)_j &\implies \mathsf{unk}(h,k)_{i+j} \quad \text{if } i+j \leq 1
\end{aligned}$$

As an example of how these predicates are used, consider the following example program (again we drop the return values to concentrate on the mapping from key k).

$$\begin{array}{c}
\{\mathsf{in}_{\mathsf{def}}(h,k,v)_1\} \\
\{\mathsf{unk}(h,k)_1\} \\
\left\{\mathsf{unk}(h,k)_{\frac{1}{3}} * \mathsf{unk}(h,k)_{\frac{1}{3}} * \mathsf{unk}(h,k)_{\frac{1}{3}}\right\} \\
\left\{\mathsf{unk}(h,k)_{\frac{1}{3}}\right\} \quad \left\{\mathsf{unk}(h,k)_{\frac{1}{3}}\right\} \quad \left\{\mathsf{unk}(h,k)_{\frac{1}{3}}\right\} \\
\mathtt{search}(h,k) \quad \mathtt{insert}(h,k,a) \quad \mathtt{insert}(h,k,b) \\
\left\{\mathsf{unk}(h,k)_{\frac{1}{3}}\right\} \quad \left\{\mathsf{unk}_{\mathsf{ins}}(h,k)_{\frac{1}{3}}\right\} \quad \left\{\mathsf{unk}_{\mathsf{ins}}(h,k)_{\frac{1}{3}}\right\} \\
\mathtt{remove}(h,k) \quad \mathtt{remove}(h,k) \quad \mathtt{remove}(h,k) \\
\left\{\mathsf{unk}_{\mathsf{rem}}(h,k)_{\frac{1}{3}}\right\} \quad \left\{\mathsf{unk}_{\mathsf{rem}}(h,k)_{\frac{1}{3}}\right\} \quad \left\{\mathsf{unk}_{\mathsf{rem}}(h,k)_{\frac{1}{3}}\right\} \\
\left\{\mathsf{unk}_{\mathsf{rem}}(h,k)_{\frac{1}{3}} * \mathsf{unk}_{\mathsf{rem}}(h,k)_{\frac{1}{3}} * \mathsf{unk}_{\mathsf{rem}}(h,k)_{\frac{1}{3}}\right\} \\
\{\mathsf{unk}_{\mathsf{rem}}(h,k)_1\} \\
\{\mathsf{out}_{\mathsf{def}}(h,k)_1\}
\end{array}$$

We are forced to convert the def predicate into an unk predicate as there are multiple inserts and removes occurring in parallel on the key k. However, since we know that all of the threads end with a deletion on this key, we can show that there will definitely be no mapping from this key at the end of the program.

The aim of our specifications is to allow us to formally prove properties of programs that we can deduce intuitively from the code. In particular, for the mapping of keys to values, we believe that our specification is optimal in the sense that it captures all of the available key data in the underlying index model. Note that this is not the case for the values mapped by the keys as we often lose this information.

5. AN INDEX IMPLEMENTATION BASED ON B^{LINK} TREES

The work presented here is defined using the concurrent abstract predicates approach. Consequently, we can verify that an implementation of an index is correct with respect to our abstract specification. This is one of the major advantages of this approach.

To prove that an implementation, Sagiv's B^{Link} tree [13], conforms to our specification, we give concrete interpretations of each of our abstract predicates, and prove the correctness of the implementation with respect to these interpretations. In order to give formal proofs we fully implement the parts of Sagiv's algorithms originally given as pseudocode and also code up his extensions that cover various corner cases. (Full details of the implementation and the proof are given in [5].)

A B^{Link} tree is a balanced search tree, an example of which is shown in Fig 4. Search operations run on the B^{Link} tree are lock-free, and insert and remove operations lock only one node (or two if they are modifying the root node), making this a highly concurrent index implementation. The tree is accessed through a prime block which holds pointers to the first node at each level in the tree. Each node in the tree contains an ordered list of key-value pairs, which at the leaves form the contents of the B^{Link} tree. Non-leaf nodes contain pointers that make up the search structure of the tree. This structure ensures that every key-value pair stored in the tree can be reached starting from the leftmost node at any level of the tree. If the value cannot be found by following a pointer down the tree, it can be found by following the rightward sibling pointer. This is important because insertion operations can create new nodes that can only be reached via the sibling pointers until the higher levels of the tree are later repaired by the operation.

5.1 Defining the B^{Link} Tree Data Structure

To begin the proof, we first define a series of predicates representing the concrete B^{Link} tree data structure. We assume two basic predicates for representing nodes in the tree: a leaf predicate, and an inner predicate.

$$x \mapsto \mathsf{leaf}(t,k,S,k',p') \qquad x \mapsto \mathsf{inner}(t,k,p,S,k',p')$$

Here, x is the address of the node. The value t holds the thread identifier of the thread locking the node; if the node is unlocked it is 0. The ordered list S contains the key-value pairs (k,v) represented by the node. The values k and k' are the lower and upper bound, respectively, on the keys contained in this list. The pointer p points to the subtree which contains all of the keys which are greater than the minimum value of this node. The pointer p', known as the link pointer, points to the node's right sibling, if it exists.

A B^{Link} tree is a superimposed structure made up of both a tree and several layers of linked lists. At the leaf level the linked list contains pointers to data entries, while at other levels the linked list contains pointers to nodes deeper in the tree structure. These linked lists always have at least one element, the first node has minimum value $-\infty$ and the last node has maximum value $+\infty$. Each node in these linked lists is disjoint, so we can use a separation logic predicate to define this structure precisely. Given key-value lists T and D, which contain the key-value pairs that point into the current level of the tree and the key-value pairs contained in the current level of the tree respectively, we can define the linked list structure for a layer of the B^{Link} tree. Let $T = [(k_1,v_1),...,(k_n,v_n)]$ then,

$$\begin{aligned}
\mathsf{leafList}(T,D) \triangleq\ & \exists S_1,...,S_n. \\
& \circledast_{i=1}^{n-1} v_i \mapsto \mathsf{leaf}(-,k_i,S_i,k_{i+1},v_{i+1}) \\
& * v_n \mapsto \mathsf{leaf}(-,k_n,S_n,+\infty,\mathsf{nil}) \\
& \wedge k_1 = -\infty \wedge n > 0 \wedge D = \textstyle\bigcup_{i=1}^{n} S_i
\end{aligned}$$

$$\begin{aligned}
\mathsf{innerList}(T,D) \triangleq\ & \exists S_1,...,S_n. \\
& \circledast_{i=1}^{n-1} v_i \mapsto \mathsf{inner}(-,k_i,v_i',S_i,k_{i+1},v_{i+1}) \\
& * v_n \mapsto \mathsf{inner}(-,k_n,v_n',S_n,+\infty,\mathsf{nil}) \\
& \wedge k_1 = -\infty \wedge n > 0 \\
& \wedge D = \textstyle\bigcup_{i=1}^{n} (k_i,v_i') \cup S_i
\end{aligned}$$

We choose not to define the tree structure directly, as at some points in time the tree structure of the B^{Link} tree can actually be broken by the `insert` operation. When the `insert` operation creates a new node in the tree, it is added to the linked list structure before it is given a reference in the layer above. If the `search` operation did not use the link pointers as well as the tree pointers, it would not be able to find this new node at this point in time. To capture this

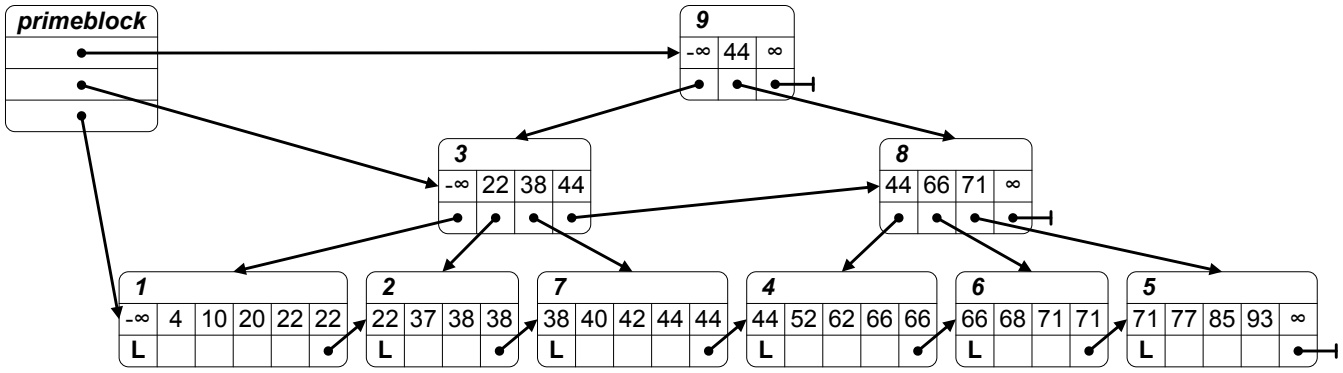

Figure 4: A full B^{Link} tree.

behaviour we instead choose to build up our tree predicate by layering our lists on top of one another. Using our linked lists predicates, we can build up a predicate for the tree-like structure of the B^{Link} tree.

$$\mathsf{Btree}_1(PB, x :: T, D) \triangleq \exists p.\, \mathsf{leafList}(x :: T, D) \wedge x = (-\infty, p) \wedge PB = [p]$$

$$\mathsf{Btree}_{n+1}(p :: PB, x :: T, D) \triangleq \exists L, L'.\, \mathsf{innerList}(x :: T, L) * \mathsf{Btree}_n(PB, L', D) \wedge x = (-\infty, p) \wedge L \subseteq L'$$

The prime block PB contains a list of pointers to the leftmost node at each level of the tree. The key-value list D is the concatenation of all key-value pairs at the fringe of the tree and corresponds to our abstract index view of the B^{Link} tree structure.

Finally, using these predicates, we can now define a predicate for the complete B^{Link} tree structure.

$$\mathsf{BLTree}(h, D) \triangleq \exists PB, n, T.\, \mathsf{Btree}_n(PB, T, D) * h \mapsto PB$$

This describes a B^{Link} tree whose prime block is stored at address h and contains a set of key-value pairs D. Figure 4 shows an example of a B^{Link} tree. The fringe of the tree forms a leafList that contains all of the key-value pairs mapped to by the index. Each of the other layers of the tree forms an innerList that makes up the search structure of the tree. Each list has minimum value $-\infty$ and maximum value ∞ and the primeblock points to the head of each layer's list.

5.2 Defining the Abstract Predicates

Now that we have a predicate for a B^{Link} tree, we turn our attention to providing concrete interpretations of our abstract predicates for concurrent indexes. For example, we give the concrete interpretation of the $\mathsf{in}_{\mathsf{def}}$ predicate as:

$$\mathsf{in}_{\mathsf{def}}(h, k, v)_i \triangleq \exists D.\, \boxed{\mathsf{BLTree}(h, D) \wedge (k, v) \in D}^b_{\mathcal{A}} * [\textsc{Change}(k)]^b_i$$

The definition for the predicate describes two different parts of the program state. The boxed assertion talks about the shared state in a region labeled b. The assertion says that the region b contains a B^{Link} tree at h and that the key-value pair (k, v) is contained somewhere in this tree. Boxed assertions on the same region behave additively under $*$, that is:

$$\boxed{P}^b * \boxed{Q}^b \iff \boxed{P \wedge Q}^b$$

The boxed assertion is also parameterised by an interference environment $\mathcal{A}$ which describes the possible effects of other threads on the shared state.

The second part of the definition for the predicate talks about the local (or private) state of the current thread. In this case the local state contains a token $[\textsc{Change}(k)]$ which is associated with shared region b and has permission i. This token allows the current thread to perform certain actions on the shared state in region b. In the case of this $[\textsc{Change}(k)]$ token, the thread may read the value at key k if it has permission $i > 0$. If the thread has full permission on the $[\textsc{Change}(k)]$ token, i.e. $i = 1$, then it is also allowed to modify (insert or remove) the value at key k. The token also describes the possible interference from the environment on this key. If the thread has a partial permission on this token ($0 < i < 1$), then it knows that no other thread can be concurrently modifying the value at this key, since no thread can have full permission on the $[\textsc{Change}(k)]$ token. If the thread has full permission on the token ($i = 1$), then it knows that no other thread is even reading this key. The concrete interpretations of the other in predicates have much the same structure.

$$\mathsf{in}_{\mathsf{ins}}(h, k, v)_i \triangleq \exists D, v'.\, (i = 1 \implies v = v') \wedge \boxed{\mathsf{BLTree}(h, D) \wedge (k, v') \in D}^b_{\mathcal{A}} * [\textsc{Insert}(k)]^b_i$$

$$\mathsf{in}_{\mathsf{rem}}(h, k, v)_i \triangleq \exists D.\, (i = 1 \implies (k, v) \in D) \wedge \boxed{\mathsf{BLTree}(h, D)}^b_{\mathcal{A}} * [\textsc{Remove}(k)]^b_i$$

The main difference here is that what we know about the shared state is dependent on the permission value i. If $i < 1$ then in the ins case it is possible that some other thread may have inserted a different value into the shared state for the key k and in the rem case it is possible that some other thread may have removed the value of key k. In the first case we do not know the value mapped from key k and in the second case we do not even know if there is a mapping from the key. However, when $i = 1$ we know that no other thread is able to modify the key in question so the key k definitely has the value v in the shared state. Also note that the thread's local state contains different tokens than in the def case. The $[\textsc{Insert}(k)]$ token allows the current thread, and the environment, to read or write to the value of key k in the shared state, but no thread is allowed to perform a remove operation. The $[\textsc{Remove}(k)]$ token allows the current thread, or the environment, to read or remove the value of key k in the shared state, but no thread is allowed to perform

an insert operation. The concrete interpretations of the out, unk and read predicates are defined in a similar fashion.

The interference environment $\mathcal{A}$ is defined in terms of a set of actions $(P \rightsquigarrow Q)$ which describe how the shared state may be modified by the program and its environment. For example, in the B^{Link} tree implementation a thread may remove a key-value pair from the index with the following action:

$$\begin{pmatrix} x \mapsto \mathsf{node}(t_{id}, k, S, k', y) \\ * [\textsc{Change}(k'')]_1^b * \\ [\textsc{Unlock}(x)]_1^l \wedge (k'', v) \in S \end{pmatrix} \rightsquigarrow \begin{pmatrix} x \mapsto \mathsf{node}(t_{id}, k, S', k', y) \\ * [\textsc{Mod}(x, k'')]_1^b \\ \wedge S' = S \setminus (k'', v) \end{pmatrix}$$

(Here node is defined as the disjunction of inner and leaf.)

The left-hand side of the action describes part of the shared state before the action and the right-hand side describes how this part of the shared state is modified by the action. The remove action above requires that there is a node x in the shared state that has been locked by the thread attempting to perform this action (with thread identifier t_{id}). This node must contain an ordered set of key-value pairs S, which includes a pair (k, v), and whose keys are greater than the node's minimum value k and less than or equal to the node's maximum value k'. The node also contains a pointer y to the next node in the list. The action then removes the key-value pair (k'', v) from the set S, but leaves the rest of the node unchanged.

Actions do not mention a thread's local state, they only describe how the shared state is modified, so there is some token transfer taking place here as well. In particular the $[\textsc{Mod}(x, k)]$ token is moved from the thread's local state to the shared state and the $[\textsc{Change}(k)]$ and $[\textsc{Unlock}(x)]$ tokens are moved from the shared state to the thread's local state. This token transfer corresponds to the abstract level permission and predicate system. In this example the current thread is giving up the right to modify the value of key k in the node x and it is gaining the right to change the value of the key k in the set as well as the right to unlock the node x. The token transfer required to gain the $[\textsc{Mod}(x, k)]$ token (gain the right to modify the value of key k in node x in the def environment) is an action that can only be performed if a thread has full permission on both the $[\textsc{Change}(k)]$ and $[\textsc{Unlock}(x)]$ tokens.

$$\forall x.\, [\textsc{Mod}(x, k)]_1^b \rightsquigarrow [\textsc{Unlock}(x)]_1^l * [\textsc{Change}(k)]_1^b$$

That is it must have the exclusive right to modify the value of key k in the index and also have the node x locked. This captures the fact that in the def environment, if a thread wants to perform a write operation on some key it must have full permission on that key.

At the abstract level, when we have full permission on a key we can change how that key is treated. This is represented in the interference environment by providing actions that let us exchange tokens with the shared state. We have already seen how we may exchange the $[\textsc{Change}(k)]$ token to acquire the right to modify a node in the B^{Link} tree. When we have full permission on this token we can also exchange it for tokens that allow different levels of interference on the shared state. We have the following actions:

$$[\textsc{Insert}(k)]_1^b \rightsquigarrow [\textsc{Change}(k)]_1^b$$
$$[\textsc{Remove}(k)]_1^b \rightsquigarrow [\textsc{Insert}(k)]_1^b$$

Recall that these actions only describe the change to the shared state, so the first action allows a thread to swap the $[\textsc{Change}(k)]$ token for the $[\textsc{Insert}(k)]$ token and the second action allows a thread to swap the $[\textsc{Insert}(k)]$ token for the $[\textsc{Remove}(k)]$ token, so long as the thread has full permission on these tokens. Similar actions exist for swapping the tokens back in the other direction. As we discussed when defining the concrete interpretations of our predicates, when a thread owns one of these tokens it affects the actions that can be carried out by the whole system. For a full description of the interference environment $\mathcal{A}$ and the actions contained within it see [5].

$\{\mathsf{in}_{\mathsf{def}}(h, k, v)_i\}$
```
search(h, k) {
```
$\left\{\exists D.\, \boxed{\mathsf{BLTree}(h, D) \wedge (k, v) \in D}_{\mathcal{A}}^{b} * [\textsc{Change}(k)]_i^b\right\}$
```
  PB := getPrimeBlock(h);
  current := root(PB);
  A := get(current);
```
$\left\{\begin{array}{l}\exists D, k', k'', s, h'.\, \boxed{\mathsf{BLTree}(h, D) \wedge (k, v) \in D}_{\mathcal{A}}^{b} * [\textsc{Change}(k)]_i^b \\ \wedge\ \mathtt{current} = h' \wedge \mathtt{A} = \mathsf{node}(-, k', s, k'', -)\end{array}\right\}$
```
  while(isLeaf(A) = false) {
    current := next(A, k);
    A := get(current);
  }
```
$\left\{\begin{array}{l}\exists D, k', k'', s, h'.\, \boxed{\mathsf{BLTree}(h, D) \wedge (k, v) \in D}_{\mathcal{A}}^{b} * [\textsc{Change}(k)]_i^b \\ \wedge\ \mathtt{current} = h' \wedge \mathtt{A} = \mathsf{leaf}(-, k', s, k'', -)\end{array}\right\}$
```
  while(k > highValue(A)) {
    current := next(A, k);
    A := get(current);
  }
```
$\left\{\begin{array}{l}\exists D, k', k'', s, h'.\, \boxed{\mathsf{BLTree}(h, D) \wedge (k, v) \in D}_{\mathcal{A}}^{b} * [\textsc{Change}(k)]_i^b \\ \wedge\ \mathtt{current} = h' \wedge \mathtt{A} = \mathsf{leaf}(-, k', s, k'', -) \wedge k \leq k''\end{array}\right\}$
```
  if(isIn(A, k)) {
    return( lookup(A, k) );
  } else {
    return null;
```
$\{\mathsf{false}\}$
```
  }
```
$\left\{\begin{array}{l}\exists D, k', k'', s, h'.\, \boxed{\mathsf{BLTree}(h, D) \wedge (k, v) \in D}_{\mathcal{A}}^{b} * [\textsc{Change}(k)]_i^b \\ \wedge\ \mathtt{current} = h' \wedge \mathtt{A} = \mathsf{leaf}(-, k', s, k'', -) \wedge k \leq k'' \wedge \mathrm{ret} = v\end{array}\right\}$
```
}
```
$\{\mathsf{in}_{\mathsf{def}}(h, k, v)_i \wedge \mathrm{ret} = v\}$

Figure 5: Proof-sketch for `search`.

5.3 Verifying the B^{Link} Tree Operations

With our concrete predicate interpretations and interference environment we can now prove that the B^{Link} tree implementation satisfies our abstract concurrent index specification. In figure 5 we give an example proof sketch which shows how the the B^{Link} tree implementation of the `search` operation behaves in the $\mathsf{in}_{\mathsf{def}}$ case.

We will not give the full specification of our low-level implementation language here, but we shall outline a couple of key points. First the `get(x)` command performs an atomic read action that returns a copy of the contents of node `x`. The `next(A,k)` command compares `k` to the keys in node contents `A` and returns the pointer to the next node that should be followed to find the leaf node that contains the key `k`. In particular this will return the link pointer to the next node at the current level if `k` is greater than the maximum value of the node contents `A`. The `isIn(A,k)` command simply checks to see if the key `k` is contained within `A`. Finally, the `lookup(A,k)` command returns the value associ-

ated with the key `k` in `A`. Note that this command faults if the key `k` is not in `A`.

So long as each of the concrete interpretations of the abstract predicates is stable, that is invariant under the actions in the interference environment $\mathcal{A}$, our abstraction of the implementation is sound and this implementation satisfies our abstract specification. The proofs that this implementation satisfies the other abstract specifications for `search` follow the same style as the proof given above, as do the proofs of the `insert` and `remove` implementations. Further details of these proofs and of our low-level implementation language can be found in [5].

6. CONCLUSIONS

We have given a simple abstract specification for concurrent indexes, and have proved that Sagiv's widely-used B^{Link} tree algorithm is correct, in the sense of satisfying our specification. Using our approach, da Rocha Pinto has also shown the correctness of a concurrent index implementation based on hash tables [5]. In both cases, our abstract specification hides the complexities of the implementation from the view of clients. Our results on verifying algorithms based on concurrent indexes hold regardless of the implementation chosen.

Our results rely heavily on the reasoning given by separation logic with concurrent abstract predicates. Although there has been much work on using separation logic to reason about concurrent programs, the work on concurrent abstract predicates [6] was the first to allow abstract reasoning about concurrent modules. However, this initial paper only considered simple examples of a concurrent set and lock module. We extend the reasoning of concurrent abstract predicates, and apply it to concurrent indexes and the B^{Link} tree algorithm. This represents the most complex example to date of concurrent reasoning using separation logic.

We will investigate B^{Link} tree implementations used, for example, by databases and file systems, to obtain a better understanding of which properties of concurrent indexes are useful in practice. We can already think of several natural extensions to the reasoning presented here. For example, at the moment when multiple inserts occur for the same key, our reasoning states that the value is unknown. However, we intuitively know the set of possible values, and it would be possible to extend our specifications to include this information if it were important. Similarly, we could track the possible return values rather than lose this information. By focusing our attention on implementations used in practice, we will obtain an understanding of what kinds of properties are important for client programs using concurrent indexes.

Our proof that the B^{Link} tree implementation is correct with respect to our specification is at the boundary of what it is possible to prove by hand. We therefore require automatic verification tools to help with our reasoning. There is now substantial experience with using separation logic to reason modularly about sequential C programs: in the past researchers could prove properties of simple programs (100 lines); with tools based on separation logic, they can now verify hundreds of thousands of lines of code [4]. In contrast, verification tools for concurrency have, up to now, lacked scalability due to the lack of modularly of the underlying reasoning. We have demonstrated that our reasoning provides strong modular reasoning, in the sense that we have an abstract specification of concurrent indexes where the complex details of the B^{Link} tree implementation can be hidden from the client's view. This strong abstraction mechanism offers the possibility of scalable tools for concurrent verification. In this paper, we have shown that abstract reasoning about such algorithms is possible; in future work, we aim to show that it is practical.

Acknowledgments

Thanks to Richard Bornat, Cliff Jones, Daiva Naudžiūnienė, Gian Ntzik, Matthew Parkinson, Viktor Vafeiadis and John Wickerson for useful discussions and feedback.

We acknowledge funding from an EPSRC DTA (da Rocha Pinto), EPSRC programme grant EP/H008373/1 (da Rocha Pinto, Dinsdale-Young, Gardner and Wheelhouse) and EPSRC grant EP/H010815/1 (Dodds).

7. REFERENCES

[1] ANSI X3.135-1992, American National Standard for Information Systems - Database Language - SQL, 1992.

[2] Blelloch, G. E. Programming parallel algorithms. *Commun. ACM 39* (March 1996), 85–97.

[3] Boyland, J. Checking interference with fractional permissions. In *Static Analysis* (2003).

[4] Calcagno, C., Distefano, D., O'Hearn, P., and Yang, H. Compositional shape analysis by means of bi-abduction. In *POPL* (2009).

[5] da Rocha Pinto, P. Reasoning about Concurrent Indexes. Master's thesis, Imperial College London, Sept. 2010.

[6] Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M., and Vafeiadis, V. Concurrent abstract predicates. In *ECOOP* (2010).

[7] Dinsdale-Young, T., Gardner, P., and Wheelhouse, M. Abstraction and Refinement for Local Reasoning. In *Verified Software: Theories, Tools, Experiments*, G. Leavens, P. OâĂŹHearn, and S. Rajamani, Eds., vol. 6217 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2010, pp. 199–215.

[8] Dodds, M., Feng, X., Parkinson, M., and Vafeiadis, V. Deny-guarantee reasoning. In *ESOP* (2009).

[9] Hoare, C. A. R. Proof of a structured program: 'The sieve of Eratosthenes'. *The Computer Journal 15*, 4 (1972), 321–325.

[10] Johnson, T., and Shasha, D. A framework for the performance analysis of concurrent B-tree algorithms. In *PODS* (1990).

[11] Malecha, G., Morrisett, G., Shinnar, A., and Wisnesky, R. Toward a verified relational database management system. In *POPL* (2010).

[12] Reynolds, J. Separation logic: a logic for shared mutable data structures. In *LICS* (2002).

[13] Sagiv, Y. Concurrent operations on B*-trees with overtaking. *Journal of Computer and System Sciences 33* (October 1986), 275–296.

[14] Sexton, A., and Thielecke, H. Reasoning about B+ trees with operational semantics and separation logic. *ENTCS 218* (2008).

APPENDIX

A. SIEVE SET LEMMA

LEMMA 2. $\circledast_{2\leq j\leq\sqrt{m}}.\ \mathsf{wpst}(j,m,idx)_{\frac{1}{\sqrt{m}-1}}$

implies $\circledast_{i\in[1\ldots m]}.\ \mathsf{isPrime}(i) \Rightarrow \mathsf{in}_{\mathsf{def}}(idx,i,0)_1$
$\wedge\ \neg\mathsf{isPrime}(i) \Rightarrow \mathsf{out}_{\mathsf{def}}(idx,i)_1.$

PROOF. The first stage of the proof is achieved by rearranging the definition of wpst as follows:

$$\begin{aligned}&\circledast_{2\leq j\leq\sqrt{m}}.\ \mathsf{wpst}(j,m,idx)_{\frac{1}{\sqrt{m}-1}}\\ &\quad\Longrightarrow\quad \circledast_{i\in[1\,..\,m]}.\circledast_{2\leq j\leq\sqrt{m}}.\\ &\qquad\qquad (i\neq j\wedge i \bmod j=0)\Rightarrow \mathsf{out}_{\mathsf{rem}}(idx,i)_{\frac{1}{\sqrt{m}-1}}\\ &\qquad\qquad \wedge\,(i=j\vee i \bmod j\neq 0)\Rightarrow \mathsf{in}_{\mathsf{rem}}(idx,i,0)_{\frac{1}{\sqrt{m}-1}}\end{aligned}$$

Consequently, to prove the result we need only prove the following implication, for an arbitrary integer $i\in[1\ldots m]$:

$$\begin{pmatrix}\circledast_{2\leq i\leq\sqrt{m}}.(i\neq j\wedge i \bmod j=0)\Rightarrow \mathsf{out}_{\mathsf{rem}}(idx,i)_{\frac{1}{\sqrt{m}-1}}\\ \qquad\qquad \wedge\,(i=j\vee i \bmod j\neq 0)\Rightarrow \mathsf{in}_{\mathsf{rem}}(idx,i,0)_{\frac{1}{\sqrt{m}-1}}\end{pmatrix}$$

$$\begin{aligned}\Longrightarrow\quad & \mathsf{isPrime}(i)\Rightarrow \mathsf{in}_{\mathsf{rem}}(idx,i,0)_1\\ & \wedge\,\neg\mathsf{isPrime}(i)\Rightarrow \mathsf{out}_{\mathsf{def}}(idx,i)_1\end{aligned}$$

There are two cases: either i is prime or it is not.

- **prime:** As i is prime there is no value j other than 1 such that $(i \bmod j)=0$. Consequently for all $j\in 2..\sqrt{m}$ there is a predicate $\mathsf{in}_{\mathsf{rem}}(idx,i,0)_{\frac{1}{\sqrt{m}-1}}$. Applying the module axioms results in the predicate $\mathsf{in}_{\mathsf{rem}}(idx,i,0)_1$.
- i **not prime:** For any i that is not prime there must exist value $j\in 2..\sqrt{m}$ in such that $(i \bmod j)=0$. In the set of predicates, there must therefore be at least one predicate $\mathsf{out}_{\mathsf{rem}}(idx,i)_{\frac{1}{\sqrt{m}-1}}$. Applying the module axioms results in the predicate $\mathsf{out}_{\mathsf{def}}(idx,i)_1$.

This completes the proof of the lemma. □