Imperial College London

A Simple Abstraction for Complex Concurrent Indexes

OOPSLA 2011

Imperial College London:

Pedro da Rocha Pinto, Thomas Dinsdale-Young, Philippa Gardner and ***Mark Wheelhouse***

University of Cambridge:

Mike Dodds

Imperial College London

Motivation

Indexes are ubiquitous in computing systems:

Databases

Caches

File systems

JavaScript Objects

And have a variety of implementations:

Linked Lists

Arrays

Hash Tables

B-trees

Intuitive Index Specification

An index is a partial function mapping keys to values:

$$H : Keys \rightharpoonup Vals$$

There are three basic operations on an index h:

```
r ≔ search(h,k)

insert(h,k,v)

remove(h,k)
```

Simple Concurrent Example

This intuitive specification is not enough to reason about concurrent access to the index.

e.g

$$\texttt{r} := \texttt{search(h,k}_2\texttt{);}$$
$$\texttt{insert(h,k}_1\texttt{,r)} \quad || \quad \texttt{remove(h,k}_2\texttt{)}$$

with $k_1 \neq k_2$

Imperial College London

Disjoint Key Concurrency

Concurrent Abstract Predicates:

$in(h, k, v)$: there is a mapping in the index h from k to v, and only the thread holding the predicate can modify k.

$out(h, k)$: there is no mapping in the index h from k, and only the thread holding the predicate can modify k.

Axioms:

e.g. $in(h, k, v) * out(h, k) \Rightarrow false$

Concurrent Index Specification

$\{\, in(h,k,v) \,\}$	`r := search(h,k)`	$\{\, in(h,k,v) \land r = v \,\}$
$\{\, out(h,k) \,\}$	`r := search(h,k)`	$\{\, out(h,k) \land r = null \,\}$
$\{\, in(h,k,v') \,\}$	`insert(h,k,v)`	$\{\, in(h,k,v') \,\}$
$\{\, out(h,k) \,\}$	`insert(h,k,v)`	$\{\, in(h,k,v) \,\}$
$\{\, in(h,k,v) \,\}$	`remove(h,k)`	$\{\, out(h,k) \,\}$
$\{\, out(h,k) \,\}$	`remove(h,k)`	$\{\, out(h,k) \,\}$

Imperial College London

Simple Concurrent Example

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

```
r := search(h,k₂);
```

$$\{out(h, k_1) * in(h, k_2, v) \wedge r = v\}$$

$$\{out(h, k_1) \wedge r = v\}$$

```
insert(h,k₁,r)
```

$$\{\ in(h, k_1, v)\ \}$$

||

$$\{in(h, k_2, v)\}$$

```
remove(h,k₂)
```

$$\{\ out(h, k_2)\ \}$$

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

More Example Programs

However, we still cannot reason about the following programs:

```
remove(h,k) || remove(h,k)

insert(h,k,v) || remove(h,k)

r := search(h,k) || remove(h,k)
```

We need to account for the sharing of keys between threads.

Real-World Client Programs

Database sanitation:

remove all patients who have been cured, transferred or released

Graphics drawing:

clip all objects outside of some horizontal and vertical bounds

Garbage collection:

parallel marking in the mark/sweep algorithm

Web caching (NOSQL):

removing a picture whilst others are attaching comments to it

Imperial College London

Shared Key Concurrency

Extended Concurrent Abstract Predicates: with $i \in (0,1]$

$$in_{def}(h, k, v)_i$$

- in_{def} : the key k definitely maps to value v
- $0 < i \leq 1$: no other thread can change the value at key k
- $i = 1$: this thread can change the value at key k
- out_{def} is analogous

Imperial College London

Shared Key Concurrency

Extended Concurrent Abstract Predicates: with $i \in (0,1]$

$$in_{rem}(h, k, v)_i$$

- rem : the key k might map to a value, and if it does that value is v
- $0 < i \leq 1$: all threads can only remove the value at key k, the current thread has not done this so far
- out_{rem} is analogous
- Similarly we have out_{ins} and out_{ins} for insert only

Concurrent Index Specification

New specification of remove(h,k):

$\{ in_{def}(h,k,v)_1 \}$ **remove(h,k)** $\{ out_{def}(h,k)_1 \}$

$\{ out_{def}(h,k)_i \}$ **remove(h,k)** $\{ out_{def}(h,k)_i \}$

$\{ in_{rem}(h,k,v)_i \lor out_{rem}(h,k)_i \}$ **remove(h,k)** $\{ out_{rem}(h,k)_i \}$

Concurrent remove

$$\{ in_{def}(h,k,v)_1 \}$$

$$\{ in_{rem}(h,k,v)_1 \}$$

$$\{ in_{rem}(h,k,v)_{0.5} * in_{rem}(h,k,v)_{0.5} \}$$

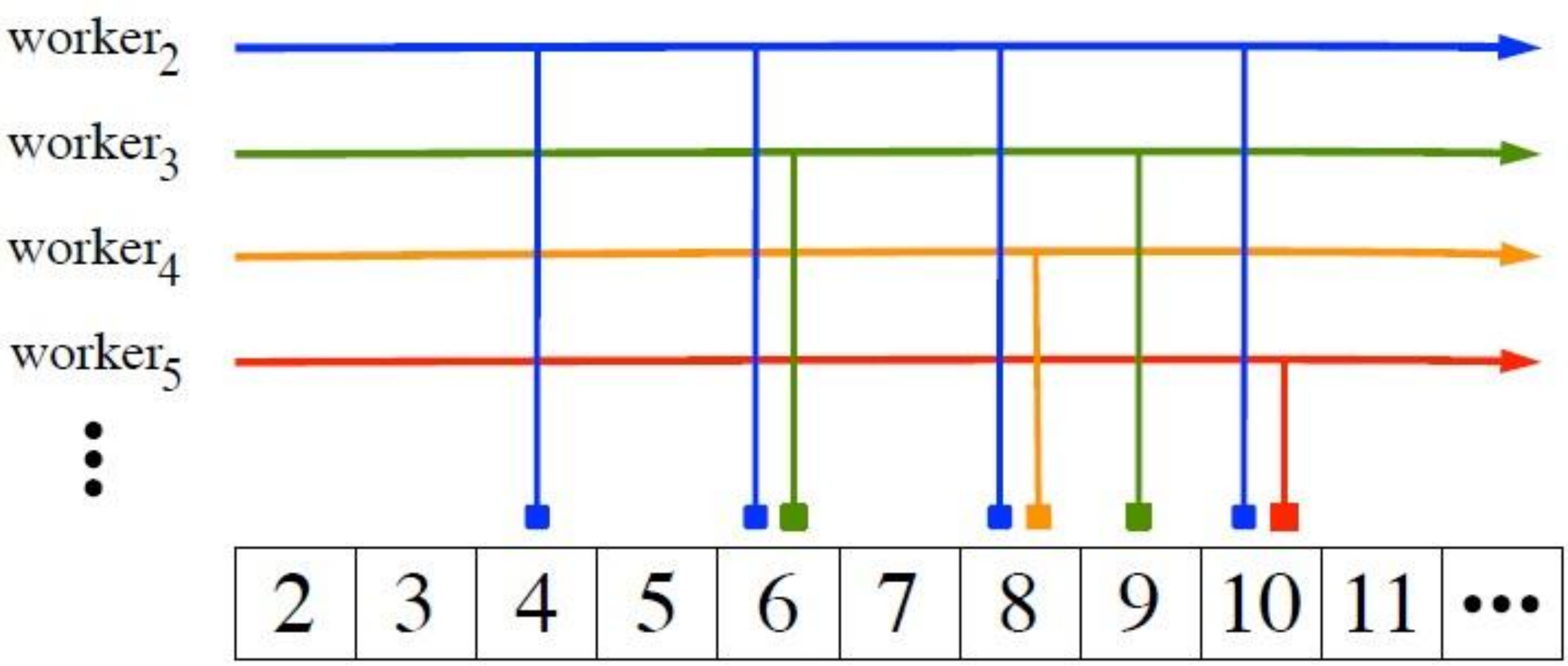
$\{ in_{rem}(h,k,v)_{0.5} \}$	||	$\{ in_{rem}(h,k,v)_{0.5} \}$
`remove(h,k)`	||	**`remove(h,k)`**
$\{ out_{rem}(h,k)_{0.5} \}$	||	$\{ out_{rem}(h,k)_{0.5} \}$

$$\{ out_{rem}(h,k)_{0.5} * out_{rem}(h,k)_{0.5} \}$$

$$\{ out_{def}(h,k)_1 \}$$

Imperial College London

Parallel Sieve of Eratosthenes



Parallel Sieve of Eratosthenes

Worker thread:

$$\{\, 2 \leq v \wedge \circledast_{2 \leq n \leq max}\, in_{rem}(h, n, 0)_i \,\}$$

```
worker(v,max,h)
        c ≔ v + v;
        while(c ≤ max)
                remove(h,c);
                c ≔ c + v;
```

$$\left\{ \circledast_{2 \leq n \leq max} \begin{array}{l} fac(n, v) \Rightarrow out_{rem}(h, n)_i \wedge \\ \neg fac(n, v) \Rightarrow in_{rem}(h, n, 0)_i \end{array} \right\}$$

Imperial College London

Combining Predicates

$$in_{rem}(h, k, v)_i * in_{rem}(h, k, v)_j \Leftrightarrow in_{rem}(h, k, v)_{i+j} \quad \text{if } i + j \leq 1$$

$$in_{rem}(h, k, v)_i * out_{rem}(h, k)_j \Rightarrow out_{rem}(h, k)_{i+j} \quad \text{if } i + j \leq 1$$

Imperial College London

Parallel Sieve of Eratosthenes

Sieve specification:

$$\{ \circledast_{2 \le n \le max} in_{def}(h, n, 0)_1 \wedge max > 1 \}$$

```
worker(2,max,h)|| worker(3,max,h) || … || worker(m,max,h)
```

$$\left\{ \circledast_{2 \le n \le max} \begin{array}{c} isPrime(n) \Rightarrow in_{def}(h, n, 0)_1 \wedge \\ \neg isPrime(n) \Rightarrow out_{def}(h, n)_1 \end{array} \right\}$$

where m = $\lfloor \sqrt{max} \rfloor$

Implementing a Concurrent Index

Our abstract concurrent index specification is sound for a number of different implementations, including:

Linked Lists

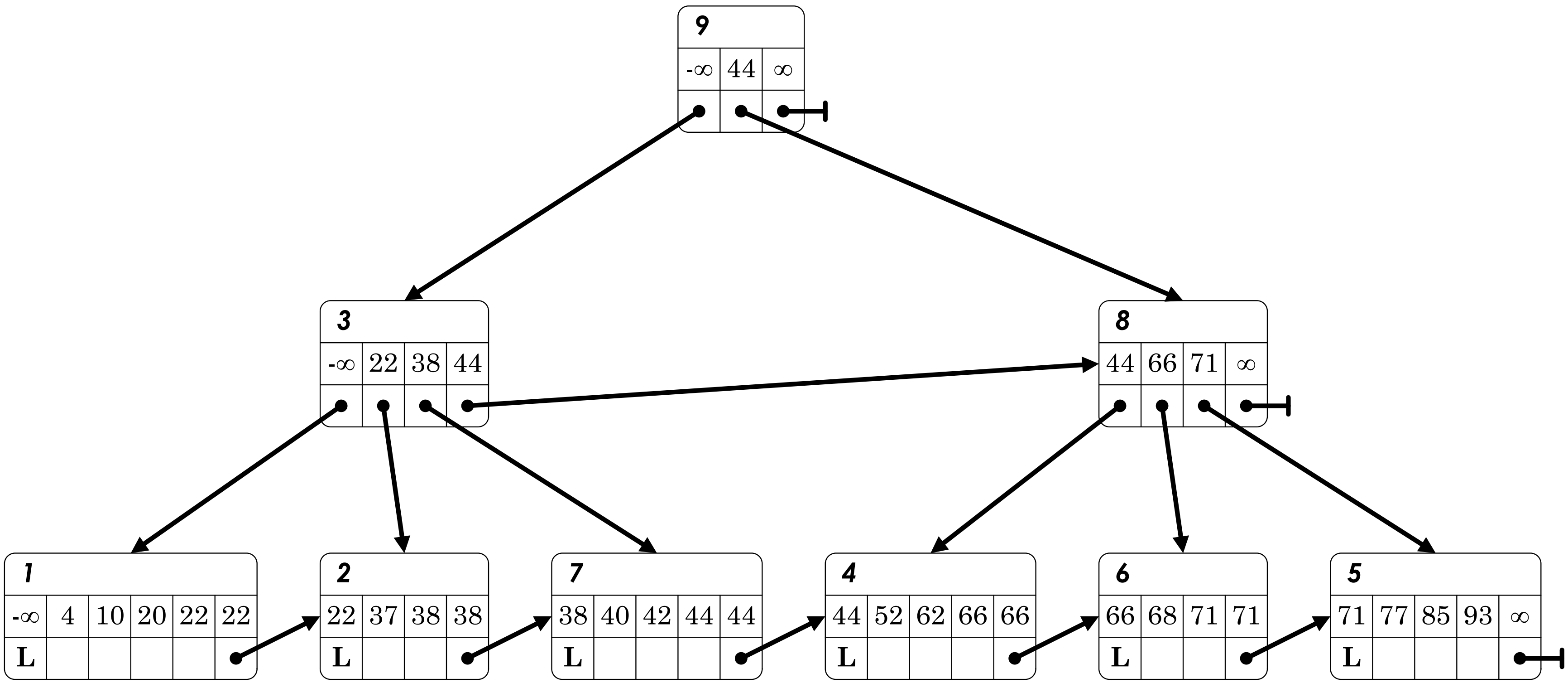
Arrays

B-trees

Hash Tables

Imperial College London

Concurrent B-tree



Concurrent B-tree

B-tree remove implementation must satisfy the specification:

$$\{\, in_{def}(h,k,v)_1 \,\}\ \texttt{remove(h,k)}\ \{\, out_{def}(h,k)_1 \,\}$$

Concrete definition of $in_{def}(h,k,v)_i$:

Interference environment

Shared state

$$in_{def}(h,k,v)_i = \exists r.\ \boxed{B_{\in}(h,k,v)}^{r}_{I(r,h)} * [\textsc{Lock}]^r_g * [\textsc{Swap}]^r_g * [\textsc{Rem}(0,\mathrm{k})]^r_{(d,i)} * \circledast_{v \in Vals} [\textsc{Ins}(0,\mathrm{k},\mathrm{v})]^r_{(d,i)}$$

Capability tokens

Concurrent B-tree

Check axioms: for example,

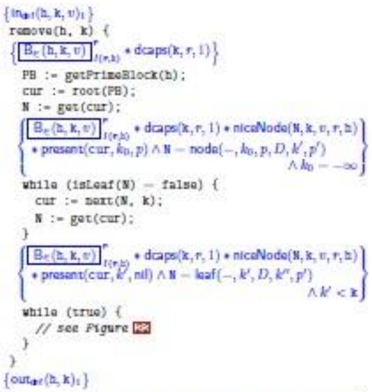
$$in_{def}(h,k,v)_i * in_{def}(h,k,v)_j \Leftrightarrow in_{def}(h,k,v)_{i+j} \quad \text{if } i+j \leq 1$$

Check stability of predicates

Check implementations satisfy abstract specifications

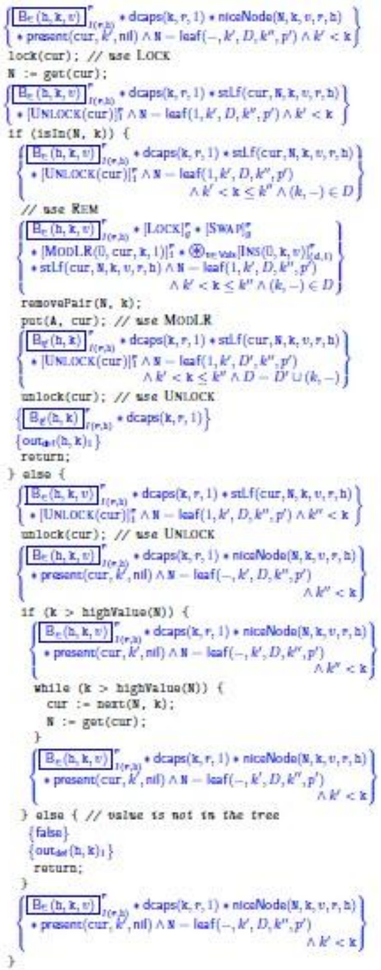
Concurrent B-tree

Proof of remove implementation:



```
{in_del(h, k, v)_1}
remove(h, k) {
  {[B_ε(h, k, v)]^r_{I(r,h)} * dcaps(k, r, 1)}
  PB := getPrimeBlock(h);
  cur := root(PB);
  N := get(cur);
  {[B_ε(h, k, v)]^r_{I(r,h)} * dcaps(k, r, 1) * niceNode(N, k, v, r, h)
   * present(cur, k_0, p) ∧ N = node(−, k_0, p, D, k', p')
                                                   ∧ k_0 = −∞}
  while (isLeaf(N) = false) {
    cur := next(N, k);
    N := get(cur);
  }
  {[B_ε(h, k, v)]^r_{I(r,h)} * dcaps(k, r, 1) * niceNode(N, k, v, r, h)
   * present(cur, k', nil) ∧ N = leaf(−, k', D, k'', p')
                                                   ∧ k' < k}
  while (true) {
    // see Figure 22
  }
}
{out_del(h, k)_1}
```

Figure 21. Proof outline for B^{Link} tree remove (excluding loop body).



```
{[B_ε(h, k, v)]^r_{I(r,h)} * dcaps(k, r, 1) * niceNode(N, k, v, r, h)
 * present(cur, k', nil) ∧ N = leaf(−, k', D, k'', p') ∧ k' < k}
lock(cur); // use LOCK
N := get(cur);
{[B_ε(h, k, v)]^r_{I(r,h)} * dcaps(k, r, 1) * stLf(cur, N, k, v, r, h)
 * [UNLOCK(cur)]^r_1 ∧ N = leaf(1, k', D, k'', p') ∧ k' < k}
if (isIn(N, k)) {
  {[B_ε(h, k, v)]^r_{I(r,h)} * dcaps(k, r, 1) * stLf(cur, N, k, v, r, h)
   * [UNLOCK(cur)]^r_1 ∧ N = leaf(1, k', D, k'', p')
                              ∧ k' < k ≤ k'' ∧ (k, −) ∈ D}
  // use REM
  {[B_ε(h, k, v)]^r_{I(r,h)} * [LOCK]^r_g * [SWAP]^r_g
   * [MODLR(0, cur, k, 1)]^r_1 * ⊛_{v∈Vals}[INS(0, k, v)]^r_{(d,1)}
   * stLf(cur, N, k, v, r, h) ∧ N = leaf(1, k', D, k'', p')
                              ∧ k' < k ≤ k'' ∧ (k, −) ∈ D}
  removePair(N, k);
  put(A, cur); // use MODLR
  {[B_∉(h, k)]^r_{I(r,h)} * dcaps(k, r, 1) * stLf(cur, N, k, v, r, h)
   * [UNLOCK(cur)]^r_1 ∧ N = leaf(1, k', D', k'', p')
                              ∧ k' < k ≤ k'' ∧ D = D' ⊔ (k, −)}
  unlock(cur); // use UNLOCK
  {[B_∉(h, k)]^r_{I(r,h)} * dcaps(k, r, 1)}
  {out_del(h, k)_1}
  return;
} else {
  {[B_ε(h, k, v)]^r_{I(r,h)} * dcaps(k, r, 1) * stLf(cur, N, k, v, r, h)
   * [UNLOCK(cur)]^r_1 ∧ N = leaf(1, k', D, k'', p') ∧ k'' < k}
  unlock(cur); // use UNLOCK
  {[B_ε(h, k, v)]^r_{I(r,h)} * dcaps(k, r, 1) * niceNode(N, k, v, r, h)
   * present(cur, k', nil) ∧ N = leaf(−, k', D, k'', p')
                                                   ∧ k'' < k}
  if (k > highValue(N)) {
    {[B_ε(h, k, v)]^r_{I(r,h)} * dcaps(k, r, 1) * niceNode(N, k, v, r, h)
     * present(cur, k', nil) ∧ N = leaf(−, k', D, k'', p')
                                                   ∧ k'' < k}
    while (k > highValue(N)) {
      cur := next(N, k);
      N := get(cur);
    }
    {[B_ε(h, k, v)]^r_{I(r,h)} * dcaps(k, r, 1) * niceNode(N, k, v, r, h)
     * present(cur, k', nil) ∧ N = leaf(−, k', D, k'', p')
                                                   ∧ k' < k}
  } else { // value is not in the tree
    {false}
    {out_del(h, k)_1}
    return;
  }
  {[B_ε(h, k, v)]^r_{I(r,h)} * dcaps(k, r, 1) * niceNode(N, k, v, r, h)
   * present(cur, k', nil) ∧ N = leaf(−, k', D, k'', p')
                                                   ∧ k' < k}
}
```

Figure 22. Proof outline for B^{Link} tree remove (main loop body).

Imperial College London

Conclusion

Summary:

- simple abstract spec for concurrent indexes
- essence of real-world client programs
- correct implementations
  - linked lists
  - hash tables
  - concurrent B-trees
- proof structure lends itself to automation

Future work:

- Automation/Proof Assistant (Dinsdale-Young)
- java.util.concurrent (da Rocha Pinto)
- File Systems (Ntzik)