Hash Table Implementations

Minke Zhang

February 27, 2014

1 Modules

Hash Tables

As was the case in the previous project, we wish to create a hash table interface which abstracts away the specifics of the implementation – this way, we will be able to hide the complexities of the `add()`, `remove()`, and `contains()` methods in the hash module, while keeping the outward-facing code as simple as possible.

```
typedef struct hash_t {
  int type;     // designates the table TYPE (oneof LOCKING, LOCKFREE, LINEAR, AWESOME)
  void *b;      // the specific blob of the hash with type = TYPE
  int heur;     // designates the heuristic the hash table will employ to determine
                //   if it is full
  int size;     // current size metric of the hash table -- referenced in is_full()
  int max_s;    // maximum size metric of the hash table
} hash;

hash *init_hash(int type, int heur, int max_s) {
  hash *table = malloc(sizeof(hash));
  table->type = type;
  table->heur = heur;
  table->size = 0;
  table->max_s = max_s;
  switch(type) {
    ...         // sets the hash blob table->b
  }
  return(table);
}

void free_hash(hash *table) {
  switch(table->type) {
    ...   // frees the hash blob table->b
  }
  free(table);
}

/* determines if the given hash table *h is full */
int is_full(hash *table) {
```

```
  return(table->max_s >= table->size);
}

/* gets the current size of the hash table */
int set_size(hash *table) {
  int success, size;
  switch(table->type) {
    ...
  }
  if(success) {
    table->size = size;
  }
  return(success);
}

/* resizes a hash table *h such that h'->size == 2 * h->size */
int resize(hash *table) {
  int success;
  switch(table->type) {
    ...
  }
  if(success) {
    table->max_s <<= 2;
  }
  return(success);
}

/* adds an element to the hash table */
int add(hash *table, void *elem) {
  int success;
  switch(table->type) {
    ...
  }
  if(is_full(table)) {
    resize(table);
  }
  set_size(table);
  return(success);
}

/* drops an element from the hash table */
int remove(hash *table, void *elem) {
  int success;
  switch(table->type) {
    ...
  }
  set_size(table);
  return(success);
}

/* returns if a given element is in the hash table */
int contains(hash *table, void *elem) {
  int success;
  switch(table->type) {
```

```
    ...
  }
  return(success);
}
```

Experiment Interfaces

```
typedef result_t {
  ...
  int fingerprint;      // the total fingerprint returned by an experiment --
                        //   as before, so we can easily check the PARALLEL
                        //   implementations with that of the SERIAL
} result;

result *init_result(...) {
  result *r = malloc(sizeof(result));
  ...
  r->fingerprint = 0;
  return(r);
}

void free_result(result *r) {
  ...
  free(r);
}

result *serial(int m, int pp, int pm, float p, float w, int s);
result *parallel(..., int n, int h);
result *noload(..., int n);

#define RUNTIME 2000

void dispatcher_rate() {
  serial(RUNTIME, _, _, _, w = 1, _, n = c, _);
  ...
}

void parallel_overhead(int h) {
  // read-heavy usage
  parallel(RUNTIME, pp, pm, p, w = 4000, _, n = 1, h);
  ...
  // write-heavy usage
  parallel(RUNTIME, pp, pm, p, w = 4000, _, n = 1, h);
  ...
}

void speedup(int n, int h) {
  // read-heavy usage
  parallel(RUNTIME, pp, pm, p, w = 4000);
  serial(RUNTIME, pp, pm, p, w = 4000, n, h);
  ...
```

```
  // write-heavy usage
  parallel(RUNTIME, pp, pm, p, w = 4000);
  serial(RUNTIME, pp, pm, p, w = 4000, n, h);
  ...
}
```

2 Hypotheses

3 Dispatcher Rate

In the first firewall project, we noticed a hit in the performance of the equivalent experiment due to the overhead of initializing the data structures which were necessary for the parallel Lamport queues. In this experiment, we will absorb the queue initialization penalty, plus the hash table initialization. However, as per the project specifications, we consider `m = 2000` to be a suffcient window in which the experiments would run, which seems to indicate that at this time interval, a significant portion of the time spent in the program should be dedicated to processing packets and not to initialization – which would heavily amortize any initial penalty while initializing the additional data structures for this project.

4 Parallel Overhead

Note that in our hash table implementations, we were not asked to shrink the size of the hash table – we will only need to grow the table when the size is insufficient. Thus, we would expect that a load configuration with higher P_+ to have a higher overhead penalty than that of runs with a load configuration with lower P_+.

5 Speedup

In order to maximize speedup, we wish to lower the amount of serial bottlenecks in our code – thus, the `LOCKING` hash table would seem to be naturally slower, as it employs a read-write lock to ensure proper operations.

6 Testing

We wish to test several properties of the hash table – that is, add() will properly insert an element into the table, remove() correctly deletes an element, and that contains() is able to search out the contents of the hash table. As such, we will implement the following tests.

```
// tests to make sure that add() is behaving as expected
int test_presence(int type) {
  hash *table = init_table(type, ...);
  void *elem;
  int success = 1;
  for(int i = 0; i < MAX_ELEMENTS; i++) {
    elem = ...;
    add(table, elem);
    success &= contains(table, elem);
```

```
  }
  return(success);
}

// tests to make sure that remove() is behaving as expected
int test_absence(int type) {
  hash *table = init_table(type, ...);
  void *reference_table[MAX_ELEMENTS];
  int success = 1;
  for(int i = 0; i < MAX_ELEMENTS; i++) {
    reference_table[i] = ...;
    add(table, reference_table[i]);
  }
  for(int i = 0; i < MAX_ELEMENTS; i++) {
    remove(table, reference_table[i]);
    success &= !contains(table, reference_table[i]);
  }
  return(success);
}

int test_correctness() {
  result *p = serial(...);
  result *q = parallel(..., n, h);
  result *r = noload(..., n);
  return((p->fingerprint == q->fingerprint) && (q->fingerprint == r->fingerprint));
}
```