Flexible Hash Table Implementations for Near Drop in Replacement

Presentation · 07. September 2017
Tobias Maier
Possible Topics/Focuses

- interface construction

- collection of small tidbits
  - increasing memory inplace (2 methods)
  - removing contention from shared variables

- more on DySECT (the paper presented at ESA)
Interface – Main Ideas

- user expectation
  - similar library functions
  - interface used in literature

- possible problems (analyzing an interface)
  - more powerful interface
  - ease of implementation
  - misusability
Interface – Functionality and Performance

- **update operation**

  ```
  auto it = table.find(k);
  if (it != end) it->second += 5;
  ```

- **using dedicated update**

  ```
  bool b = table.update(k,
  [](value_type& cur) {
    cur+=5;
  });
  ```
Interface – Misusability

- iterator as return type
  - iterators give pointers
  - moving elements can lead to errors

- operations invalidate iterators
  - refresh iterators

- change iterators to not return pointers
Overallocation – For Inplace Resize

- very large virtual memory allocation
- only changed cells are mapped to physical memory
- writing to virtual memory \(\approx\) increasing local allocation
  - inplace growing
- limited portability

used memory 1 mem page offcut allocation (initially unmapped)
Overallocation – Multi Table Inplace Resize

- Subtables are islands of physical memory in a virtual allocation.
- Equally space tables in virtual memory:
  \[ tab_i @ M/T \cdot i \]
  \[ M = \text{overalloc size} \]
  \[ T = \text{number of subtables} \]
- No explicit indirection.

Diagram:
- Tab 0
- Tab 1
- Tab 2
- Allocation (initially unmapped)
- Mapped memory
Parallelization – Some Initial Thoughts

- Parallele Datenstrukturen für Informationsaustausch

- Große Tabelle ⇒ viele unabhängige Zugriffe

- Wachsende Tabellen benötigen Koordination
  - Häufiger Zugriff auf die selben Variablen
  - häufig vermeidbar durch lokale Duplizierung

- Beschränkt durch Speicherbandbreite
Parallelization – Optimizing Often Used Members

- `table_ptr` is smart
  - copying is expensive

- `n` has contentious updates
  - exact value is changing

```
global object

table_ptr
n ?
...
insert(....)
```
- `table_ptr` is smart
  - copying is expensive
  - cache the pointer copy

- `n` has contentious updates
  - exact value is changing
  - update `approx_n` with local counts

- move interface to handle to avoid misusability
Final Size Not Known A Priori

- conservative estimate
- strict bound might not be reasonable
- less space efficient

\[ n \leq n' \]
Final Size Not Known A Priori

- conservative estimate
- optimistic estimate $n \approx n'$
  - might overfill
  - needs growing strategy

\[ n \approx n' \]

\[ \varepsilon n \]

\[ \varepsilon n' \]

slow
needs growing
Final Size Not Known A Priori

- conservative estimate
- optimistic estimate
- number of elements changes over time
  - cannot be initialized with max size
Resizing

- Growing has to be in small steps

- Basic approaches
  - Additional table
  - Full migration
  - Inplace + reorder

Most common in libraries.
Secondary Contribution – Efficient Growing

- addressing the table (no powers of two)
  - conventional wisdom: modulo table size
  - faster: use hash value as scaling factor
    \[ \text{idx}(k) = h(k) \cdot \frac{\text{size}}{\text{maxHash} + 1} \]

- very fast migration due to cache efficiency
Secondary Contribution – Efficient Growing

- addressing the table (no powers of two)
- conventional wisdom: modulo table size
- faster: use hash value as scaling factor
  \[ idx(k) = h(k) \cdot \frac{size}{maxHash + 1} \]
- very fast migration due to cache efficiency
Secondary Contribution – Efficient Growing

- addressing the table (no powers of two)
  - conventional wisdom: modulo table size
  - faster: use hash value as scaling factor
    \[ \text{idx}(k) = h(k) \cdot \frac{\text{size}}{\max\text{Hash} + 1} \]

- very fast migration due to cache efficiency

- inplace variant going from right to left

not portable
Multi Table Approach

- \( T = 2^c \) subtables with expected equal count
- reduces memory during subtable migration

- \( h(k) \Rightarrow h_t(k) \) for the subtable \( h_p(k) \) within the table
Cuckoo Displacement

**H-ary B-Bucket Cuckoo Hashing**

[Pagh, Dietzfelbinger, Mehlhorn, Mitzenmacher, ...]

- **buckets** of \( B \) cells

- **\( H \) alternative buckets** per element
  
  \( h_1(k), \ldots, h_H(k) \)

- if buckets are full, move existing elements
  
  breadth-first-search

\( h_1(k) \)

\( h_2(k) \)
Cuckoo Displacement

*H*-ary *B*-Bucket Cuckoo Hashing

[Pagh, Dietzfelbinger, Mehlhorn, Mitzenmacher, ...]

- **buckets** of *B* cells
- **H alternative buckets** per element
  \[ h_1(k), \ldots, h_H(k) \]
- if buckets are full, move existing elements
  - breadth-first-search
Contribution – Dynamic Space Efficient Cuckoo Table

- use subtables of unequal size (use powers of 2)
  - $h_i(k) \Rightarrow h_{it}(k)$ table and $h_{ip}(k)$ position in table
  - doubling one subtable $\iff$ small overall factor

- use displacements to equalize load imbalance
Contribution – Dynamic Space Efficient Cuckoo Table

- use subtables of unequal size (use powers of 2)
  - $h_i(k) \Rightarrow h_{it}(k)$ table and $h_{ip}(k)$ position in table
  - doubling one subtable $\Leftrightarrow$ small overall factor

- use displacements to equalize load imbalance
Contribution – Dynamic Space Efficient Cuckoo Table

- use subtables of unequal size (use powers of 2)
  - $h_i(k) \Rightarrow h_{it}(k)$ table and $h_{ip}(k)$ position in table
  - doubling one subtable $\iff$ small overall factor

- use displacements to equalize load imbalance
Result – Insertion into Growing Table

![Graph showing time per operation vs. enforced min load δ for different hash table algorithms. The graph includes lines for DySECT $B=8, H=3$, Cuckoo $B=8, H=3$, Lin Prob, and Robin Hood. The y-axis represents time per operation $\cdot (1-\delta)$ in nanoseconds, and the x-axis represents the enforced min load $\delta$. The "expected time" per insertion is indicated by the $\frac{1}{1-\delta}$ formula.](image)
Result – Word Count Benchmark

- DySECT $B=8, H=3$
- Cuckoo $B=8, H=3$
- Lin Prob
- Robin Hood

- CommonCrawl (avg. $12 \times$)
- not normalized
Result – Load Bound

![Graph showing load bound with different parameter combinations: B=8, H=3; B=8, H=2; B=4, H=3; B=4, H=2.]

- We are in cooperation to prove bounds.
Conclusion

- only **dynamic** tables offer true **space efficiency**

- **lack** of published work on **dynamic** hash tables
  - even simple techniques are largely unpublished

- **DySECT**  
  - no overallocation  
  - constant lookup  
  - addressing uses bit operations

- **cuckoo displacement** offers more untapped potential

- code available: [https://github.com/TooBiased/DySECT](https://github.com/TooBiased/DySECT)
(Ab)using Overallocation

- subtables are islands of physical memory in a virtual allocation
- writing to virtual memory $\approx$ increasing local allocation
  - inplace growing
  - no explicit indirection
  - limited portability

allocation (initially unmapped)  \hspace{2cm} mapped memory
Result - Successful Find

- DySECT $B=8, H=3$
- Cuckoo $B=8, H=3$
- Lin Prob
- Robin Hood

Load factor vs. time per op [ns]

- DySECT
- Cuckoo
- Lin Prob
- Robin Hood
Result - Unsuccessful Find

![Graph showing the time per operation and load factor for different algorithms. The x-axis represents the load factor ranging from 0.8 to 1.0, and the y-axis represents the time per operation in nanoseconds (ns). The graph compares DySECT (B=8, H=3), Cuckoo (B=8, H=3), Lin Prob, and Robin Hood algorithms.]