The hash tables

- Google’s dense and sparse hash tables
  - Use open addressing
  - Quadratic probing
- SGI
  - Is a chained hash table
  - Referred to as gnu in graphs
- One-table and Two-table
  - Doubly linked: Rather large compartments
  - Buckets: Two pointers delimit a section of a circular list.
  - One-table uses alternative vector implementation

- Chained
  - Singly linked: small compartments
  - lookup uses a tight inner loop by copying into sentinel.

- The same hash functions were used for all hash tables. A string table lookup method for string data and a division method for integer data.

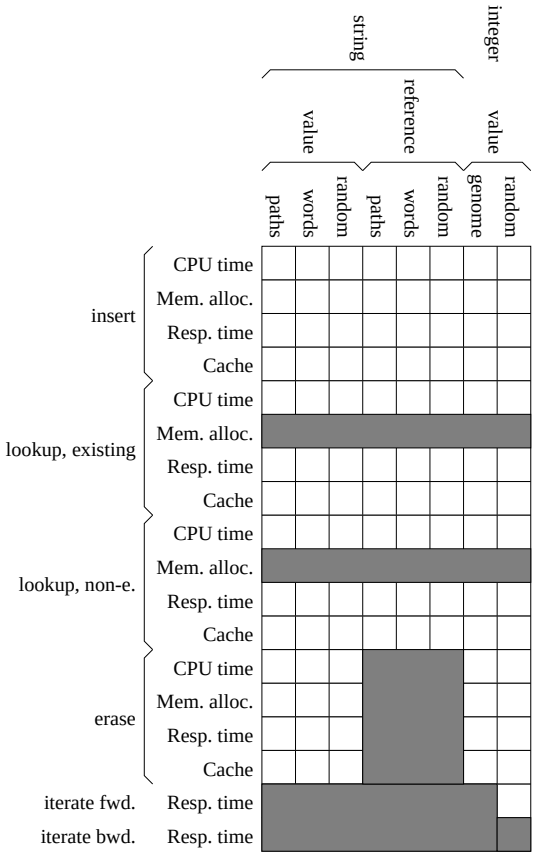
The machines

- 32 bit app intel machines at DIKU without PAPI
- 64 bit amd with PAPI extensions

Kinds of benchmarks

- Memory allocated
  - Not Google
- Timing
  - CPU time + Total CPU cycles
  - Variability measured as std. deviation
  - CPU time and CPU cycle graphs are very alike, CPU cycles are used.
- Cache behaviour
  - Number of L1 Cache misses.
  - L1 Cache miss ratio: Percentage of cache accesses that are misses.

Benchmarks skipped



- The meaurements using “by value” string data are not included here because they were problemetic.

Data used for the benchmarks

- Random strings (10 bytes) and integers - reflect the behavior of the data structure with a perfect distribution of in-data.

- Ordered data: Gene sequences, words and output of the `locate` command.

- Realism of data: range 100000 - 800000 elements

- Data was loaded into memory from a file and then scanned.

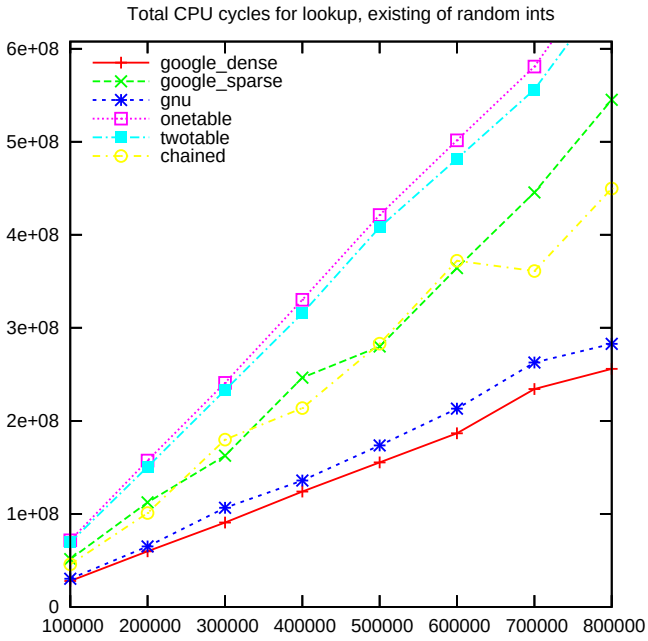
- The maximum load factor
  - Google: 0.8
  - SGI (gnu on graphs): 1.0
  - Us: initially 5.0, then 1.0 - we focus on max load factor of 1.0

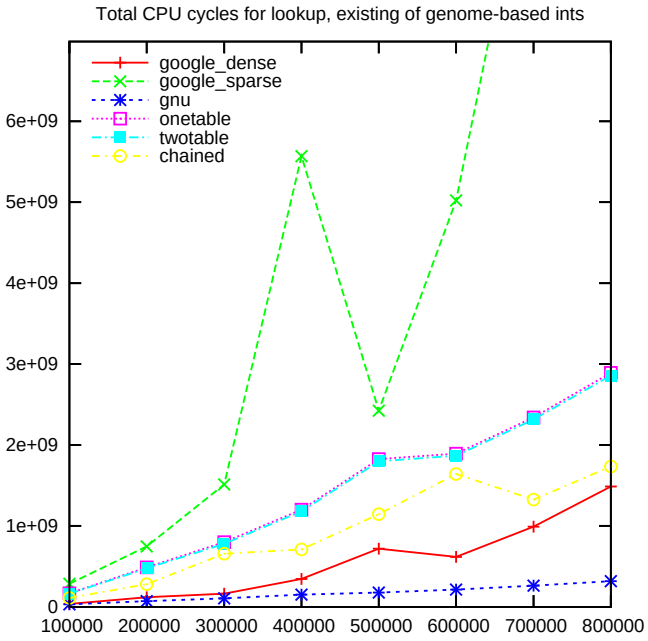
- Timing
  - Linear hash tables all save the hash value which saves time on string data.
  - Iteration on the linear hash tables should be efficient because of the circular chain of elements.
  - The chained hash table uses a tight inner loop containing only one test. First compartment in chain is stored in vector.
  - Google has a cache advantage in using open addressing.

- Allocation
  - Saving the hash value takes more memory.
  - Doubly linked lists vs. singly linked lists.
  - Two tables vs. one table.
  - Alternative allocation scheme used by one table

The `Lookup` operation

- `lookup` non-existing is called for each `insert`.
- `lookup` existing is called for each `delete`
- Saving the hash value saves time on `lookup` of string data.
- Lookup non-existing: each odd entry was inserted, each even was looked up.

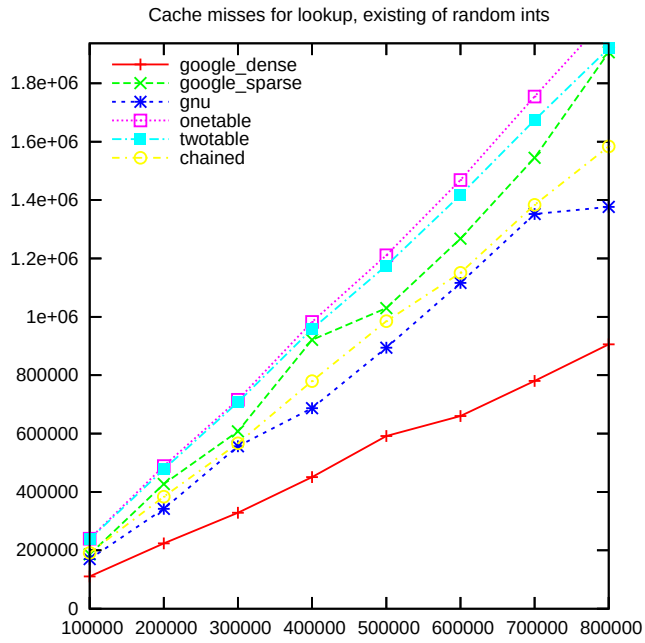


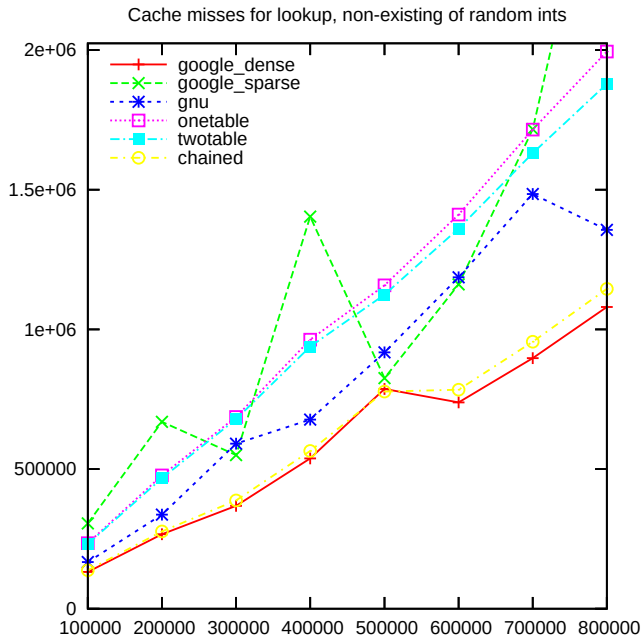


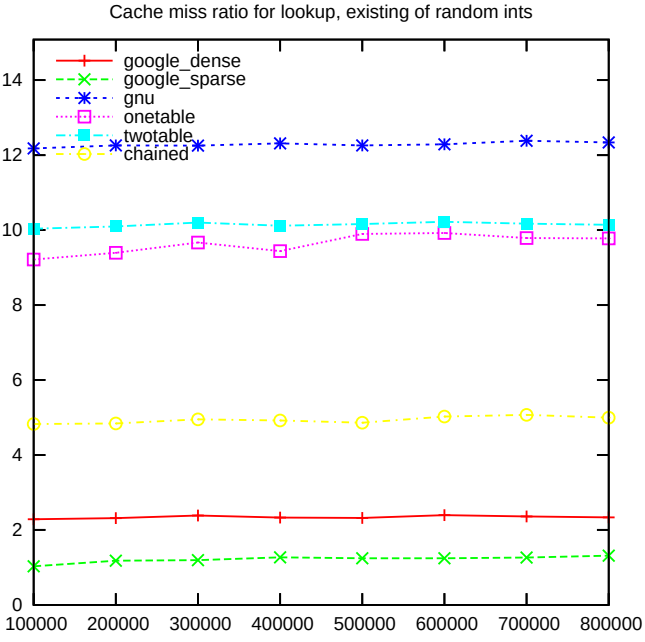
- One- and two-table are very similar.
- A factor 10 difference between the two graphs y axis
- a non uniform distribution particularly effects the sparse hash table

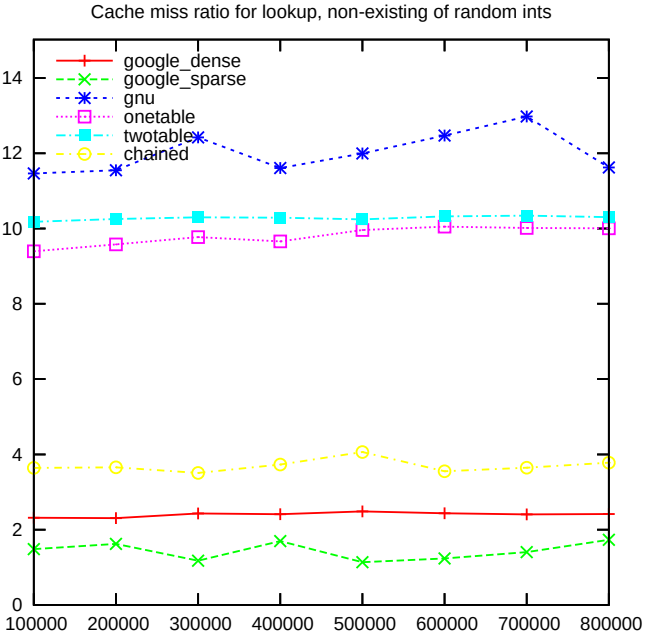
Cache

- Few cache misses with open addressing
- lookup of non-existing elements causes more cache misses because more elements are traversed on average.
- Linear hash tables have a quite high cache miss ratio.
- The good cache miss ratio of the chained hash table implementation is likely due to the storing of the first compartment within the vector.

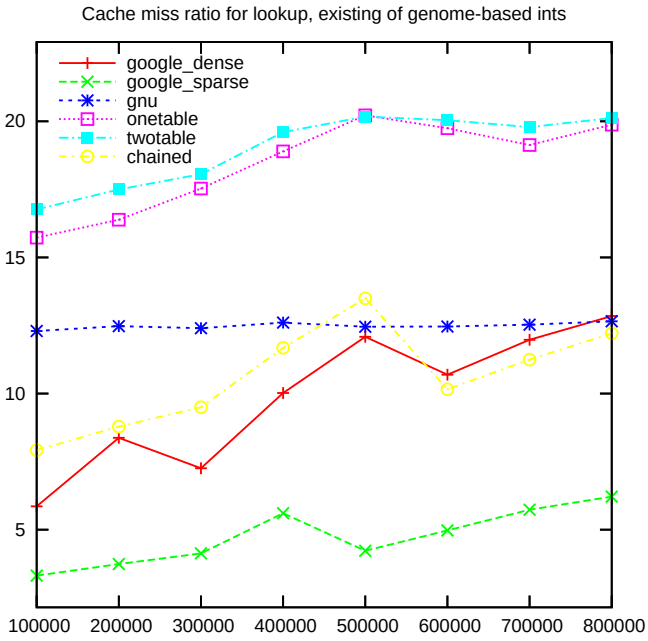

Cache misses for lookup, existing of random ints
google_dense
google_sparse
gnu
onetable
twotable
chained
0
200000
400000
600000
800000
1e+06
1.2e+06
1.4e+06
1.6e+06
1.8e+06
100000
200000
300000
400000
500000
600000
700000
800000

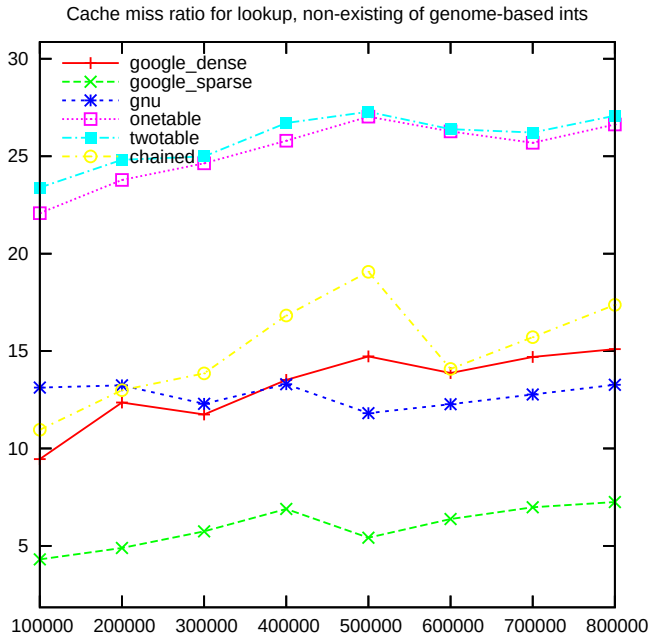

Cache misses for lookup, non-existing of random ints
google_dense
google_sparse
gnu
onetable
twotable
chained
0
500000
1e+06
1.5e+06
2e+06
100000
200000
300000
400000
500000
600000
700000
800000


Cache miss ratio for lookup, existing of random ints
google_dense
google_sparse
gnu
onetable
twotable
chained
0
2
4
6
8
10
12
14
100000
200000
300000
400000
500000
600000
700000
800000


Cache miss ratio for lookup, non-existing of random ints
google_dense
google_sparse
gnu
onetable
twotable
chained
0
2
4
6
8
10
12
14
100000
200000
300000
400000
500000
600000
700000
800000

- When using data that is not uniformly distributed the cache miss ratio is higher for all hash tables, and significantly higher for the linear hash tables.



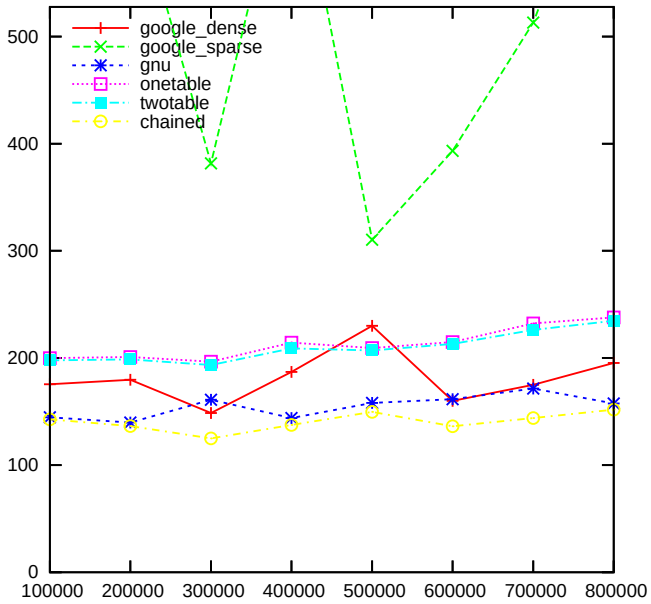


Variability

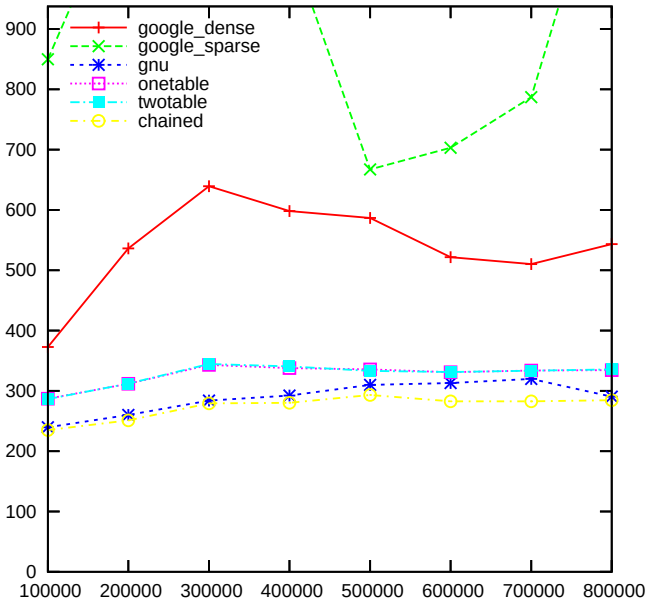
- The chained hash table exhibits the least amount of variability. Again the storing of the first element within the vector may be the reason.

- Google sparse fluctuates a lot.

Standard deviation of CPU cycle count per operation for lookup, existing of pointers to random strings



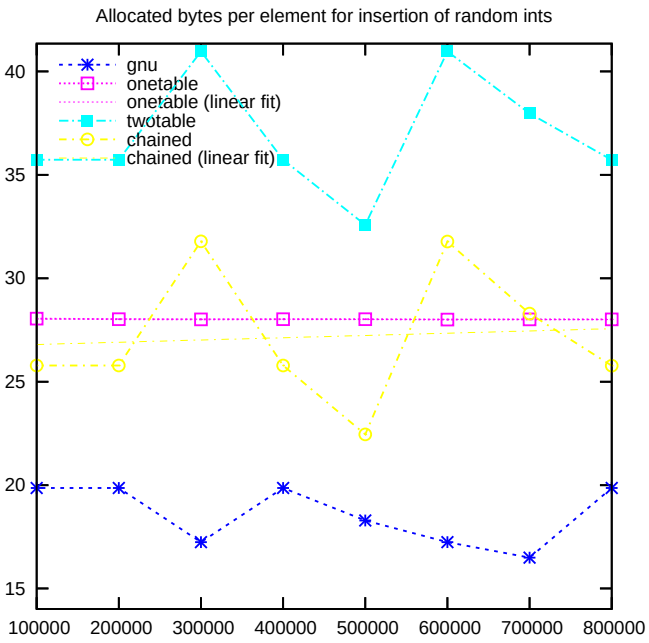
Standard deviation of CPU cycle count per operation for lookup, existing of pointers to filenames

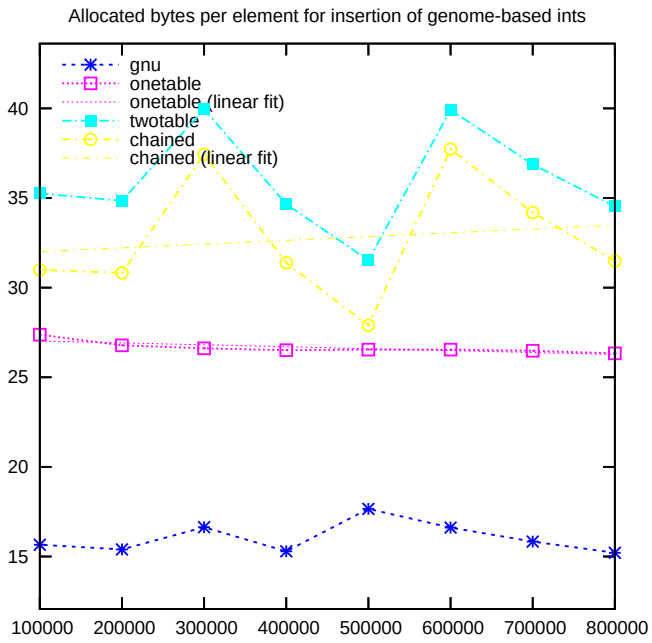


Memory allocation

- Graphs are similar for different data types.
- Graphs from the amd64 are similar, but more memory is allocated.
- The onetable hash table uses a constant amount of memory per element.
- Small decline in memory allocated per element for the onetable implementation is due to duplicates in the genome data.

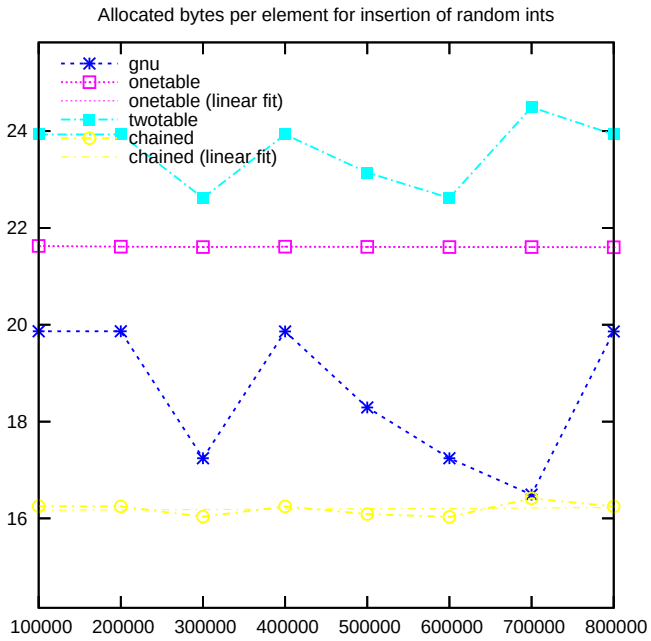
Max load factor 1

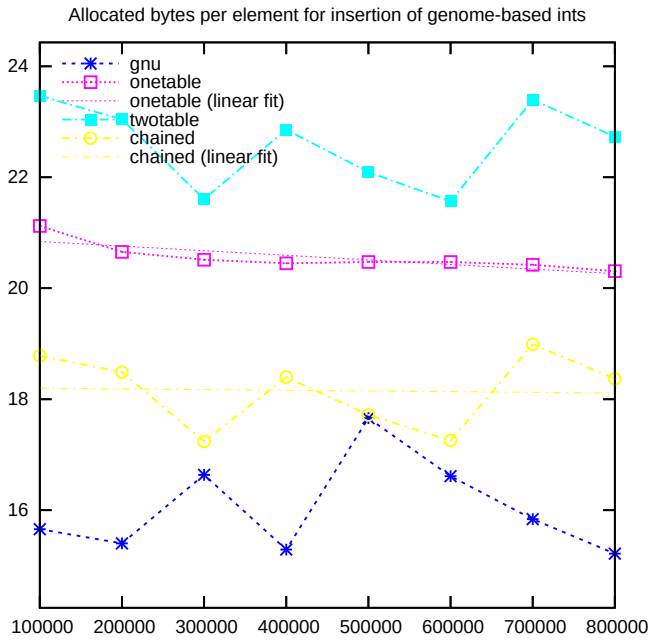




Max load factor 5

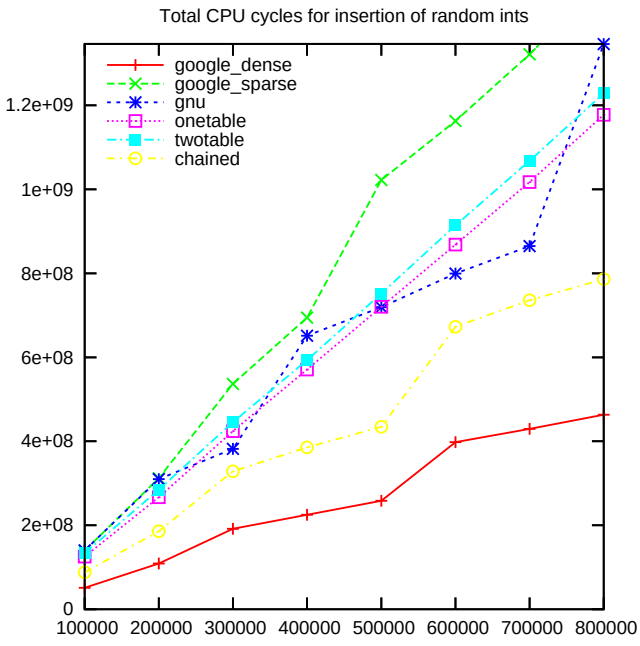
- All our implementations use less memory at a load factor of 5 because more buckets need to be allocated.

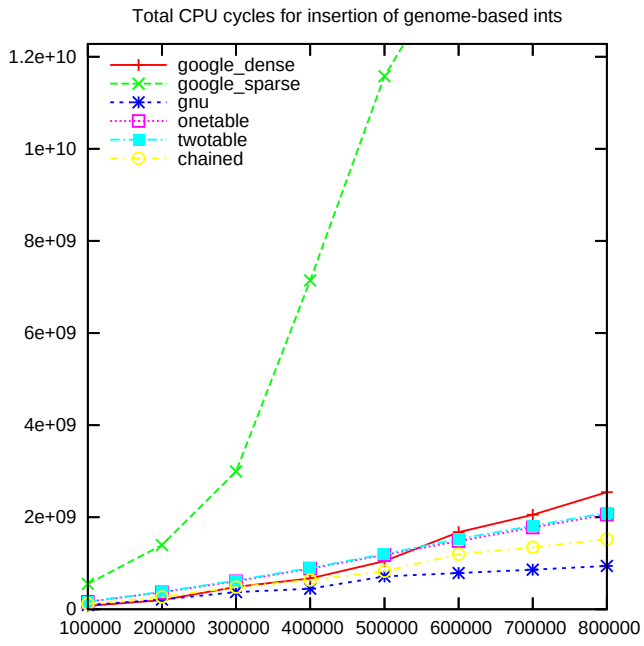

Allocated bytes per element for insertion of random ints
gnu
onetable
onetable (linear fit)
twotable
chained
chained (linear fit)
24
22
20
18
16
100000
200000
300000
400000
500000
600000
700000
800000

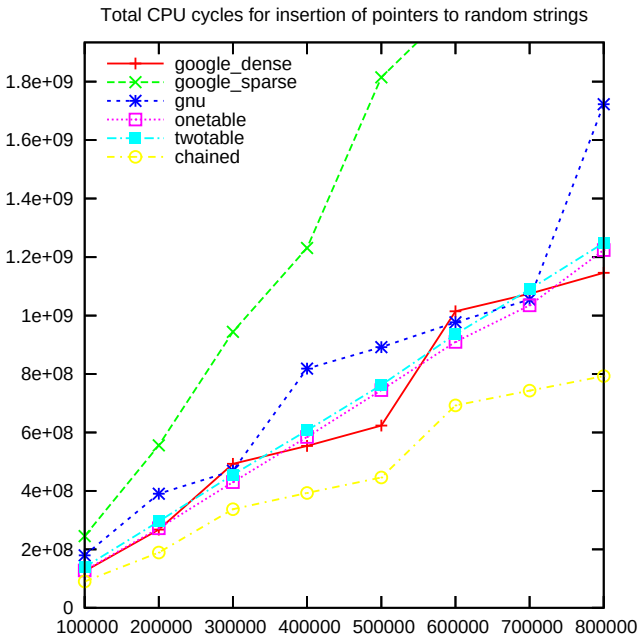

Allocated bytes per element for insertion of genome-based ints
gnu
onetable
onetable (linear fit)
twotable
chained
chained (linear fit)
24
22
20
18
16
100000
200000
300000
400000
500000
600000
700000
800000

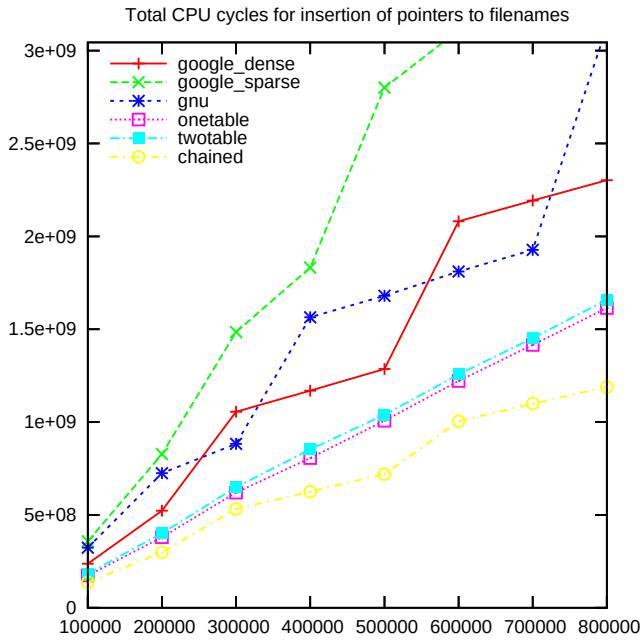
The `insert` operation

- The sparse hash table uses a lot of time, especially on the genome based integers.
- The chained hash table does well on string data.
- The google hash tables loose cache benefits when inserting strings.

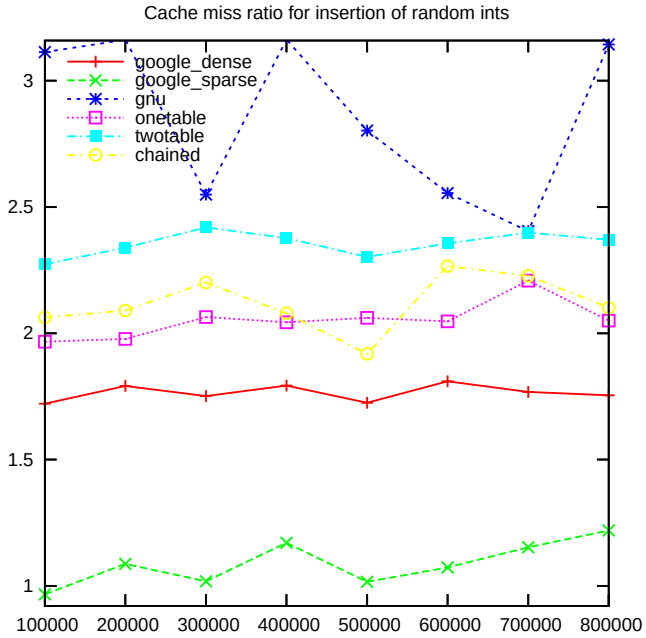


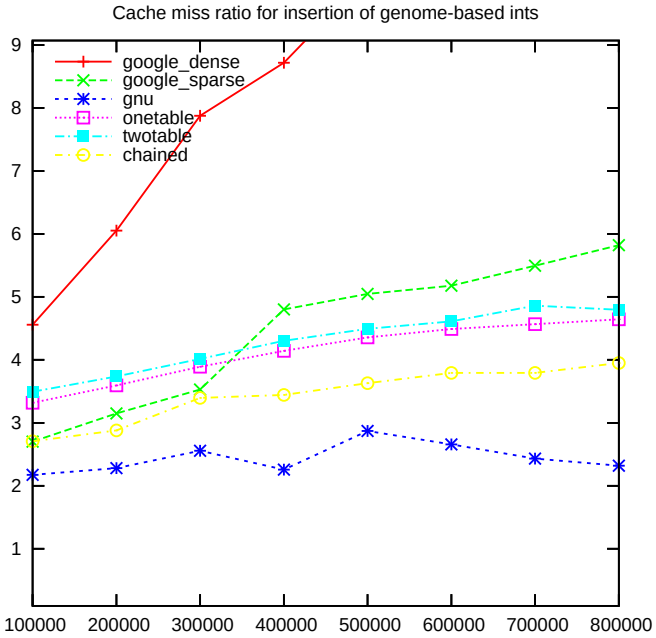


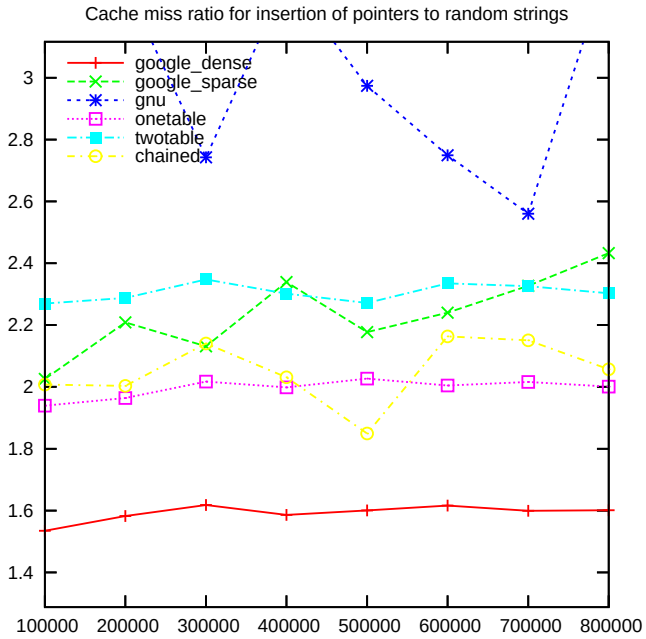

Total CPU cycles for insertion of pointers to random strings
google_dense
google_sparse
gnu
onetable
twotable
chained
1.8e+09
1.6e+09
1.4e+09
1.2e+09
1e+09
8e+08
6e+08
4e+08
2e+08
0
100000
200000
300000
400000
500000
600000
700000
800000

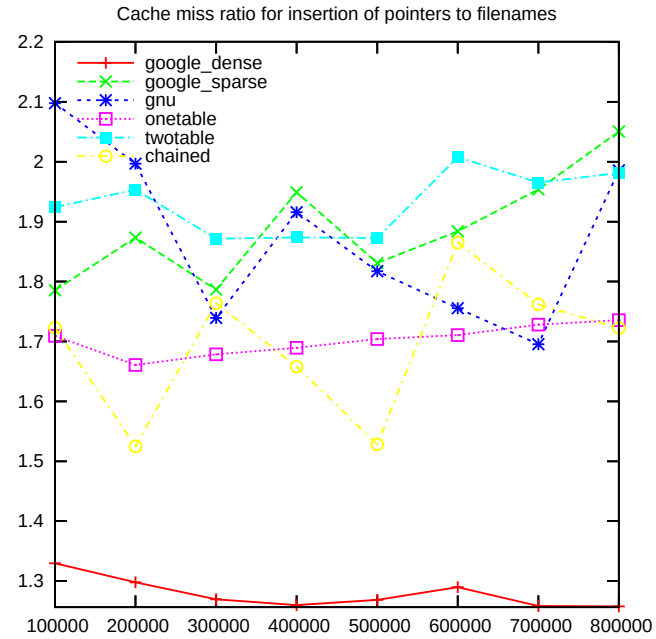

Total CPU cycles for insertion of pointers to filenames
google_dense
google_sparse
gnu
onetable
twotable
chained
3e+09
2.5e+09
2e+09
1.5e+09
1e+09
5e+08
0
100000
200000
300000
400000
500000
600000
700000
800000

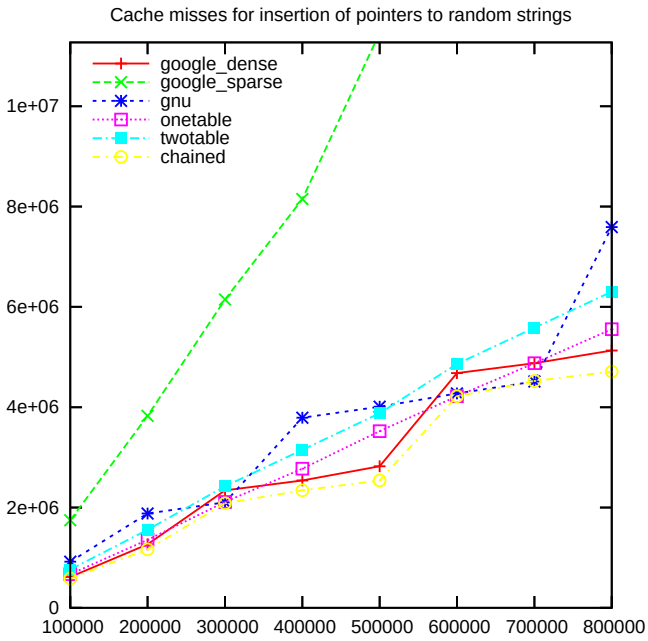
Cache

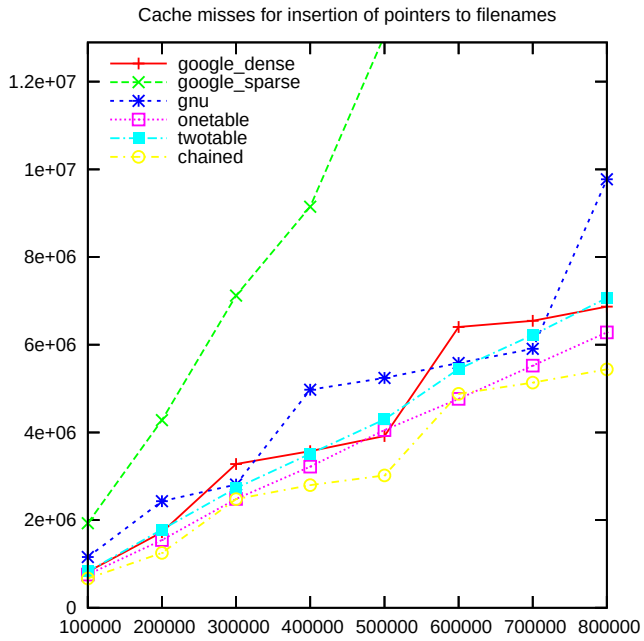

Cache miss ratio for insertion of random ints
google_dense
google_sparse
gnu
onetable
twotable
chained
3
2.5
2
1.5
1
100000
200000
300000
400000
500000
600000
700000
800000


Cache miss ratio for insertion of genome-based ints
google_dense
google_sparse
gnu
onetable
twotable
chained
9
8
7
6
5
4
3
2
1
100000
200000
300000
400000
500000
600000
700000
800000


Cache miss ratio for insertion of pointers to random strings
google_dense
google_sparse
gnu
onetable
twotable
chained
3
2.8
2.6
2.4
2.2
2
1.8
1.6
1.4
100000
200000
300000
400000
500000
600000
700000
800000

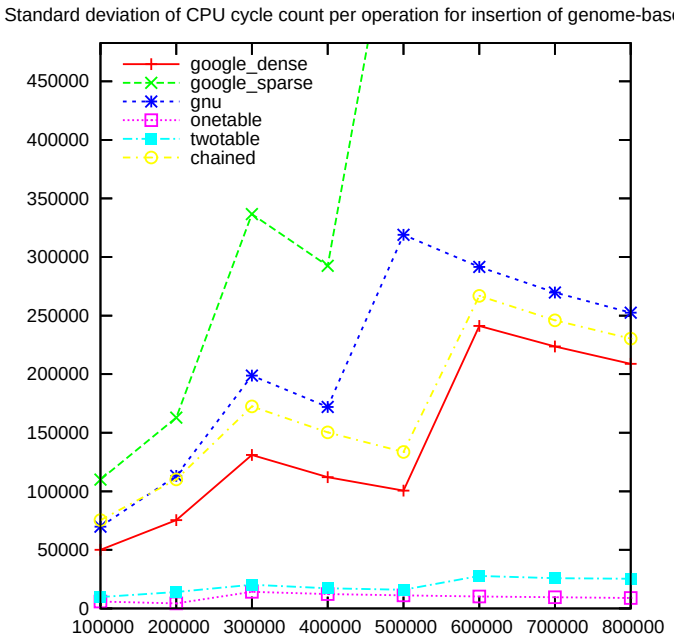

Cache miss ratio for insertion of pointers to filenames
google_dense
google_sparse
gnu
onetable
twotable
chained
2.2
2.1
2
1.9
1.8
1.7
1.6
1.5
1.4
1.3
100000
200000
300000
400000
500000
600000
700000
800000

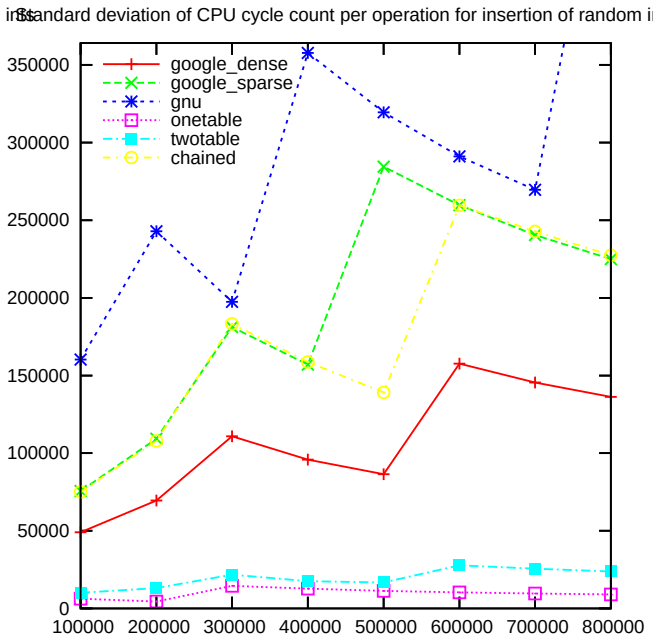

Cache misses for insertion of pointers to random strings
google_dense
google_sparse
gnu
onetable
twotable
chained
0
2e+06
4e+06
6e+06
8e+06
1e+07
100000
200000
300000
400000
500000
600000
700000
800000


Cache misses for insertion of pointers to filenames
google_dense
google_sparse
gnu
onetable
twotable
chained
0
2e+06
4e+06
6e+06
8e+06
1e+07
1.2e+07
100000
200000
300000
400000
500000
600000
700000
800000

Variability

- Linear hash tables show very little variability.

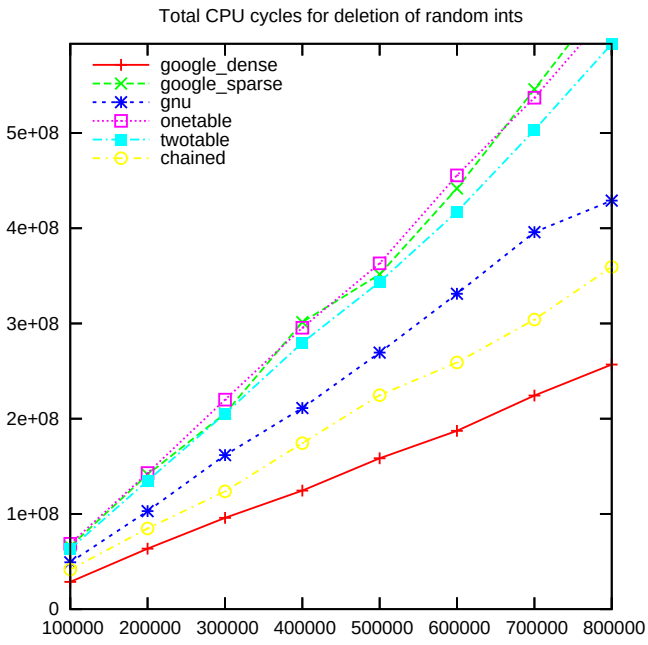


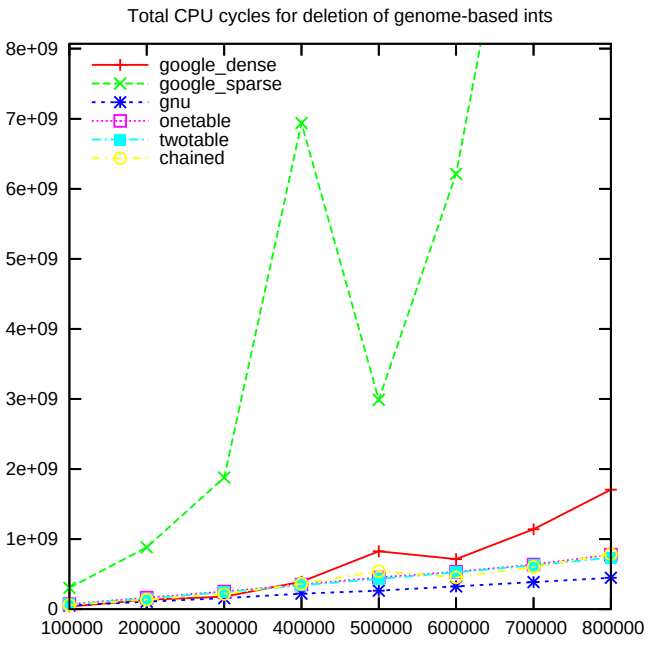


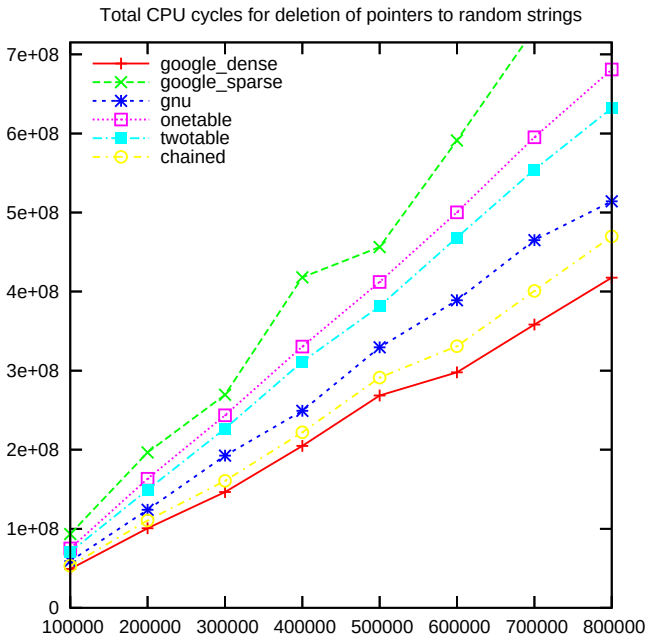
The `delete` operation

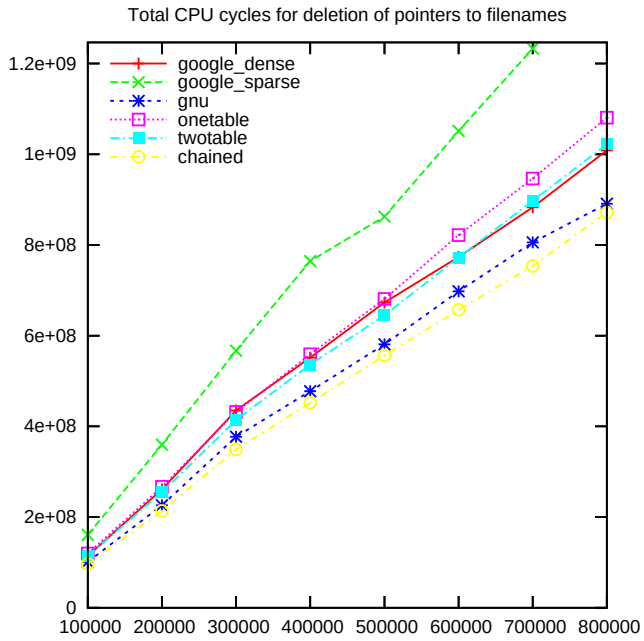
- Deletion is surprisingly effective on the google dense hash table

- The chained hash table is also very effective



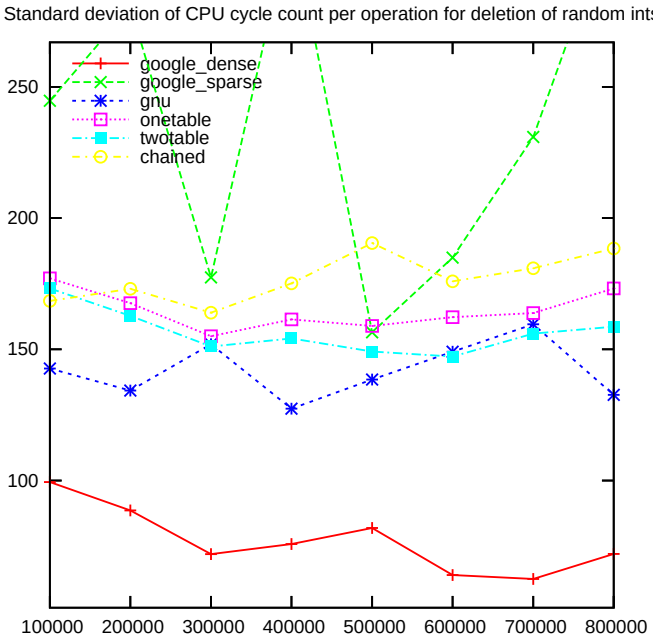


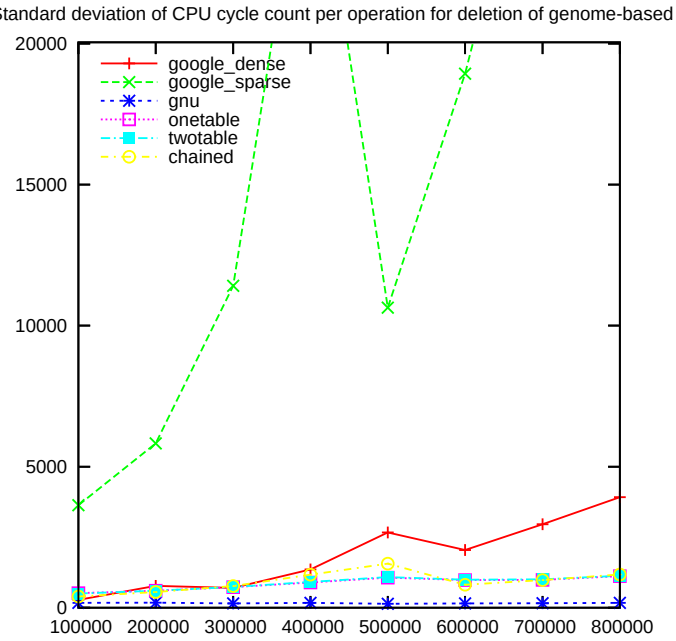

Total CPU cycles for deletion of pointers to random strings
google_dense
google_sparse
gnu
onetable
twotable
chained
7e+08
6e+08
5e+08
4e+08
3e+08
2e+08
1e+08
0
100000
200000
300000
400000
500000
600000
700000
800000

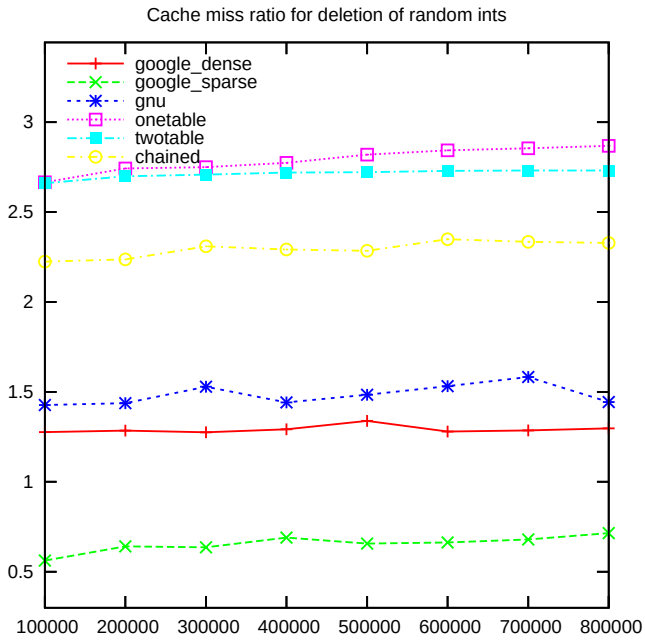

Total CPU cycles for deletion of pointers to filenames
google_dense
google_sparse
gnu
onetable
twotable
chained
1.2e+09
1e+09
8e+08
6e+08
4e+08
2e+08
0
100000
200000
300000
400000
500000
600000
700000
800000

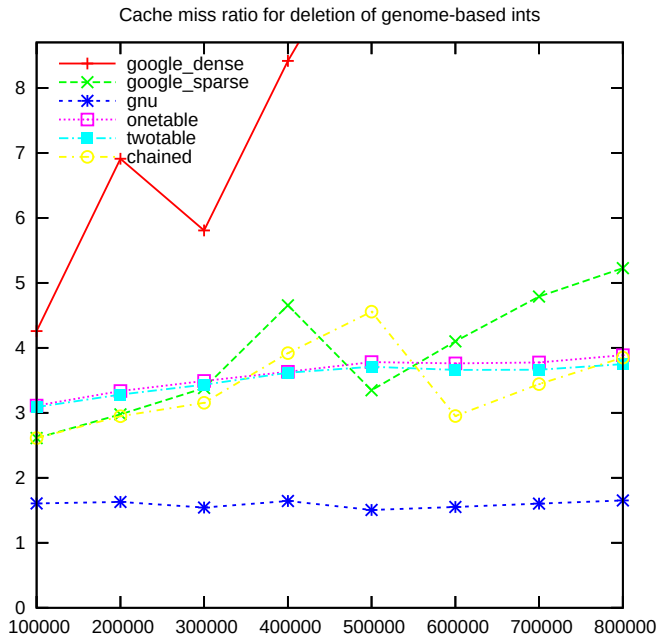
Variability

- Sparse does very bad on genome based data

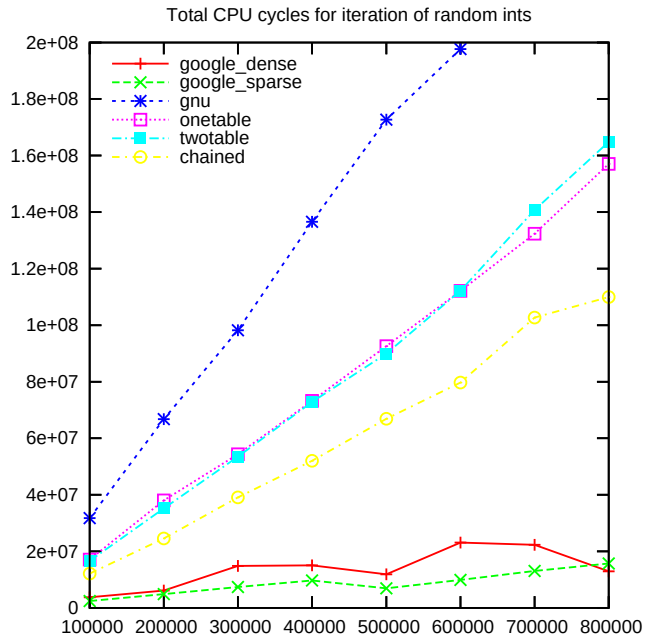


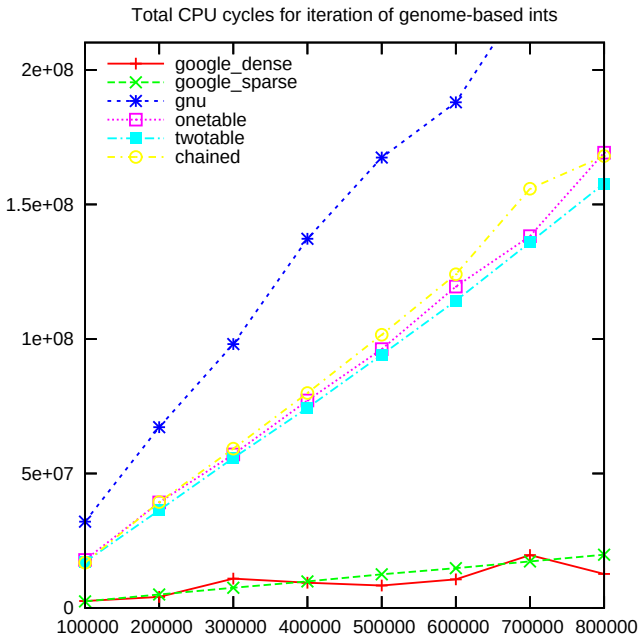


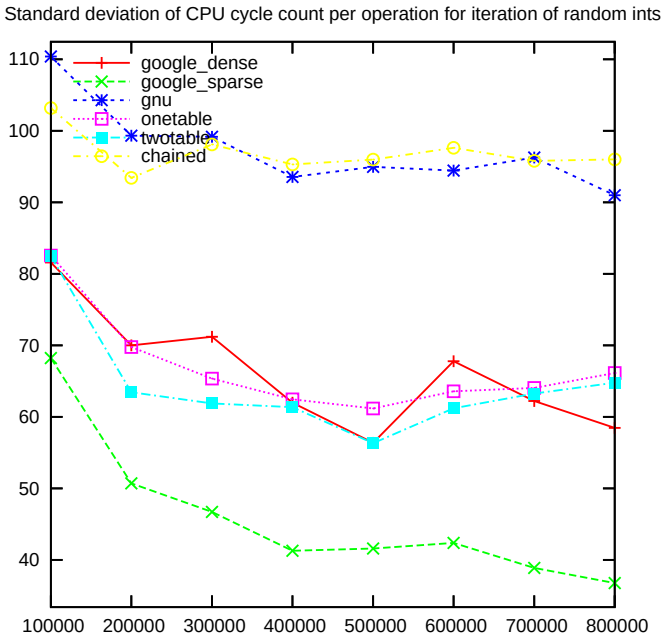

Cache miss ratio for deletion of random ints
google_dense
google_sparse
gnu
onetable
twotable
chained
3
2.5
2
1.5
1
0.5
100000
200000
300000
400000
500000
600000
700000
800000


Cache miss ratio for deletion of genome-based ints
google_dense
google_sparse
gnu
onetable
twotable
chained
8
7
6
5
4
3
2
1
0
100000
200000
300000
400000
500000
600000
700000
800000

Forward Iteration







Standard deviation of CPU cycle count per operation for iteration of genome-based ints