Subject: Map implementation
Posted by cbpporter on Thu, 21 Mar 2019 15:32:17 GMT
View Forum Message <> Reply to Message

I have implemented plenty of stuff since I'm a programmer, including many containers and even multiple small GUI toolkits (not as good as U++ of course :lol:) but I never implemented a hash maps, let alone one that is accessibly as an array like in U++.

Can I shamelessly dissect and steal VectorMap? :p

I'm currently studying HashBase, trying to figure out how it works and how the masking process works. Then Index...

Or are you aware of some other easy to learn and re-implement version of hash maps out there that will also perform more than adequately?

And how would you compare the U++ version to a more "standard" one?

Thanks!

Subject: Re: Map implementation
Posted by mirek on Fri, 22 Mar 2019 06:18:10 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Thu, 21 March 2019 16:32I have implemented plenty of stuff since I'm a programmer, including many containers and even multiple small GUI toolkits (not as good as U++ of course :lol:) but I never implemented a hash maps, let alone one that is accessibly as an array like in U++.

Can I shamelessly dissect and steal VectorMap? :p

It is opensource, is not it? And I have not applied for any patents.... :)

Quote:
Or are you aware of some other easy to learn and re-implement version of hash maps out there that will also perform more than adequately?

And how would you compare the U++ version to a more "standard" one?

Compared to standard one, it is "alien technology" :)

Mirek

Subject: Re: Map implementation
Posted by Novo on Wed, 27 Mar 2019 02:33:31 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Thu, 21 March 2019 11:32
Or are you aware of some other easy to learn and re-implement version of hash maps out there that will also perform more than adequately?

Useful links:
https://en.wikipedia.org/wiki/Open_addressing
https://probablydance.com/2017/02/26/i-wrote-the-fastest-has htable/
http://www.idryman.org/blog/2017/05/03/writing-a-damn-fast-h
ash-table-with-tiny-memory-footprints/
https://aras-p.info/blog/2016/08/02/Hash-Functions-all-the-w ay-down/
https://preshing.com/20160201/new-concurrent-hash-maps-for-c pp/
http://szelei.me/constexpr-murmurhash/
https://opensource.googleblog.com/2014/03/introducing-farmha sh.html

Hope this helps. :)

Subject: Re: Map implementation
Posted by cbpporter on Tue, 02 Apr 2019 12:13:18 GMT
View Forum Message <> Reply to Message

Thank you very much!

There is a lot to go through...

Subject: Re: Map implementation
Posted by Novo on Tue, 02 Apr 2019 15:48:08 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Tue, 02 April 2019 08:13
There is a lot to go through...
This is actually not that hard. All aspects of the hash table design are very well described in "I Wrote The Fastest Hashtable". You just need to choose your design. The easiest way to start with is "Powers of Two", linear probing, internal chaining. "internal chaining" means that you explicitly store a pointer to the next element in a chain, and "linear probing" means that you are using linear memory scan to find first available slot.

Ideally, your hash table should be completely policy-based, so you can easily replace linear probing with quadratic probing, for example.

It is not hard to implement a table having one fixed design. What is really hard is to make it completely policy-based.

Subject: Re: Map implementation
Posted by Novo on Tue, 02 Apr 2019 16:39:47 GMT
View Forum Message <> Reply to Message

Actually, the right name for "internal chaining" seems to be "Coalesced hashing".

Subject: Re: Map implementation
Posted by cbpporter on Mon, 08 Apr 2019 11:57:09 GMT
View Forum Message <> Reply to Message

I'm dissecting Index and noticed that Reindex calls ClearIndex and Free, while ClearIndex calls Free :).

Subject: Re: Map implementation
Posted by mirek on Mon, 08 Apr 2019 20:37:27 GMT
View Forum Message <> Reply to Message

Yeah, that is uncessary.

Anyway, Index will probably get overhaul soon. I plan to drop unused features and slightly change the semantics to gain more performance and cleaner API and operation.

Mirek

Subject: Re: Map implementation
Posted by cbpporter on Tue, 09 Apr 2019 12:57:46 GMT
View Forum Message <> Reply to Message

I re-implemented Index (copy&paste, trancode and cleanup) and should be a good starting point since it is smaller and simpler than a lot of maps out there.

I'll test mine thoroughly and benchmark, but before I benchmarked U++ vs. set vs. multiset, just to set my expectations accordingly.

U++ is pretty fast when compared to STL, but it does have exponential growth. With many items, in some tests, when multiplying items by 10, STL will go from like 250ms to 320, while Index will go from 250ms to 2 minutes :).

I used this simple test:

```
const int SEQ = 10;
const int MAX = 3000000 * SEQ;
const int JUMP = 100;
```

```
{
 Index<int> ind;

 StopWatch ts;

 for (int j = 0; j < MAX; j += JUMP)
  for (int i = j; i < j + SEQ; i++) {
   ind.Add(i);
   //ind.Debug(Cout());
  }

 Cout() << "U++ add " << ts.Elapsed() << "\n";

 ts.Reset();

 int count = 0;
 for (int i = 0; i < MAX; i++)
  if (ind.Find(i) != -1)
   count++;

 Cout() << "U++ find " << ts.Elapsed() << "\n";
 Cout() << count << "\n";
}

{
 std::set<int> ind;

 StopWatch ts;

 for (int j = 0; j < MAX; j += JUMP)
  for (int i = j; i < j + SEQ; i++) {
   ind.insert(i);
   //ind.Debug(Cout());
  }

 Cout() << "set add " << ts.Elapsed() << "\n";

 ts.Reset();

 int count = 0;
 for (int i = 0; i < MAX; i++)
  if (ind.find(i) != ind.end())
   count++;

 Cout() << "set find " << ts.Elapsed() << "\n";
 Cout() << count << "\n";
}
```

```
{
 std::multiset<int> ind;

 StopWatch ts;


 for (int j = 0; j < MAX; j += JUMP)
  for (int i = j; i < j + SEQ; i++) {
   ind.insert(i);
   //ind.Debug(Cout());
  }

 Cout() << "multiset add " << ts.Elapsed() << "\n";

 ts.Reset();

 int count = 0;
 for (int i = 0; i < MAX; i++)
  if (ind.find(i) != ind.end())
   count++;

 Cout() << "multiset find " << ts.Elapsed() << "\n";
 Cout() << count << "\n";
}
```

Is this an OK first artificial benchmark?

For MAX as large as the example, this is the starting point where U++ begins to loose vs. STL.

Next I need to compare memory usage...

Edit: I broke the benchmark into add and find, and the exponential growth for many items is at the find phase.

Subject: Re: Map implementation
Posted by mirek on Tue, 09 Apr 2019 15:16:16 GMT
View Forum Message <> Reply to Message

You have accidentally succeeded in attack on used hash mechanism - what you see is result of hash collisions. This is sad and something to deal with, however this would be much harder to achieve with String and quite unlikely to happen with real (aka random) data.

Still it is something I am worried about and will fix properly in the next version. For now, try to replace

```
inline dword FoldHash(dword h)
{
 return h - 362437 * SwapEndian32(h);
}

inline int& HashBase::Maph(unsigned _hash) const
{
 unsigned h = _hash & ~UNSIGNED_HIBIT;
 return map[mask & FoldHash(h)];
}
```

(In future, I will make '362437' number random prime, which will likely make this kind of attack impossible).

Mirek

EDIT: Forgot the change in Map.h

Subject: Re: Map implementation
Posted by mirek on Wed, 10 Apr 2019 09:07:09 GMT
View Forum Message <> Reply to Message

Update: Upon further investigation... The proposed change is not perfect. Better solution:

```
inline dword FoldHash(dword h)
{
 return SwapEndian32(2833151717 * h);
}
```

Interestingly, in this particular case it runs slower. It took me a couple of hours to figure out why: It is cache issue. This much better FoldHash actually spreads hashes nicely through hash space (thus causing cache misses), while the previous one tended to put them close (cache hits). It had accidentally fixed this particular benchmark's collisions, but would probably fail in some other scenario.

Conclusion: At this number of elements, the benchmark is memory bound (for int), so well working hasmap will perform similary to std::set (actually, I think the benchmark favors std::set here, as sequential numbers will repeat the same path in the binary tree, which is more cache friendly).

Original FoldHash was too simplistic for ink keys, this new version should be harder to attack.

In future version of Index, I will try to randomize FoldHash (and other hashing ops).

Mirek

Subject: Re: Map implementation
Posted by cbpporter on Wed, 10 Apr 2019 09:39:25 GMT
View Forum Message <> Reply to Message

I tested the "h - 362437 * SwapEndian32(h);" version and so far it seems to fix the collision issue and also regularly blows std::set out of the water. Needs more testing to cover enough cases. I also talked with a colleague and his hashmap version is supposedly 3x+ faster than stl, so it looks like it is very possible to greatly outperform it.

I will test the 2833151717 version too.

And with other types, like points and strings.

Anyway, Index has been highly educational. A lot of resources out there are either very basic, talking more about the principles of managing buckets or are about taking something that works very well and squeezing the last bits of performance out of it.

I'm still not experienced enough to tell how well things should be distributed when "map" inside HashBase grows, but I added plenty of debug methods...

Subject: Re: Map implementation
Posted by Novo on Wed, 10 Apr 2019 14:13:42 GMT
View Forum Message <> Reply to Message

mirek wrote on Wed, 10 April 2019 05:07Update: Upon further investigation... The proposed change is not perfect. Better solution:

```
inline dword FoldHash(dword h)
{
 return SwapEndian32(2833151717 * h);
}
```

Clang. Cpp14.

```
uppsrc/Core/Ops.h:289:9: error: call to 'SwapEndian32' is ambiguous
        return SwapEndian32(2833151717 * h);
               ^~~~~~~~~~~~
/home/ssg/dvlp/cpp/upp/git/uppsrc/Core/Ops.h:44:15: note: candidate function
inline dword  SwapEndian32(dword v)   { __asm__("bswap %0" : "=r" (v) : "0" (v)); return v; }
              ^
/home/ssg/dvlp/cpp/upp/git/uppsrc/Core/Ops.h:45:15: note: candidate function
inline int    SwapEndian32(int v)     { __asm__("bswap %0" : "=r" (v) : "0" (v)); return v; }
              ^
1 error generated.
```

Subject: Re: Map implementation
Posted by Novo on Wed, 10 Apr 2019 14:48:24 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Tue, 09 April 2019 08:57Is this an OK first artificial benchmark?

More scenarios for testing:

* Fill: insert a randomly shuffled sequence of unique keys into the table.
* Presized fill: like Fill, but first reserve enough memory for all the keys we'll insert, to prevent rehashing and reallocing during the fill process.
* Lookup: perform 100K lookups of random keys, all of which are in the table.
* Failed lookup: perform 100K lookups of random keys, none of which are in the table.
* Remove: remove a randomly chosen half of the elements from a table.
* Destruct: destroy a table and free its memory.

Subject: Re: Map implementation
Posted by Novo on Wed, 10 Apr 2019 15:09:47 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Wed, 10 April 2019 05:39I also talked with a colleague and his hashmap version is supposedly 3x+ faster than stl, so it looks like it is very possible to greatly outperform it.

IMHO, it is impossible to create one ideal hash table which will greatly outperform STL in all possible scenarios.
Let's take a look at two scenarios.

1. One million hash tables containing one hundred records.
2. One hash table containing one hundred million records.

You will need two completely different implementations in these cases.

Two more scenarios.

1. Add data once and search for data most of the time.
2. Add/remove data most of the time and search for it occasionally.

Again, you will need two completely different implementations.

Subject: Re: Map implementation
Posted by mirek on Wed, 10 Apr 2019 15:37:07 GMT
View Forum Message <> Reply to Message

Novo wrote on Wed, 10 April 2019 17:09cbpporter wrote on Wed, 10 April 2019 05:39I also talked with a colleague and his hashmap version is supposedly 3x+ faster than stl, so it looks like it is very possible to greatly outperform it.

IMHO, it is impossible to create one ideal hash table which will greatly outperform STL in all possible scenarios.

That is probably true, but mostly because at some point the limiting factor becomes cache / memory performance.

Quote:
Let's take a look at two scenarios.

1. One million hash tables containing one hundred records.
2. One hash table containing one hundred million records.

You will need two completely different implementations in these cases.

That might be true, however I do not see a way how to improve Index for either (I see some accumulated knowledge how to improve it for both, but that is another story).

Quote:
1. Add data once and search for data most of the time.
2. Add/remove data most of the time and search for it occasionally.

Ditto.

About the only thing that is in question is how to deal with collisions. Some advanced hashmaps

might e.g. use binary trees to resolve collisions. I believe that it is not an overal gain (and the fact that it is not widely used in industry makes it likely) and that we should rather invest time to investigate proper hashing techniques.

Anyway, the real benchmark would be to create real world scenario and test there. IMO U++ Index/VectorMap wins that easily.

Mirek

Subject: Re: Map implementation
Posted by mirek on Wed, 10 Apr 2019 15:50:13 GMT
View Forum Message <> Reply to Message

[quote title=Novo wrote on Wed, 10 April 2019 16:13]mirek wrote on Wed, 10 April 2019 05:07Update: Upon further investigation... The proposed change is not perfect. Better solution:

```
inline dword FoldHash(dword h)
{
 return SwapEndian32(2833151717 * h);
}
```

OK, seems I have run out of dword range accidentally here... :)

```
inline dword FoldHash(dword h)
{
 return SwapEndian32(0xa3613c16 * h);
}
```

Subject: Re: Map implementation
Posted by Novo on Sat, 13 Apr 2019 18:58:27 GMT
View Forum Message <> Reply to Message

mirek wrote on Wed, 10 April 2019 11:37
That is probably true, but mostly because at some point the limiting factor becomes cache / memory performance.

There are ways to deal with this. From the top of my head: Cache-Conscious Data Structures,

Cache-oblivious algorithms.

Subject: Re: Map implementation
Posted by Novo on Sat, 13 Apr 2019 19:07:00 GMT
View Forum Message <> Reply to Message

mirek wrote on Wed, 10 April 2019 11:37
Quote:
Let's take a look at two scenarios.

1. One million hash tables containing one hundred records.
2. One hash table containing one hundred million records.

You will need two completely different implementations in these cases.

That might be true, however I do not see a way how to improve Index for either (I see some accumulated knowledge how to improve it for both, but that is another story).

IMHO, this is impossible because in the first case your main concern is the memory usage. Tiny overhead multiplied by million is a huge problem. And in the second case performance is the main issue.

Subject: Re: Map implementation
Posted by Novo on Sat, 13 Apr 2019 19:25:22 GMT
View Forum Message <> Reply to Message

mirek wrote on Wed, 10 April 2019 11:37About the only thing that is in question is how to deal with collisions. Some advanced hashmaps might e.g. use binary trees to resolve collisions. I believe that it is not an overal gain (and the fact that it is not widely used in industry makes it likely) and that we should rather invest time to investigate proper hashing techniques.

Anyway, the real benchmark would be to create real world scenario and test there. IMO U++ Index/VectorMap wins that easily.

AFAIK, STL can't use open addressing or other such techniques because it is specified to maintain stable key/value addresses. U++ Index doesn't support that. This is why it is possible to make it faster.

About collisions. There are many ways to deal with them. A classic approach is to store chain either explicitly (a list) or implicitly (you know how to calculate address of the next colliding slot). I saw a paper from Microsoft recommending an overflow area ...

Basically, what I want to say is that there is a million of different ways to design a hash table. IMHO, the best way is to split it into multiple policies, so you can easily replace one part of it

without rewriting the whole data structure. :)
I personally couldn't figure out how to do that. :)

Subject: Re: Map implementation
Posted by mirek on Sun, 14 Apr 2019 06:03:45 GMT
View Forum Message <> Reply to Message

Novo wrote on Sat, 13 April 2019 21:07mirek wrote on Wed, 10 April 2019 11:37
Quote:
Let's take a look at two scenarios.

1. One million hash tables containing one hundred records.
2. One hash table containing one hundred million records.

You will need two completely different implementations in these cases.

That might be true, however I do not see a way how to improve Index for either (I see some accumulated knowledge how to improve it for both, but that is another story).

IMHO, this is impossible because in the first case your main concern is the memory usage. Tiny overhead multiplied by million is a huge problem. And in the second case performance is the main issue.

If the performance is the issue, then the memory is the issue too. The game starts at L1 cache size, which can correspond to hunderds of records.

Subject: Re: Map implementation
Posted by Novo on Sun, 14 Apr 2019 16:55:14 GMT
View Forum Message <> Reply to Message

mirek wrote on Sun, 14 April 2019 02:03If the performance is the issue, then the memory is the issue too. The game starts at L1 cache size, which can correspond to hunderds of records.
How to deal with the memory hierarchy is more or less clear (Cache-Conscious Data Structures, Cache-oblivious algorithms).
What I'm trying to say is that by using a little bit more memory you can significantly improve performance. For example, you can create a bitset of unoccupied slots. That would be overkill for a small hash table.

I made a simple test.

```
Vector<Index<int> > v;
Cout() << "sizeof(Index<int>): " << sizeof(Index<int>) << " bytes" << EOL;
Cout() << "Mem used: " << MemoryUsedKb() - curMU << " Kb" << EOL;
v.SetCount(v_num);
Cout() << "Created " << v_num << " empty Index<int>" << EOL;
```

```
 Cout() << "Mem used: " << MemoryUsedKb() - curMU << " Kb" << EOL;
 const int isize = 100;
 for (int i = 0; i < isize; ++i) {
  const int jsize = v_num;
  for (int j = 0; j < jsize; ++j)
   v[j].Add(i);
  Cout() << "Added " << i + 1 << " elements" << EOL;
  Cout() << "Mem used: " << MemoryUsedKb() - curMU << " Kb" << EOL;
 }
```

Result:
sizeof(Index<int>): 80 bytes
Mem used: 0 Kb
Created 1000000 empty Index<int>
Mem used: 78128 Kb
Added 1 elements
Mem used: 237028 Kb
Added 2 elements
Mem used: 237028 Kb
Added 3 elements
Mem used: 237028 Kb
Added 4 elements
Mem used: 237028 Kb
Added 5 elements
Mem used: 237028 Kb
Added 6 elements
Mem used: 237028 Kb
Added 7 elements
Mem used: 237028 Kb
Added 8 elements
Mem used: 237028 Kb
Added 9 elements
Mem used: 397796 Kb
Added 10 elements
Mem used: 397796 Kb
...
Added 99 elements
Mem used: 2592740 Kb
Added 100 elements
Mem used: 2592740 Kb

IMHO, it is possible to do much better than this ...

File Attachments

1) test_ht_perf.zip, downloaded 213 times

Subject: Re: Map implementation
Posted by mirek on Sun, 14 Apr 2019 17:52:22 GMT
View Forum Message <> Reply to Message

Novo wrote on Sun, 14 April 2019 18:55mirek wrote on Sun, 14 April 2019 02:03If the performance is the issue, then the memory is the issue too. The game starts at L1 cache size, which can correspond to hunderds of records.
How to deal with the memory hierarchy is more or less clear (Cache-Conscious Data Structures, Cache-oblivious algorithms).
What I'm trying to say is that by using a little bit more memory you can significantly improve performance. For example, you can create a bitset of unoccupied slots. That would be overkill for a small hash table.

I made a simple test.

```
 Vector<Index<int> > v;
 Cout() << "sizeof(Index<int>): " << sizeof(Index<int>) << " bytes" << EOL;
 Cout() << "Mem used: " << MemoryUsedKb() - curMU << " Kb" << EOL;
 v.SetCount(v_num);
 Cout() << "Created " << v_num << " empty Index<int>" << EOL;
 Cout() << "Mem used: " << MemoryUsedKb() - curMU << " Kb" << EOL;
 const int isize = 100;
 for (int i = 0; i < isize; ++i) {
  const int jsize = v_num;
  for (int j = 0; j < jsize; ++j)
   v[j].Add(i);
  Cout() << "Added " << i + 1 << " elements" << EOL;
  Cout() << "Mem used: " << MemoryUsedKb() - curMU << " Kb" << EOL;
 }
```

Result:

```
sizeof(Index<int>): 80 bytes
Mem used: 0 Kb
Created 1000000 empty Index<int>
Mem used: 78128 Kb
Added 1 elements
Mem used: 237028 Kb
Added 2 elements
Mem used: 237028 Kb
Added 3 elements
Mem used: 237028 Kb
Added 4 elements
Mem used: 237028 Kb
Added 5 elements
Mem used: 237028 Kb
Added 6 elements
Mem used: 237028 Kb
Added 7 elements
Mem used: 237028 Kb
Added 8 elements
```

Mem used: 237028 Kb
Added 9 elements
Mem used: 397796 Kb
Added 10 elements
Mem used: 397796 Kb
...
Added 99 elements
Mem used: 2592740 Kb
Added 100 elements
Mem used: 2592740 Kb

IMHO, it is possible to do much better than this ...

Well, keep in mind that lower bound here is 400MB - that is what will cost to store keys itself. If these keys were String, which in reality is much more typical, it would be 1600MB just for values (if they are small).

Current Index overhead is on average ~20 bytes per element, which about matches what we see here. Hard to say you can do much better. E.g. typical 'collisions are linked list' implementation will use about the same number of bytes. Even open addressing will need at least 8 bytes per node if you want to have any meaningful 'payload'.

Anyway, thanks for test. You are right that you can reduce that for very specific scenarios. And next index version will probably do about 20% better.

BTW, test putting 100 * 1000000 elements into single Index will produce the same results.

Mirek

P.S.: Overall I am happy that you both started digging here as I am thinking about refactoring Index, deprecating and detuning some unsused features (ever used FindPrev? :) in favor of those used most frequently.

Subject: Re: Map implementation
Posted by mirek on Tue, 16 Apr 2019 10:38:40 GMT
View Forum Message <> Reply to Message

Novo wrote on Sun, 14 April 2019 18:55
Added 100 elements
Mem used: 2592740 Kb
[/code]

IMHO, it is possible to do much better than this ...

Just for reference (Visual C++ 64 bit release):

```
#include <Core/Core.h>
#include <set>
#include <vector>

using namespace Upp;

CONSOLE_APP_MAIN
{
 int curMU = MemoryUsedKb();
 int v_num = 1000000;
 std::vector< std::set<int> > v;
 Cout() << "sizeof(Index<int>): " << sizeof(Index<int>) << " bytes" << EOL;
 Cout() << "Mem used: " << MemoryUsedKb() - curMU << " Kb" << EOL;
 v.resize(v_num);
 Cout() << "Created " << v_num << " empty Index<int>" << EOL;
 Cout() << "Mem used: " << MemoryUsedKb() - curMU << " Kb" << EOL;
 const int isize = 100;
 for (int i = 0; i < isize; ++i) {
  const int jsize = v_num;
  for (int j = 0; j < jsize; ++j)
   v[j].insert(i);
  Cout() << "Added " << i + 1 << " elements" << EOL;
  Cout() << "Mem used: " << MemoryUsedKb() - curMU << " Kb" << EOL;
 }
}
```

```
Added 100 elements
Mem used: 3222476 Kb
```

Replace set with unorderd_set and

```
Added 100 elements
Mem used: 12412392 Kb

sizeof(std::unordered_set<int>) = 64
```

GCC 64bits, unorderd_set

```
Added 100 elements
Mem used: 4621148 Kb

sizeof(std::unordered_set<int>) = 56
```

BTW, I am now banging my head about reducing overhead from 20 bytes to 16 per element. It looks like it might not be possible without performance degradation (in all scenarios). So I would say it is really hard to get below 20 per element.

Mirek

Subject: Re: Map implementation
Posted by Novo on Tue, 16 Apr 2019 14:38:39 GMT
View Forum Message <> Reply to Message

BTW, In the project that I attached there are two configurations. A default one, which is using an U++ allocator, and a second one, which is using a standard allocator.
You should get different numbers with different configurations.

Subject: Re: Map implementation
Posted by mirek on Tue, 16 Apr 2019 17:33:19 GMT
View Forum Message <> Reply to Message

Novo wrote on Tue, 16 April 2019 16:38BTW, In the project that I attached there are two configurations. A default one, which is using an U++ allocator, and a second one, which is using a standard allocator.
You should get different numbers with different configurations.

Yeah, the only problem is that MemoryUsedKb does not work with standard allocator...

Subject: Re: Map implementation
Posted by Novo on Tue, 16 Apr 2019 20:36:24 GMT
View Forum Message <> Reply to Message

mirek wrote on Tue, 16 April 2019 13:33Novo wrote on Tue, 16 April 2019 16:38BTW, In the project that I attached there are two configurations. A default one, which is using an U++ allocator, and a second one, which is using a standard allocator.
You should get different numbers with different configurations.

Yeah, the only problem is that MemoryUsedKb does not work with standard allocator...
I was looking at top :)
I'll try to find a memory profiler ...

Subject: Re: Map implementation

Posted by Novo on Wed, 17 Apr 2019 03:40:13 GMT
View Forum Message <> Reply to Message

Novo wrote on Tue, 16 April 2019 16:36
I'll try to find a memory profiler ...
Massif. Release conf + debug info. Index<int>
std::set<int> is using 3.8Gb ...

File Attachments

1) Screenshot_2019-04-16_23-33-31.png, downloaded 536 times

Subject: Re: Map implementation
Posted by mirek on Wed, 17 Apr 2019 06:54:17 GMT
View Forum Message <> Reply to Message

Novo wrote on Wed, 17 April 2019 05:40Novo wrote on Tue, 16 April 2019 16:36
I'll try to find a memory profiler ...
Massif. Release conf + debug info. Index<int>
std::set<int> is using 3.8Gb ...

Well, nothing surprising there, right?

Mirek

Subject: Re: Map implementation
Posted by Novo on Wed, 17 Apr 2019 14:13:01 GMT
View Forum Message <> Reply to Message

mirek wrote on Wed, 17 April 2019 02:54
Well, nothing surprising there, right?

Yes, but ...

$ ./test_ht_perf
Mem used: 78128 Kb
Index<int> Add: 7.035
Index<int> Unlink: 12.272
Mem used: 2592740 Kb
Mem used after Sweep: 3800292 Kb
Mem used: 3769040 Kb
std::set<int> insert: 7.381
std::set<int> erase: 4.296

Mem used: 7603920 Kb

```
 if (true) {
  Vector<Index<int> > v;
  v.SetCount(v_num);
  const int isize = 100;
  Cout() << "Mem used: " << MemoryUsedKb() - curMU << " Kb" << EOL;
  TimeStop ts;
  for (int i = 0; i < isize; ++i) {
   const int jsize = v_num;
   for (int j = 0; j < jsize; ++j)
    v[j].Add(i);
  }
  Cout() << "Index<int> Add: " << ts.ToString() << EOL;
  ts.Reset();
  for (int i = 0; i < isize; ++i) {
   const int jsize = v_num;
   for (int j = 0; j < jsize; ++j)
    v[j].UnlinkKey(i);
  }
  Cout() << "Index<int> Unlink: " << ts.ToString() << EOL;
  Cout() << "Mem used: " << MemoryUsedKb() - curMU << " Kb" << EOL;
  const int jsize = v_num;
  for (int j = 0; j < jsize; ++j)
   v[j].Sweep();
  Cout() << "Mem used after Sweep: " << MemoryUsedKb() - curMU << " Kb" << EOL;
 }

 if (true) {
  std::set<int>* v;
  v = new std::set<int>[v_num];
  const int isize = 100;
  Cout() << "Mem used: " << MemoryUsedKb() - curMU << " Kb" << EOL;
  TimeStop ts;
  for (int i = 0; i < isize; ++i) {
   const int jsize = v_num;
   for (int j = 0; j < jsize; ++j)
    v[j].insert(i);
  }
  Cout() << "std::set<int> insert: " << ts.ToString() << EOL;
  ts.Reset();
  for (int i = 0; i < isize; ++i) {
   const int jsize = v_num;
   for (int j = 0; j < jsize; ++j)
    v[j].erase(i);
  }
  Cout() << "std::set<int> erase: " << ts.ToString() << EOL;
```

```
 Cout() << "Mem used: " << MemoryUsedKb() - curMU << " Kb" << EOL;
}
```

Project is attached.

std::set<int> erase is three times faster than Index<int> Unlink.
After calling Index::Sweep even more memory is used. I guess this is a problem with the allocator.
And Index invalidates pointers.
So ...

File Attachments

1) test_ht_perf.zip, downloaded 200 times

Subject: Re: Map implementation
Posted by mirek on Wed, 17 Apr 2019 16:39:05 GMT
View Forum Message <> Reply to Message

Novo wrote on Wed, 17 April 2019 16:13mirek wrote on Wed, 17 April 2019 02:54
Well, nothing surprising there, right?

Yes, but ...

```
$ ./test_ht_perf
Mem used: 78128 Kb
Index<int> Add: 7.035
Index<int> Unlink: 12.272
Mem used: 2592740 Kb
Mem used after Sweep: 3800292 Kb
Mem used: 3769040 Kb
std::set<int> insert: 7.381
std::set<int> erase: 4.296
Mem used: 7603920 Kb
```

```
 if (true) {
  Vector<Index<int> > v;
  v.SetCount(v_num);
  const int isize = 100;
  Cout() << "Mem used: " << MemoryUsedKb() - curMU << " Kb" << EOL;
  TimeStop ts;
  for (int i = 0; i < isize; ++i) {
   const int jsize = v_num;
   for (int j = 0; j < jsize; ++j)
    v[j].Add(i);
  }
```

```
 Cout() << "Index<int> Add: " << ts.ToString() << EOL;
 ts.Reset();
 for (int i = 0; i < isize; ++i) {
  const int jsize = v_num;
  for (int j = 0; j < jsize; ++j)
   v[j].UnlinkKey(i);
 }
 Cout() << "Index<int> Unlink: " << ts.ToString() << EOL;
 Cout() << "Mem used: " << MemoryUsedKb() - curMU << " Kb" << EOL;
 const int jsize = v_num;
 for (int j = 0; j < jsize; ++j)
  v[j].Sweep();
 Cout() << "Mem used after Sweep: " << MemoryUsedKb() - curMU << " Kb" << EOL;
}

if (true) {
 std::set<int>* v;
 v = new std::set<int>[v_num];
 const int isize = 100;
 Cout() << "Mem used: " << MemoryUsedKb() - curMU << " Kb" << EOL;
 TimeStop ts;
 for (int i = 0; i < isize; ++i) {
  const int jsize = v_num;
  for (int j = 0; j < jsize; ++j)
   v[j].insert(i);
 }
 Cout() << "std::set<int> insert: " << ts.ToString() << EOL;
 ts.Reset();
 for (int i = 0; i < isize; ++i) {
  const int jsize = v_num;
  for (int j = 0; j < jsize; ++j)
   v[j].erase(i);
 }
 Cout() << "std::set<int> erase: " << ts.ToString() << EOL;
 Cout() << "Mem used: " << MemoryUsedKb() - curMU << " Kb" << EOL;
}
```

Project is attached.

std::set<int> erase is three times faster than Index<int> Unlink.
After calling Index::Sweep even more memory is used. I guess this is a problem with the allocator.
And Index invalidates pointers.
So ...

Cool intresting catch.

Took me a while digging into the memory issue and it is really interesting - it looks like the problem is in this line

Quote:
```
void HashBase::FinishIndex()
{
 int q = link.GetCount();
 link.Reserve(hash.GetAlloc()); <==== HERE
 link.AddN(hash.GetCount() - q);
 for(int i = q; i < hash.GetCount(); i++)
  LinkTo(i, link[i], hash[i] & UNSIGNED_HIBIT ? unlinked : Mapi(i));
}
```

If you comment it out, all works fine. The reason is that at that point, 'overreservation' of link in this particular leads to going from small blocks to large ones. Those small ones then are left free for further use, which in this example never materializes.

I will definitely keep this scenario for testing with the new Index... That said, this really is not likely to happen in real app.

Going to look into Unlink issue now.

Subject: Re: Map implementation
Posted by mirek on Fri, 07 Jun 2019 11:58:31 GMT
View Forum Message <> Reply to Message

Index and allocator are now refactored, should behave better in synthetic benchmarks as is this one.

Subject: Re: Map implementation
Posted by Novo on Tue, 25 Jun 2019 13:20:28 GMT
View Forum Message <> Reply to Message

mirek wrote on Fri, 07 June 2019 07:58Index and allocator are now refactored, should behave better in synthetic benchmarks as is this one.

It performs better, but Unlink is still ~2.5 times slower than std::set::erase.

Mem used: 39064 Kb
Index<int> Add: 5.691
Index<int> Unlink: 9.380
Mem used: 2959732 Kb
Index<int> Sweep: 0.204

Mem used after Sweep: 2959732 Kb
Index<int> Shrink: 0.122
Mem used after Shrink: 39096 Kb
Mem used: 46908 Kb
std::set<int> insert: 5.975
std::set<int> erase: 3.612
Mem used: 46904 Kb

Subject: Re: Map implementation
Posted by mirek on Wed, 26 Jun 2019 10:07:56 GMT
View Forum Message <> Reply to Message

Well, with benchmark constructed like this, beating set<int>::erase is accidentally hard.

Interesting things is that if you change the order of loops:

```
#include <Core/Core.h>
#include <set>

using namespace Upp;

CONSOLE_APP_MAIN
{
#ifdef _DEBUG
 const int v_num = 10000;
#else
 const int v_num = 100000;
#endif

 const int isize = 100;

 {
  Vector<Index<int> > v;
  v.SetCount(v_num);
  {
   RTIMING("FindAdd v_num outer");
   for (int j = 0; j < v_num; ++j)
    for (int i = 0; i < isize; ++i)
     v[j].FindAdd(i);
  }
  {
   RTIMING("UnlinkKey v_num outer");
   for (int j = 0; j < v_num; ++j)
    for (int i = 0; i < isize; ++i)
     v[j].UnlinkKey(i);
```

```
 }
 RTIMING("Sweep v_num outer");
 const int jsize = v_num;
 for (int j = 0; j < jsize; ++j)
  v[j].Sweep();
}
{
 Vector<Index<int> > v;
 v.SetCount(v_num);
 {
  RTIMING("FindAdd v_num inner");
  for (int i = 0; i < isize; ++i)
   for (int j = 0; j < v_num; ++j)
    v[j].FindAdd(i);
 }
 {
  RTIMING("UnlinkKey v_num inner");
  for (int i = 0; i < isize; ++i)
   for (int j = 0; j < v_num; ++j)
    v[j].UnlinkKey(i);
 }
 RTIMING("Sweep v_num inner");
 const int jsize = v_num;
 for (int j = 0; j < jsize; ++j)
  v[j].Sweep();
}

{
 std::set<int> *v = new std::set<int>[v_num];
 {
  RTIMING("insert v_num outer");
  for (int j = 0; j < v_num; ++j)
   for (int i = 0; i < isize; ++i)
    v[j].insert(i);
 }

 {
  RTIMING("erase v_num outer");
  for (int j = 0; j < v_num; ++j)
   for (int i = 0; i < isize; ++i)
    v[j].erase(i);
 }
 delete[] v;
}

{
 std::set<int> *v = new std::set<int>[v_num];
 {
```

```
   RTIMING("insert v_num inner");
   for (int i = 0; i < isize; ++i)
    for (int j = 0; j < v_num; ++j)
     v[j].insert(i);
  }

  {
   RTIMING("erase v_num inner");
   for (int i = 0; i < isize; ++i)
    for (int j = 0; j < v_num; ++j)
     v[j].erase(i);
  }
  delete[] v;
 }
}
```

results are very different:

```
* C:\upp\out\benchmarks\MINGWx64\Index.exe 26.06.2019 11:36:42, user: cxl

TIMING erase v_num inner: 480.00 ms - 480.00 ms (480.00 ms / 1 ), min: 480.00 ms, max:
480.00 ms, nesting: 0 - 1
TIMING insert v_num inner: 702.00 ms - 702.00 ms (702.00 ms / 1 ), min: 702.00 ms, max:
702.00 ms, nesting: 0 - 1
TIMING erase v_num outer: 427.00 ms - 427.00 ms (427.00 ms / 1 ), min: 427.00 ms, max:
427.00 ms, nesting: 0 - 1
TIMING insert v_num outer: 399.00 ms - 399.00 ms (399.00 ms / 1 ), min: 399.00 ms, max:
399.00 ms, nesting: 0 - 1
TIMING Sweep v_num inner: 22.00 ms - 22.00 ms (22.00 ms / 1 ), min: 22.00 ms, max: 22.00 ms,
nesting: 0 - 1
TIMING UnlinkKey v_num inner: 995.00 ms - 995.00 ms (995.00 ms / 1 ), min: 995.00 ms, max:
995.00 ms, nesting: 0 - 1
TIMING FindAdd v_num inner: 683.00 ms - 683.00 ms (683.00 ms / 1 ), min: 683.00 ms, max:
683.00 ms, nesting: 0 - 1
TIMING Sweep v_num outer: 28.00 ms - 28.00 ms (28.00 ms / 1 ), min: 28.00 ms, max: 28.00 ms,
nesting: 0 - 1
TIMING UnlinkKey v_num outer: 118.00 ms - 118.00 ms (118.00 ms / 1 ), min: 118.00 ms, max:
118.00 ms, nesting: 0 - 1
TIMING FindAdd v_num outer: 242.00 ms - 242.00 ms (242.00 ms / 1 ), min: 242.00 ms, max:
242.00 ms, nesting: 0 - 1
```

which probably means that set<int> is more cache friendly in the original benchmark....

Subject: Re: Map implementation
Posted by Novo on Mon, 01 Jul 2019 04:26:44 GMT
View Forum Message <> Reply to Message

Initial profiling showed this:

Almost all time of Unlink is spent in inline dword FoldHash(dword h)

Most expensive are if-statements:

```
if(i >= 0)
if(key[i] == k) {
```

That is all I can tell at the moment ...

File Attachments
1) Screenshot_2019-07-01_00-20-11.png, downloaded 391 times

Subject: Re: Map implementation
Posted by mirek on Mon, 01 Jul 2019 15:53:08 GMT
View Forum Message <> Reply to Message

Things are quite different if instead of incremental pattern you feed in random data:

```
#include <Core/Core.h>
#include <set>

using namespace Upp;

CONSOLE_APP_MAIN
{
#ifdef _DEBUG
 const int v_num = 10000;
#else
 const int v_num = 1000;
#endif

 const int isize = 100;
 const int N = 100;

 Vector<int> data;
 for(int i = 0; i < isize * v_num; i++)
  data.Add(Random());
```

```
for(int ii = 0; ii < N; ii++) {
 {
  Vector<Index<int> > v;
  v.SetCount(v_num);
  {
   RTIMING("inner FindAdd v_num");
   int *s = data;
   for (int i = 0; i < isize; ++i)
    for (int j = 0; j < v_num; ++j)
     v[j].FindAdd(*s++);
  }
  {
   RTIMING("inner UnlinkKey v_num");
   int *s = data;
   for (int i = 0; i < isize; ++i)
    for (int j = 0; j < v_num; ++j)
     v[j].UnlinkKey(*s++);
  }
  RTIMING("inner Sweep v_num");
  const int jsize = v_num;
  for (int j = 0; j < jsize; ++j)
   v[j].Sweep();
 }
 {
  Vector<Index<int> > v;
  v.SetCount(v_num);
  {
   RTIMING("outer FindAdd v_num");
   int *s = data;
   for (int j = 0; j < v_num; ++j)
    for (int i = 0; i < isize; ++i)
     v[j].FindAdd(*s++);
  }
  {
   RTIMING("outer UnlinkKey v_num");
   int *s = data;
   for (int j = 0; j < v_num; ++j)
    for (int i = 0; i < isize; ++i)
     v[j].UnlinkKey(*s++);
  }
  RTIMING("outer Sweep v_num");
  const int jsize = v_num;
  for (int j = 0; j < jsize; ++j)
   v[j].Sweep();
 }

 {
  std::set<int> *v = new std::set<int>[v_num];
```

```
  {
   RTIMING("outer insert v_num");
   int *s = data;
   for (int j = 0; j < v_num; ++j)
    for (int i = 0; i < isize; ++i)
     v[j].insert(*s++);
  }

  {
   RTIMING("outer erase v_num");
   int *s = data;
   for (int j = 0; j < v_num; ++j)
    for (int i = 0; i < isize; ++i)
     v[j].erase(*s++);
  }
  delete[] v;
 }

 {
  std::set<int> *v = new std::set<int>[v_num];
  {
   RTIMING("inner insert v_num");
   int *s = data;
   for (int i = 0; i < isize; ++i)
    for (int j = 0; j < v_num; ++j)
     v[j].insert(*s++);
  }

  {
   RTIMING("inner erase v_num");
   int *s = data;
   for (int i = 0; i < isize; ++i)
    for (int j = 0; j < v_num; ++j)
     v[j].erase(*s++);
  }
  delete[] v;
 }
 }
}
```

I guess incremental data somehow favors set, my guess is that it works as accidental prefetch here...