Subject: Writing Bits object to disk
Posted by crydev on Wed, 11 Jan 2017 16:07:50 GMT
View Forum Message <> Reply to Message

Hello,

I was thinking about using Bits as an efficient data structure to write my data to disk. However, I noticed that this data structure is quite closed and does not allow callers to retrieve a pointer to the internal buffer of bits and neither does it allow itself being constructed from existing buffer and alloc variables.

Would it be a good idea to allow this, or have a similar data structure that allows retrieval of the data structure? If not, why do you think so? I now manually edited some support in, to see if my efficient idea works out well!

Thanks!

crydev

Subject: Re: Writing Bits object to disk
Posted by mr_ped on Thu, 12 Jan 2017 00:20:40 GMT
View Forum Message <> Reply to Message

If your target is small disk space, you should rather allocate data in memory in reasonable common sizes, semantically grouped together, and run that through zlib compression, if those data resemble at least some patterns, this should save lot more, than packing them into bits and later having headache when you will want to extend them a bit.

Subject: Re: Writing Bits object to disk
Posted by crydev on Thu, 12 Jan 2017 08:03:50 GMT
View Forum Message <> Reply to Message

mr_ped wrote on Thu, 12 January 2017 01:20If your target is small disk space, you should rather allocate data in memory in reasonable common sizes, semantically grouped together, and run that through zlib compression, if those data resemble at least some patterns, this should save lot more, than packing them into bits and later having headache when you will want to extend them a bit.

For random values this would be a good idea, but not for structured data that I want to write out. I have a sorted set of memory addresses, identified by the memory page they reside in. I can reduce space by at most 32 times if I save it in bits, and at most 8 times (in x64) if I use a Vector<bool>. Compression takes too much time, I tried. :)

Subject: Re: Writing Bits object to disk
Posted by mirek on Fri, 13 Jan 2017 07:39:10 GMT
View Forum Message <> Reply to Message

crydev wrote on Wed, 11 January 2017 17:07Hello,

I was thinking about using Bits as an efficient data structure to write my data to disk. However, I noticed that this data structure is quite closed and does not allow callers to retrieve a pointer to the internal buffer of bits and neither does it allow itself being constructed from existing buffer and alloc variables.

Would it be a good idea to allow this, or have a similar data structure that allows retrieval of the data structure? If not, why do you think so? I now manually edited some support in, to see if my efficient idea works out well!

Thanks!

crydev

I guess this is valid idea. Adding to RM.

Mirek

Subject: Re: Writing Bits object to disk
Posted by crydev on Wed, 18 Jan 2017 07:51:16 GMT
View Forum Message <> Reply to Message

mirek wrote on Fri, 13 January 2017 08:39crydev wrote on Wed, 11 January 2017 17:07Hello,

I was thinking about using Bits as an efficient data structure to write my data to disk. However, I noticed that this data structure is quite closed and does not allow callers to retrieve a pointer to the internal buffer of bits and neither does it allow itself being constructed from existing buffer and alloc variables.

Would it be a good idea to allow this, or have a similar data structure that allows retrieval of the data structure? If not, why do you think so? I now manually edited some support in, to see if my efficient idea works out well!

Thanks!

crydev

I guess this is valid idea. Adding to RM.

Mirek

Great, thanks!

crydev

Subject: Re: Writing Bits object to disk
Posted by crydev on Mon, 24 Apr 2017 17:19:34 GMT
View Forum Message <> Reply to Message

Hello,

I changed the Bits class to expose its buffer and allocation variables, and ported my code using Vector<bool> to its equivalent using Bits. However, I realized that setting billions of bits in hot loops is very inefficient in Bits. The speedup I get from having to write 8 times less data to disk is eliminated by the slow computations.

Why is it so inefficient? Would it be a good idea to keep using Vector<bool>, and convert to Bits before writing to the file?

Thanks,

crydev

Subject: Re: Writing Bits object to disk
Posted by mirek on Mon, 24 Apr 2017 19:08:56 GMT
View Forum Message <> Reply to Message

crydev wrote on Mon, 24 April 2017 19:19Hello,

I changed the Bits class to expose its buffer and allocation variables, and ported my code using Vector<bool> to its equivalent using Bits. However, I realized that setting billions of bits in hot loops is very inefficient in Bits. The speedup I get from having to write 8 times less data to disk is eliminated by the slow computations.

Why is it so inefficient? Would it be a good idea to keep using Vector<bool>, and convert to Bits before writing to the file?

Thanks,

crydev

I guess it depends on what you do. What is ratio between Vector<bool> speed and Bits?

(This is actually a sort of nice problem to optimize :)

Mirek

Subject: Re: Writing Bits object to disk
Posted by crydev on Tue, 25 Apr 2017 07:36:55 GMT
View Forum Message <> Reply to Message

A ran the following code in a performance test 100,000 times.

```
// Original function implementing Vector<bool>.
const int VectorBoolOrBitsetTestOriginal(bool* const buffer, const bool* const rand)
{
 int x = 0;
 for (int i = 0; i < 4096; ++i)
 {
  if (rand)
  {
   ++x;
  }
  buffer[i] = rand;
 }
 return x;
}

// Different function implementing Bits.
const int VectorBoolOrBitsetTestBitSet(Bits& buffer, const bool* const rand)
{
 int x = 0;
 for (int i = 0; i < 4096; ++i)
 {
  if (rand)
  {
   ++x;
  }
  buffer.Set(i, rand);
 }
 return x;
}

// Different function implementing std::bitset.
const int VectorBoolOrBitsetTestStdBitSet(std::bitset<4096>& buffer, const bool* const rand)
{
 int x = 0;
 for (int i = 0; i < 4096; ++i)
 {
  if (rand)
  {
   ++x;
  }
  buffer.set(i, rand);
 }
 return x;
}
```

The result is as follows, Bits being approximately a factor 10 slower. std::bitset already seems to be a twice as fast:

crydev

File Attachments

1) Capture.PNG, downloaded 840 times

Subject: Re: Writing Bits object to disk

Posted by dolik.rce on Tue, 25 Apr 2017 07:48:07 GMT

View Forum Message <> Reply to Message

By the way, I just read an article on very similar topic: http://www.bfilipek.com/2017/04/packing-bools.html. It might contain some useful insights, especially in the second part which was not published yet.

Best regards,
Honza

Subject: Re: Writing Bits object to disk

Posted by mirek on Tue, 25 Apr 2017 08:31:35 GMT

View Forum Message <> Reply to Message

crydev wrote on Tue, 25 April 2017 09:36A ran the following code in a performance test 100,000 times.

```
// Original function implementing Vector<bool>.
const int VectorBoolOrBitsetTestOriginal(bool* const buffer, const bool* const rand)
{
 int x = 0;
 for (int i = 0; i < 4096; ++i)
 {
  if (rand)
  {
   ++x;
  }
  buffer[i] = rand;
 }
 return x;
}

// Different function implementing Bits.
const int VectorBoolOrBitsetTestBitSet(Bits& buffer, const bool* const rand)
{
 int x = 0;
 for (int i = 0; i < 4096; ++i)
 {
  if (rand)
  {
   ++x;
  }
  buffer.Set(i, rand);
 }
 return x;
}

// Different function implementing std::bitset.
const int VectorBoolOrBitsetTestStdBitSet(std::bitset<4096>& buffer, const bool* const rand)
{
 int x = 0;
 for (int i = 0; i < 4096; ++i)
 {
  if (rand)
  {
   ++x;
  }
  buffer.set(i, rand);
 }
 return x;
}
```

The result is as follows, Bits being approximately a factor 10 slower. std::bitset already seems to be a twice as fast:

crydev

The first quick observation of Bits code (after all these years) reveals that there are pretty good oportunities to optimize this. Woohoo, optimization time!

BTW, could you please post me your benchamrk package zipped? Would save me a bit of time.

Also, I am still undecided about correct interface to expose raw data. Currectly I am thinking along something like:

const byte *ReadRaw(int& count_of_bytes);
byte *WriteRaw(int count_of_bytes);

WriteRaw would make sure that there is at least count bytes available for bits (without reallocating array down).

Mirek

Subject: Re: Writing Bits object to disk
Posted by mirek on Tue, 25 Apr 2017 08:34:08 GMT
View Forum Message <> Reply to Message

P.S.: Are you sure that your Test code is doing what you wanted it to do? It looks to me like there should be something like *rand++ in it...

Subject: Re: Writing Bits object to disk
Posted by crydev on Tue, 25 Apr 2017 09:37:19 GMT
View Forum Message <> Reply to Message

mirek wrote on Tue, 25 April 2017 10:34P.S.: Are you sure that your Test code is doing what you wanted it to do? It looks to me like there should be something like *rand++ in it...

The above testing code is a subset of my real code, where x is a value I just introduced to have a return value to show. It does not represent anything, and shouldn't be taken into account. The rand variable is a pointer to an array of random bools. I wanted to use it to create some randomness, but I didn't really succeed there. :d I solely wanted to point out the runtime of the different implementations. :)

As Bits optimization, I also figured that there is no Reserve(int) function to pre-allocate the internal buffer. I think it would be a good idea to add such, because it sharply reduces the number of reallocates necessary. I also think that setting a bitmask can be vectorized. However, that would be a task for myself, because U++ is made to be portable. :)

Thanks,

crydev

Subject: Re: Writing Bits object to disk
Posted by mirek on Tue, 25 Apr 2017 09:59:24 GMT
View Forum Message <> Reply to Message

crydev wrote on Tue, 25 April 2017 11:37mirek wrote on Tue, 25 April 2017 10:34P.S.: Are you sure that your Test code is doing what you wanted it to do? It looks to me like there should be something like *rand++ in it...

The above testing code is a subset of my real code, where x is a value I just introduced to have a return value to show. It does not represent anything, and shouldn't be taken into account. The rand variable is a pointer to an array of random bools. I wanted to use it to create some randomness, but I didn't really succeed there. :d I solely wanted to point out the runtime of the different implementations. :)

I have no problem with 'x', but it looks to like you are setting all bits to 1 (because you are actually not reading rand values).

Mirek

Subject: Re: Writing Bits object to disk
Posted by mirek on Tue, 25 Apr 2017 10:38:52 GMT

View Forum Message <> Reply to Message

Actually, I think this might be the reason for value you are getting.

I am benchmarking with this:

```
CONSOLE_APP_MAIN
{
 Vector<bool> data;
 for(int i = 0; i < 100000; i++)
  data.Add(Random() & 1);

 int N = 1000;
 for(int k = 0; k < N; k++) {
  {
   RTIMING("Vector<bool>");
   Vector<bool> h;
   for(int j = 0; j < data.GetCount(); j++)
    h.At(j, false) = data[j];
  }
  {
   RTIMING("Bits");
   Bits h;
   for(int j = 0; j < data.GetCount(); j++)
    h.Set(j, data[j]);
  }
 }
}
```

This showed Bits to be only about 15% slower than Vector<bool>.

Then I have tried some very basic optimization (reorganize with inline Set) and suddenly, Bits are about 20% FASTER than Vector.

I hope I am not getting anything wrong...

Any ideas about the proper "raw data" interface?

Subject: Re: Writing Bits object to disk
Posted by crydev on Tue, 25 Apr 2017 11:35:41 GMT
View Forum Message <> Reply to Message

Thanks for your replies Mirek! I fixed my testcases, I realized I had a wrong check in my loop. The results now are:

Then, I applied your optimizations, and Bits became a little faster. However, It still is not as fast as Vector<bool>.

Quote:Any ideas about the proper "raw data" interface?

I thought about making a constructor that allows construction of Bits from an existing buffer.

I also thought about vectorizing Bits set method. In theory, we could gather 16 bools, invert the bits in this bool, such that the most significant bit is 1 if the bool value is true, and 0 if it is false. Then, the _mm_movemask_epi8 intrinsic will generate an instruction that directly converts these 16 bools to a bitmask. We can also assume that 0x80 is true, for our inverted bool. :)

crydev

File Attachments
1) new.PNG, downloaded 773 times
2) Capture.PNG, downloaded 773 times

Subject: Re: Writing Bits object to disk
Posted by mirek on Tue, 25 Apr 2017 11:39:40 GMT
View Forum Message <> Reply to Message

It is still weird that you are getting different numbers than me.

Could you perhaps try my benchmark?

Are you benchmarking "release" mode?

What CPU / Compiler are you using? Do you have latest theide (with FAST release mode always on)?

Subject: Re: Writing Bits object to disk
Posted by crydev on Tue, 25 Apr 2017 12:03:26 GMT
View Forum Message <> Reply to Message

mirek wrote on Tue, 25 April 2017 13:39It is still weird that you are getting different numbers than me.

Could you perhaps try my benchmark?

Are you benchmarking "release" mode?

What CPU / Compiler are you using? Do you have latest theide (with FAST release mode always on)?

I updated my TheIDE to the latest version, but it did not make a difference. I am using the Visual C++ compiler from Visual Studio 2015. My CPU is a Core i7 2600k. I compiled with Release mode, and the following compiler flags: -O2 /GS- /Qvec-report:2

What is FAST release mode? I also tried your RTIMING option, but it gives me the same results as my own measurement. :)

Subject: Re: Writing Bits object to disk
Posted by mirek on Tue, 25 Apr 2017 14:40:40 GMT
View Forum Message <> Reply to Message

crydev wrote on Tue, 25 April 2017 14:03mirek wrote on Tue, 25 April 2017 13:39It is still weird that you are getting different numbers than me.

Could you perhaps try my benchmark?

Are you benchmarking "release" mode?

What CPU / Compiler are you using? Do you have latest theide (with FAST release mode always on)?

I updated my TheIDE to the latest version, but it did not make a difference. I am using the Visual C++ compiler from Visual Studio 2015. My CPU is a Core i7 2600k. I compiled with Release mode, and the following compiler flags: -O2 /GS- /Qvec-report:2

What is FAST release mode? I also tried your RTIMING option, but it gives me the same results as my own measurement. :)

Weird the only difference seems to be CPU (i7 4771 here)...

Have you tried my benchmark as it is?

That said, even if those numbers you are getting were real, I guess it is now close to Vector<bool> anyway.

Subject: Re: Writing Bits object to disk
Posted by crydev on Wed, 26 Apr 2017 06:27:15 GMT

View Forum Message <> Reply to Message

mirek wrote on Tue, 25 April 2017 16:40crydev wrote on Tue, 25 April 2017 14:03mirek wrote on Tue, 25 April 2017 13:39It is still weird that you are getting different numbers than me.

Could you perhaps try my benchmark?

Are you benchmarking "release" mode?

What CPU / Compiler are you using? Do you have latest theide (with FAST release mode always on)?

I updated my TheIDE to the latest version, but it did not make a difference. I am using the Visual C++ compiler from Visual Studio 2015. My CPU is a Core i7 2600k. I compiled with Release mode, and the following compiler flags: -O2 /GS- /Qvec-report:2

What is FAST release mode? I also tried your RTIMING option, but it gives me the same results as my own measurement. :)

Weird the only difference seems to be CPU (i7 4771 here)...

Have you tried my benchmark as it is?

That said, even if those numbers you are getting were real, I guess it is now close to Vector<bool> anyway.

I haven't tried your benchmark, I could try that too. I see that Bits is coming closer to Vector<bool> in set performance, but at http://www.bfilipek.com/2017/04/packing-bools.html they provided a method that is even faster. How do you feel about a Reserve(int) function for Bits?
 (Good that Upp:Bits is now faster than std::bitset btw :).) Since I set billions of bits, maybe a vector set method accepting 16 bools (with value 0x80) is an even bigger performance

improvement.

Thanks,

crydev

Subject: Re: Writing Bits object to disk
Posted by crydev on Wed, 26 Apr 2017 06:50:34 GMT
View Forum Message <> Reply to Message

Mirek,

I ran your benchmark, and I found out why I get different results! Your benchmark gives me:

```
TIMING Bits          : 194.98 ms - 194.98 us (195.00 ms / 1000 ), min:  0.00 ns, max:  1.00 ms, nesting: 1 - 1000
TIMING Vector<bool>   : 362.98 ms - 362.98 us (363.00 ms / 1000 ), min:  0.00 ns, max:  1.00 ms, nesting: 1 - 1000
```

Constructing the Vector<bool> inside the loop also slows down the whole thing by a lot. After I moved the construction out, I got:

```
TIMING Bits          : 183.98 ms - 183.98 us (184.00 ms / 1000 ), min:  0.00 ns, max:  1.00 ms, nesting: 1 - 1000
TIMING Vector<bool>   : 102.98 ms - 102.98 us (103.00 ms / 1000 ), min:  0.00 ns, max:  1.00 ms, nesting: 1 - 1000
```

I pre-allocate the Vector<bool> with the Reserve(int) function, because I know what the size of the vector is going to be. This is also the case in my production application. That's why the Vector is still faster. We need a reserve function for Bits too. :)

crydev

Subject: Re: Writing Bits object to disk
Posted by mirek on Wed, 26 Apr 2017 11:38:03 GMT
View Forum Message <> Reply to Message

OK. So I think I am finished with Bits for now. This is now available:

```
 void   Reserve(int nbits);
 void   Shrink();

 dword       *CreateRaw(int n_dwords);
 const dword *Raw(int& n_dwords) const { n_dwords = alloc; return bp; }
 dword       *Raw(int& n_dwords)       { n_dwords = alloc; return bp; }
```

Mirek

Subject: Re: Writing Bits object to disk
Posted by crydev on Thu, 27 Apr 2017 08:31:43 GMT
View Forum Message <> Reply to Message

Thanks Mirek!

I have been trying a thing or two myself. I wrote a version of the Set(int, bool) method that accepts four bool values at the same time. This version _probably_ has more pipelining capabilities, and is faster than the regular set method!

```
union BitBoolPipeline
{
 struct
 {
  bool b1;
  bool b2;
  bool b3;
  bool b4;
 };

 dword dw;
};

// Different function implementing Bits, but with a pipelined set method.
const int VectorBoolOrBitsetTestBitSetPipeline(Bits& buffer, const Vector<bool>& rand)
{
 const int count = rand.GetCount();
 for (int i = 0; i < count; i += 4)
 {
  // We can now set four bools at the same time. However, we must assume that a bool that
  // is set to true has value 0x1, and a bool set to false has value 0x0.
  BitBoolPipeline b;
  b.b1 = rand[i];
  b.b2 = rand[i + 1];
  b.b3 = rand[i + 2];
  b.b4 = rand[i + 3];
  buffer.PipelineSet(i, b.dw);
 }
 int alloc = 0;
 buffer.Raw(alloc);
 return alloc;
}

const dword PowersOfTwo[] =
{
 0x1, 0x2, 0x4, 0x8, 0x10, 0x20, 0x40, 0x80, 0x100, 0x200, 0x400, 0x800, 0x1000, 0x2000,
 0x4000, 0x8000, 0x10000, 0x20000, 0x40000, 0x80000, 0x100000, 0x200000, 0x400000, 0x800000,
 0x1000000, 0x2000000, 0x4000000, 0x8000000, 0x10000000, 0x20000000, 0x40000000, 0x80000000
};

// Testing pipelined version of Bits::Set(int, bool)
void Bits::PipelineSet(int i, const dword bs)
{
 // Check whether i is within the bounds of the container.
 ASSERT(i >= 0 && alloc >= 0);

 // Get the DWORD index for the internal buffer.
 int q = i >> 5;

 // Get the bit index of the next available DWORD.
 i &= 31;

 // Do we need to expand the internal buffer first?
 // Also check whether we can place 4 bits in the existing DWORD. If not, we should expand.
 if(q >= alloc)
  Expand(q);

 // Get integer bit values according to existing DWORD value and indices.
 // Assuming default value of bool is 0x1 if true!
 bp[q] = (bp[q] | bs & PowersOfTwo[1] | bs & PowersOfTwo[8] | bs & PowersOfTwo[16] | bs & PowersOfTwo[24]);
}
```

This function assumes that a bool set to true has value 0x1, and 0x0 otherwise. I made the array of constant powers of two in advance. Replacing the array indices with the array constant itself doesn't improve any further, because the compiler propagates the constants by itself. The measurement is as follows:

I'm also working on a SIMD version that should be able to set 16 bools in parallel, having some assumptions about the input. :)

crydev

File Attachments

1) Capture.PNG, downloaded 749 times

Subject: Re: Writing Bits object to disk

Posted by mirek on Thu, 27 Apr 2017 21:50:04 GMT

View Forum Message <> Reply to Message

Problem with this approach is that you have to create intput Vector<bool> argument first, which is likely to spoil any benefits from faster Bits...

Really, this is the issue - to improve speed here, the interface is problem.

Some possible solutions that came to my mind:

```
void Bits::Set(int pos, int count, dword bits);
```

Here count <= 32 and you are passing values in bits dword; that would work if you are packing some normal data into Bits.

```
template
void Bits::Set(int pos, int count, auto lambda /* [=] (int pos) -> bool */)
```

Here we would provide lambda that returns value for given position - if compiler is good, it should inline well.

```
tempalte
void Bits::Set(int pos, bool x ...)
```

Maybe vararg template is a possible solution to the problem too.

Now the question is: How are you using Bits?

Mirek

Subject: Re: Writing Bits object to disk

Posted by crydev on Fri, 28 Apr 2017 17:04:45 GMT

View Forum Message <> Reply to Message

mirek wrote on Thu, 27 April 2017 23:50Problem with this approach is that you have to create intput Vector<bool> argument first, which is likely to spoil any benefits from faster Bits...

Really, this is the issue - to improve speed here, the interface is problem.

Some possible solutions that came to my mind:

void Bits::Set(int pos, int count, dword bits);

Here count <= 32 and you are passing values in bits dword; that would work if you are packing some normal data into Bits.

template
void Bits::Set(int pos, int count, auto lambda /* [=] (int pos) -> bool */)

Here we would provide lambda that returns value for given position - if compiler is good, it should inline well.

tempalte
void Bits::Set(int pos, bool x ...)

Maybe vararg template is a possible solution to the problem too.

Now the question is: How are you using Bits?

Mirek

Thanks Mirek,

I am building a memory scanner. The results of this scanner consist of a set of addresses and a set of corresponding values. The addresses are fairly well structured. That is: they live in pages, and always live at a specific offset from a base address. Therefore, I can efficiently store this data by using Bits. Every bit is an address, and I keep track of the metadata like base address and offsets. However, that means that I have to write billions of bits to the Bits structure. Therefore, my application benefits from vectorized approaches of setting. :)

Having to create an input vector of bools may not be very efficient, but it is more efficient in my test case. Moreover, it is also a nice solution to have a Bits::Set method do this in another way, not having to do it yourself. :)

I understand that vectorized versions of the Bits::Set function are not portable and should not be in U++ for portability reasons, but instruction pipelining is available on many architectures. I think it can be well exploited in this case!

crydev

Subject: Re: Writing Bits object to disk
Posted by omari on Fri, 28 Apr 2017 17:31:56 GMT

View Forum Message <> Reply to Message

Hi,
Is it possible to add this function:

```
 void Zero() { if(bp) Fill(bp, bp + alloc, (dword)0);}
```

It allows the reuse of a reserved object.

And if possible ToString (I doubt of it's portability in BIG ENDIAN)

```
String Bits::ToString() const
{
 String ss;
 for(int i = alloc-1; i >= 0; i--)
  ss << FormatIntHex(bp[i]);
 return ss;
}
```

Subject: Re: Writing Bits object to disk
Posted by mirek on Sat, 29 Apr 2017 07:05:12 GMT

View Forum Message <> Reply to Message

[quote title=crydev wrote on Fri, 28 April 2017 19:04][quote title=mirek wrote on Thu, 27 April 2017 23:50]Problem with this Quote:
I understand that vectorized versions of the Bits::Set function are not portable and should not be in U++ for portability reasons,

That is not a problem at all. I just still cannot sort out how with Vector<bool> interface this can be faster.

My understanding is that setting Vector<bool> alone is only marginally faster than setting Bits. So setting Vector<bool> AND then setting this Vector<bool> to Bits can hardly be faster....

Subject: Re: Writing Bits object to disk
Posted by mirek on Sat, 29 Apr 2017 07:53:30 GMT
View Forum Message <> Reply to Message

Quote:

```
String Bits::ToString() const
{
 String ss;
 for(int i = alloc-1; i >= 0; i--)
  ss << FormatIntHex(bp[i]);
 return ss;
}
```

OK, but not so sure that this format is good...

Subject: Re: Writing Bits object to disk
Posted by crydev on Tue, 02 May 2017 20:54:52 GMT
View Forum Message <> Reply to Message

Hi Mirek,

I realized that my pipelined function was wrong. I made a silly mistake in the test case. The function is now fixed as:

```
// Testing pipelined version of Bits::Set(int, bool)
void Bits::PipelineSet(int i, const dword bs)
{
 // Check whether i is within the bounds of the container.
 ASSERT(i >= 0 && alloc >= 0);

 // Get the DWORD index for the internal buffer.
 int q = i >> 5;

 // Get the bit index of the next available DWORD.
 i &= 31;

 // Do we need to expand the internal buffer first?
 // Also check whether we can place 4 bits in the existing DWORD. If not, we should expand.
 if(q >= alloc)
  Expand(q);

 // Get integer bit values according to existing DWORD value and indices.
 // Assuming default value of bool is 0x1 if true!
 const dword d1 = !!(bs & PowersOfTwo[0]) << i;
 const dword d2 = !!(bs & PowersOfTwo[8]) << (i + 1);
 const dword d3 = !!(bs & PowersOfTwo[16]) << (i + 2);
 const dword d4 = !!(bs & PowersOfTwo[24]) << (i + 3);
 bp[q] = (bp[q] | d1 | d2 | d3 | d4);
}
```

I did some more tests, and I am really surprised by the amount of difference the compiler and CPU make in this situation. I had to switch from my i7 2600k CPU to an Intel Core i7 4710MQ CPU because I was missing AVX2 (AVX2 really made it fast) :) When compiled with the Visual C++ compiler, I got the following result.

I was surprised to see how the newer CPU runs the simple pipelined version a lot faster than the 2600k! I also saw that a vectorized version of the Set function is almost simpler than the regular one. :) However, when I use the Intel C++ compiler, the results are very different:

It seems that the Intel compiler generates way different code, making the SSE2 version blazingly fast. 10 times faster than the regular Set function. The strange thing is, that when I use the Visual C++ compiler, the timing of the different test cases is as I expected them to be. I expected the AVX2 function to be faster than the SSE2 one, which clearly is not the case with the Intel compiler.

The sources of my vectorized functions is as following:

```
// Testing vectorized version of Bits::Set(int, bool)
// We require that the input bools have value 0x80, e.g. most significant byte set if true.
void Bits::VectorSet(int i, const unsigned char vec[16])
{
 // Check whether i is within the bounds of the container.
 ASSERT(i >= 0 && alloc >= 0);

 // Get the DWORD index for the internal buffer.
 int q = i >> 5;

 // Do we need to expand the internal buffer first?
 if(q >= alloc)
  Expand(q);

 // Get the bit index of the next available DWORD.
 i &= 31;

 // Create a bitmask with vector intrinsics.
 __m128i boolVec = _mm_set_epi8(vec[15], vec[14], vec[13], vec[12], vec[11], vec[10], vec[9],
vec[8]
  , vec[7], vec[6], vec[5], vec[4], vec[3], vec[2], vec[1], vec[0]);
 const int bitMask = _mm_movemask_epi8(boolVec);

 // Set the resulting WORD.
 LowHighDword w;
 w.dw = bp[q];
 if (i == 16)
 {
  w.w2 = (short)bitMask;
 }
 else
 {
  w.w1 = (short)bitMask;
 }
 bp[q] = w.dw;
}

// The same vectorized function, but with AVX2 instructions.
void Bits::VectorSetAVX2(int i, const unsigned char vec[32])
{
 // Check whether i is within the bounds of the container.
 ASSERT(i >= 0 && alloc >= 0);

 // Get the DWORD index for the internal buffer.
 int q = i >> 5;

 // Do we need to expand the internal buffer first?
 if(q >= alloc)
  Expand(q);

 // Get the bit index of the next available DWORD.
 i &= 31;

 // Create a bitmask with vector intrinsics.
 __m256i boolVec = _mm256_set_epi8(vec[31], vec[30], vec[29], vec[28], vec[27], vec[26],
vec[25], vec[24], vec[23]
  , vec[22], vec[21], vec[20], vec[19], vec[18], vec[17], vec[16], vec[15], vec[14], vec[13], vec[12],
vec[11]
  , vec[10], vec[9], vec[8], vec[7], vec[6], vec[5], vec[4], vec[3], vec[2], vec[1], vec[0]);
 const int bitMask = _mm256_movemask_epi8(boolVec);

 // Set the resulting DWORD.
 bp[q] = bitMask;
}
```

Is it feasible to make a vectorized version for U++ by default, or should I provide it for myself?

Thanks,

evo

File Attachments

1) Capture.PNG, downloaded 693 times
2) Capture2.PNG, downloaded 746 times

Subject: Re: Writing Bits object to disk

Posted by mirek on Tue, 02 May 2017 21:55:53 GMT
View Forum Message <> Reply to Message

crydev wrote on Tue, 02 May 2017 22:54Hi Mirek,

I realized that my pipelined function was wrong. I made a silly mistake in the test case. The function is now fixed as:

```
// Testing pipelined version of Bits::Set(int, bool)
void Bits::PipelineSet(int i, const dword bs)
{
 // Check whether i is within the bounds of the container.
 ASSERT(i >= 0 && alloc >= 0);

 // Get the DWORD index for the internal buffer.
 int q = i >> 5;

 // Get the bit index of the next available DWORD.
 i &= 31;

 // Do we need to expand the internal buffer first?
 // Also check whether we can place 4 bits in the existing DWORD. If not, we should expand.
 if(q >= alloc)
  Expand(q);

 // Get integer bit values according to existing DWORD value and indices.
 // Assuming default value of bool is 0x1 if true!
 const dword d1 = !!(bs & PowersOfTwo[0]) << i;
 const dword d2 = !!(bs & PowersOfTwo[8]) << (i + 1);
 const dword d3 = !!(bs & PowersOfTwo[16]) << (i + 2);
 const dword d4 = !!(bs & PowersOfTwo[24]) << (i + 3);
 bp[q] = (bp[q] | d1 | d2 | d3 | d4);
}
```

I did some more tests, and I am really surprised by the amount of difference the compiler and

CPU make in this situation. I had to switch from my i7 2600k CPU to an Intel Core i7 4710MQ CPU because I was missing AVX2 (AVX2 really made it fast) :) When compiled with the Visual C++ compiler, I got the following result.

I was surprised to see how the newer CPU runs the simple pipelined version a lot faster than the 2600k! I also saw that a vectorized version of the Set function is almost simpler than the regular one. :) However, when I use the Intel C++ compiler, the results are very different:

It seems that the Intel compiler generates way different code, making the SSE2 version blazingly fast. 10 times faster than the regular Set function. The strange thing is, that when I use the Visual C++ compiler, the timing of the different test cases is as I expected them to be. I expected the AVX2 function to be faster than the SSE2 one, which clearly is not the case with the Intel compiler.

The sources of my vectorized functions is as following:

```
// Testing vectorized version of Bits::Set(int, bool)
// We require that the input bools have value 0x80, e.g. most significant byte set if true.
void Bits::VectorSet(int i, const unsigned char vec[16])
{
 // Check whether i is within the bounds of the container.
 ASSERT(i >= 0 && alloc >= 0);

 // Get the DWORD index for the internal buffer.
 int q = i >> 5;

 // Do we need to expand the internal buffer first?
 if(q >= alloc)
  Expand(q);

 // Get the bit index of the next available DWORD.
 i &= 31;

 // Create a bitmask with vector intrinsics.
 __m128i boolVec = _mm_set_epi8(vec[15], vec[14], vec[13], vec[12], vec[11], vec[10], vec[9],
vec[8]
  , vec[7], vec[6], vec[5], vec[4], vec[3], vec[2], vec[1], vec[0]);
 const int bitMask = _mm_movemask_epi8(boolVec);

 // Set the resulting WORD.
 LowHighDword w;
 w.dw = bp[q];
 if (i == 16)
 {
  w.w2 = (short)bitMask;
 }
 else
 {
  w.w1 = (short)bitMask;
 }
 bp[q] = w.dw;
}

// The same vectorized function, but with AVX2 instructions.
void Bits::VectorSetAVX2(int i, const unsigned char vec[32])
{
 // Check whether i is within the bounds of the container.
 ASSERT(i >= 0 && alloc >= 0);

 // Get the DWORD index for the internal buffer.
 int q = i >> 5;

 // Do we need to expand the internal buffer first?
 if(q >= alloc)
  Expand(q);

 // Get the bit index of the next available DWORD.
 i &= 31;

 // Create a bitmask with vector intrinsics.
 __m256i boolVec = _mm256_set_epi8(vec[31], vec[30], vec[29], vec[28], vec[27], vec[26], vec[25], vec[24], vec[23]
  , vec[22], vec[21], vec[20], vec[19], vec[18], vec[17], vec[16], vec[15], vec[14], vec[13], vec[12], vec[11]
  , vec[10], vec[9], vec[8], vec[7], vec[6], vec[5], vec[4], vec[3], vec[2], vec[1], vec[0]);
 const int bitMask = _mm256_movemask_epi8(boolVec);

 // Set the resulting DWORD.
 bp[q] = bitMask;
}
```

Is it feasible to make a vectorized version for U++ by default, or should I provide it for myself?

Thanks,

evo

I still think there is a glitch somewhere benchmarking this. Would it possible to post a whole package here?

Mirek

Subject: Re: Writing Bits object to disk
Posted by crydev on Wed, 03 May 2017 07:00:49 GMT
View Forum Message <> Reply to Message

mirek wrote on Tue, 02 May 2017 23:55
I still think there is a glitch somewhere benchmarking this. Would it possible to post a whole package here?

Mirek

The testing application is on Bitbucket here (only runs on Windows):
https://bitbucket.org/evolution536/cry-performance-test

You still have to add the following functions to the Bits class:

```
void PipelineSet(int i, const dword bs);
void VectorSet(int i, const unsigned char vec[16]);
void VectorSetAVX2(int i, const unsigned char vec[32]);
```

crydev

Subject: Re: Writing Bits object to disk
Posted by mirek on Wed, 03 May 2017 07:53:03 GMT
View Forum Message <> Reply to Message

crydev wrote on Wed, 03 May 2017 09:00mirek wrote on Tue, 02 May 2017 23:55
I still think there is a glitch somewhere benchmarking this. Would it possible to post a whole package here?

Mirek

The testing application is on Bitbucket here (only runs on Windows):
https://bitbucket.org/evolution536/cry-performance-test

You still have to add the following functions to the Bits class:

```
void PipelineSet(int i, const dword bs);
void VectorSet(int i, const unsigned char vec[16]);
void VectorSetAVX2(int i, const unsigned char vec[32]);
```

crydev

Well, it is what I thought:

```
const int VectorBoolOrBitsetTestBitSetVectorized(Bits& buffer, const Vector<unsigned char>&
randVec)
{
 const int count = randVec.GetCount();
 for (int i = 0; i < count; i += 16)
 {
  // Use the vectorized set method.
  buffer.VectorSet(i, &randVec[i]);
 }
 int alloc = 0;
 buffer.Raw(alloc);
 return alloc;
}
```

This is not the proper benchmark. The large part of work is already done by grouping input bits into randVec, which is not included in the benchmark time IMO.

If you wanted the proper benchmark, you could e.g. set randVec to random values and then set bits in Bits for those values that satisfy some condition - that IMO will benchmark scenario closer to the real use. E.g. (for original Bits and random 0..100):

```
for(int i = 0; i < randValue.GetCount(); i++)
    bits.Set(i, randValue[i] > 50);
```

Subject: Re: Writing Bits object to disk
Posted by crydev on Wed, 03 May 2017 09:12:54 GMT

View Forum Message <> Reply to Message

mirek wrote on Wed, 03 May 2017 09:53
Well, it is what I thought:

```
const int VectorBoolOrBitsetTestBitSetVectorized(Bits& buffer, const Vector<unsigned char>&
randVec)
{
 const int count = randVec.GetCount();
 for (int i = 0; i < count; i += 16)
 {
  // Use the vectorized set method.
  buffer.VectorSet(i, &randVec[i]);
 }
 int alloc = 0;
 buffer.Raw(alloc);
 return alloc;
}
```

This is not the proper benchmark. The large part of work is already done by grouping input bits into randVec, which is not included in the benchmark time IMO.

If you wanted the proper benchmark, you could e.g. set randVec to random values and then set bits in Bits for those values that satisfy some condition - that IMO will benchmark scenario closer to the real use. E.g. (for original Bits and random 0..100):

```
for(int i = 0; i < randValue.GetCount(); i++)
    bits.Set(i, randValue[i] > 50);
```

Why is it the case? The random vector is precomputed, and the only thing both testing functions do is set the random values to the underlying Bits:

```
buffer.Set(i, rand[i]);

// vs.

buffer.VectorSet(i, &randVec[i]);
```

The problem is that the regular Set function only accepts one bool as parameter. The difference is that I can put a pointer to a buffer of bools directly into the Set function, which is exactly my use case (bulk set) :)

crydev

Subject: Re: Writing Bits object to disk
Posted by mirek on Wed, 03 May 2017 09:27:14 GMT
View Forum Message <> Reply to Message

> crydev wrote on Wed, 03 May 2017 11:12
> Why is it the case? The random vector is precomputed, and the only thing both testing functions do is set the random values to the underlying Bits:

Will you get that Vector<bool> (or bool *) for free in your app?

IMO, usual usage pattern will always be "check some condition, set the bit to result".

So the point I am trying to make is that with Bits, as they are, you are not required to "precompute" Vector<bool>. Which is why I think the benchmark is not telling the whole truth.

Mirek

Subject: Re: Writing Bits object to disk
Posted by crydev on Wed, 03 May 2017 10:24:41 GMT

View Forum Message <> Reply to Message

mirek wrote on Wed, 03 May 2017 11:27crydev wrote on Wed, 03 May 2017 11:12
Why is it the case? The random vector is precomputed, and the only thing both testing functions do is set the random values to the underlying Bits:

Will you get that Vector<bool> (or bool *) for free in your app?

IMO, usual usage pattern will always be "check some condition, set the bit to result".

So the point I am trying to make is that with Bits, as they are, you are not required to "precompute" Vector<bool>. Which is why I think the benchmark is not telling the whole truth.

Mirek

I see what you mean! What test case would you propose? I implemented the Bits usage in my primary application again, and now with your changes, it finally starts to pay off using Bits instead. Also, thanks a lot for implementing the Raw and CreateRaw functions. :)

I'm still trying to get my own vectorized version working in the primary application. The setting indeed takes some overhead, but for now it seems to speed up the entire process (not just setting bits) by around 25%.

crydev

Subject: Re: Writing Bits object to disk
Posted by mirek on Wed, 03 May 2017 10:37:41 GMT

View Forum Message <> Reply to Message

crydev wrote on Wed, 03 May 2017 12:24
I see what you mean! What test case would you propose? I implemented the Bits usage in my primary application again, and now with your changes, it finally starts to pay off using Bits instead. Also, thanks a lot for implementing the Raw and CreateRaw functions. :)

The one above... :) Set the primary array to Random(100), then sets bits to 1 to those >50. This of course will work without creating buffer single bit Set, but will require recomputing values into 0x0 / 0x80 buffer for vectorised version...

That said, I really am not opposed to vectorised version, I just do not think the interface is right. I would rather see something like

Set(int pos, bool b0, ...)

(varargs Set). That way it would be perhaps possible to work without precreating the buffer.

In either case, I think we should start with

Set(int pos, dword bits, int count);

Subject: Re: Writing Bits object to disk
Posted by crydev on Wed, 03 May 2017 16:33:19 GMT
View Forum Message <> Reply to Message

mirek wrote on Wed, 03 May 2017 12:37crydev wrote on Wed, 03 May 2017 12:24
I see what you mean! What test case would you propose? I implemented the Bits usage in my primary application again, and now with your changes, it finally starts to pay off using Bits instead. Also, thanks a lot for implementing the Raw and CreateRaw functions. :)

The one above... :) Set the primary array to Random(100), then sets bits to 1 to those >50. This of course will work without creating buffer single bit Set, but will require recomputing values into 0x0 / 0x80 buffer for vectorised version...

That said, I really am not opposed to vectorised version, I just do not think the interface is right. I would rather see something like

Set(int pos, bool b0, ...)

(varargs Set). That way it would be perhaps possible to work without precreating the buffer.

In either case, I think we should start with

```
Set(int pos, dword bits, int count);
```

I see. :) I made the 0x80 pre-computed buffer because in my use case, I have full control over what the input value is. Using 0x1 or 0x80 for a true value, will not require more work in my situation. However, I understand that you would not want such interface in U++.

I'll try to work out an example with such interface. :)

crydev

Subject: Re: Writing Bits object to disk
Posted by crydev on Sat, 06 May 2017 08:28:39 GMT
View Forum Message <> Reply to Message

Hi Mirek,

I worked out the test you suggested. The pipelined set function is not faster anymore. I can understand that this is the case. The SSE2 vectorized set function now is around 30% faster:

```
// Non naive vector test :D
const int VectorBoolOrBitsetTestBitSetVectorizedNonNaive(Bits& buffer, const Vector<unsigned char>& rand)
{
 unsigned char vec[32];
 const int count = rand.GetCount();
 for (int i = 0; i < count; i += sizeof(vec))
 {
  for (int j = 0; j < sizeof(vec); ++j)
  {
   vec[j] = rand[i + j] > 50 ? 0x80 : 0x0;
  }

  // Use the vectorized set method.
  buffer.VectorSetNonNaive(i, vec);
 }
 int alloc = 0;
 buffer.Raw(alloc);
 return alloc;
}

// The non naive vector set function, that actually implements only the vector
// setting, and not also the comparison of the input!
void Bits::VectorSetNonNaive(int i, const unsigned char vec[32])
{
 // Check whether i is within the bounds of the container.
 ASSERT(i >= 0 && alloc >= 0);

 // Get the DWORD index for the internal buffer.
 int q = i >> 5;

 // Do we need to expand the internal buffer first?
 if(q >= alloc)
  Expand(q);

 // Get the bit index of the next available DWORD.
 i &= 31;

 // Create a bitmask with vector intrinsics.
 __m128i boolVecLow = _mm_load_si128((__m128i*)vec);
 __m128i boolVecHigh = _mm_load_si128((__m128i*)(vec + 16));
 const int bitMaskLow = _mm_movemask_epi8(boolVecLow);
 const int bitMaskHigh = _mm_movemask_epi8(boolVecHigh);

 // Set the resulting WORD.
 LowHighDword w;
 w.dw = bp[q];
 w.w1 = (short)bitMaskLow;
 w.w2 = (short)bitMaskHigh;
 bp[q] = w.dw;
}
```

However, I was thinking: if I use vectorized code, I should be allowed to vectorize everything, including the comparison of the input values! I thought about my own use case (where I indeed also have to perform the kind of comparison you suggested), and I realized that this comparison could be vectorized. When I did so, the speeds of the SSE2 and AVX2 vectorized code examples skyrocketed. I understand that you may think of the vectorized comparison as a 'proof-of-concept' rather than a realistic implementation for U++, but I can actually utilize this implementation for my memory scanner. Besides, it was fun to write code like this. I am surprised how easy it can be done and how easily the game is played out. :)

The source code is of the other tests is once again located at my Bitbucket repository:
https://bitbucket.org/evolution536/cry-performance-test/src/212c3b4f51a29efa0e70841e76cceb11bcaacf06/VectorBoolOrBitsetTest.cpp?at=master&fileviewer=file-view-default

What do you think about a possible 'multi-set' implementation with an interface like:

```
void Set(int i, const Vector<bool>& vec, const int count);
```

Such function could contain a loop for a vector implementation, and the sequential Set function for the remainder. Maybe also a set function that allows you to manually prepare the dwords and have the set function solely manage the allocation and positioning for you. :)

crydev

File Attachments

1) Capture.PNG, downloaded 656 times

Subject: Re: Writing Bits object to disk
Posted by crydev on Tue, 16 May 2017 07:12:07 GMT
View Forum Message <> Reply to Message

Hi Mirek,

How do you feel about adding a Bits::Set function that allows the user to put in packed bools? I was thinking about a function such as the following:

```
void Bits::VectorSet(int i, const dword bits32)
{
 ASSERT(i >= 0 && alloc >= 0);
 int q = i >> 5;

 if(q >= alloc)
  Expand(q);

 i &= 31;

 // Just place the input DWORD in position q in the Bits data buffer.
 bp[q] = bits32;
}
```

Maybe with some checks, this is just an idea. I mean, this function works in my testcase because I only work with multiples of 32. Even though, this could be used as an assumption to make it fast. It accepts a bitset dword and directly sets it. This allows efficient use of the Bits structure while preparing the actual packed bools structure with SSE2 instructions like _mm_movemask_epi8. :)

Thanks!

crydev

Subject: Re: Writing Bits object to disk
Posted by mirek on Tue, 16 May 2017 08:39:44 GMT

View Forum Message <> Reply to Message

crydev wrote on Tue, 16 May 2017 09:12Hi Mirek,

How do you feel about adding a Bits::Set function that allows the user to put in packed bools? I was thinking about a function such as the following:

```
void Bits::VectorSet(int i, const dword bits32)
{
 ASSERT(i >= 0 && alloc >= 0);
 int q = i >> 5;

 if(q >= alloc)
  Expand(q);

 i &= 31;

 // Just place the input DWORD in position q in the Bits data buffer.
 bp[q] = bits32;
}
```

Maybe with some checks, this is just an idea. I mean, this function works in my testcase because I only work with multiples of 32. Even though, this could be used as an assumption to make it fast. It accepts a bitset dword and directly sets it. This allows efficient use of the Bits structure while preparing the actual packed bools structure with SSE2 instructions like _mm_movemask_epi8. :)

Thanks!

crydev

I feel good about it, but I definitely think that it should be able to handle boundaries correctly...

I mean, this only works if i % 32 == 0. Should work in all other cases too, with variable number of bits.

Subject: Re: Writing Bits object to disk
Posted by mirek on Tue, 16 May 2017 11:12:12 GMT
View Forum Message <> Reply to Message

Added these:

```
 void   Set(int i, dword bits, int count);
 void   Set64(int i, uint64 bits, int count);
 void   SetN(int i, int count, bool b = true);
```

Slight disadvantage is that while Set detects "aligned" operation (i % 32 == 0, count == 32), there is some penalty, but it seems to be quite small (about 3-4 opcodes). I guess if you want to be faster than that, you have to use Raw....

Subject: Re: Writing Bits object to disk
Posted by crydev on Tue, 16 May 2017 18:17:44 GMT

View Forum Message <> Reply to Message

> mirek wrote on Tue, 16 May 2017 13:12Added these:
>
> ```
>  void   Set(int i, dword bits, int count);
>  void   Set64(int i, uint64 bits, int count);
>  void   SetN(int i, int count, bool b = true);
> ```
>
> Slight disadvantage is that while Set detects "aligned" operation (i % 32 == 0, count == 32), there is some penalty, but it seems to be quite small (about 3-4 opcodes). I guess if you want to be faster than that, you have to use Raw....

Hi Mirek,

The changes look wonderful! Thanks a bunch. :)

crydev