

Re: Suggestions for double-hashing scheme

Source: <http://coding.derkeiler.com/Archive/General/comp.programming/2005-06/msg00262.html>

- *From:* websnarf@xxxxxxxxxx
 - *Date:* 8 Jun 2005 14:32:22 -0700
-

Tim Rentsch wrote:

> websnarf@xxxxxxxxxx writes:

>> Clint Olsen wrote:

>>> I'm interested in coding up a double-hash implementation (rather than
>>> chaining). Why? Well, because it is an interesting problem, and I'm
>>> trying to see if an implmentation can work just as good as chaining to
>>> handle collisions.

>>

>> [...snip...]

>>

>> Once you sufficiently optimize a hash table, the big cost in hashing
>> turns out to be the cost of *COMPARING* the entries (*copying* entries
>> (if you aren't using just refs) is the next secondary effect, followed
>> by computing of the hash function). So don't obsess too much over
>> "clever" techniques for reducing delete impact. You should think of
>> "Delete" as adding design complications, not performance problems.

>

> An over-generalization. Comparing can be relatively expensive,
> but it often (or even mostly) isn't. Besides the case where
> elements in the hash table are pointers, which is a common case
> in its own right, [...]

This is true, but you are not carrying my statement to its logical conclusion. The point is that, once optimized, the entire overhead of managing the hash table is very low. If nothing costs anything in your hash table, then indeed, your performance bottleneck may end up going anywhere, but more than likely in these situations your performance hit isn't going to be hash table access at all.

> often compares of, for example, structures, differ in the first
> few bytes. Doing a trivial reject by doing a 4-byte compare on
> the first word of the two elements often lets structure
> comparison run at almost the full speed of pointer comparison.

Ok, but we can see that from a $O(*)$ point of view this is all equivalent. The problem with any variable length structures, or entries where you are hoping the first 4 bytes to be a sufficient differentiator is that you are at the mercy of the internal structural bit determinability. And you'll eat an additional branch miss penalty

Re: Suggestions for double-hashing scheme

when you are wrong.

By comparison, any good hash function basically always has a minimal probability of "false positives" regardless of internal structure. It also doesn't add anything to the raw cost, because you have to calculate the hash value for every entry you search for or insert anyway.

So you may have to do some analysis to *know* whether or not raw comparison is good enough. Simply using prescreening removes this uncertainty uniformly. This more closely follows my personal focus of making universal libraries that *always* work well regardless of the situation, as opposed to libraries that only work under narrow situations that you try to justify/impose upon your consumers.

>> Anyhow, coming back to what *IS* important. Since comparing is where
>> the real bottleneck is, you have to do anything possible to reduce this
>> impact. How can you do this? First of all, the "swap to front"
>> heuristic can help, but more importantly you need to do hash-value
>> *PRE-SCREENING*. That is to say, entries should retain their full
>> hash-value along with them in the table, and as you scan, you first
>> check that the hashes match (between the one in the table and the entry
>> you are searching for) before incurring the expensive cost of comparing
>> entries (if $\text{hash}(e1) \neq \text{hash}(e2)$, which is fast to compute, then you
>> already know that $e1 \neq e2$). In this way, so long as your hash
>> function is reasonable, the comparison cost is absolutely minimized
>> (you basically only perform comparisons when its practically a forgone
>> conclusion that the entry is a match). (Remember, you keep the entire
>> *original* hash function along with each entry, not just the index for
>> $h0()$ or $h1()$.)

>
> Keeping the hash value is a good technique to know, but
> whether it's a good idea in any particular instance depends
> on other things, including the size of the elements being
> hashed. If the elements being hashed are 8 bytes and a hash
> value is 4 bytes, it's probably better to keep just the
> element being hashed, which means for the same space the
> load factor will be 33% less. The smaller load factor will
> mean fewer collisions and almost certainly translate into
> better performance.

This is important for the cases you describe if you are memory constrained, or if you expect the size to be right on the edge of a cache size.

>>> In double hashing, the overall hash is calculated via:
>>>
>>> $H(k) = h0(k) + n * h1(k)$
>>>
>>> Where $h0$ and $h1$ are the hash functions and n is the probe attempt (0, 1, 2,
>>> ...).

Re: Suggestions for double-hashing scheme

>>
>> Yes. But its important to note that the total number of bits required
>> to produce h0 and h1 is almost certainly < 32. This requires some
>> analysis:
>>
>> Computing h1() is interesting because on the one hand it must not be 0
>> and each possibility must be coprime with the current table size and
>> other possible values for it. And on the other hand the range of
>> possible values of h1() does not need to be all that great.
>>
>> First of all, if you don't already know about it, familliarize yourself
>> with the "Birthday Paradox":
>> http://en.wikipedia.org/wiki/Birthday_Paradox . Understanding this is
>> key to understanding how to properly design your reprobe strategy. If
>> you don't understand the Birthday Paradox, then you are going to have a
>> hard time designing a trully sound and efficient hash table.
>>
>> Basically in a hash table that's starting to fill up, the worst
>> colliding initial h0() positions might have around 10 entries aliasing.
>> What you want is for each of these entries to have a different reprobe
>> skip value. The Birthday Paradox basically says that if you pick
>> random choices amoungst a palette of about 100 entries, then you can
>> expect about "1" additional collision in skip values. So this is quite
>> tolerable.
>>
>> I also said above that the possible h1() choices should be coprime with
>> each other. Hopefully this is easy to understand -- if h1() were to
>> output 3 then 6, for example, then chains that followed those probe
>> values from the same h0() would intersect quit a bit! Whereas, if it
>> were to output 5 and 7, then their common intersections are a lot less
>> minimal. So being coprime with each other makes a big difference.
>>
>> Ok, so what does this all mean? It means that h1() essentially doesn't
>> need more than about 7 bits worth of degrees of freedom, and should be
>> taken from a table of numbers which are non-zero and coprime which each
>> other, as well as the size of the hash table itself. An array of the
>> first 128 primes starting from, say 11 (to set a lowest expected
>> interesection length that is greater than the greatest expected reprobe
>> list), indexed from a uniform 7 bit hash function is sufficient. There
>> are other choices, but this is clearly the most obvious one that fits
>> the above described criteria.
>
> The idea that the choices for h1() should be restricted to
> sets that have only prime values is flawed. There will be fewer
> collisions if *all* values that are relatively prime to the table
> size (but not necessarily to each other) are in the set of
> choices for h1(). For example, assuming a table size of 512
> entries, and looking at the first 5 re-probes, the two cases
> calculate out as follows:
>
> 256 relatively prime values => collision density of 0.234%

Re: Suggestions for double–hashing scheme

> 93 primes ≥ 11 and $< 512 \Rightarrow$ collision density of 0.355%

What the hell is this measuring? Explain the scenario. Have you entered more than 5 entries off the same initial offset and are you measuring *average* number of collisions or something like that?

You've also made a critical analytical mistake. The range of the primes must also satisfy (in this case) the restriction that $p \cdot 5 < sz$. Otherwise it will just wrap around the table and actually lose coprimality property it was designed for. I.e., for tables of size 512, you should pick primes between 11 and 102, which is actually far fewer than 93 choices, but will actually have fewer collisions.

Look. 1) Practical testing indicates that reprobe length should not exceed about 10. 2) Average case is not so important because the effectiveness of the initial offset dominates — the average case has, in fact, a reprobe length < 2 .

The value of the secondary hash value is that it should be effective on average given the < 10 collision assumption, as well as being as effective as possible in the *worst* cases. The fact is that 2 and 4 are going to be relatively prime with a prime sized table — but those choices have a 50%/100% intersection rate, which is a horrible worst case. I have explained this in another post on this subject.

Once again, my focus is on universal solutions, not particular or average cases. By maximally mitigating the worst case scenario, you can have an expectation that the average case performance measurements will remain consistent over the widest possible scenarios. If you don't do this then you are just accepting a larger number of potential worst case scenarios, where the max of 10 assumption simply will not hold.

> [...]

>

> Incidentally, the above are theoretical calculations, not
> experimental measurements; they don't depend on specific input
> values. The numbers above should hold across large numbers of
> hash table usage, if the hash functions being used are good.

But the numbers are also wrong, as explained above.

>> So why go into all that much detail just on $h1()$? The key thing is to
>> realize that it uses so few bits (only 7) from a hash function which is
>> likely to output so many more bits. Furthermore the $h0()$ function
>> itself is generally going to use far less than 32 bits. So hopefully
>> this makes it clear that the right design is to use a *SINGLE* hash
>> function that outputs, say, 32 bits, and split off as many bits as is
>> required for $h0()$, and use the remaining bits (up to 7 of them) as the
>> index for $h1()$. If you look at Bob Jenkin's page, or my page on hash
>> functions you will see that basically all high performance hash table

Re: Suggestions for double–hashing scheme

Re: Suggestions for double-hashing scheme

>> hash functions output 32 bits (or more.) So this seems to be the
>> precise sweet spot for hash function/table design.
>>
>> Although this seems to limit the hash table to 33 million entries,
>> actually, the "7 bits for h1()" is overspecified. That is to say, as
>> your hash table grows beyond 33 million entries, you can simply
>> cannibalize "h1 bits" for use in h0. My own testing indicates that
>> h1() is still effective with as low as 3(!) bits for indexing (for half
>> a billion entries) and of course it still technically functions even if
>> there no bits available for h1() (at which point you should be thinking
>> about 64 bit solutions anyways -- this is the one redeeming aspect of
>> the FNV hash function, since 64 a bit version of it exists.)
>
> If N bits are needed to choose among the complete set of h1()
> values for this table size, we could choose h1() based on "low
> N bits of hash" and h0() based on "hash ^ hash>>N".

Its the same difference, except you may not need h1(). Just as a matter of micro-optimization, you would rather make sure h0() gets all the performance attention, while h1() can eat a minor penalty as a result. The balance is a net win.

>> Another important note is about the size of the hash table itself. A
>> common mistake in the literature and elsewhere is to pick the table
>> size as a prime number. They do this in order to relax the constraints
>> on h1() in order to guarantee that it traverses the table with a mere
>> != 0 criteria. Hopefully my explanation above shows how naive and
>> pedestrian such reasoning is. Worse yet is that the -> index =
>> hashfunction(entry) % table_size; line all of a sudden has a new
>> bottleneck, namely the "%" function. As you well know, "%" (or
>> equivalently division) by a non-constant number is basically one of the
>> slowest operations you can do on any CPU outside of transcendentals or
>> IO.
>
> This analysis is superficial. First, the cost needs to be
> paid only on the initial calculation

The majority of entries have a probe length of exactly 1. The initial cost is the whole kit and kaboodle. Who is being superficial here?

> -- reprobates can be done using
>
> hash = hash + h1;
> if(hash >= table_size) hash -= table_size;

Because branches never cost anything right? You simply aren't looking at this correctly. I don't know if you've ever done hand-optimization or micro-optimizations before, but you are really missing it here. Once you tweak the hash table design, the only thing you are left to focus your attention on, wrt to performance, are this micro-optimization issues.

Re: Suggestions for double-hashing scheme

- > Second, the benefits of lower collision density for reprobates
- > when using prime table sizes is ignored.

Only if we ignore the fact that you've messed up the prime table range.

- > Third, just because the table size is prime doesn't mean that
- > a runtime modulo operation must be done. Chuck Falconer's
- > hashlib, for example, uses just a small number of prime table
- > sizes (about 20, but it could just as easily be 50 or 100) --
- > it would be easy to produce a routine with a 'switch()'
- > statement where each of the modulo operations were done using
- > a compile time constant.

The performance of "switch()" is equivalent to a branch miss, or a full pipeline flush. So its a little faster than divide, but not much. Add to that the cost of the simulated divide, and you are not going to come out ahead. Interesting that you are willing to mismeasure "collison densities" but unwilling to measure this. Trust me, the results will not inspire you.

- > [...] Depending on the architecture, this improvement might be
- > better accomplished using a pointer to function (one function
- > for each of the fixed choices of table size).

You, of course, are referring to some architecture will a really fast indirect/indexed jump? Can you name one? Otherwise I'm going to have to dispute your use of the word "improvement".

- > Another advantage to using prime table sizes is that they naturally
- > use all bits of the hash value.

How can you, on the one hand, claim to expect arbitrary bit masking versus first mixing then masking to perform equivalently (true for any good hash function), then turn around and claim here that using all bits is some advantage?

- > [...] Hashing a pointer 'p', for example,
- > might be as simple as '(uint32_t) p'. This approach could easily
- > perform poorly in a power-of-two-sized hash table, but would very
- > likely do just fine in a hash table with a prime number of slots.

This is going to do poorly no matter what. Remember, my design recommendation is to use one good hash function, then extract bits for both h0 and h1 from it. So using '(uint32_t) p' is not an acceptable hash function at all -- certainly you can't use it for h0(), where most of the performance costs are focused. Since you have to pay the price to get a good h0() anyways, and since a good h1() basically pops out almost for free, this idea just doesn't apply at all.

- > If your hash tables are mostly empty (load factor of less than 50%)

Re: Suggestions for double-hashing scheme

Re: Suggestions for double-hashing scheme

- > then the cost of the initial probe is likely to dominate, and that
- > first modulo operation is going to look pretty big. But some
- > applications do better with large hash tables that are almost full
- > (eg, where the hash table is approximately the size of the L2 cache),
- > with load factors of more than 90%; in these cases, choosing a table
- > size that is prime may very well pay off.

Fits in the L2 cache ... what's with you and CBFalconer? Why are such tiny hash tables so important to you? You're letting these very particular scenarios affect the design decisions for something that is supposed to be a scalable ADT. How can you, on the one hand here, be concerned about on-chip performance which is a concern *SOMETIMES*, and yet be totally oblivious to the modulo cost for a prime sized table which is a concern *ALL THE TIME*?

- >> The more sensible approach, of course, is to pick a table size that is
- >> a power of 2. Then you have `index = hashfunction(entry) &`
- >> `(tablesize-1)`; which removes the "%" bottleneck ("&", as you know,
- >> basically costs nothing.) The earlier analysis of `h1()` above makes it
- >> clear that traversing the table will be a non-issue, if you simply
- >> choose them from a small table of primes.

>

- > Certainly there are advantages of using a table size that is a
- > power of two. But there also are drawbacks, and these haven't
- > been mentioned. Using power of two size tables, the set of
- > choices is limited: when growing a 128 MB table, for example,
- > the smallest next size is 256 MB. Using prime size tables, on
- > the other hand, allows more economical growth, because the
- > density of primes is so much higher than the density of powers
- > of two – we can easily find a prime 25%, 10%, or even only 2%
- > larger. Obviously, whether this economy is needed depends on
- > the application, but it's nice to have it available when we
- > need it; limiting hash table code to use only table sizes
- > that are powers of two removes that choice.

Talk about not mentioning the drawbacks — if you use a "25%", "10%" or "2%" rehash criteria, then guess where all your performance goes? It goes into *REBUILDING THE TABLE*. I've posted elsewhere, that in fact, serious consideration should be put into just doing *quadruplications* of the hash size (obviously you would fall back to doubling if the quadrupling malloc failed), since this will greatly reduce the number of times an entry is re-inserted due to a rebuild.

Remember that a doubling of memory is equivalent to 1 year of technology (Moore's law) and all these things are to be balanced against hash-key prescreening, or reprobe counts, or statistics or whatever else you are adding to your hash table design. Except in embedded environments or if you are obsessed with very small hash tables that actually fit in your caches, this sort of "size massage" doesn't have any tangible value.

—
Paul Hsieh

<http://www.pobox.com/~qed/>

<http://bstring.sf.net/>

-
- *Follow–Ups:*
 - ◆ ***Re: Suggestions for double–hashing scheme***
 - ◇ *From:* Tim Rentsch
 - *References:*
 - ◆ ***Re: Suggestions for double–hashing scheme***
 - ◇ *From:* Tim Rentsch
 - Prev by Date: ***Re: sort on more than one key?***
 - Next by Date: ***Re: Font problem using Selection.TypeText in Word/VBA***
 - Previous by thread: ***Re: Suggestions for double–hashing scheme***
 - Next by thread: ***Re: Suggestions for double–hashing scheme***
 - Index(es):
 - ◆ ***Date***
 - ◆ ***Thread***