A generic hash table

By **Jake Edge**
August 8, 2012

A data structure implementation that is more or less replicated in 50 or more places in the kernel seems like some nice low-hanging fruit to pick. That is just what Sasha Levin is trying to do with his generic hash table patch set. It implements a simple fixed-size hash table and starts the process of changing various existing hash table implementations to use this new infrastructure.

The interface to Levin's hash table is fairly straightforward. The API is defined in `linux/hashtable.h` and one declares a hash table as follows:

```
DEFINE_HASHTABLE(name, bits)
```

This creates a table with the given `name` and a power-of-2 size based on `bits`. The table is implemented using buckets containing a kernel `struct hlist_head` type. It implements a chaining hash, where hash collisions are simply added to the head of the `hlist`. One then calls:

```
hash_init(name, bits);
```

to initialize the buckets.

Once that's done, a structure containing a `struct hlist_node` pointer can be constructed to hold the data to be inserted, which is done with:

```
hash_add(name, bits, node, key);
```

where `node` is a pointer to the `hlist_node` and `key` is the key that is hashed into the table. There are also two mechanisms to iterate over the table. The first iterates through the entire hash table, returning the entries in each bucket:

```
hash_for_each(name, bits, bkt, node, obj, member)
```

The second returns only the entries that correspond to the `key`'s hash bucket:

```
hash_for_each_possible(name, obj, bits, node, member, key)
```

In each case, `obj` is the type of the underlying data, `node` is a `struct hlist_head` pointer to use as a loop cursor, and `member` is the name of the `struct hlist_node` member in the stored data type. In addition, `hash_for_each()` needs an integer loop cursor, `bkt`. Beyond that, one can remove an entry from the table with:

```
hash_del(node);
```

Levin has also converted six different hash table uses in the kernel as examples in the patch set. While the code savings aren't huge (a net loss of 16 lines), they could be reasonably significant after converting the 50+ different fixed-size hash tables that Levin found in the kernel. There is also the obvious advantage of restricting all of the hash table implementation bugs to one place.

There has been a fair amount of discussion of the patches over the three revisions that Levin has posted so far. Much of it concerned implementation details, but there was another more global concern as well. Eric W. Biederman was not convinced that replacing the existing simple hash tables was desirable:

> For a trivial hash table I don't know if the abstraction is worth it. For a hash table that starts off small and grows as big as you need it the [incentive] to use a hash table abstraction seems a lot stronger.

But, Linus Torvalds disagreed. He mentioned that he had been "playing around" with a directory cache (dcache) patch that uses a fixed-size hash table as an L1 cache for directory entries that provided a noticeable performance boost. If a lookup in that first hash table fails, the code then falls back to the existing dynamically sized hash table. The reason that the code hasn't been committed yet is because "filling of the small L1 hash is racy for me right now" and he has not yet found a lockless and race-free way to do so. So:

> [...] what I really wanted to bring up was the fact that static hash tables of a fixed size are really quite noticeably faster. So I would say that Sasha's patch to make *that* case easy actually sounds nice, rather than making some more complicated case that is fundamentally slower and more complicated.

Torvalds posted his patch (dropped diff attachment) after a request from Josh Triplett. The race condition is "almost entirely theoretical", he said, so it could be used to generate some preliminary performance numbers. Beyond just using the small fixed-sized table, Torvalds's patch also circumvents any chaining; if the hash bucket doesn't contain the entry, the second cache is consulted. By avoiding "pointer chasing", the L1 dcache "really improved performance".

Torvalds's dcache work is, of course, something of an aside in terms of Levin's patches, but several kernel developers seemed favorably inclined toward consolidating the various kernel hash table implementations. Biederman was unimpressed with the conversion of the UID cache in the user namespace code and Nacked it. On the other hand, Mathieu Desnoyers had only minor comments on the conversion of the tracepoint hash table and Eric Dumazet had mostly stylistic comments on the conversion of the 9p protocol error table. There are several other maintainers who have not yet weighed in, but so far most of the reaction has been positive. Levin is trying to attract more reviews by converting a few subsystems, as he notes in the patch.

It is still a fair amount of work to convert the other 40+ implementations, but the conversion seems fairly straightforward. But, Biederman's complaint about the conversion of the namespace code is something to note: "I don't have the time for a new improved better hash table that makes the code buggier." Levin will need to prove that his implementation works well, and that the conversions don't introduce regressions, before there is any chance that we will see it in the mainline. There is no reason that all hash tables need to be converted before that happens—though it might make it more likely to go in.

Index entries for this article
Kernel Hash table

(Log in to post comments)

A generic hash table

Posted Aug 9, 2012 5:02 UTC (Thu) by **alonz** (subscriber, #815) [Link]

Re the pointer-chasing vs. simple table argument—has anyone tried to modify the hash tables to implement Cuckoo hash behavior? From my experience these provide nice gains over a simple hash table, and with zero added complexity.

Reply to this comment

A generic hash table

Posted Aug 9, 2012 23:26 UTC (Thu) by **nix** (subscriber, #2304) [Link]

Ooo. I'd failed to notice cuckoo hashes: very nice (particularly with more than one item: 50% loading is rather cruddy, but 91% is close enough to 100% as makes no odds. Combine that with using three hash functions and you could get it *very* close to 100% for only a small constant slowdown. Downside: you still do need to resize now and then, which is hateful because it breaks active iterators unless you have crazy complexity to deal with that case alone. Yes, you *can* ban modification of hashes being iterated over, but I consider that a bad-quality implementation.)

Reply to this comment

A generic hash table

Posted Aug 10, 2012 2:17 UTC (Fri) by **nybble41** (subscriber, #55106) [Link]

> Downside: you still do need to resize now and then, which is hateful because it breaks active iterators unless you have crazy complexity to deal with that case alone.

Even without resizing, keys can change location when other items are inserted, so you can't make very many guarantees. Even with a normal hash table it's difficult to say whether you will see the inserted key later on; with a cuckoo hash table you may miss _other_ keys, or even double-iterate them.

If you must modify a hash table with active iterators, perhaps you should collect the list of keys up front, atomically, and iterate over that instead. It may not be quite as efficient as iterating over the hash table directly, in memory or CPU time, but the result will be much more predictable. Or you could make a shallow copy of the hash table to iterate over.

Reply to this comment

A generic hash table

Posted Aug 10, 2012 16:28 UTC (Fri) by **nix** (subscriber, #2304) [Link]

Yeah. There are hash designs which don't have this problem -- e.g. any based on trees (you can even arrange that deletion of the key you're currently iterating over will only impose the same overhead on the next iteration as a normal hash lookup). Downside: trees often involve a lot of pointer chasing, and some of them are write-heavy on update and even on read (e.g. splay trees: nice self-optimizing data structure, shame about your cacheline contention).

Reply to this comment

A generic hash table

Posted Aug 9, 2012 14:32 UTC (Thu) by **pj** (subscriber, #4506) [Link]

Coming right after the 'kernel regressions' article, this made me wonder... does this new generic hash table implementation come with a test suite? If so, that would be a measurable, notable improvement over the other N implementations that Sasha could point to.

Reply to this comment

A generic hash table

Posted Aug 14, 2012 17:02 UTC (Tue) by **sashal** (subscriber, #81842) [Link]

Indeed this is a great suggestion. Some sort of CONFIG_HASHTABLE_TEST (similar to what happens with list sorting for example) is definitely something I'd like to add.

Thanks for the suggestion!

Reply to this comment

User space unit tests?

Posted Aug 15, 2012 15:01 UTC (Wed) by **alex** (subscriber, #1355) [Link]

While enabling run time testing of some kernel features is useful (the VM torture tests spring to mind) it would be much better if library code was testable in a user space build that didn't involve bringing a kernel up.

How about adding "make unittests" as a target for the kernel build?

Reply to this comment

A generic hash table

Posted Aug 9, 2012 20:17 UTC (Thu) by **HelloWorld** (guest, #56129) [Link]

One has to wonder why the kernel is only now gettin such a basic data structure as a generic hash table. I think it's because generic data structures aren't all that useful in a language like C. It's not really possible to make them both generic and typesafe (at least not without horrible macro hackery), and making them customizable (so you can apply different policies regarding memory allocation, collision handling etc.) is even harder.
Yes, C++ is an ugly language and it has many warts. But it's *useful* in that it lets me write code in the way I want to write it: generic, type-safe, flexible and just as efficient as C, if not more so.

Reply to this comment

A generic hash table

Posted Aug 13, 2012 21:40 UTC (Mon) by **liljencrantz** (guest, #28458) [Link]

C++11 is the first version of the language to feature a built in hash table. One has to wonder why the language is only now gettin (sic) such a basic data structure as a generic hash table. I think it's because generic data structures are insanely hard in a language like C++. It's not really possible to make them both generic and performant (at least not without hortrible template hackery), and making them customizable (so you can apply different policies regarding memory allocation, collision handling, etc.) is even harder.

Yes, C is an ugly language and it has many warts. But it's *useful* in that it lets me write code in the way I want to write it: without hidden costs, reasonably close to the hardware and just as efficient as C++, if not more so.

Seriously though, writing generic data structures in C++ is a hard enough problem that even stl is pretty crappy for a lot of applications. Every insert into the C++ linked list causes a memory allocation. Until recently, LinkedList::size() was an O(n) operation. The attempts to make vector<bool> fast were a horrible

failure. The COW abilities of basic_string are mostly a joke because of weak C++ aliasing rules. C++ provides you with half the tools you need to write good generic data structures and all the rope you need to hang yourself while chasing that pipe dream.

Reply to this comment

A generic hash table

Posted Aug 13, 2012 22:18 UTC (Mon) by **Cyberax** (★ **supporter** ★, #52523) [Link]

>C++11 is the first version of the language to feature a built in hash table. One has to wonder why the language is only now gettin (sic) such a basic data structure as a generic hash table.

That's because C++11 is the first major revision of C++ since 1998 (yeah, blame the ISO committee). Hash maps have been present in various non-standard ways in C++ stdlibs since 90-s. That's also the reason they are called 'unordered_maps' in the C++11 to avoid all kinds of name collisions.

You might also notice, that tree based std::map has been present since C++98.

Reply to this comment

A generic hash table

Posted Aug 13, 2012 23:42 UTC (Mon) by **liljencrantz** (guest, #28458) [Link]

Hash maps were invented in the fifties. C++ has been a popular language since the eighties. The fact that they couldn't design a hash table implementation that was good enough to slap a sticker on it in 1998 (that's almost 20 years down the line) is a strong testament to just how hard the trade offs are when trying to design generic C++ data structures.

Do you want an open addressing implementation? What type of probing? A Cuckoo hash? Do you use a linked list to allow ordered iteration? Do you allow hash table modification during iteration? If so, only using that iterator or arbitrary insertions? What resize strategy do you use when growing the table? What resize strategy do you use when shrinking the table? Do you store precalculated hash codes in the table? Do you use tombstones to mark deleted entries?

No matter what you answer to the above questions, you will make your hash table nearly useless for a large swath of people. And you can try to make the answer to all those questions decidable at compile time through templates, but that solution comes with it's own set of painful drawbacks.

You might also notice, that generic tree based red black trees have been present in the Linux kernel since forever.

Reply to this comment

A generic hash table

Posted Aug 14, 2012 0:04 UTC (Tue) by **Cyberax** (★ **supporter** ★, #52523) [Link]

> Hash maps were invented in the fifties. C++ has been a popular language since the eighties. The fact that they couldn't design a hash table implementation that was good enough to slap a sticker on it in 1998
Not in 1998 but in late 1995 (that's when the final drafts were produced). And the whole idea of STL was pretty new at that point, it's actually a small miracle they managed to put it into the C++ Standard as it is.

> Do you want an open addressing implementation? What type of probing? A Cuckoo hash? Do you use a linked list to allow ordered iteration? Do you allow hash table modification during iteration? If so, only using that iterator or arbitrary insertions? What resize strategy do you use when growing the table? What resize strategy do you use when shrinking the table? Do you store precalculated hash codes in the table? Do you use tombstones to mark deleted entries?

You've described about 100 different structures (considering various permutations). Some of the functionality can be split into policies (they work wonderfully with templates, incidentally). Otherwise, library designers just pick a generic enough version and go along with it.

If you need some special tuned version for your very specific need - you're welcome to write it. If you abide by STL's standards then you can even use them with STL/Boost algorithms. Without any loss of performance, I might add.

Reply to this comment

A generic hash table

Posted Aug 13, 2012 22:39 UTC (Mon) by **dlang** (guest, #313) [Link]

> C++11 is the first version of the language to feature a built in hash table. One has to wonder why the language is only now gettin (sic) such a basic data structure as a generic hash table.

It's not that C++ is getting a "generic hash table"

It's that the Linux Kernel project is implementing something they call a "generic hash table" that combines all the various features that the different hash tables in the kernel implement.

C++ has a very generic hash table, it's also not very useful by itself, and as a result it does 80% of the work and you implement the remaining 20% to match your application.

In the Linux Kernel, this 20% has been implemented a dozen different ways (working to solve a dozen different types of problems). This is a unified implementation that proposes to cover all dozen use cases with one set of code.

Reply to this comment

A generic hash table

Posted Aug 13, 2012 23:26 UTC (Mon) by **HelloWorld** (guest, #56129) [Link]

> One has to wonder why the language is only now gettin (sic) such a basic data structure as a generic hash table

The reason is trivial: The committee ran out of time when C++98 was standardized, thus hash tables didn't get in. However, boost and various STL vendors have been shipping hash tables for a long time. And let's not talk about the fact that C11 doesn't ship any containers at all.

> <pointless trolling removed>

> Every insert into the C++ linked list causes a memory allocation.

One can specify an allocator if the default one isn't fast enough, and if that still doesn't cut it, one can use something like boost.intrusive, where one has full control over all memory (de)allocation. The STL isn't all things to all people. The point is that in C++ it's actually *possible* to write good (i. e. type-safe, generic, fast) containers, while it isn't in C.

> Until recently, LinkedList::size() was an O(n) operation.
Having size() be an O(1) operation requires the size to be stored in the list and updated when elements are added or removed. This means that one can either move an iterator range from one list to another in constant time, or have size() run in constant time, but not both, because if you need to update the list sizes, you need to count the elements in the range, and requires linear time. So it actually makes sense for size() to run in linear time depending on what you want to do.

> The attempts to make vector<bool> fast were a horrible failure.
Well, vector<bool> was a mistake, shit happens. It tends to be a minor problem in the real world. One can just use a std::vector<uint8_t> or std::bitset or boost::dynamic_bitset or whatever else does the job.

> The COW abilities of basic_string are mostly a joke
C11 not having a string type at all, that's what is a joke.

> <yet more pointless trolling removed>

Reply to this comment

A generic hash table
Posted Aug 13, 2012 23:46 UTC (Mon) by **liljencrantz** (guest, #28458) [Link]

I find it hilarious that I posted a s/Linux/C++/ of your own comment and you call me a troll. Do they have mirrors in your world?

Reply to this comment

A generic hash table
Posted Aug 13, 2012 23:50 UTC (Mon) by **HelloWorld** (guest, #56129) [Link]

> I find it hilarious that I posted a s/Linux/C++/ of your own comment and you call me a troll.
That's because the arguments don't make sense any more when they're applied the other way around (and that's a good thing, as they'd be meaningless otherwise).

Reply to this comment

A generic hash table
Posted Aug 13, 2012 23:55 UTC (Mon) by **liljencrantz** (guest, #28458) [Link]

The part where you say that it is impossible to write type-safe, generic and fast containers in C is demonstratably false. All of the above can be had with macros. I look forward to you arguing that such implementations aren't good while ignoring that most of the problems with macros also exist with templates.

Whenever the many deficiencies of STL is pointed out, you defensively say that "they fixed that in boost::XXX", but ignore that any missing containers, string types and other omissions from the standard C library can be easily rectified through the use of a small set of additional libraries.

Reply to this comment

A generic hash table
Posted Aug 14, 2012 19:56 UTC (Tue) by **HelloWorld** (guest, #56129) [Link]

> I look forward to you arguing that such implementations aren't good while ignoring that most of the problems with macros also exist with templates.
Nonsense, there are numerous problems with macros. Here's a simple example:
template<typename T> bool max(T x, T y) { return x > y ? x : y; }
#define max(x, y) x > y ? x : y

The macro version will evaluate one argument twice. Bad things happen when you do max(a, max(b, c)). Yes, that can be fixed by parenthesizing each macro argument, but I shouldn't need to deal with that kind of crap. The template version will complain when the types don't match, the macro won't. Other people have tried to make it suck less by using macros to generate functions: http://www.canonware.com/rb/
That sucks too, because I really don't feel that I should be responsible for the bookkeeping that approach requires. It also doesn't work if you need to deal with complex types like int(*)(void) or whatever. Yes, one can use a typedef in that case. No, that's not how it should be. Generic algorithms and data structures are important enough to warrant language features to support them.

> Whenever the many deficiencies of STL is pointed out, you defensively say that "they fixed that in boost::XXX",
Boost doesn't usually "fix" the STL, it simply provides solutions to other problems than the STL does. std::list (de)allocates memory for you but works with arbitrary types. boost.intrusive don't work with all types (because they need the pointers to be embedded within the type), but it's more flexible with regards to memory management.

> but ignore that any missing containers, string types and other omissions from the standard C library can be easily rectified through the use of a small set of additional libraries.
It can't, generic data structures suck when the language doesn't support them. And besides, it's not like one always needs boost. Often enough, STL containers and algorithms do the job just fine.

Reply to this comment

A generic hash table

Posted Aug 17, 2012 8:18 UTC (Fri) by **jzbiciak** (guest, #5246) [Link]

```
template<typename T> bool max(T x, T y) { return x > y ? x : y; }
```

Erm... I don't think that's the right return type, unless you want `max(a, max(b, c))` to have rather unusual semantics.

Reply to this comment

A generic hash table

Posted Aug 17, 2012 9:02 UTC (Fri) by **jzbiciak** (guest, #5246) [Link]

Actually... any use of that `max` that returns `bool` would have rather unusual semantics with that return type. Almost reminds me of PHP. ;-)

As far as the whole C / C++ debate goes: Up until a few years ago I was an ardent C proponent and avoided C++. Since then, I've really given C++ a chance, and I now tend to prefer it for new projects. I still use C for quick one-offs, but for anything larger I use C++.

For one thing, C++ compilers are much better now than when I first looked at them ~15 years ago. You can throw some seriously complicated template messes at a modern compiler and it'll sort it out and give you rather tight code more often than not. 5 to 10 years ago, good code in that circumstance was more the exception than the rule. Bringing STL into the language in the

90s, plus the rise of Boost and template metaprogramming really forced compilers to up their game here.

I remember around 10 years ago playing with the "Stepanov" benchmark. It's a benchmark that measures a compiler's ability to deal with abstraction in a C++ program by wrapping a simple addition with a `double` in increasing amounts of C++ abstraction. G++ performed rather poorly at the time (incurring a 10x - 15x slowdown at the worst, IIRC) and our own compiler at work showing a 50x slowdown. Now I believe both compilers show no slowdown. They are able to flatten away the abstractions. It's beautiful.

For the problems I'm tackling in my day job, I really appreciate the compositional style templates offer me. Too often I find myself needing "an X with detail Y". In C, I'd have to code up a separate function that combined the two, or a macro that instantiated X subbing in a Y where necessary. In C++, it's just X<Y>, if X is a suitable template. I get type safety and an awful lot of compile-time computation and help.

This can be really wild but useful at the limit. I have a truly compelling example I'd love to share, but since it's my day job and it's proprietary technology, I don't know just how much I can share. I will say that the code I generate relies on a ton of compile-time computation to generate code that's sufficiently efficient that I don't mind running on design simulations of our chips that run at speeds on the order of 100Hz, plus/minus an order of magnitude. You read that right: anywhere from 10 to 1000 CPU cycles per second.

(Note: I've completely ignored error messages. C error messages are rarely obtuse and usually easily grokked. C++ with multiple nested templates is another ball of yarn altogether... They generally suck, and I'm with you if you think they're an abomination.)

Reply to this comment

A generic hash table

Posted Aug 17, 2012 9:49 UTC (Fri) by **HelloWorld** (guest, #56129) [Link]

You're right of course :). Oops.

Reply to this comment