

 rust-lang / rfcs

## support alloca #618

New issue

 Open

steveklabnik opened this issue on 21 Jan 2015 · 40 comments



steveklabnik commented on 21 Jan 2015

Member



Issue by thestinger

Friday May 03, 2013 at 01:35 GMT

For earlier discussion, see [rust-lang/rust#6203](#)

This issue was labelled with: A-llvm, I-wishlist in the Rust repository

It's probably going to be tricky to get this working with the segmented stack implementation.



5



steveklabnik referenced this issue in rust-lang/rust on 21 Jan 2015

support alloca #6203

 Closed

steveklabnik added the A-wishlist label on 21 Jan 2015



steveklabnik referenced this issue on 3 Apr 2015

Allow dynamically sized arrays on stack #1031

 Open

nagisa commented on 14 May 2015

Contributor

While we don't have segmented stack anymore, I don't know how we could make it work with stack probes. Maybe do another probe every time alloca gets called?



ticki commented on 29 Feb 2016

Contributor

Is anyone drafting an rfc for this?



steveklabnik commented on 29 Feb 2016

Member

I don't think so. I believe there's a pretty large contingent that doesn't want to see alloca at all.

On Feb 29, 2016, 07:19 -0500, [Tickinotifications@github.com](mailto:Tickinotifications@github.com), wrote:

Is anyone drafting an rfc for this?

—

Reply to this email directly or view it on GitHub([#618 \(comment\)](#)).

1



5



ticki commented on 29 Feb 2016

Contributor

But why?



2



stalkerg commented on 14 Mar 2016

## Assignees

No one assigned

## Labels

T-lang

## Projects

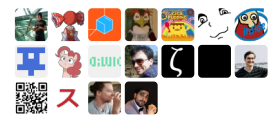
None yet

## Milestone

No milestone

## Notifications

18 participants



Simple link: <https://github.com/tomaka/vulkano/blob/master/TROUBLES.md>

👍 5



golddranks commented on 14 Mar 2016

TL;DR for the link: it would be nice to be able to dynamically allocate an array on the stack, for passing a reference down. This is to avoid more costly heap allocation. @tomaka has a real-world use case with the Vulkano wrappers.

👍 7



petrochenkov commented on 14 Mar 2016

Contributor

That particular use case would be better covered by `SmallVec<N>` as far as I can see from that short description.



golddranks commented on 14 Mar 2016

I thought so too.

But I think there is many other use cases too, especially in embedded and kernel development.



tomaka commented on 14 Mar 2016

Ha! I didn't even advertise that file. I think I linked it once on IRC.

I could use smallvec for most situations. In fact I already use it in glium. It's more of an ideological complaint as using smallvec would waste a lot of space, and not a pragmatic problem.

😊 4



ticki commented on 14 Mar 2016

Contributor

Type-level integers will certainly solve *some* parts of the problem. But alloca would certainly be ideal.

My usecases largely stems from kernel code (Redox), etc. Heap allocation is never ideal in these environments.

👍 7



kirillkh commented on 30 May 2016 • edited ▾

I find myself wishing there were stack-allocated arrays quite often. One use-case that comes up: allocate a temporary array, do some processing, tear it down (in the same function). The alternatives to doing it on stack are either temporary heap allocation (heavy price) or allocate and cache (inconvenience, sometimes not even feasible, e.g. when the array is allocated by `Iterator::collect()`; with stack-allocated arrays it would be possible to extend the collections API to allow collecting into stack-based arrays).

👍 5



tomaka commented on 20 Aug 2016 • edited ▾

Another problem of `SmallVec` is that the generated assembly is really more verbose and sub-optimal.

The following code, that uses fixed-size arrays:

```
fn test_array(val1: u32, val2: u32) {  
    let array = [val1, val2];  
    ptr(array.as_ptr());  
}
```

...Generates the following assembly:

```
subq    $40, %rsp
movabsq $42949672965, %rax
movq    %rax, 32(%rsp)
leaq    32(%rsp), %rcx
callq   ptr2
addq    $40, %rsp
retq
```

Very nice and clean. Using `alloca` would probably do something similar but without using a constant for the `subq`.

The following code, which uses a `Vec`:

```
fn test_vec(val1: u32, val2: u32) {
    let array = vec![val1, val2];
    ptr(array.as_ptr());
}
```

...Generates this:

```
pushq   %rsi
subq    $32, %rsp
movl    $8, %ecx
movl    $4, %edx
callq   __rust_allocate
movq    %rax, %rsi
testq   %rsi, %rsi
je      .LBB2_1
movabsq $42949672965, %rax
movq    %rax, (%rsi)
movq    %rsi, %rcx
callq   ptr2
movl    $8, %edx
movl    $4, %r8d
movq    %rsi, %rcx
addq    $32, %rsp
popq    %rsi
rex64   jmp    __rust_deallocate
.LBB2_1:
callq   _ZN5alloc3oom3oom17h72360ae2d301e7a7E
ud2
```

Again you can't do much better. Using a `Vec` obviously has the drawback that it allocates memory.

Finally, this code:

```
fn test_smallvec(val1: u32, val2: u32) {
    let mut array: SmallVec<[_; 2]> = SmallVec::new();
    array.push(val1);
    array.push(val2);
    ptr(array.as_ptr());
}
```

...Generates this: <https://gist.github.com/tomaka/dca0e20760486b7933b0409add4434e6>

I'm leaving in 10 minutes so I didn't take time to benchmark, but seeing the assembly I already know that the `smallvec` version is far worse than the fixed-sized-arrays version. Now I'm not even sure it's better than the `vec` version.




**tomaka** commented on 20 Aug 2016

Alternatively, this code:

```
fn test_smallvec(val1: u32, val2: u32) {
    let array: SmallVec<[_; 2]> = [val1, val2].iter().cloned().collect();
    ptr(array.as_ptr());
}
```

Generates this: <https://gist.github.com/tomaka/838e28e480a198d7c4d4bd448d0bf6be>  
Which is a bit better.

📌  **tomaka** referenced this issue in `vulkano-rs/vulkano` on 20 Aug 2016

### Some improvements to the generated assembly #227

 Merged



**comex** commented on 21 Aug 2016

...Generates this: <https://gist.github.com/tomaka/dca0e20760486b7933b0409add4434e6>

Ew...

It should be possible to significantly improve this by tweaking the implementation of `smallvec`. Right now, `push` is not getting inlined into its callers (but `grow` is being inlined into it). It should be the opposite: `push` should be fully inlined as a small helper that calls a non-inlined, perhaps `#[cold] grow` if the capacity's out of range - which should allow LLVM to constant fold all the size calculations and, in the case of your example with a fixed number of pushes, generate trivial assembly.

To avoid excessive code in non-constant cases, it would probably help to switch from the current high-level enum representation, which requires switching on the variant each time the capacity or base pointer is loaded, to having dedicated fields for capacity and base pointer which would initially be set as appropriate for the stack buffer. (though that would break moves, sigh.) Either that or split `push` into an inline fast path that only works for the stack case and a separate function for the heap case.



**diwic** commented on 29 Aug 2016

Would it be possible to support `alloca` through a closure approach, like:

```
fn alloca<T: Clone, R, F: FnOnce(&mut [T]) -> R>(count: usize, init_val: T, f: F);

fn test_array(val1: u32, val2: u32) {
    alloca(2, 0, |array| {
        array[0] = val1;
        array[1] = val2;
        ptr(array.as_ptr());
    });
}
```

`Alloca` could potentially be implemented through (in order of preference) some LLVM intrinsic, built-in assembly, or even FFI.



**jmesmon** commented on 29 Aug 2016

@diwic

- at the lowest level, we need something that works with any unsized type, not just arrays.
- having to have a default initialized value is an inefficiency that some people will want to avoid (due to the cost of initializing or dropping some objects)
- `T: Clone` bound might be too restrictive.

My thoughts are:

- If we can't expose a fully generic safe interface, it should be OK to expose a fully generic unsafe interface with specific safe wrappers (though ideally we'd have a safe interface).



**eddyb** commented on 29 Aug 2016

Member

We can probably "just" make `[x; n]` work with runtime values of `n` - besides, VLAs have their own LLVM support which probably optimizes better than the general C `alloca`.

OTOH, I've heard that anything like that affects optimizations and machine codegen, so it may be a bad idea to just allow it by default, it could end up to optimization-losing accidental usage.

👍 1



**diwic** commented on 29 Aug 2016

@jmesmon Sure - it would be nicer with something that takes `FromIterator` instead of just cloning it. Or at least allow `mem::uninitialized()` to work as expected for the initialization.

As for the unsized type instead of arrays - I haven't seen an example of that up the thread, maybe you could add one so we don't forget this use case?

@eddyb, oh, interesting. As for not allowing it by default, maybe a `VLA<T>` could be our next lang item then? (Which derefs to a slice.)



comex commented on 30 Aug 2016 • edited ▾

It's pretty unclear to me why people want `alloca` over reserving a worst-case uninitialized buffer on the stack, in the form of a fixed-size array, and using however much of it is required.

Kernel code is mentioned; but if you're in the kernel, your stacks are probably not pageable, and taking up RAM whether you use them or not. And if you don't account for worst case stack usage in general, you're liable to crash. So the only way `alloca` can help is if certain possible call stacks will fit in the available stack space with the actual array sizes required, but *not* if each function on the stack uses its individual worst case. And either you've carefully measured this and you're *sure* there's no way for unexpected input to overflow the stack, or else you use some complicated scheme to check at runtime, keeping in mind that after calling `alloca`, you need enough space left not just for the current function but for anything it might call (including recursively).

Oh, and if you mess this up it may be exploitable (depending on whether stack probes are enabled and you have guard pages around your stacks).

Of course, allocating large fixed-size buffers can overflow the stack too, but it's easier to catch, because it'll crash whenever the call stack in question is reached rather than depending on the size of the arrays.

User code has a bit more of a case. Concerns about unpredictable stack overflow still apply, which is one reason usage of `alloca` and VLAs in C tends to be heavily discouraged. But it's not a completely binary choice between "fits" and "doesn't fit", since previously unused stack pages may not be taking up real RAM. If you have a big stack buffer, as long as you keep it uninitialized, it won't fault any fully unused 4kb-or-so pages contained in it (even with stack probes, if properly implemented), but whatever goes on the stack /after/ it might be touching pages past the normal high water mark, wasting memory.

But I'm a bit skeptical that that matters much in most cases. Especially if your worst case buffer isn't all that large - if it doesn't hit the previous high water mark, it doesn't matter. And if you don't have a lot of threads, the absolute worst possible outcome is "only" faulting the entirety of each thread's stack at a few MB each.

👍 1



ticki commented on 30 Aug 2016

Contributor

@comex, thing is, there is many cases where there simply is no worst-case size.

Oh, and if you mess this up it may be exploitable (depending on whether stack probes are enabled and you have guard pages around your stacks).

You can already exploit the stack when guard pages are disabled, so `alloca` isn't particularly special in that sense.



comex commented on 30 Aug 2016

@ticki Can you elaborate? You say there's no worst-case size, but there's certainly a size past which you'll overflow the stack. So are you dealing with purely trusted input, or do you intend to set a limit based on dynamically calculating the amount of stack space remaining?

You can already exploit the stack when guard pages are disabled, so `alloca` isn't particularly special in that sense.

(It's special in that if you have guard pages but *not* stack probes, as tends to be the case for C/C++ programs in userland and some kernels, unchecked allocas let you easily jump past the guard page. Very large fixed-size arrays (4kb is the theoretical minimum but larger tends to be required in practice) can also do that, but they're not that common, and trickier because: whether or not there is a guard page, unchecked allocas allow relatively precise control over where the stack ends up, making exploitation considerably easier. If there *isn't* a guard page, the biggest threat is probably recursion, just because it's more common than either large arrays or `alloca`. Of course, as long as you have both guard pages and stack probes, none of this matters since the worst case is "just" a crash.)

jmesmon commented on 30 Aug 2016

there's certainly a size past which you'll overflow the stack

Which I'd expect to abort the same way other allocation failures abort on memory exhaustion.



diwic commented on 31 Aug 2016

@comex, thanks for your well written thoughts. Makes me wonder why I wanted alloca in the first place.

But if you're making a library which is supposed to work on both, say, a small device with very little RAM, and a big laptop, and so the small device might have up to 10 items in the array and the laptop might have up to 1000, you can't just hard-code "1000" for your fixed size array, because that'll overflow the small device.

You could then possibly configure your library for your target somehow...or just use VLA/alloca, which is quick and easy, and will use just as much memory as it needs.

👍 2



ticki commented on 31 Aug 2016

Contributor

@comex

Can you elaborate? You say there's no worst-case size, but there's certainly a size past which you'll overflow the stack. So are you dealing with purely trusted input, or do you intend to set a limit based on dynamically calculating the amount of stack space remaining?

Obviously there is an upperbound, but what I mean is that it isn't reasonable to, say, make an array of that size, since it is in a reasonable size say 99% of the time. Yet, a special input could yield the upperbound, which would result in stack overflow. Reserving this much space for that worst-case is simply not plausible.



comex commented on 1 Sep 2016

I'm a bit confused about the meaning of your comment - by "is in" and "yet" do you mean "isn't" and "yes"? I guess you're saying your input is indeed trusted?



whitequark commented on 3 May 2017

Looks like the RFC PR was never posted here, so: [#1808](#)

🔖 bestouff referenced this issue in [petertodd/rust-obstack](#) on 30 Aug 2017  
**alloca() #2**

🔗 Open

🔖 TheDan64 referenced this issue in [TheDan64/inkwell](#) on 11 Sep 2017  
**Improve speed by moving function collections to stack arrays #12**

🔗 Open



TheDan64 commented on 13 Sep 2017

I've recently ran into an issue similar to the ones mentioned earlier. I have a very common pattern in my library where I take a slice of my own types as a function argument, map and collect that slice to a vec of raw pointers, and pass that vec's raw pointer to a FFI call (along with the size), returning the (independent) result of that call.

So, it seemed to me that the obvious optimization would be to allocate that contiguous block of memory on the stack instead of the heap, since it is only needed for the scope of the function call and isn't ever returned. Because this pattern is so common, it means many heap allocations are performed on the regular, even though they don't need to be.

I get that overflowing the stack is a legitimate concern and I don't know the answer to that. But I do know that for my library, the average input slice is very likely to be small enough *not* to overflow the stack, and so not being able to allocate a runtime sized array on the stack is a big hindrance to performance.

I also feel strongly that setting some arbitrarily chosen compile-time upper bound to the input slice size just to be able to allocate a fixed size stack array (ie via the use of `arrayvec`, `smallvec`, or other similar crates) seems like an artificial limitation upon my crate's users and so is a lackluster workaround.

Just throwing out an idea that I don't think was mentioned: Would it at all be possible for rust to see if there is enough space on the stack to allocate such an array, and if not, panic? Or some sort of allocation function/macro that returns `Result<[T], NotEnoughStackSpace>` I'm not knowledgeable on the capabilities of stack probes and the like - but that seems like a reasonable compromise to me.

**whitequark** commented on 13 Sep 2017 • edited ▼

Or some sort of allocation function/macro that returns `Result<[T], NotEnoughStackSpace>`

You can definitely add such a function on every Tier 1 platform Rust currently supports. It'd just have to read its own SP and compare it with the OS-defined stack limit.

This however raises questions for `no_std` code, because stack probes are transparent--they check for whether it's possible to access the page by accessing the page. However, such a function would have to interface with the OS explicitly.

This is probably worth exploring as a separate crate.

**TheDan64** commented on 13 Sep 2017 • edited ▼

Well, an `alloca` only lives for the scope of the function it's created in, right? So I don't think a separate crate could return a stack allocated `[T]` from a function (unless it's always inlined or something). I guess I was thinking something more along the lines of handwavey compiler magic, though, a macro might still work. Either way, I think it would still need some compiler support, since there currently is no notion of a stack allocated array (so how could you work with such a thing?).

**kevincox** commented on 14 Sep 2017 • edited ▼

Since `llvm` has support for eliding allocations now I think it would be much better to ensure the `Vec` can receive this optimization (in at least some cases) and recommend that. This leads to natural code and simple APIs. For example I think the following should be optimized to a stack allocation.

```
fn checksum(len: usize) -> u32 {
    let mut buf = Vec::with_capacity(len);
    for _ in 0..len {
        buf.push(getbyte());
    }
    crc32(&buf[..])
}
```

We could make a couple of functions like this and add a test that there are no heap allocations to prevent regressions. Of course this wouldn't be a guaranteed optimization but it can be made reliable enough for almost all use cases.

👍 2    🗨 1

**whitequark** commented on 14 Sep 2017

Since `llvm` has support for eliding allocations now I think it would be much better to ensure the `Vec` can receive this optimization (in at least some cases) and recommend that.

This doesn't help `no_std` code at all since there's no `Vec`.


👍 4

**kevincox** commented on 14 Sep 2017

This is a good point. While I think my solution should still be the recommended approach for `std` code I see that we would need some sort of no-allocation guaranteed solution.

**briansmith** commented on 25 Sep 2017

@kevincox Your idea about automatically eliding allocations in Vec is good. It doesn't require an RFC (AFAICT) and it is complementary to this feature request, so I recommend filing a new issue in the rust-lang/rust issue tracker requesting that feature.

 **Centril** added the **T-lang** label on 23 Feb

 **petrochenkov** removed the **A-wishlist** label on 24 Feb



**TheDan64** commented on 13 May • edited ▾

I've started to try out some of the proposed ideas in a crate here: [https://github.com/TheDan64/scoped\\_alloca/](https://github.com/TheDan64/scoped_alloca/)

It works by creating a wrapper function to clang alloca; and hoping rust's LTO optimizations will inline the wrapper. Seems to work in release(at least in 1.26), but not in debug. (I wonder if we can force the required optimizations for debug?)

In particular I've started with a version of @diwic's idea which collects an `ExactSizeIterator` into an `alloca` so it should be safe to use (since the iterator is the initializer) and looks like this:

```
let v = vec![1, 2, 3, 4];
let iter = v.iter().map(|v| v + 4);
let res: Result<i32, ()> = alloca_collect(iter, |alloca_slice| alloca_slice.iter().sum::<i32>());
```

Next, I'm planning on looking at the more unsafe/uninitialized variants. I would be happy to get feedback/contributions from anyone who is interested in this topic



**whitequark** commented on 13 May

This is really not the right way to go (that would be to implement #1909). Your solution is inherently fragile.



**TheDan64** commented on 13 May

Sure; it's more exploratory as to how it could look if something similar was integrated into the language rather than being for actual use right now.



**whitequark** commented on 13 May

You could implement a Rust alloca intrinsic very easily, but that would (I think) need to go through an RFC first to be integrated.



**phaazon** commented 22 days ago

Contributor

We need this feature! Is there any RFC around it? If not, I might write it! 🤪



**whitequark** commented 22 days ago

@phaazon #1909



**phaazon** commented 22 days ago

Contributor

Yeah I just read it! Amazing work peeps! <3