Giving Rust a chance for in-kernel codecs

[LWN subscriber-only content]

April 26, 2024

This article was contributed by Daniel Almeida

Video playback is undeniably one of the most important features in modern consumer devices. Yet, surprisingly, users are by and large unaware of the intricate engineering involved in the compression and decompression of video data, with codecs being left to find a delicate balance between image quality, bandwidth, and power consumption. In response to constant performance pressure, video codecs have become complex and hardware implementations are now common, but programming these devices is becoming increasingly difficult and fraught with opportunities for exploitation. I hope to convey how Rust can help fix this problem.

Some time ago, I proposed to the Linux media community that, since codec data is particularly sensitive, complex, and hard to parse, we could write some of the codec drivers in Rust to benefit from its safety guarantees. Some important concerns were raised back then, in particular that having to maintain a Rust abstraction layer would impose a high cost on the already overstretched maintainers. So I went back to the drawing board and came up with a new, simpler proposal; it differs a bit from the general flow of the Rust-for-Linux community so far by realizing that we can convert error-prone driver sections without writing a whole layer of Rust bindings.

The dangers of stateless decoders

Most of my blog posts at Collabora have focused on the difference between stateful and stateless codec APIs. I recommend a quick reading of this one for an introduction to the domain before following through with this text. This talk by my colleague Nicolas Dufresne is also a good resource. Stateless decoders operate as a clean slate and, in doing so, they require a lot of metadata that is read directly from the bit stream before decoding each and every frame. Note that this metadata directs the decoding, and is used to make control-flow decisions within the codec; erroneous or incorrect metadata can easily send a codec astray.

User space is responsible for parsing this metadata and feeding it to the drivers, which perform a best-effort validation routine before consuming it to get instructions on how to proceed with the decoding process. It is the kernel's responsibility to comb through and transfer this data to the hardware. The parsing algorithms are laid out by the codec specification, which are usually hundreds of pages long and subject to errata like any other technical document.

Given the above, it is easy to see the finicky nature of stateless decoder drivers. Not long ago, some researchers crafted a program capable of emitting a syntactically correct but semantically non-compliant H.264 stream exploiting the weaknesses that are inherent to the decoding process. Interested readers can refer themselves to the actual paper.

The role of Video4Linux2 codec libraries

A key aspect of hardware accelerators is that they implement significant parts of a workload in hardware, but often not all of it. For codec drivers in particular, this means that the metadata is not only used to control the decoding process in the device, it is also fed to codec algorithms that run in the CPU.

These codec algorithms are laid out in the codec's specification, and it would not make sense for each driver to have a version of that in their source code, so that part gets abstracted away as kernel libraries. To see the code implementing these codecs, look at files like `drivers/media/v4l2-core/v4l2-vp9.c`, `v4l2-h264.c`, and `v4l2-jpeg.c` in the kernel sources.

What's more, with the introduction of more AV1 drivers and the proposed V4L2 Stateless Encoding API, the number of codec libraries will probably increase. With the stateless encoding API, a new challenge will be to capture parts of the metadata in the kernel successfully, bit by bit, while parsing data returned from the device. For more information on the stateless encoding initiative, see this talk, by my colleague Andrzej Pietrasiewicz or the mailing-list discussion. A tentative user-space API for H.264 alongside a driver for Hantro devices was also submitted last year, although the discussion is still on a RFC level.

Why Rust?

Security and reliability are paramount in software development; in the kernel, initiatives aimed at improving automated testing and continuous integration are gaining ground. As much as this is excellent news, it does not fix many of the hardships that stem from the use of C as the chosen programming language. The work being done by Miguel Ojeda and others in the Rust-for-Linux project has the potential to finally bring relief to problems such as complex locking, error handling, bounds checking, and hard-to-track ownership that span a large number of domains and subsystems.

Codec code is also plagued by many of the pitfalls listed above and we have discussed at length about the finicky and error-prone nature of codec algorithms and metadata. Said algorithms, as we've seen, will use the metadata to guide the control flow on the fly and also to index into various memory locations. That has been shown to be a major problem in the user-space stack, and the problem is even more critical at the kernel level.

Rust can help by making a whole class of errors impossible, thus significantly reducing the attack surface. In particular, raw pointer arithmetic and problematic `memcpy()` calls can be eliminated, array accesses can be checked at run time, and error paths can be greatly simplified. Complicated algorithms can be expressed more succinctly through the use of more modern abstractions such as iterators, ranges, generics, and the like. These add up to a more secure driver and, thus, a more secure system.

Porting codec code to Rust, piece by piece

If adding a layer of Rust abstractions is deemed problematic for some, a cleaner approach can focus on using Rust only where it matters by converting a few functions at a time. This technique composes well, and works by instructing the Rust compiler to generate code that obeys the C calling convention by using the `extern "C"` construct, so that existing C code in the kernel can call into Rust seamlessly. Name-mangling also has to be turned off for whatever symbols the programmer plans to expose, while a `[repr(C)]` annotation ensures that the Rust compiler will lay out structs, unions, and arrays as C would for interoperability.

Once the symbol and machine code are in the object file, calling these functions now becomes a matter of matching signatures and declarations between C and Rust. Maintaining the ABI between both layers can be challenging but, fortunately, this is a problem that is solved by employing `cbindgen`, a standalone tool from Mozilla that is capable of generating an equivalent C header from a Rust file.

With that header in place, the linker will do the rest, and a seamless transition into Rust will take place at run time. Once in Rust land, one can freely call other Rust functions that do not have to be annotated with `#[no_mangle]` or `extern "C"`, which is why it's advisable to use the C entry point only as a facade for the native Rust code:

```
// The C API for C drivers.
pub mod c {
    use super::*;

    #[no_mangle]
    pub extern "C" fn v4l2_vp9_fw_update_probs_rs(
        probs: &mut FrameContext,
        deltas: &bindings::v4l2_ctrl_vp9_compressed_hdr,
        dec_params: &bindings::v4l2_ctrl_vp9_frame,
    ) {
        super::fw_update_probs(probs, deltas, dec_params);
    }
```

In this example, `v4l2_vp9_fw_update_probs_rs()` is called from C, but immediately jumps to `fw_update_probs()`, a native Rust function where the actual implementation lives.

In a C driver, the switch is as simple as calling the `_rs()` version instead of the C version. The parameters needed by a Rust function can be neatly packed into a struct on the C side, freeing the programmer from writing abstractions for a lot of types.

Putting that to the test

Given the ability to rewrite error-prone C code into Rust one function at a time, I believe it is now time to rewrite our codec libraries, together with any driver code that directly accesses the bit stream parameters. Thankfully, it is easy to test codec drivers and their associated libraries, at least for decoders.

The Fluster tool by Fluendo can automate conformance testing by running a decoder and comparing its results against that of the canonical implementation. This gives us an objective metric for regressions and, in effect, tests the whole infrastructure: from drivers, to codec libraries and, even, the V4L2 framework. My plan is to see Rust code being tested on KernelCI in the near future so as to assess its stability and establish a case for its upstreaming.

By gating any new Rust code behind a `KConfig` option, users can keep running the C implementation while the continuous-integration system tests the Rust version. It is by establishing this level of trust that I hope to see Rust gain ground in the kernel.

Readers willing to judge this initiative may refer to the patch set I sent to the Linux media mailing list. It ports the VP9 library written by Pietrasiewicz into Rust as a proof of concept, converting both the `hantro` and `rkvdec` drivers to use the new version. It then converts error-prone parts of `rkvdec` itself into Rust, which encompasses all code touching the VP9 bit stream parameters directly, showing how Rust and C can both coexist within a driver.

So far, only one person has replied, noting that the patches did not introduce any regressions for them. I plan on discussing this idea further in the next Media Summit, the annual gathering of the kernel media developers that is yet to take place this year.

In my opinion, not only should we strive to convert the existing libraries to Rust, but we should also aim to write the new libraries that will invariably be needed directly in Rust. If this proves successful, I hope to show that there will be no more space for C codec libraries in the media tree. As for drivers, I hope to see Rust used where it matters: in places where its safety and improved ergonomics proves worth the hassle.

Getting involved

Those willing to contribute to this effort may start by introducing themselves to video codecs by reading the specification for their codec of choice. A good second step is to refer to GStreamer or FFmpeg to learn how stateless codec APIs can be used to drive a codec accelerator. For GStreamer, in particular, look for the v4l2codecs plugin. Learning `cbindgen` is better accomplished by referring to the `cbindgen` documentation provided by Mozilla. Lastly, reading through a codec driver like `rkvdec` and the V4L2 memory-to-memory stateless video decoder interface documentation can also be helpful.

Index entries for this article

Kernel	Development tools/Rust
Kernel	Video4Linux2
GuestArticles	Almeida, Daniel

Send a free link

(Log in to post comments)

Giving Rust a chance for in-kernel codecs

Posted Apr 27, 2024 8:16 UTC (Sat) by **gmatht** (subscriber, #58961) [Link]

It seems to me that the reason to write is Rust would be quite obvious to anyone who hasn't lived under a rock for the last nine years. What is less clear to me is why we have in-kernel codecs in the first place. Is it faster to blit video to a Weyland server from the kernel, or do the codecs need low-level access to the GPU for acceleration?

Reply to this comment

Giving Rust a chance for in-kernel codecs

Posted Apr 27, 2024 13:02 UTC (Sat) by **atnot** (subscriber, #124910) [Link]

> What is less clear to me is why we have in-kernel codecs in the first place. Is it faster to blit video to a Weyland server from the kernel, or do the codecs need low-level access to the GPU for acceleration?

The article touches on it pretty well I think, but the reason basically comes down to stateful vs stateless decoding hardware. In the olden days hardware media decoders were pretty simple as far as programmers were concerned. You just put the bytes from your file in one end and got pixel data out the other. And vice versa, which is e.g. how those high resolution IP cameras are so cheap.

However, among other things implementing an entire complex codec including the file parsing logic this way is pretty inflexible and kind of wasteful when you have perfectly good CPU cores sitting there anyway. So in the newer stateless model, you instead favor implementing only the "hot loops" of the codec in hardware (some of which may even be shared by multiple codecs) and rely on the driver to pass in the required state. That requires a much deeper understanding of how the codec works, which can't really be fully offloaded to underspace because similar to GPUs, the kernel still needs to validate that the potentially dangerous commands it's getting actually make sense before passing them to the hardware.

Reply to this comment

Giving Rust a chance for in-kernel codecs

Posted Apr 27, 2024 21:20 UTC (Sat) by **ocrete** (subscriber, #107180) [Link]

These are codecs which are largely implemented in hardware, so they need a driver. These are also not the codec accelerators that are part of the GPU that you see on desktop platforms, but they're independent hardware blocks on all the non-x86 chips.

Reply to this comment

Giving Rust a chance for in-kernel codecs

Posted Apr 29, 2024 4:15 UTC (Mon) by **flussence** (subscriber, #85566) [Link]

v4l2 has to do a lot of container format framing and bitstream marshalling, and the hardware usually just does the compression and decompression which is computationally heavy but rigid in its scope. Media formats are also sometimes link-layer formats, as in DVB MPEG-TS packets, so it also has to handle corrupted or missing data robustly. It'd be pretty bad if someone figured out a kernel 0-day that could be broadcast over DVB-T airspace.

It's almost exactly the same deal as encryption - you can have silicon that does RSA or SHA2 really fast, but it probably doesn't know what X.509 or ASN.1 are and so the responsibility for juggling them lies with the kernel (and historically those have also been a security headache).

Reply to this comment

Giving Rust a chance for in-kernel codecs

Posted Apr 29, 2024 14:44 UTC (Mon) by **leromarinvit** (subscriber, #56850) [Link]

I get the impression that the reason we have in-kernel codecs is an impedance mismatch between (modern) hardware and the v4l2 api. Handing the raw bitstream to the kernel and expecting decoded frames in return is a concept that made sense with decoders that handle everything in hardware, but it doesn't with stateless decoders that only implement the expensive parts.

From my perspective (as someone who has never worked with v4l2 or video codecs in general, so take this with a grain of salt) it would make more sense for the kernel to expose the operations the hardware actually implements, and let a user space library deal with mangling the raw bitstream into whatever the hardware accepts. What is to be gained by doing this in the kernel? Is transparently supporting stateless codecs in v4l2 really worth the downsides of having full blown media format parsers in the kernel?

Reply to this comment

Giving Rust a chance for in-kernel codecs

Posted Apr 29, 2024 16:00 UTC (Mon) by **farnz** (subscriber, #17727) [Link]

There is an alternative option, in `drivers/gpu/drm`, where your in-kernel component handles device management (buffer handling, command submission etc), and you supply an open-source userspace component (doesn't have to be part of Mesa3D) that implements the majority of the codec driver.

Reply to this comment

Giving Rust a chance for in-kernel codecs

Posted Apr 29, 2024 20:03 UTC (Mon) by **stormer** (subscriber, #62536) [Link]

DRM/KMS drivers is clearly another option and an option that I keep an eye on for the future. For me, V4L2 uAPI choice was made before my time and it does work well enough. But in DRM/KMS subsystem, the notion of scheduling work from various user process onto a certain set of fixed function cores is already solved. And this is exactly what modern stateless codecs are becoming. Rasperry Pi HEVC decoders process two task concurrently (entropy decoding and reconstruction). Mediatek VCODEC do that same, but with two independent cores. Rockchip 3588 have quad core jpeg decoders, but also dual core HEVC decoders were you can use both independently or bind the cores to process 8K frames. This in V4L2 framework is a true challenge and huge gap for which DRM/KMS do have solution for.

Of course, the validation is still mandatory, but we can easily design a common command bitstream for this type of hardware, and then internally validate and reconstruct a validated command that is simply the register layout in RAM that can be used by the scheduler to applied. Might be a steep dive, but what I'm saying is that DRM/KMS is a very viable solution for future

CODEC and may help fixing all the legacy and short coming of V4L2 uAPI and hopefully support Vulkan Video on this type of hardware.

Reply to this comment

Giving Rust a chance for in-kernel codecs

Posted Apr 29, 2024 16:20 UTC (Mon) by **flussence** (subscriber, #85566) [Link]

I see what you're proposing, though that'd be an enormous amount of work. It'd be doing mostly the same thing as the DRM KMS/GBM/TTM/DRI3/etc transition, and although that has delivered solid improvements it's also been an ongoing effort for 20 years now.

With modern media hardware the main obstacle is becoming middle managers desperately trying to put ever more useless buzzword-chasing junk in the devices. Webcam utility plateaus at "looks okay in standard room illumination, provides the functionality of a point and click camera from 2005, does not eat cpu like a winmodem" and anything added past that is fluff and an extra point of failure the driver has to account for, even if nothing in userspace will ever take advantage of it. Sound cards went through the same blight at the turn of the century.

Reply to this comment

Giving Rust a chance for in-kernel codecs

Posted Apr 29, 2024 16:42 UTC (Mon) by **farnz** (subscriber, #17727) [Link]

I think it's also worth calling out the DMA-BUF UAPI as a prerequisite for change. Without DMA-BUF, the only ways to move data between hardware blocks involve either copies into userspace memory, or having a subsystem-specific mechanism like the V4L2 media controller API to move data around in kernelspace within a single subsystem.

DMA-BUF enables you to export a handle to data to userspace, which can then pass it to a different kernel subsystem. This allows you to elide the copies, and export data from V4L2 to DRM, or from DRM to V4L2.

Reply to this comment

Giving Rust a chance for in-kernel codecs

Posted Apr 29, 2024 19:11 UTC (Mon) by **ocrete** (subscriber, #107180) [Link]

What you are describing is more or less exactly how stateless drivers work. The userspace parses the bitstream and fills this information into C structures that the kernel then reads and copies this information almost directly into the registers to program the hardware. Most of the drivers is similar to DRM drivers in that the kernel mostly deals with allocating buffers, lifetime, telling the hardware where it's allowed to read and write, etc.

And although the legacy v4l2 API allows one to mmap the buffers directly, the modern way is just to export them as a dmabuf which works like any other dmabuf.

Reply to this comment

Giving Rust a chance for in-kernel codecs

Posted Apr 29, 2024 19:50 UTC (Mon) by **stormer** (subscriber, #62536) [Link]

I'm quite suspicious about what you mean by "having in kernel codecs". Stateless decoders refer to some hardware that does not track the sate of the decoding process. The benefit of such hardware represent a special processing core that can be scheduled to different tasks (task being streams in the

context) in a very flexible way. In contrast, the stateful kind of decoding hardware needs to maintain the state of each concurrent streams and this is often limited by firmware and co-processor resources. The scheduling usually cannot be adapted to any third party constraints (consider cgroup and other kind of quotas).

All in call, the layer placed in front of these drivers through V4L2 Stateless Decoder interface does not constitute an "in-kernel" codec". The hardware implements the heavy processing, userspace implements the high level decoding logic and parsing. The responsibility of such Linux kernel driver is to ensure that the parsed parameters and state is valid and matches the pre-allocate resources size. This isn't something Rust can solve and will ever solve, this is pure logic and logic can be broken even in Rust. For each codec, specific stream parameters passed to the hardware imply specific auxiliary or image sizes. It is the responsibility of the Linux kernel to ensure that the hardware will not overflow these for a given decoder command. As this is a mix of code and hardware, Rust brings nothing here.

Though, in order to adapt to all kind of hardware, we are forced to implement small bits of the codec spec. This is implemented in the form of different codec specific libraries. For H.264 and HEVC, we transform and reorder references lists to match each hardware requirement. For VP9 and AV1, we need to post process some of the probability tables in order to combine bitstream probability updates with observations made during the decoding process. Just a quick read at these C helpers, you'll notice its made of tones of C arrays which if overrun will overwrite each other silently. I do hope our implementation is right and safe in C, but real confidence could come from the guaranties offered by the Rust language/compilers.

Another study that Daniel has been doing is the inner part of the stateless hardware programming. This is not very specific at this point, but this kind of hardware have hundreds of variable sized parameters packed into registers at different bit location. What the study revealed is that this code often misses some integer sign and sizes consideration. This may lead to miss-programming of the hardware in corner cases, errors that would generally be prevented by the Rust compiler. I personally think we could do more then just safety, and improve how we program these register with the Rust language, but at this step, this pure choice and preference.

Reply to this comment

Giving Rust a chance for in-kernel codecs

Posted Apr 30, 2024 8:32 UTC (Tue) by **leromarinvit** (subscriber, #56850) [Link]

Thanks for the detailed explanation! Somehow, I misread the article and was left with the impression that v4l2 contains at least significant parts of full codec implementations, to support the parts that some decoders don't implement in hardware. That's what I commented on, but I see now that what looked like a strange design to me was simply a misunderstanding.

Like I said, so far I've never had the need to handle video in my own code, so I know next to nothing about how all the components work and interact.

Reply to this comment

Giving Rust a chance for in-kernel codecs

Posted Apr 30, 2024 9:38 UTC (Tue) by **farnz** (subscriber, #17727) [Link]

Note that even without the little quirks, stateless codecs will always need something to manage three chunks of state for them:

1. Position in the bitstream. Something needs to track how far through the bitstream you are, and avoid either skipping bits that the stateless codec needs to see, or sending it the same bits repeatedly when it doesn't need them again.

2. Picture reordering. Video codecs can predict the "current" picture from both past pictures, and future pictures. If you have pictures in output order 1 2 3 4, where pictures 2 and 3 can use picture 4 as a reference, the bitstream will contain pictures in the order 1 4 2 3, and something needs enough state to rearrange that into 1 2 3 4.
3. Reference picture buffering. Video codecs aren't allowed to refer to arbitrary pictures when they predict the current picture; they're only allowed to refer to "reference pictures". Every codec has rules for when a picture enters the set of reference pictures, and when it leaves, and the stateful wrapper has to ensure that the stateless decoder has the "right" set of reference pictures available to it.

And then you get into more complicated things like stormer described, but also things like MPEG-2 having per-sequence and per-picture state, which needs to be available to the decoder with every slice it's decoding. The wrapper thus has to understand enough of the bitstream to know what state the decoder needs to be given with each slice it's decoding.

Reply to this comment

Giving Rust a chance for in-kernel codecs

Posted Apr 27, 2024 14:37 UTC (Sat) by **dvdeug** (subscriber, #10998) [Link]

I wonder at what point we'll get a programming language in kernel that simply doesn't do bounds checks because it can prove they're not needed. SPARK/Ada can do it, and could be used; Coq and Idris are more powerful and advanced, but hardly kernel usable. It just seems like runtime bounds checks are a waste of time, when they can be made explicit under the control of the programmer, and the compiler can prove they're sufficient.

Reply to this comment

Giving Rust a chance for in-kernel codecs

Posted Apr 27, 2024 16:07 UTC (Sat) by **walters** (subscriber, #7396) [Link]

There's also https://github.com/google/wuffs in this space. I've only seen it referenced in this space before. I suspect the tradeoff boils down to the costs of introducing a 3rd programming language; bridging Rust and C is already hard enough.

Reply to this comment

Giving Rust a chance for in-kernel codecs

Posted Apr 27, 2024 20:16 UTC (Sat) by **tialaramex** (subscriber, #21167) [Link]

WUFFS-the-language is currently implemented as a transpiler, which produces C. It could in principle produce anything ("unsafe" Rust, Python, Java byte code) but since WUFFS isn't finished that's what it does today.

I don't know enough about the details of the work being done here to figure out whether WUFFS is the right tool for the job. Today WUFFS is an excellent (very fast yet entirely safe) way to produce codecs in software. It has no idea what a "string" is, which isn't a problem for this application space, and as you observed it doesn't emit bounds checks since it has necessarily checked your code can't have any bounds misses, so they would be redundant.

[If the only way to avoid bounds misses in your implementation is to check for them, which is probably a sign you've designed it wrong, you have to write them, and then WUFFS will see that your checks are sufficient and the code compiles, or maybe it won't and you just found a bug in your bounds checks...]

Reply to this comment

Giving Rust a chance for in-kernel codecs

Posted Apr 27, 2024 23:04 UTC (Sat) by **tialaramex** (subscriber, #21167) [Link]

I think it might be beneficial to expand, given the context, on what exactly WUFFS is (and is not) doing here to pull this off.

The main trick is that WUFFS is only _checking_ a proof, so while a proof assistant is very complicated and barely plausible at the scale we need so it will be annoyingly slow, WUFFS is just checking a proof which is much faster. But how do we get a proof? Well, because it was conceived specifically for this purpose, to the exclusion of even basic features not needed for such work (e.g. it literally can't do "Hello, World") the language is designed so that mostly you are writing the things needed for a proof as you write code e.g. in WUFFS we specify not only the mechanical size of a data type when declaring a variable, like it's an 8-bit integer, but also specify the range, maybe this particular 8-bit integer must have values between 10 and 186 inclusive. WUFFS will ensure everywhere that this variable is assigned that the value is guaranteed to be between 10 and 186, and equally, when assigning from (or comparing against) this value, it will be known to be between 10 and 186 inclusive again.

You will need to write stuff that just wouldn't exist in other languages, to justify yourself, but it's still not violently dissimilar from the assertions or similar you might write in Rust or even C - it's just that they're always checked, whereas a C assertion is not checked in release code, and they're in mathematical terms, not imperative terms, if we say that x < 10 in this block of code we mean that *in every possible case* this is true, and WUFFS will refuse to compile the software unless it can see why that's true.

To this end, WUFFS knows a bunch more intro level discrete mathematics than a compiler often does and it can be told to apply specific rules it has been taught (which mathematicians have long proved true) to what it knows to achieve new knowledge e.g. if a < b and b < c, then a < c when writing such an assertion.

Because it isn't a prover or proof assistant, WUFFS won't try to figure out whether your code *could* be proved correct. It just checks the proof you've in effect written in the code itself.

Reply to this comment

Giving Rust a chance for in-kernel codecs

Posted Apr 27, 2024 16:30 UTC (Sat) by **dwlsalmeida** (subscriber, #144268) [Link]

The Rust compiler will optimize away bound checks if it can prove that they are not needed through static analysis. You can also opt out of bound checks but that has to go into an unsafe{} block, see https://doc.rust-lang.org/std/vec/struct.Vec.html#method....

Reply to this comment

Giving Rust a chance for in-kernel codecs

Posted Apr 27, 2024 16:44 UTC (Sat) by **atnot** (subscriber, #124910) [Link]

I'd say this is basically already the case for rust with e.g. capable iterators that remove almost every case where you'd normally use indexing in languages without them. Of course that doesn't help you when you do for whatever reason need to for whatever reason, but to be honest I think most of my codebases contain few if any instances of array indexing.

Reply to this comment

Giving Rust a chance for in-kernel codecs

Posted Apr 29, 2024 20:40 UTC (Mon) by **gspr** (subscriber, #91542) [Link]

Since multimedia codecs are the focus of this thread, I think they constitute a type of code where explicit indexing very much remains necessary. It's often not possible to express things like "add the red value of this pixel plus c times the blue value of the pixel k rows directly above it" using iterators, let alone iterators where the compiler can automatically elide bounds checking.

Reply to this comment

Giving Rust a chance for in-kernel codecs

Posted Apr 29, 2024 23:19 UTC (Mon) by **tialaramex** (subscriber, #21167) [Link]

But it certainly _is_ possible to develop a language where a machine tool can follow along with the discrete maths needed to see that we're in bounds when we do this, or point to where our code doesn't do what we thought it did (e.g. it's mandatory for videos in our encoding to have width multiple of 16, and so we didn't check that but our code assumes it's true, leaving a gap for anybody to just lie)

We only need to: Develop that language (for conventional software codecs it exists, it's named WUFFS) and teach the handful of people who write new codecs how to use this purpose made language and then sit back and enjoy the high performance totally safe results.

Reply to this comment

Giving Rust a chance for in-kernel codecs

Posted Apr 27, 2024 17:03 UTC (Sat) by **Cyberax** (★ **supporter** ★, #52523) [Link]

> SPARK/Ada can do it

Not really. Ada sucks for anything that uses dynamic allocations or generic code. It only recently copy-pasted Rust's approach with borrowing, but it's still not nearly as advanced. Wuffs is probably the best practical tool for parsing.

Reply to this comment