TEXT ONLY

Concurrency in WebAssembly

EXPERIMENTS IN THE WEB AND BEYOND

CONRAD WATT

As Herb Sutter once famously observed in his article *The Free Lunch is Over*, CPU clock speeds have trended toward a plateau in recent years, while the *number* of CPU cores per chip continues to increase. As such, modern software reaches ever more urgently for multicore concurrency (alongside other strategies such as data parallelism) in order to capture the full performance capabilities of modern hardware.

As a separate trend, more and more software is now built for the web platform—the collection of open standards implemented in every web browser and underpinning the operation of every website. The web platform offers an exceptional value proposition for developers: a sandboxed and (mostly) uniform virtual machine environment for application deployment that is supported by billions of devices worldwide by default, through their installed web browsers. The web platform is not without its limitations, however. The browser virtual machine can only execute code written in one of the standardized Web languages (before WebAssembly, just JavaScript, HTML and CSS), and such code can access only a restricted, safe abstraction of the underlying capabilities of the local system. Therefore, application developers must often choose between the uniformity and reach of the web platform and the flexibility, performance, and expressivity attainable only through native deployment.

The web platform's constraints are logical considering the security model of the web—untrusted code from a website may be downloaded and executed by a visitor's computer without their explicit intervention, so the capabilities of any downloaded code must be limited. Moreover, the constraints make sense from a social perspective—in order to maintain the uniformity of the web platform, new features are generally standardized only if all browser vendors agree to implement them, creating an understandable bottleneck in the capabilities of the web based on the priorities, resources, and technical circumstances of these companies.

With the advent of Wasm (WebAssembly) as an extension of the web platform, browser virtual machines now offer a uniform, developer-facing bytecode language and compilation target. This raises the tantalizing narrative that developers are now free to develop their applications in whichever programming language they like, so long as this language can be compiled to (or otherwise executed on top of) Wasm, in the same way that compilers might traditionally support separate x86 and ARM instruction-set targets.

Certainly, it is true that Wasm marks an exciting new era for developer engagement with the web platform, and many projects have created profoundly impressive Web applications backed by Wasm as a compilation target for their preferred source language. Such a compilation target must still respect the constraints of the web platform as a whole, however. Mismatches between the interfaces promised to programmers by source languages and the capabilities of the underlying web platform are a constant trap in compiling to Wasm. Even simple examples such as a C program using the language's native file-system API present difficulties—for obvious reasons the web platform does not allow its programs arbitrary access to a client's file system, so the behavior of this API must be carefully virtualized during the compilation process if it is supported at all.

Often such gaps can be papered over by the compilation toolchain somewhat automatically, without the developer needing to know all of the details so long as their code runs correctly end to end. This state of affairs is strained to its limits when compiling programs for the web that use multicore concurrency features. This article aims to describe how concurrent programs are compiled to Wasm today given the unique limitations that the Web operates under with respect to multi-core concurrency support and also to highlight some of the current discussions of standards that are taking place around further expanding Wasm's concurrency capabilities.

The capability to create a new thread of execution on the web that can execute user-defined code is carefully limited as it's a powerful and security-sensitive operation.

THE STATUS QUO

What does multicore concurrency on the web platform look like today? The capability to create a new thread of execution on the web that can execute user-defined code is carefully limited as it's a powerful and security-sensitive operation. Currently the only way to do this is through the Web Workers API (including specialized variants such as Service Workers and Worklets, not explicitly covered by this article). Web Workers can be contrasted with native OS threads in that there is a pervasive assumption throughout the web platform that objects allocated in one Worker (or the main thread) *cannot be transferred by reference* to another Worker.

There are two main causes of this constraint. First, JavaScript and DOM (Document Object model) objects are complicated enough that it's not possible to make their accesses thread-safe without severely compromising performance (e.g., by putting locks on everything). JavaScript objects in particular have highly dynamic representations in most browser implementations. Second, modern garbage collectors (GCs) in web browsers are generally built for speed, making use of a longstanding assumption that a lot of garbage-collection work can be safely performed thread-locally, without needing to worry about stopping the world or following cross-Worker references to ensure object liveness. Invalidating these deep-rooted implementation assumptions would create a monumental performance and security headache for web browsers, and so they are more or less entrenched as pervasive limitations on the capabilities of new Web concurrency features.

These limitations mean that general shared-memory concurrency, where multiple threads may access the same object concurrently, is not possible as a rule, with one major exception to be discussed here shortly. Most communication between Workers happens through asynchronous message passing that simply prevents the transfer of object references by construction (either by explicitly erroring if they are present in the message or implicitly creating a separate copy in the other thread rather than sharing the reference directly).

Because of this restriction, each newly created Web Worker must allocate a fresh global context on start-up (since it cannot share its creator's context by reference, which would allow concurrent access). This context must include (at a minimum) a new JavaScript global object specific to that Worker, and so the process of creating a Web Worker is many times more expensive than creating a new native thread. There are tricks that browser engines can play to ameliorate this issue, but, in practice, if your web code wants to create many concurrently executing jobs, it is recommended that you create a somewhat fixed pool of Web Workers on start-up, and then implement load balancing and pooling of jobs among them in user code rather than creating a new Worker for each job. The overhead of being required to perform this management in user space, especially layered on top of some underlying OS scheduler that is ignorant of the workload, has not been fully quantified (contributions welcome!).

When coding directly in JavaScript or another web-first language, the web platform's restrictions against cross-thread reference sharing are exposed directly to the programmer, so their code can be directly written with these restrictions in mind. However, if I simply take a C program that calls `pthread_create()` and attempt to compile it to Wasm, there seems to be a mismatch. I could try to map `pthread_create()` to the web platform's new `Worker()` functionality, but imagine that the C program allocates a struct in one thread and tries to pass a pointer to that struct as an argument of `pthread_create` (figure 1 shows a simple C program that allocates a struct and then accesses the struct in two other threads through a pointer). How can I

FIGURE 1: **A SIMPLE C PROGRAM THAT ALLOCATES AND ACCESSES A STRUCT**

```
#include <stdio.h>
#include <pthread.h>

typedef struct {
    _Atomic int bar;
} foo;

void* incr(void* myFoo) {
    ((foo*)myFoo)->bar++;
    return NULL;
}

int main() {

    foo myFoo = {0}; // how do we compile this allocation in
                        WebAssembly?
    incr(&myFoo);
    printf("%d\n", myFoo.bar); // print 1

    // run two threads that take &myFoo as an argument and
    call incr on it
    pthread_t thread_id1, thread_id2;
    pthread_create(&thread_id1, NULL, &incr, &myFoo);
    pthread_create(&thread_id2, NULL, &incr, &myFoo);
    pthread_join(thread_id1, NULL);
    pthread_join(thread_id2, NULL);

    printf("%d\n", myFoo.bar); // print 3

    return 0;
}
```

faithfully map this functionality to Wasm when we seem to lack the concept of shareable allocations?

As mentioned in the introduction of this article, concurrent programs (including C programs using `pthreads`) *can* in fact be compiled to Wasm. This is because there is an exception to the general principle that objects cannot be transferred between or accessed across multiple Web Workers (leaving aside certain simple immutable objects where the difference between reference sharing and copying is more or less semantically unobservable): the Wasm *shared memory* and the analogous JavaScript *SharedArrayBuffer*. Both of these objects are thin wrappers for the special allocation of a simple integer-indexed buffer of bytes—sometimes called the *backing store* for these objects—that can be allocated in one Worker and transferred by reference to another Worker (backing stores are not directly user-accessible, but when one of its user-accessible wrappers is included in a message to a Web Worker, the underlying backing store is passed by reference and a new wrapper object for it is allocated by the receiving Worker).

These special buffers/backing stores allow true shared-memory concurrent access and can fit within the previously mentioned implementation assumptions of web browsers since they have a simple layout in memory and cannot themselves hold references to any regular objects—they can only hold raw byte data. This means that their accesses can be implemented in a thread-safe way, and the shared buffers do not need to participate in the liveness analyses of other objects when thread-local garbage collection is performed. One Worker can write to a shared buffer, and another Worker can concurrently read from it and observe this write (at least subject to the horrifically complicated cross-core caching/synchronization behaviors that are exposed by every shared-memory feature[1,2]—both objects provide a suite of atomic access operations to facilitate user-level synchronization). As a brief aside, this low-level model of concurrent memory access was of particular concern when mitigating the notorious Spectre and Meltdown vulnerabilities on the web, and today, as a result, shared buffers can only be allocated by websites that enable a security policy known as *cross-origin isolation,* which restricts the use of third-party scripts.

In the context of compilation, and preserving the source-level semantics of the original program, this capability allows us (and in fact near-*obligates* us!) to bootstrap all of the source language's shared-memory concurrency behavior purely from the Wasm-level shared memory. This approach allows a feasible compilation scheme from the program from figure 1 to a Wasm module.

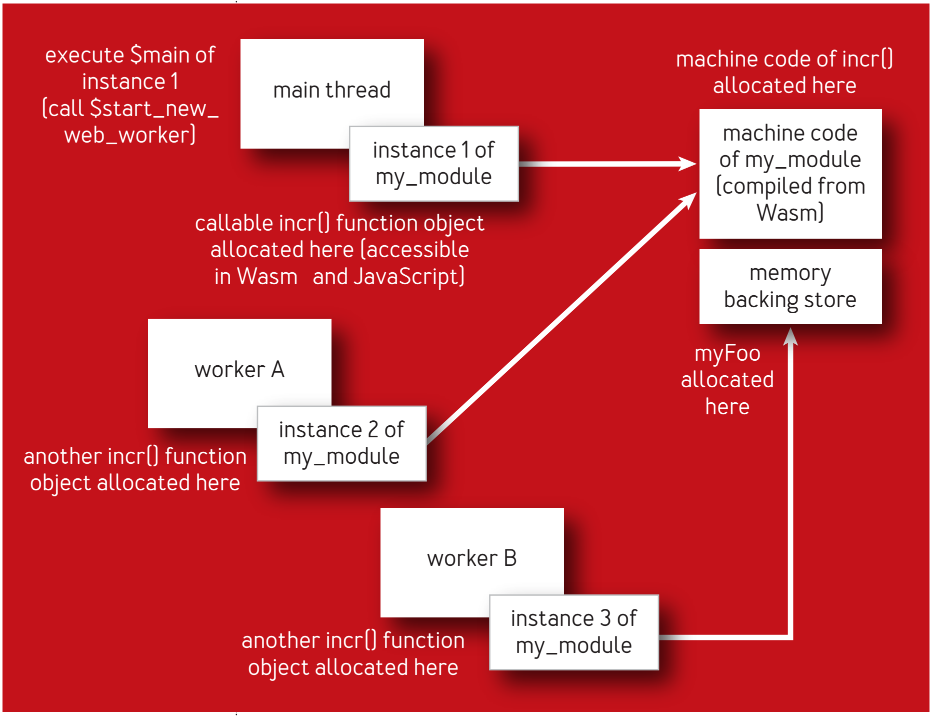
Before we get into the full detail of how this scheme works, it's necessary to explain a little background on the lifecycle of a Wasm module, once it has been produced by compilation of some source code. In order to execute such a Wasm module, the module must first itself be compiled to machine code by the Wasm engine (e.g., a web browser engine such as V8). Then, a process called instantiation gathers the module's requested imports and wraps the generated machine code with function objects that can be called and manipulated by host code such as JavaScript—the result of instantiation is a user-accessible *instance* *object* that contains these ready-for-execution function objects, among other things.

Our compilation scheme to Wasm produces a central module that imports a shared memory from the host. When the main thread of figure 1 allocates `myFoo`, because its pointer can be shared with other threads, the compiled Wasm code must allocate its representation into the imported shared memory. The address `&myFoo` is represented as an integer index into the memory in the compiled code, and likewise the function address `&incr` becomes an integer index into the list of Wasm function objects declared in the compiled code—note that for security reasons Wasm is a "Harvard architecture" where executable functions are only user-accessible in the form of these function objects, which have opaque representations and live in their own index space that is separate from the directly user-writable byte memory.

When `pthread_create`(... `&incr`, `&myFoo`) is called, we will create a new Web Worker (or request one from a pre-allocated pool), but as one crucial complication this Web Worker cannot access the existing Wasm instance or its functions due to the previously mentioned restriction that such objects cannot be shared between Workers. Therefore, we must initialize a new instance in the freshly created Web Worker, essentially *a separate copy of the same compiled program* (as an optimization, the compiled machine code can be cached/shared across instantiations, but the wrapping instance objects must still be separately allocated). To do this we must transfer the shared memory by reference to the new Web Worker to be imported as part of the new instantiation process for the module performed in that Web Worker. We also message the new Worker with two integers—the first representing the address of the function to begin executing (`&incr`) once instantiation is completed, and the second representing its argument: the address `&myFoo`.

The Wasm standards and toolchain community has taken to referring to this approach as an *instance-per-thread* compilation scheme, because it relies on creating a separate Wasm instance object representing a user-accessible view of the compiled program in each thread. Figure 2 depicts a

FIGURE 2: **DIAGRAM OF WEB WORKERS**


execute $main of instance 1 (call $start_new_web_worker)
main thread
instance 1 of my_module
machine code of incr() allocated here
machine code of my_module (compiled from Wasm)
memory backing store
callable incr() function object allocated here (accessible in Wasm and JavaScript)
worker A
instance 2 of my_module
another incr() function object allocated here
myFoo allocated here
worker B
instance 3 of my_module
another incr() function object allocated here

snapshot of the Wasm program described earlier executing in a browser. After instantiation, and during the subsequent execution of the compiled `main()` function, a chunk of the shared memory is used to store the byte representation of the C-allocated `myFoo` struct. Two Web Workers in a pool are created with their own Wasm instances since the instance (and constituent function objects) allocated in the main thread cannot be transferred to the Web Workers by reference. The Wasm implementation of `pthread_create` involves signaling to an idle Web Worker that it should execute the `incr()` function, as described earlier. The diagram in figure 2 (Web Workers executing the program shown in figure 1 when compiled to Wasm) depicts this point in the execution where the two Web Workers have created their separate instances that both reference the same underlying shared memory (and as an optimization, the compiled machine code).

THE FUTURE?

Several limitations to this approach remain, many of which are being discussed as part of the gargantuan *shared-everything threads* project that I am championing in the Wasm standards community alongside Andrew Brown (Intel) and Thomas Lively (Google). This umbrella project collects a number of interrelated standards proposals of varying complexity that are being investigated as possible ways to expand the concurrency capabilities of Wasm and the wider web platform. This effort also intersects with and takes inspiration from the JavaScript structs proposal from the JavaScript standards community, championed by Shu-yu Guo (Google) and Roy Buckton (Microsoft); this proposal looks at some of the limitations from a JavaScript perspective.

At this point I should also acknowledge the invaluable efforts of Andreas Rossberg (independent), Luke Wagner (Fastly), Ryan Hunt (Mozilla), and all the other members of the web standards community who have contributed to and participated in the development of shared-everything threads and related proposals in other standards bodies. I should also caution that all of the ideas discussed here are merely proposals, and no presumption should be made that they will be standardized in the web platform. I can only hope to give a flavor of the currently active discussions in the community, which I expect to inform the development of Wasm in the years to come.

Limitation 1—limited variety of atomics

Languages such as C/C++ and Rust offer a range of different "strengths" of different atomic operations, which allow expert programmers to selectively weaken the cross-thread synchronization guarantees of certain memory accesses in exchange for improved runtime performance. When accessing a shared memory, Wasm and JavaScript offer only two choices of access strength at the extreme ends of the spectrum—*unordered* and *sequentially consistent*. While these two access strengths are by far the most commonly used in real code, even in languages where other options are available, expertly written programs using other intermediate access strengths lose out on some performance when compiled to Wasm, since all such accesses must be compiled to Wasm's stronger and slower sequentially consistent accesses.

Adding some other intermediate access strengths to Wasm, such as release-acquire, would unlock more of this performance without requiring major structural changes to the language. These efforts are ongoing as one of the less onerous parts of the shared-everything threads project.

Limitation 2—inability to share Wasm instances

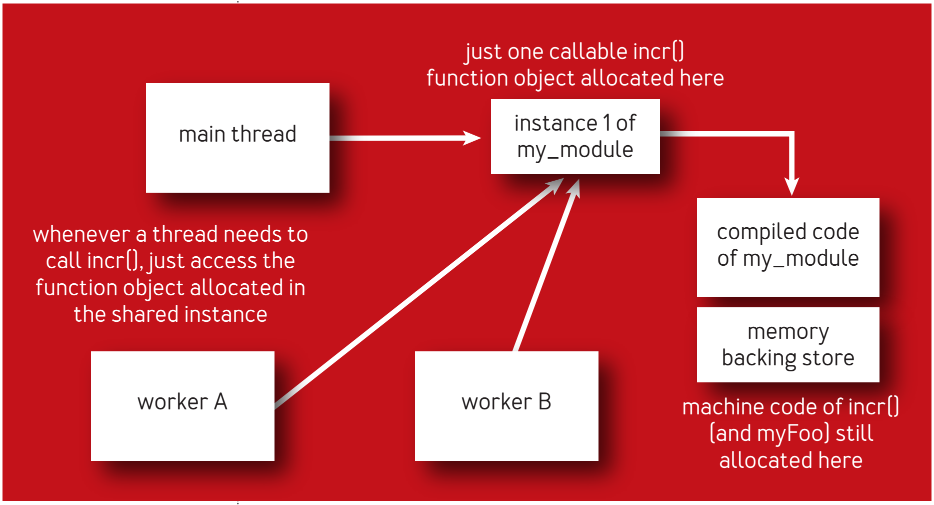
As previously described, when multithreaded code is compiled to Wasm, each Web Worker used in the compiled program must create a separate Wasm instance (see figure 2). As already noted, this leads to increased thread start-up costs (mitigated by pooling of Web Workers), but a more fundamental issue arises when attempting to support dynamic code loading, such as when compiling a source C program that calls `dlopen` to Wasm. This system call dynamically loads new code and data into memory and returns a bag of new pointers, including function pointers to the loaded code that may be called as normal.

Recall the approach for compiling C function pointers to Wasm that was previously sketched: Each thread must keep a consistent list of loaded functions, and then function pointers can just be compiled to integers that index this list, which can be safely passed between threads. If one thread executes `dlopen`, however, the compiled code in Wasm must laboriously pause the execution of every thread and update the list of loaded functions in order to ensure that any new function pointers introduced by `dlopen` behave correctly if they are used in another thread. This bookkeeping can be done automatically by the toolchain, but it is clearly not an ergonomic experience.

The widely-used Emscripten toolchain for Wasm offers experimental support for this approach when compiling C code that uses `pthreads+dlopen`.

To address this and related issues, discussions in the Wasm community have focused around the feasibility of introducing a concept of shared instances to the Wasm language. Instead of requiring each Web Worker to keep a separate Wasm instance and list of loaded functions, such a capability would allow the allocation of a single shared instance that all Web Workers point to. With this, dynamic code loading could be implemented in Wasm in a way that is closer to its expected behavior in native compilation without requiring that all threads be paused. See figure 3 for a sketch of this. If this code needs to dynamically load a new function (e.g., through `dlopen`),

FIGURE 3: **THE SUPPORT OF A HYPOTHETICAL SHARED INSTANCES FEATURE**


just one callable incr()
function object allocated here
main thread
instance 1 of my_module
compiled code of my_module
memory backing store
whenever a thread needs to call incr(), just access the function object allocated in the shared instance
worker A
worker B
machine code of incr() (and myFoo) still allocated here

only one shared list of functions (maintained by the shared instance) needs to be updated (in contrast to the current scheme shown in figure 2).

This extension, naively introduced, would violate the core invariant of the web as previously mentioned—now any object accessed by a function in the shared list would be accessible by multiple threads simultaneously. Discussions on possible designs of such an extension have therefore relied on mechanisms that would prevent such cross-thread–shared functions from holding references to objects that are not thread-safe, possibly through static restrictions enforced by Wasm's validation algorithm. Shared instances would still need new special handling in implementations, but at least the effect on existing objects in the web platform would be minimized. Several ideas in this space are currently the subject of a lively standards debate, taking account of the known constraints of the Web platform and the implementation resources of browser vendors.

Limitation 2+—inability to share JavaScript objects

Wasm programs often import capabilities from the JavaScript host. Created Wasm instances must therefore often hold persistent references to JavaScript functions and other objects that will be accessed during execution. As noted earlier, this is problematic when considering a possible extension to shared instances—without careful protections, it would be possible for a JavaScript function object to be accessed through the same shared instance in multiple threads, breaking the pervasive assumptions made by browser implementations about the thread-(un) safety of JavaScript objects and the safety of carrying out thread-local garbage collections. The knee-jerk reaction would be simply to ban any Wasm shared instance from importing or accessing a JavaScript object, but clearly this would severely limit the expressiveness of any compiled Wasm program making use of such a shared instance.

It is therefore natural to ask whether the capabilities of JavaScript could be expanded in some limited way to create objects that are more safely shareable. There is ongoing discussion in the JavaScript community around the ambitious *shared structs* proposal, authored by Shu-yu Guo, which would introduce exactly this capability. Of course, the usual caveat applies that this is an early stage proposal subject to significant debate in the standards community.

Limitation 3—inability to share Wasm garbage-collected structs

The discussions here have focused mostly on the compilation of C-style languages to Wasm, which primarily use the (shared) Wasm memory—a buffer of bytes. Wasm has recently been extended with new support for allocating structs and arrays that can piggyback off of a host environment's existing GC—on the web, this means JavaScript's GC. These new primitives are used to provide enhanced support for compiling garbage-collected languages to Wasm since they can, in principle, remove the need for the compiled code to ship its own (likely far less efficient) GC as part of the Wasm binary. Because these allocations are managed by the host's GC, they are subject to the same limitations as other allocated objects—

namely, the inability to share references to the allocations between threads/Web Workers. Compilation of a source language such as Java to Wasm must therefore disallow the use of source-level concurrency features (or mimic their behavior with a single-threaded simulation) if GC structs are used in the compiled code, since there is no way to faithfully support Java's much more permissive cross-thread reference-sharing behavior.

To address this issue, the Wasm standards community is discussing the feasibility of shared GC structs, which could be safely shared by reference across Web Workers. This capability would essentially be a Wasm-level view of the ongoing and closely related JavaScript shared structs proposal. As noted in the previous limitation, such shareable objects, if standardized, would need to be prevented from accidentally introducing the ability to concurrently access ordinary JavaScript objects through transitive references.

Limitation 4—lack of lightweight threads

Web Workers carry some overhead because of their need to allocate a separate JavaScript context on start-up. It is tempting to ask whether this overhead is necessary in the context of supporting compilation to Wasm, and whether a more lightweight thread creation primitive would be appropriate. Wasm standards-body discussions around this concept have focused on the performance of the existing Web Worker pooling compilation strategy and the significant engagement with the wider Web standards ecosystem that would be necessary to pursue such a new feature. It has also been observed that generated Wasm programs need to call out to the host (e.g., JavaScript) surprisingly often during execution, and therefore the utility of a lightweight thread without a JavaScript context (where such calls would not be possible) might be limited.

BEYOND THE WEB

While this article has focused mainly on the limitations and future of WebAssembly concurrency, it is worth emphasising that thanks to the existing efforts of the many web platform contributors, plenty of use cases for concurrency on the web already work *today*. Tools like Emscripten, where applicable, allow the arcane process of Web Worker organization and communication to be treated (mostly) as a black box, with high-profile projects such as Google Earth using this approach to bring vast "native" codebases to the Web.

It's also worth emphasizing that *shared-everything threads* is not the only standards project seeking to enhance WebAssembly's capabilities. Two somewhat-related recent extensions especially worth highlighting, *SIMD instructions* and *JavaScript Promise Integration* (JSPI), introduce enhanced support for data parallelism and asynchronous host interaction respectively, while the early stage *Stack Switching* proposal hopes to extend WebAssembly with core primitives for asynchronous computation. These efforts (along with the aforementioned shared-everything threads umbrella proposal and many others) show the best of the web standards community, with many different participants actively contributing towards making WebAssembly as powerful and expressive as it possibly can be, all without compromising the open, consensus-driven, and backwards-compatible nature of the Web platform.

The Web platform has proven to be popular enough that it is often the preferred abstraction even for other use cases such as server-side and cloud computation — consider for example the ubiquity of server-side JavaScript. In principle these environments can offer more tailored development experiences according to their unique priorities and technical constraints, but in practice the rigorous security model of the Web platform offers a lot "out of the box" to these environments, and economies of scale around Web development expertise mean that it just makes sense to focus efforts around providing a Web-like development environment rather than something more bespoke. WebAssembly specifically has attracted interest from even more varied environments related to embedded systems and blockchain. While all of these environments can offer limited additional capabilities to executing programs, such as file system access, on top of the base capabilities of the Web platform, they are still incidentally affected by several of its constraints — there are many benefits that come from adopting a widely used and implemented standard, but this is the trade-off! Many of the limitations discussed above (especially shared instances) will require core WebAssembly language extensions to address, which must be agreed among all stakeholders, even though many of these platforms are not subject to the same technical constraints as Web browsers.

To end on a positive note, it may be that these "off-Web" platforms, *because* they don't suffer from the same technical constraints as Web browsers, will become fruitful environments for experimentation with new WebAssembly concurrency features that would be too onerous for Web browsers to speculatively prototype. This in turn can create evidence and precedent that may be used to inform the standards process. Seeing the energy and creativity that so many are bringing to the Web standards community on this topic leaves me optimistic about the future of WebAssembly's concurrency capabilities, and I look forward to seeing how the efforts of everyone involved will pay off over the coming years.

References

1. Watt, C., Pulte, C., Podkopaev, A., Barbier, G., Dolan, S., Flur, S., Pichon-Pharabod, J., Guo, S.-y. 2020. Repairing and mechanizing the JavaScript relaxed memory model. *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 346–361; https://dl.acm.org/doi/abs/10.1145/3385412.3385973.
2. Watt, C., Rossberg, A., Pichon-Pharabod, J. 2019. Weakening WebAssembly. *Proceedings of the ACM on Programming Languages* 3 (OOPSLA), Article 133, 1–28; https://dl.acm.org/doi/10.1145/3360559.

Conrad Watt *is a co-chair of the W3C WebAssembly Community Group, the main industrial decision-making body for the language. He is also an assistant professor at Nanyang Technological University, Singapore. His work focuses on the definition of WebAssembly's shared-memory concurrency*

model, and more broadly on the machine-assisted verification of WebAssembly's language design and related artifacts. He obtained his Ph.D. from the University of Cambridge, where he also spent three years as a Research Fellow of Peterhouse. His doctoral dissertation on formal verification of WebAssembly was awarded the EAPLS (European Association for Programming Languages and Systems) Best Dissertation Award 2021 and an ACM Doctoral Dissertation Award Honorable Mention.