Cooperative Concurrency Needs `atomic`

…and It Can Be Implemented Efficiently

Anonymous Author(s)

Abstract

The sophistication of multitasking in mainstream software has increased in recent years, thanks to the increasing richness of network communication, physical world interaction and multi-process software architectures. These trends have pushed programming languages to offer better ways of writing such code. As just one example of this trend, in recent years some flavor of coroutine has been standardized (or is heading towards standardization) in JavaScript, Swift, and C++.

This paper makes two primary contributions. The first is an empirical study of multitasking in modern web applications. Our primary finding is that in current practice, web JavaScript code tends to exhibit risky patterns, from the perspective of atomicity. We argue that an `atomic` block primitive should be added to languages to help programmers avoid atomicity bugs.

We also experiment with two implementations of cooperative threads plus atomic: a JavaScript library and a dialect of C. This investigation shows that this language design can be implemented in a flexible and efficient way.

Keywords keyword1, keyword2, keyword3

1 Introduction

Many modern applications are highly multitasked. They have lots of balls in the air at the same time: communications over the internet, queries to databases, multiple I/O modalities to monitor and manage. Applications use a wide range of language and core library features to implement this multitasking: threads, event loops, coroutines, async functions, functional reactive programming, etc. We briefly note that some of these features are also relevant to *parallelism*, but that is *not* the subject of this paper or the project it describes.

Many popular application programming frameworks have elected to use some form of cooperative multitasking, because of the fear that preemption carries too much risk of concurrency bugs. While the authors of this paper support that line of reasoning, it does not mean that applications built on a cooperative multitasking framework are automatically free of concurrency defects. In fact there has been a great deal of research in the last decade on cataloging, exposing, diagnosing and fixing concurrency bugs in web and mobile applications that use a cooperative concurrency model (for example, [12, 17, 22]).

This paper starts with a measurement study of widely used web applications[1]. We observe that patterns that seem risky from an atomicity perspective are common on the web. In particular, there are many chains of short callbacks with short gaps between them. We believe it would be easy for a developer to test such code and effectively assume atomicity for the chain, even though it is possible with a small perturbation in timing for a conflicting task to execute in one of the gaps.

To reduce the risk of atomicity violations, we argue that languages designed for multitasking should have an atomic block primitive. We identify a conflict between atomic blocks and the anonymous concurrency that arises from event loop, coroutine or async function based systems. We conclude that languages with atomic blocks should have an explicit notion of task/thread naming.

This line of reasoning leads to a language design closely related to cooperative threads, with the addition of atomic blocks and two classes of functions: async/interruptible and atomic. We explore what such a design could look like in the context of two implementations: a JavaScript library and a C dialect. We address one interesting semantic issue, which is the nesting of atomic regions and task spawning. We also explore efficiency in the context of the C implementation, in particular the allocation of call frames and the implementation of yield/context switching.

All code and data described in this paper is available publicly on GitHub [URL removed for blind review].

2 Multitasking on the Web Today

We begin with some brief remarks on the history of JavaScript. At the beginning, the designers of JavaScript decided that preemptive threads are unacceptably concurrency-bug-prone. One good entry into the vast literature on how hard it is to diagnose and fix multithreading defects is [15]. These challenges have been understood for a long time; two decades ago Ousterhout wrote a popular critique of threads [19].

Early versions of JavaScript supported only event loop based concurrency. The simple picture of the implementation of JavaScript is that there is a single event queue that stores a collection of functions waiting for some enabling condition. These functions can be registered by builtin callback APIs, like `setTimeout`. In all cases we refer to these functions as

PL'18, January 01–03, 2018, New York, NY, USA
2018.

[1]We chose to focus on web applications for the simple reason that a large amount of deployed production code is easily available for the studying.

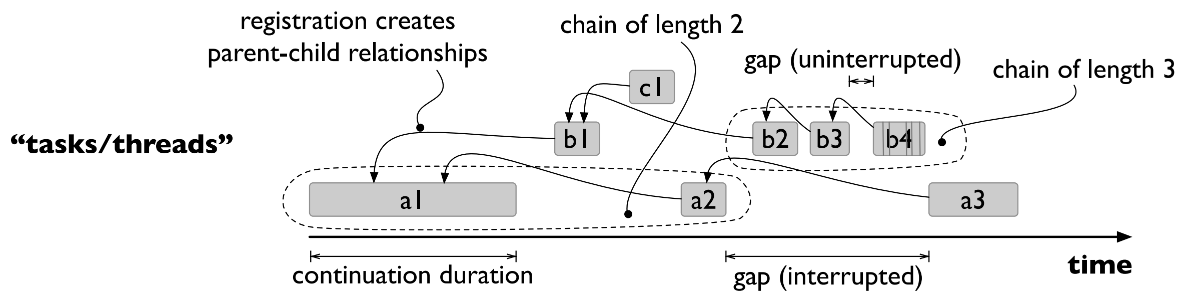

Figure 1. The features of JavaScript event execution that our modified Chromium browser measures. The gray boxes are continuations. The continuations are arranged in rows to suggest that they belong to the same task/thread. However, JavaScript does not have this concept, so this is purely a human interpretation. For example, a2 and b1 are both children of a1, and there is no way for the system to know what logical task either belongs to.

continuations. We refer to a continuation plus its enabling condition as an *event*. When an event becomes ready, the queue may execute its continuation whenever it chooses (generally as soon as possible).

Simple event loop programming has some serious drawbacks. It can make management of resources whose lifetime spans multiple callbacks quite tricky. This leads to a style of programming referred to as *stack ripping* [2], or more colloquially *callback hell*. These and more subtle problems are well analyzed and criticized by von Behren et al. in [26].

The increasingly sophisticated multitasking in modern applications and the well-known problems with events and threads have forced language designers to explore alternatives in recent years. The most widely used alternatives are *coroutines* and *async procedures*. Examples include async/await, which was added to C#/.Net in 2012; function generators (`function*`), which were added to JavaScript with ECMAScript 6 (standard published in 2015); even the C++ community is considering adding support for coroutines (currently on the short list for inclusion in the C++20 standard). Coroutine implementations are not *new*; for example, Modula-2 had coroutines in the 1980s. However, relatively few of the most widely used languages natively supported coroutines until recently.

The most recent versions of JavaScript support both coroutines and async functions.

2.1 Measurements

We modified a version of Chromium to gather data on JavaScript behavior from web applications. Figure 1 illustrates the features of JavaScript execution that our modified browser measures:

- Continuation duration is the time from the beginning of the execution of a continuation to the time it returns control to the event loop.
- Parent-child relationships are inferred by recording which continuation (the parent) is running when potential child continuations are registered/scheduled.

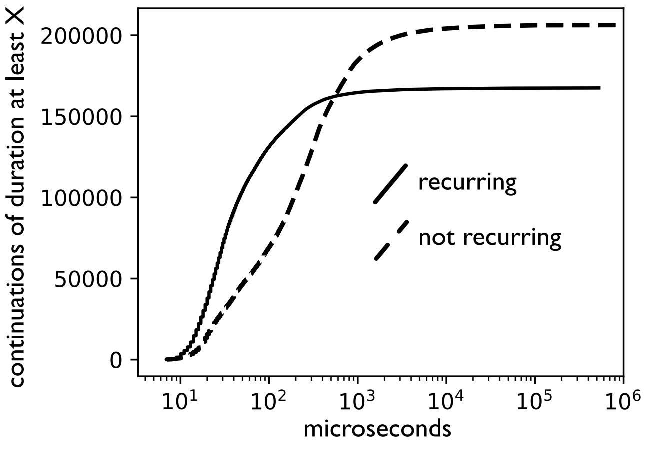

Figure 2. Most continuations are short. Many are very short.

- A *gap* is the difference between the start of a continuation and the end of its parent. A gap is *interrupted* if *any* other continuation runs between parent and child.
- We measure both the duration and number of microtasks in microtask batches. More on microtasks below. (The vertical lines in continuation b4 in the Figure are meant to suggest individual microtasks within a batch.)
- Figure 1 suggests with the horizontal rows of continuations that we can determine whether a continuation should be intuitively understood as continuing its parent's task or spawning a concurrent task. As far as we know it is *not* possible to automatically and precisely infer this.

2.1.1 Duration and Branching

Figures 2 and 3 start to give a sense of the shapes of the continuation trees in widely used JavaScript code. The first important observation is that most continuations are quite short. A large majority take less than one millisecond, and

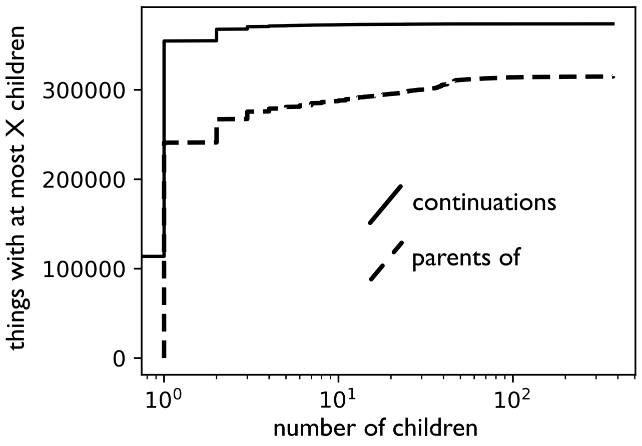

Figure 3. Most continuations have one child. A few have many, many children.

many are much shorter than that. Figure 2 shows both microtask batches and normal continuations combined into a single data set.

Short continuations are *not* direct evidence of an atomicity problem. However, it is interesting to see that JavaScript programmers seem to have taken very seriously the rule that in order to keep applications responsive, all function executions must complete quickly. This ecosystem norm will be important when we discuss library function design later.

The difference between the solid and dashed lines in Figure 2 highlights the importance of recurring events. In JavaScript, certain primitive APIs, like `setInterval` and `XMLHttpRequest` can register continuations that are called multiple times. As you can see, these recurring continuations (especially network request updates) account for a majority of the shortest.

Next, notice in Figure 3 that a majority of continuations have exactly one child. This suggests that JavaScript programmers are in many cases breaking linear sequences of logic up into multiple continuations. Presumably they are doing this either consciously to avoid responsiveness problems, or because the libraries they are using do not offer any other choice.

A large majority of continuations have at most one child, but a very few have a much higher branching factor. To visualize this long tail more clearly, the dashed line in Figure 3 shows not each continuation's branching factor, but each continuation's parent's branching factor. In principle this is not different information, but it emphasizes high-branching factor continuations more. This view of the data makes it clearer that there is a substantial amount of concurrency, in the sense that a substantial minority of continuations have a parent with more than one child.

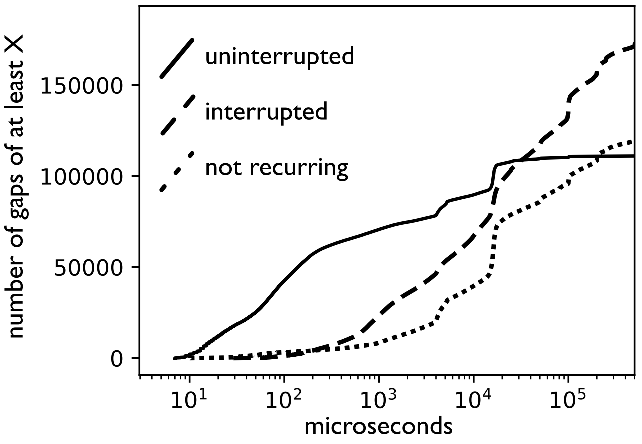

Figure 4. Most gaps are of reasonable length, but a nontrivial percentage are quite short (under one millisecond, or even 100 microseconds). This graph is truncated at half a second, because we are not interested in long gaps. There is only a small number of longer gaps, anyway.

2.1.2 Gaps

Figure 4 further clarifies the risk of atomicity violations. You can see that many of the time gaps between continuations are quite short. If it was always the case that there was a substantial time gap between parent and child, it would be more reasonable to believe that programmers are consistently being careful about the possibility of interruptions. There is no hard cut-off between long and short gaps, but it is interesting to note that roughly half of the measured gaps are shorter than one millisecond. The distribution tails off fairly quickly, but there are still many gaps shorter than 100 microseconds, and a few shorter than 10 microseconds.

It is especially interesting to observe that there are many uninterrupted gaps shorter than 100 microseconds and only a very few interrupted gaps in the same range. This means that while it is possible for such short gaps to be interrupted, actual interruptions are rare. We can further infer that even if applications are heavily tested, following common good practices, that some of these gaps will have never been interrupted during a test. However, all it takes is an unluckily timed network or UI event to interrupt a short gap, potentially triggering an atomicity bug.

We can see, by looking at the dotted line, that a large majority of the shortest gaps arise from recurring events. It is perhaps the case that these continuations tend to have simpler logic that represents a smaller risk of atomicity bugs. However, note that even though the relative number of short non-recurring gaps is small, there are still thousands of such gaps shorter than 1 millisecond across our data set.

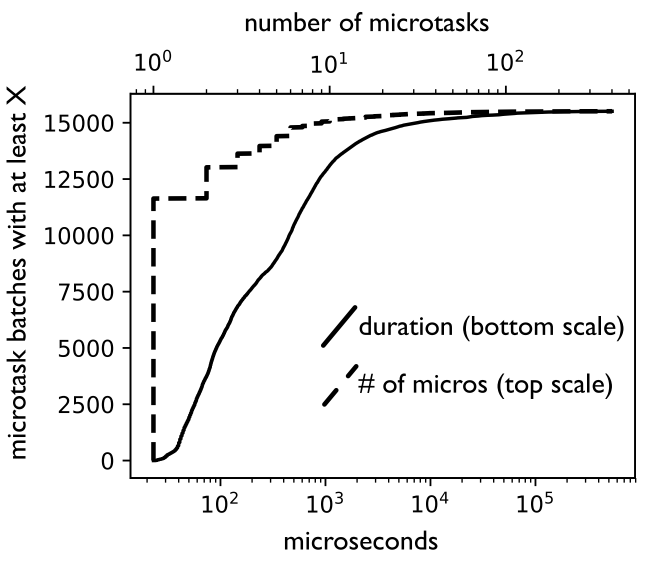

Figure 5. Most microtask batches are short, but a nontrivial number are long enough to possibly exceed the time and/or microtask count limits of some reasonable implementations.

2.1.3 Microtasks

Microtasks are a special subset of events in JavaScript. Unlike regular events, as soon as at least one microtask is placed in the microtask queue, the application switches to running microtasks until the queue is empty. In principle it is possible for an application to become unresponsive because microtasks keep enqueuing more microtasks.

Figure 5 shows information about the durations and microtask counts of microtask batches. Most batches are fairly short/small by both metrics, which is good from a concurrency bug perspective. However, a nontrivial minority of microtask batches get into the 10s and even 100s of milliseconds and/or dozens or hundred of microtasks. At that size, these microtasks start to be potential unresponsiveness problems. When batches of microtasks get too long, programmers may make ad hoc decisions about where to break them up, thus potentially creating more opportunities for atomicity problems.

2.2 Is This Actually a Problem?

In this paper we do not present any direct evidence of actual atomicity violations, only that patterns that seem risky are common. There has been a great deal of research on atomicity violations in general (for example, [15]). More importantly, Davis, et al. studied real world concurrency bugs in JavaScript projects by looking at bug databases [8]. They found several atomicity violations.

Given that the risk of atomicity violations is real, it is natural to wonder if there is anything about current languages and libraries that encourages these risky patterns. We believe there is.

There is a dilemma that library authors face in existing cooperative concurrency frameworks: When designing an API, should it be a regular atomic function or async in some way (callback, Promise, whatever)? If it is certain to complete in a small amount of time, an atomic function is preferable, because avoids the possibility of atomicity violations. However, if it might run for a long time, then the possibility of unresponsiveness comes in to play, and the library author is strongly incentivized to make the API async.

A similar line of reasoning led the developer's of Microsoft Windows' core Windows Runtime (WinRT) library to replace many atomic procedures with interruptible ones in Windows 8. In a post on the Microsoft Developer Blog [25], the team explains that they did this for procedures that "could likely take longer than 50 milliseconds to execute".

The dilemma gets especially interesting when you consider that many library functions can be fast or slow, depending parameters and/or some environmental factor. Consider, for example, any of the classic collection processing functions like map, filter, fold or reduce. If the size of the collection is small and the function being applied is reasonably fast, then the whole aggregate operation should also be fairly fast. On the other hand, if the collection is large or the function being applied is slow, then the whole thing will be slow. Because there is a real possibility of these aggregate operations being slow, the only good choice for a library author forced to choose between atomic and async is async. But this means we have introduced an undesirable interruption point when applying such a library function in a context where it will be fast. The same line of reasoning applies to any data transformation API (encrypt/decrypt, serialize/deserialize, compress/expand, etc).

In some recently defined JavaScript APIs, even some procedures that are very likely to execute quite quickly are defined in an interruptible style. As one example among many, the PBKDF2 function in the cryptography API returns a promise, which means that it is interruptible. PBKDF2 is a function that takes a number of iterations as a parameter. The iterations parameter must be set over a million to cause the PBKDF2 function to take more than a second on a newish computer. If the iterations parameter is closer to one thousand (which is appropriate for some applications), the time is in the millisecond range. In such contexts, the caller would almost certainly prefer that the call be atomic.

Consider a more extreme example: sending a network request. Sending a message over a wide area network will inevitably be slow, and network operations are generally defined to be asynchronous to accommodate this. However, in a microservices software architecture, applications use network protocols to communicate with other processes running on the same computer. Depending on the details of a specific request to a peer process, it may actually be

```
async function f_risky( params )
{
    let x1 = yield api1();
    let x2 = yield api2();
    return some_compute( x1, x2 );
}

async function f_safer( params )
{
    atomic {
        let x1 = yield api1();
        let x2 = yield api2();
    }
    return some_compute( x1, x2 );
}
```

Figure 6. A simple example of code that could lead to the risky gaps, and a version with an atomic block to eliminate the risk of an atomicity violation.

fast enough that the requester would prefer to block other internal tasks while waiting for the response.

Our conclusion from this analysis is that whether an API is defined to be atomic or async is often a dilemma. Many APIs need to be async in some calling contexts, but should be atomic in others. It seems that current design practice leans towards defining APIs in an async way to avoid responsiveness problems. But this has lead to what seems to be quite a lot of exposure to atomicity problems. Therefore, we believe that the language should accommodate flexible definitions that leave the async/atomic decision up to each individual caller.

3 An Atomic Block for JavaScript

We believe that the gaps described in the previous section represent a risk of atomicity violations, because of the low, but non-zero, probability that such gaps will be interrupted in any given execution. Figure 6 shows a very simple example of how atomicity violations can arise, and our proposed solution: an atomic block primitive. Previous research has shown that atomic blocks can be an effective and bug-resistant concurrency control mechanism in the mutlithreading context (for example, [10, 11, 21]). Unfortunately, implementing atomic blocks efficiently for preemptive threads is extremely hard. Perhaps more importantly, there are semantic questions that are unclear how to resolve satisfactorily for mainstream programming languages (for example how do actions performed within an atomic block relate to actions performed outside of any atomic block?). In this section we discuss ways that atomic could be added to JavaScript and similar languages that use cooperative concurrency.

3.1 A Simple Strategy and Its Fatal Flaw

One way atomic could be implement in event queue systems is as follows:

- When some code enters an atomic block, all currently registered continuations are removed from the queue and saved elsewhere.
- The code of the atomic block runs as usual, perhaps registering new continuations.
- When the code reaches the end of the atomic block it pauses and waits until the event queue is completely empty (note that continuations can themselves register more continuations).
- Once the event queue is empty, the code continues executing whatever comes after the atomic block and the saved continuations from the first step are restored to the event queue.

We believe that this strategy would work in many cases, but it has a fatal flaw related to nesting of atomic blocks and task spawning.

3.2 Background on Nested Atomics and Spawns

In this subsection we give a brief review of research on atomic blocks in the context of multithreading and transactional memory. We will relate this back to JavaScript and event queue concurrency below. The tricky semantic issue is how to define thread spawns nested within atomic blocks (and potential further nesting of both). One good treatment of the formal semantics of this issue is [16]. It seems there is no simple definition that works well in all cases. Some of the possibilities that have been explored are:

- Thread spawn within an atomic block is an error.
- Threads spawned within an atomic block are delayed until just after the block has completed. (*delay-spawn*)
- Threads spawned within a block are allowed to run concurrently, and the block is not complete until such threads are finished. (*spawn-contained*)

In the research on this subject there are good examples where each of these options is appropriate, and suggestions of more complex hybrid options.

3.3 Atomic and Event Queues

An issue with event queue concurrency is that there is no explicit notion of a task or thread that defines the identity of specific chains of continuations. In other words, when a continuation is registered, the system cannot be sure whether it is logically continuing the execution of its parent or spawning a concurrent task.

The simple strategy sketched above effectively assumes that all continuations spawned during an atomic block belong to their parent's task. Therefore, the dynamic scope of the atomic block must include all children, grandchildren, etc until all newly registered continuations have completed. This corresponds closely to the *spawn-contained* policy.

A quick example of when the *spawn-contained* policy is problematic: Consider a function that performs a few quick database queries and then sends a potentially slow network

```
let ex1 = asyncFn( function*( actx, p1, p2 ) {
   ... body ...
   return whatever;
} );

let ex2 = asyncFn( function*( actx, px, py ) {
   ...
   actx.atomic( function*() {
      x = yield ex1( actx, 42, px );
      y = yield fetch( "localhost/foo" );
   } );
   ...
} );

let actx = newActivityContext();
let actId = actx.spawn( ex2 );
```

Figure 7. A sketch of how the JavaScript library implementation of activities is used. The syntactic clutter is fairly high because we are using JavaScript function generators (`function*`) to simulate regular functions in an activities framework.

request based on the results of the queries. A caller may want to enforce atomicity among the database queries, but wrapping a call to this function in atomic would cause the application to become unresponsive until the network request completed.

We conclude that adding an atomic block directly to an event queue system is possible, but problematic. We believe that a much better option is to introduce an explicit notion of tasks/threads, which is what the next section explores.

We briefly note that there are other options for defining an atomic block in an event queue system. For example, the system could assume that the last child registered by a continuation is part of its parent's task and any other children are concurrent tasks. We do not investigate these options further, because they seem too ad hoc.

3.4 Adding Explicit Tasks to Event Queue Systems

In this section we discuss a library implementation of explicit tasks in JavaScript, built on top of async functions. Async functions were standardized in ECMAScript 2017. It is also possible to define async functions as a library, built on top of JavaScript's function generators (`function*`) and Promises. In fact, there are many examples of such libraries added to JavaScript frameworks for convenience.

Our library has two essential functions: spawn and atomic. Spawn takes an async function as a parameter and returns a handle to a concurrent task. In our library we call these concurrent tasks *activities.* Atomic takes a function generator as a parameter and ensures that it runs atomically (i.e., it is not interrupted by an concurrent activities).

Figure 7 gives a tiny example of what our library actually looks like. Notice that each asynchronous function needs an additional context parameter so that it can know what activity it belongs to. If this were implemented natively in a language, this parameter would be unnecessary.

Without atomic and spawn nesting, the implementation of this library would be relatively simple. There would be a single global variable that contained either null or the identity of the activity in atomic mode. Each async function would need to know the identity of the activity it belongs to, and would check the global variable before resuming. If the variable contains the identity of some other activity, the async function would need to wait. Upon exiting an atomic block, any waiting async functions would be released.

Because we want it to be legal to spawn inside of an atomic block, the single global variable indicating *the* activity that is in atomic mode is no longer sufficient. Instead we need a stack of sets of activity identifiers, which is initially empty. When an activity enters atomic mode, it pushes a singleton set on the stack with its own identifier. When an async function attempts to resume it now needs to check whether the stack is empty (i.e. no activity is in atomic mode) or if it is in the set on top of the stack; if neither is the case, then it must wait.

As mentioned previously, there are several reasonable strategies for handling spawns inside atomic. By default, activities spawned while the program is in atomic mode will be put on hold until the program has exited atomic mode. This strategy is the default because it is relatively simple, and as long as the atomic block executes relatively quickly (which is how the feature is supposed to be used), the newly spawned activities will experience relatively little delay. Unlike the multithreading and transactional memory context, where there might be a performance benefit to letting multiple threads simultaneously execute in atomic mode, we are dealing with single-threaded multitasking here.

The only situation in which the *delay-spawn* strategy is problematic is when the code executing the atomic block waits (directly or indirectly) for the newly spawned activity to do something before leaving the atomic block. In this situation, forcing the new activity to wait could cause the program to deadlock. For this reason, the spawn function accepts an optional parameter to override the default nesting strategy. Newly spawned activities can be spawned with the *spawn-contained* strategy. This allows the newly spawned activity to run, and forces the program to wait at the end of the atomic block for any such activities to complete.

We briefly note an efficiency opportunity that atomic blocks create. In addition to being an atomicity hazard, chains of short continuations create event queue overhead. In a version of JavaScript with an atomic block, it would be possible for a just-in-time compiler to fuse continuations run atomically into a single function, reducing this overhead.

The *activities* design we ended up with is essentially cooperative threads plus automatic yield insertion around async functions and atomic blocks. In this paper we do not present a formal semantics for activities. Rather, we refer to existing work on the semantics of cooperative threads [1] and atomic blocks [16]. The semantics of activities are a combination of these two ideas.

3.5 Integration with Legacy Code

Our activities library integrates relatively cleanly with legacy JavaScript code. Any JavaScript Promise can be yielded, just as one would with regular async function implementations. One small awkwardness is that our library cannot prevent continuations registered by legacy code from running whenever the event queue chooses. So if one activity is in an atomic mode, a continuation previously registered by legacy code run in another activity may still run.

3.6 Building an App with Activities

Our experience with the JavaScript Activities library consists primarily of building a simple single-page web app that lets users share information. It makes heavy use of the cryptography API to implement an end-to-end encryption protocol. We wrapped the Promise-returning API functions with simple `asyncFn`s. There is a high density of calls to such functions in the code.

We make a couple observations about this experience. First, compared to an earlier version of this app that used explicit callbacks, the direct style is dramatically easier to read and reason about. Of course, using async functions would also give this benefit.

More than half of the calls we expect to be short-lived. These we can execute in atomic mode to ensure that there are no atomicity violations around them. Using standard async functions, it would not be possible to do that.

4 An Atomic Block for C (Design)

In the previous section we did not discuss the efficiency of the JavaScript implementation of activities in any detail. However, we are interested in whether activities can be implemented in a highly efficient way, so we added them to a dialect of C that we call DblBlind. In this section, we discuss the design of DblBlind. In the next section we discuss the implementation of DblBlind.

Like the JavaScript version, the primary additions are spawn and atomic primitives. We also needed to add a yield primitive, since this does not exist in plain C. The main differences with the JavaScript version are related to the manual memory management of C.

To illustrate our design, we use an example stolen from [14]; a plain C version is in Figure 8. This example performs `N` DNS lookups using the standard `getaddrinfo` procedure.

The DblBlind version performs in Figure 9 performs these lookups concurrently. It has a parameter (`max_conc`) to limit the number of DNS requests that will be sent concurrently to avoid flooding the network with too many requests.

```
int multi_getaddrinfo1(
   const char **nodes, const char **services,
   const struct addrinfo *hints,
   struct addrinfo **ress, int n )
{
   int num_failed = 0;
   for( int i = 0; i < n; ++i ) {
      if( getaddrinfo( nodes[ i ],
         services[ i ], hints, ress[ i ] ) ) {
            ress[ i ] = NULL;
            ++num_failed;
   }  }
   return num_failed;
}
```

Figure 8. This is a plain C version of the DNS fetcher running example. This version is entirely *sequential.*

In DblBlind we add some syntactic convenience to the plain spawn to get the *activate* statement, an example of which is on line 10 of the example. The body of the activate statement runs concurrently (but *not* in parallel) with the statement's continuation. Only one activity can be running at a time and the system can only context switch between activities on a yield.

Each DblBlind program has a scheduler that chooses when a yield should lead to a context switch, and which activity to switch to. The current DblBlind implementation uses a simple FIFO scheduler; clearly this is something that could be looked at for further refinement in terms of performance, fairness guarantees and/or greater application control.

The activate statement takes as a parameter a pointer to application-allocated memory for storing metadata about the new activity. In the example, the memory for storing activity information is allocated locally. Because of this the example procedure must wait for all the activities it spawns to finish. (Returning before they finish could cause the program to access deallocated memory). It would be possible to change the interface so that the caller passes in the backing memory, or to use dynamic allocation and leave the deallocation to the caller. In either of these cases, the procedure could return while the activities were still fetching DNS information, allowing the application to go on with other work.

4.1 Shared Variables

The parentheses after the `activate` keyword are for controlling whether local variables are accessed by-value or by-reference. The default is by-reference, which is how most of the variables are used in this example. This means that all the activities read and write a shared instance of that variable. The exception is `i` which is used as a name of sorts in the `printf` call. Each activity gets its own copy of `i`, whose initial value is whatever the variable's value was at activity creation time. If there were a modification to `i` inside `activate`, each activity would be modifying its own copy.

```
int multi_getaddrinfo3(
   const char **nodes, const char **services,
   const struct addrinfo *hints,
   struct addrinfo **ress, int n, int max_conc )
{
   activity_t acts[ n ];
   semaphore_t governor;
   sem_init( &governor, max_conc );
   for( int i = 0; i < n; ++i ) {
      activate[ &acts[i] ]( i ) {
         sem_decr( &governor );
         printf( "activity %i lookup %s\n",
                 i, nodes[ i ] );
         if( getaddrinfo( nodes[ i ],
            services[ i ], hints, ress[ i ] ) ) {
            ress[ i ] = NULL;
            ++num_failed;
         }
         sem_incr( &governor );
   }  }
   for( int i = 0; i < n; ++i )
      activity_wait( &acts[i] );
   return num_failed;
}
```

Figure 9. This version of the DNS fetcher limits the number of concurrent requests to `max_conc`.

4.2 Yield Insertion

One of the two primary features that distinguish activities from conventional cooperative threads is that there is a translation step that automatically inserts yield statements before and after every procedure call. This rule helps ensure responsiveness by default.

There are two ways a program can still become unresponsive. The first is by wrapping a long-running part of the program in atomic. The second is long-running loops with no calls (or explicit yields) in their body. Both of these should be considered concurrency bugs to be managed with the usual software reliability tools (testing, code review, verification, etc).

In the example code above, the system will automatically insert four yields: one each before and after the two calls. Without further information, it should be assumed that the called functions themselves have more yields in their implementations.

Automatic yield insertion makes activities more like preemptive threads than conventional cooperative threads. However, activities are still more resistant to concurrency bugs than threads by default, because the granularity of scheduling is coarser. Between calls and explicit yield invocations, code is guaranteed to execute atomically. For example, classic data races are impossible with activities.

4.3 Atomic

Any statement, expression or procedure declaration in DblBlind can be "wrapped" with atomic. In the dynamic scope of a atomic block, the current activity cannot be interrupted. In other words, any yields that would have happened are overridden/suppressed by atomic. This is a simple and powerful tool for enforcing atomicity. Most locking in current multithreading practice can be either removed entirely, thanks to the coarser scheduling granularity, or directly replaced by the simpler and safer atomic.

In the example, the compiler will insert yields before and after the calls to `printf` and `getaddrinfo`. If the programmer wanted to suppress some of these yields, they could use `atomic`. For example, it would be reasonable to wrap the `printf` call in `atomic`, since it will complete quickly (relative to `getaddrinfo`), and there is little to be gained by switching to another task at that point. Wrapping the `getaddrinfo` call in atomic would be a performance bug, because that would force the DNS lookups to be executed sequentially.

As with the JavaScript implementation, spawns inside atomics are a challenge. Our solution here is the same: a stack of sets of activities running inside the atomic. There is an additional challenge here with stack management that we address in the next section.

4.4 System Calls

System calls present a challenge, because clearly a language cannot force the operating system to yield in the middle of kernel code execution. To implement activities correctly, the system must ensure that system calls are performed asynchronously. That is, during a system call the system may switch to another activity. If a system call completes while some other activity is running, the calling activity must wait for a yield before it can resume execution. There are more details on this in the implementation section. Legacy code can call blocking system calls, which will cause the whole application to block.

5 An Atomic Block for C (Implementation)

There are a few interesting features of our implementation of activities in the context of DblBlind, which we describe in this section: the allocation of call frames, the implementation the yield primitive, and compiling a version of each procedure for yielding and atomic mode.

5.1 Call Frame Allocation

One of the tricky issues in the implementation of any thread-like abstraction is the allocation of procedure call frames. The first few sections below describe existing strategies for frame allocation in multithreaded code. Then we describe a new approach that we call *hot stacking*.

5.1.1 Contiguous Allocation

The most common strategy for multithreading is to pre-allocate a moderately large area of memory in the heap for each thread's frames. Individual frames are allocated contiguously within this area. This strategy is fast and simple, but is memory inefficient.

5.1.2 Individual Heap Allocation

At the other extreme of the time/memory efficiency spectrum, systems can individually allocate each frame in the heap, linking them together with pointers. This avoids the memory concerns associated with contiguous allocation. However, heap allocation of frames comes at a time cost for calls and returns.

Simple implementations of heap allocation can be more than an order of magnitude slower than contiguous allocation. More sophisticated implementations can be substantially more efficient (e.g. [23]). However, even the most efficient implementations that we are aware of are still slower than contiguous allocation for call-heavy code.

5.1.3 Split/Segmented Stacks

Some languages have experimented with a hybrid strategy called *split*, *segmented*, or *linked* stacks. The idea is that stack space is allocated in small segments or chunks. The common case call/return execution looks like traditional contiguous allocation. When a thread reaches the end of its segment it allocates a new one and links them together with pointers. This idea is appealing; as fairly recent examples, early implementations of Rust and Go both used it. Unfortunately it has unpleasant *uncommon* case behavior called *stack thrashing* or the *hot split* problem. When there are frequent calls/returns right at the boundary of a segment, the overhead can be quite high. The implementers of Rust [3] and Go (search "contiguous stacks in go") both abandoned split stacks in later versions.

5.1.4 Growable Contiguous Stacks

Since version 1.3, the Go language has used a novel stack allocation strategy, where each thread's stack memory is initially small and is resized on-demand. This is only possible because the Go language implementation knows about all pointers to stack memory and can update them if necessary (i.e., if it is necessary to relocate the whole stack). Clearly this would not work for C/C++ and close cousins. This strategy still has an unpredictable slowness problem when the stack has to be copied.

5.1.5 Hot Stacking

The call frame allocation strategy introduced in this paper is a hybrid of contiguous and individual heap allocation. The key observation is that the speed of individual allocation is only important for short-lived calls. For long-lived calls, the overhead of the call and return operations can be amortized over the long running time of the call. So the main idea of hot stacking is that long-lived frames are individually allocated and (most) short-lived frames are contiguously allocated in memory that is shared by all activities.

For this strategy to work correctly, the frames allocated in the shared area must all be deallocated before context switching to another task. For regular threads it is not clear how this deallocation could be enforced easily. The use of atomic blocks in programs makes it practical to implement hot stacking, because the implementation can use atomic blocks as an indicator of when it should use contiguous allocation.

In the microbenchmarking section below we provide evidence that this strategy captures the benefits of heap allocation (per-activity memory overhead is small) and contiguous allocation (fast calls and returns when it matters). It is certainly possible for short-lived calls to happen in yielding mode, which means that the frames will be individually allocated. This should be considered a performance bug, but its worst-case overhead is not terrible and it should be a relatively easy pattern to identify with a profiler.

Nested spawns present a challenge for hot stacking, because multiple activities can execute concurrently in an atomic block. The implementation accommodates this by allocating additional regions for stack allocation on demand. Note that this is not an issue for activities that use the default strategy of blocking until the end of the atomic block. We expect that in practice multiple concurrent activities in an atomic block would be quite rare. As mentioned earlier, since activities do not run in parallel, there is little to be gained.

Other than the nesting issue, we are not aware of any uncommon case performance problems like the hot split problem with segmented stacks.

The name *hot stacking* is a reference to a practice used in some organizations called *hot racking* or *hot desking*. Some limited resource (e.g. a bunk or desk) is used in shifts by multiple people. In our case, the resource is the memory area for contiguous frame allocation, and it is shared in shifts by multiple activities. The word *hot* is also a reference to the fact that this top of stack area should remain hot in the memory hierarchy/cache sense as long as any code is running, because all activities use the same block of memory for the frames for their short-lived calls.

5.2 Yield Implementation

Automatic yield insertion makes frequent yielding common. So we need to ensure that yields are as efficient as possible. In particular, we implemented three ideas:

- Zero overhead yields in atomic mode
- Infrequent context switching
- Very fast non-switching yield

5.2.1 Yield in Atomic Mode

There are important differences at the implementation level between how code executes in yielding versus atomic mode, not least among them the call frame allocation strategy. To implement this dual mode concept in a reasonably simple and efficient way, the current DblBlind implementation generates two versions of each procedure: one for each mode. The yielding mode implementation includes inserted yields and assumes its own frame was individually allocated. The atomic mode implementation does not include yields; for most intents and purposes it is a simple translation to plain C.

This implementation strategy could lead to a substantial increase in code size (more than 2×). We believe this problem can be managed, based on the assumption that most procedures in real-world programs would be called only in either yielding or atomic mode, not both. This means that we could either rely on a smart linker to perform a dead code analysis and delete the unused versions, or use runtime optimizations to generate the appropriate version on-demand.

5.2.2 Function Pointers

One challenge with a dual implementation strategy is how to handle function pointers. When the address of a function is taken it is not easy to know which mode the function will eventually be called in. The current implementation generates a small piece of code for every function that can be called in either yielding or atomic mode. It accepts an additional implicit mode parameter, which controls which actual version gets called. This makes indirect calls somewhat more expensive than in plain C. However, indirect calls are already expensive enough that there is an extensive body of research on how to convert them to direct calls (e.g. [9]), so adding a modest amount of overhead to indirect calls should not have a large performance impact on most applications.

5.2.3 Actual Yields

The most important factor in the implementation of the yield primitive is that most yield invocations should not result in context switching, even if other activities are ready to run. In well designed activity code, the time between yield invocations should be in the range of microseconds to milliseconds. Context switching has the moderate direct cost of manipulating a handful of data structures in the runtime system, and the potentially higher indirect cost of cache thrashing. Therefore in normal operation yields should lead to context switches at a relatively low frequency, perhaps every few milliseconds.

The speed of the yield primitive itself is somewhat important. As described in the previous section, code that is compiled in atomic mode does not have yields at all, so yield performance is not an issue for the most performance critical loops. However, we expect that moderately frequent yielding (perhaps as frequently as many per microsecond) would be common in real-world code. Therefore, the performance of the yield primitive does matter to some degree.

The simplest implementation of yield would check a counter or clock of some sort. Reasonably efficient implementations of this strategy would certainly be non-portable (e.g. using processor-specific counter registers) and probably still be somewhat expensive. Instead the current DblBlind implementation uses periodic system timers/alarms that deliver a signal to the program. The handler for these signals atomically modifies a global variable. The yield primitive atomically reads this global variable; as long as it has not changed the program continues execution immediately. Therefore, in the common case the cost of a yield is only an atomic read and a (highly predictable) branch, plus a fast call and return to get to the yield code itself. (Of course the yield code can be inlined, but that is *not* obviously a good idea because branch predictors work better if all yields share a single branch instruction.)

5.3 System Calls

An important part of making activities work properly is asynchronous system calls. The current DblBlind implementation uses libuv for this. At startup, the runtime system spawns a thread that runs a libuv event loop. When an activity wants to make a system call, it sends an asynchronous message to the event loop thread, which makes the appropriate libuv API call. When the event completes, the activity is notified and the system puts it back in the runnable queue.

This architecture works, but it adds a non-trivial amount of overhead, especially for fast system calls. We speculate that integrating the event processing with application code in the same thread could remove some performance overhead. The libuv maintainers have discussed this as a "pull", versus the current "push" architecture, which they may implement in a future version.

5.4 Translation

We use a modified version of Cil [18] to translate DblBlind to plain C with calls to our runtime system, which is written in C.

6 An Atomic Block for C (Measurement)

To investigate the performance of activities, we wrote microbenchmarks with the DblBlind implementation. All tests were run on a machine with the specs listed in Table 1. Depending on the benchmark, we compare the DblBlind implementation against plain C, C compiled with gcc's split stack implementation, Go or C plus the default Linux threads implementation. All speed-related tests were compiled at optimization level `-O2`. For speed-related tests we ran the benchmark 5 times and report the fastest result.

6.1 Memory Overhead

Like coroutines (e.g. in their modern C++ incarnation), activities have very low memory overhead; approximately whatever is needed to store a dozen pointers. It is interesting to compare this with goroutines. The Go implementation (as of version 1.3) uses a sophisticated stack growing process

OS	Ubuntu Linux 14.04
Kernel	3.13.0-68-generic
Processor	4 GHz Intel Core i7
Memory	8 GB 1600 MHz DDR3
Compiler	gcc 4.8.4
Go	1.5.3

Table 1. Specs of the test system

to avoid pre-allocating a large amount of memory for call frames. However, the minimum size is still relatively large (in the neighborhood of a kilobyte), more than a factor of 6 greater than activities in DblBlind.

6.2 Task Spawn and Switching Speed

Unsurprisingly, activities are dramatically faster than system threads when it comes to spawning and switching between tasks. The speed difference is over an order of magnitude, as measured by microbenchmarks. The primary difference here is that threads must make system calls to do these things, whereas with activities they can be done entirely in user mode.

In these benchmarks, Go was slightly faster than DblBlind. We expect this is because the Go team has put a lot of engineering effort into optimizing these concurrency primitives. It seems likely that DblBlind could be optimized to be at least as fast.

The speed of these primitives have a big impact on the practical minimum granularity of tasks. In both DblBlind and Go that overhead is closer to that of a method call than the equivalent thread operations. That means that concurrency can be used in interesting ways in those languages that they cannot with threads.

6.3 Just Calling

Our next microbenchmark measures the overhead of hot stacking. This test is a simple recursive function that calls itself twice and performs a very small computation at the "leaves"; the code is in Figure 10.

	user	sys	wall
Plain C	0.77	0.00	0.77
Individual Allocation	42.4	0.04	42.6

These numbers are in nanoseconds per call/return. This difference is nearly two orders of magnitude, which is painful for DblBlind. However, there are three caveats to keep in mind. First, the DblBlind implementation is not highly optimized; the allocator used in this test was the default system malloc. Surely a more tuned implementation would close the gap to some extent. Second, this is a microbenchmark; no application spends all of its time just calling and returning,

```
int f( int d, int x ) {
    if( d > 0 )
        return f( d - 1, f( d - 1, x ) );
    else
        return small_computation( x );
```

Figure 10. The microbenchmark for measuring call frame allocation overhead.

so these numbers are a fairly high upper bound on the real application performance impact of calling overhead.

Most importantly, all the calls in the first version of this test were performed in yielding mode. In atomic mode, calls and returns in DblBlind are just as efficient as plain C. In well-tuned DblBlind code, most of the leaf and near-leaf calls should be performed in atomic mode. To simulate this effect, we modified the benchmark to the version in Figure 11. The parameter N controls how deep in the call tree the benchmark uses yielding mode, before switching to atomic.

	user	sys	wall
Plain C	0.77	0.00	0.77
gcc split stacks[2]	1.00	0.15	1.15
Go	2.80	0.20	3.00
Hot Stacking, N = 4	11.00	0.01	11.00
Hot Stacking, N = 8	1.54	0.00	1.55

This shows that with a modest fraction of the (static) calls performed in atomic mode, the calling overhead gets down to less than a factor of 2. Combined with the observations above about tuning the language implementation, we believe this overhead would be barely noticeable for most applications. Also it is worth noting that this is the price to be paid for the extremely low memory overhead of activities.

Hot stacking with N = 8 outperforms Go on this benchmark by a modest margin. It is interesting that the gap between plain C and Go is as large as it is, given the Go implementation team's famous focus on low-level performance details.

gcc's split stack implementation outperforms hot stacking by a small margin on this benchmark. However, as noted previously split stacks have bad performance in the pathological case where calls are made at high frequency right at the edge of a segment.

6.4 Yielding

As described in section 5.2, yield frequency can be managed fairly easily and the yield primitive is designed to have a very fast common path. Nevertheless, it is important to minimize yielding in hot inner loops.

[2]This benchmark does not create stacks deep enough to exceed the default size of a single segment of gcc's split stack. So we modified the benchmark for the split stack case to make a linear chain of calls instead of a binary tree.

```
int f( int d, int x ) {
    if( d > N )
        return f( d - 1, f( d - 1, x ) );
    else if( d > 0 )
        atomic {
            return f( d - 1, f( d - 1, x ) );
        }
    else
        return small_computation( x );
```

Figure 11. The call/return microbenchmark modified to make some calls in atomic mode.

To quantify yielding overhead, we benchmarked a simple but problematic function from the C standard library: strcmp. strcmp is tricky for a few reasons: First, the input strings can be arbitrarily long, so never yielding could cause unresponsiveness. Second, the body of the loop is extremely simple; good implementations are just a few assembly instructions. This means that yielding every iteration causes significant performance overhead. Third, the continued execution of the loop depends on the data read in each iteration, so simple loop tiling/blocking tricks do not work. The best-performing implementation we have found so far appears in Figure 12.

```
int strcmp( const char *s1, const char *s2 ) {
    while( *s1 && !( *s1 - *s2 ) ) {
        ++s1; ++s2;
    }
    return *s1 - *s2;
}
```

```
int strcmp( const char *s1, const char *s2 ) {
   while( *s1 && !( *s1 - *s2 ) ) {
      ++s1; ++s2;
      if( !( s1 & ( ( 1 << YIELD_FREQ ) - 1 ) )
         yield;
   }
   return *s1 - *s2;
}
```

Figure 12. Two implementations of strcmp. Top is a plain C version. Bottom is a DblBlind implementation aimed at minimizing yield overhead.

In the code YIELD_FREQ controls how frequently the loop yields. The effect is that once every 2^{YF} iterations there is a yield.

Plain C	430
DblBlind, YIELD_FREQ = 4	1070
DblBlind, YIELD_FREQ = 8	519
DblBlind, YIELD_FREQ = 12	446

These numbers are in microseconds per strcmp of two identical strings of 1MB length. These data show that yielding is not free, but with a little careful tuning the yield overhead can be brought quite low.

When the inputs to strcmp are known to be short, the caller can wrap the call in atomic. The atomic version should perform identically to the plain C implementation.

6.5 Microbenchmark Summary

Like all well implemented coroutines, the DblBlind implementation has extremely low memory and time overhead for basic concurrency primitives. Hot stacking and frequent yielding are potential performance issues, but our tests indicate that at these overheads can be managed.

Though we do not have experience with writing large programs in DblBlind yet, we expect that relatively little code will require the kinds of contortions shown in the strcmp example. Moreover, almost all such code will be in tight inner loops in library code. This is exactly where writing somewhat weird code for performance reasons is acceptable.

7 Related Work

Our activities are clearly closely related to many flavors of cooperative threads. Most commonly, cooperative threads rely on programs explicitly invoking yield to permit task switching. We believe that automatically inserting yields before and after calls by default creates a more predictable environment for programmers. Boudol proposes a slightly different approach to automatic yield insertion in cooperative threads [5].

The other major difference is the atomic block, which this paper shows is not a trivial addition to a language. We are not aware of any other research on the combination of cooperative threads and atomic blocks.

One weakness that activities share with most multitasking frameworks is that activities cannot easily be run in parallel. The authors are not optimistic about running cooperative threads in parallel, but there are researchers pursuing this idea [6, 7, 20].

8 Summary and Discussion

In this paper we presented evidence that web applications seem to be highly exposed to atomicity violations. We note that adding an atomic block would help programmers avoid these problems, and discussed how that addition can be achieved. Perhaps the most important slogan to take from this paper is that the atomic versus async decision should be made by callers, not implementers of APIs. We showed that our design can be implemented efficiently in a dialect of C. We believe that our design combines the benefits of existing cooperative concurrency frameworks in a new and interesting way.

References

[1] Martin Abadi and Gordon Plotkin. 2009. A Model of Cooperative Threads. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09)*. ACM, New York, NY, USA, 29–40.

[2] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. 2002. Cooperative Task Management Without Manual Stack Management. In *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference (ATEC '02)*. USENIX Association, Berkeley, CA, USA, 289–302.

[3] Brian Anderson. 2013. Abandoning segmented stacks in Rust. rust-dev mailing list. (November 2013).

[4] Hans-J. Boehm. 2011. How to Miscompile Programs with "Benign" Data Races. In *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism (HotPar'11)*. USENIX Association, Berkeley, CA, USA, 3–3.

[5] Gérard Boudol. 2007. Fair Cooperative Multithreading: Typing Termination in a Higher-order Concurrent Imperative Language. In *Proceedings of the 18th International Conference on Concurrency Theory (CONCUR'07)*. Springer-Verlag, Berlin, Heidelberg, 272–286.

[6] Frédéric Boussinot. 2006. FairThreads: Mixing Cooperative and Preemptive Threads in C: Research Articles. *Concurr. Comput. : Pract. Exper.* 18, 5 (April 2006), 445–469.

[7] FrÃĺdÃĺric Dabrowski and FrÃĺdÃĺric Boussinot. 2006. Cooperative Threads and Preemptive Computations. In *in &quot;Proceedings of TVâĂŹ06, Multithreading in Hardware and Software: Formal Approaches to Design and Verification*.

[8] James Davis, Arun Thekumparampil, and Dongyoon Lee. 2017. Node.Fz: Fuzzing the Server-Side Event-Driven Architecture. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. ACM, New York, NY, USA, 145–160.

[9] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP '95)*. Springer-Verlag, London, UK, UK, 77–101.

[10] Dan Grossman. 2007. The Transactional Memory / Garbage Collection Analogy. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA '07)*. ACM, New York, NY, USA, 695–706.

[11] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. 2005. Composable Memory Transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '05)*. ACM, New York, NY, USA, 48–60.

[12] Chun-Hung Hsiao, Jie Yu, Satish Narayanasamy, Ziyun Kong, Cristiano L. Pereira, Gilles A. Pokam, Peter M. Chen, and Jason Flinn. 2014. Race Detection for Event-driven Mobile Applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 326–336.

[13] Gabriel Kerneis. 2013. *Continuation-Passing C: Program Transformations for Compiling Concurrency in an Imperative Language*. Ph.D. Dissertation. Université Paris Diderot - Paris 7.

[14] Maxwell Krohn, Eddie Kohler, and M. Frans Kaashoek. 2007. Events can make sense. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference (ATC'07)*. USENIX Association, Berkeley, CA, USA, Article 7, 14 pages.

[15] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems (ASPLOS XIII)*. ACM, New York, NY, USA, 329–339.

[16] Katherine F. Moore and Dan Grossman. 2008. High-level Small-step Operational Semantics for Transactions. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. ACM, New York, NY, USA, 51–62.

[17] Erdal Mutlu, Serdar Tasiran, and Benjamin Livshits. 2015. Detecting JavaScript Races That Matter. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 381–392.

[18] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. 2002. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of the 11th International Conference on Compiler Construction (CC '02)*. Springer-Verlag, London, UK, UK, 213–228.

[19] John Ousterhout. 1996. Why Threads Are A Bad Idea (for most purposes). In *USENIX Technical Conference*.

[20] Melissa E. O'Neill and Christopher A. Stone. 2015. Making Impractical Implementations Practical: Observationally Cooperative Multithreading Using HLE. In *Proceedings of the 10th ACM SIGPLAN Workshop on Transactional Computing*.

[21] Victor Pankratius and Ali-Reza Adl-Tabatabai. 2014. Software Engineering with Transactional Memory Versus Locks in Practice. *Theor. Comp. Sys.* 55, 3 (Oct. 2014), 555–590.

[22] Boris Petrov, Martin Vechev, Manu Sridharan, and Julian Dolby. 2012. Race Detection for Web Applications. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 251–262.

[23] Zhong Shao and Andrew W. Appel. 2000. Efficient and Safe-for-space Closure Conversion. *ACM Trans. Program. Lang. Syst.* 22, 1 (Jan. 2000), 129–161.

[24] Abhayendra Singh, Satish Narayanasamy, Daniel Marino, Todd Millstein, and Madanlal Musuvathi. 2012. End-to-end sequential consistency. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*. IEEE Computer Society, Washington, DC, USA, 524–535.

[25] Windows 8 Team. 2012. Keeping apps fast and fluid with asynchrony in the Windows Runtime. (2012). https://blogs.msdn.microsoft.com/windowsappdev/2012/03/20/keeping-apps-fast-and-fluid-with-asynchrony-in-the-windows-runtime/

[26] Rob von Behren, Jeremy Condit, and Eric Brewer. 2003. Why Events Are a Bad Idea (for High-concurrency Servers). In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9 (HOTOS'03)*. USENIX Association, Berkeley, CA, USA, 4–4.