TigerQuoll: Parallel Event-based JavaScript

Daniele Bonetta Walter Binder Cesare Pautasso
Faculty of Informatics, University of Lugano – USI
Lugano, Switzerland
{name.surname}@usi.ch

Abstract

JavaScript, the most popular language on the Web, is rapidly moving to the server-side, becoming even more pervasive. Still, JavaScript lacks support for shared memory parallelism, making it challenging for developers to exploit multicores present in both servers and clients. In this paper we present TigerQuoll, a novel API and runtime for parallel programming in JavaScript. TigerQuoll features an event-based API and a parallel runtime allowing applications to exploit a mutable shared memory space. The programming model of TigerQuoll features automatic consistency and concurrency management, such that developers do not have to deal with shared-data synchronization. TigerQuoll supports an innovative transaction model that allows for eventual consistency to speed up high-contention workloads. Experiments show that TigerQuoll applications scale well, allowing one to implement common parallelism patterns in JavaScript.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel Programming

General Terms Languages, Performance

Keywords JavaScript, Event-based Programming, Parallelism, Eventual Transactions

1. Introduction

JavaScript was initially designed as a single-threaded programming language for executing small client-side scripts within Web browsers. To support increasingly complex Web applications such as Facebook or GMail, recent research has focused on increasing the performance of JavaScript engines. As a result, modern engines such as Google's V8 and Mozilla's SpiderMonkey feature advanced optimizations such as just-in-time compilation and inline caching.

Despite of these advancements, one fundamental issue with JavaScript-based applications remains unsolved, that is, the poor support for parallelism. As the Web continues evolving towards a mature application-hosting platform, the lack of parallelism in JavaScript would eventually correspond to a serious technological barrier. Furthermore, the wide-spread deployment of JavaScript-based frameworks for server-side development has increased the pervasiveness of the language, making JavaScript a central language for the entire Web development stack. As a result, frameworks such as Microsoft Azure and Node.JS [2] offer a convenient solution to develop Cloud-hosted Web applications using JavaScript as the sole language for both the client and the server.

The limited support for parallelism in JavaScript constrains the class of applications which can be executed in the browser, thus preventing Web developers from fully exploiting parallel infrastructures such as multicore machines commonly available both on the client and on the server. Current actor-based solutions such as WebWorkers [1] limit the way developers can exploit parallelism, as they force them to reason in terms of parallel processes, message passing, and share-nothing parallelism. Although well suited for master-worker parallelism, the Actor model of concurrency appears to be too limited for algorithms and applications that can benefit from more complex parallel patterns [24]. Moreover, share-nothing parallelism represents a strong limitation for JavaScript developers, as it clashes with the programming style of JavaScript, based on single-threaded asynchronous programming and shared memory for accessing the Document Object Model (DOM) of a Web page.

We advocate a different solution, claiming that the parallel support for JavaScript should come *without* violating the programming style of JavaScript, that is, asynchronous programming [16]. To substantiate our claim, in this paper we present TigerQuoll, a fully JavaScript-compatible execution engine which allows scripts to be executed in parallel using a flexible event-based API.

TigerQuoll makes the following contributions to advancing the state-of-the-art in the field of parallel programming for the Web:

(1) *Programming Model*. The TigerQuoll API extends single-threaded event-based JavaScript programming with support for parallelism. The resulting programming model allows the developer to exploit parallelism by means of *parallel event handlers*. Executing multiple event handlers in parallel allows the JavaScript developer to adopt a familiar programming style based on asynchronous callback invocation.

(2) *Eventual Transactions*. Parallel execution is protected using a software transactional memory (STM). In addition, TigerQuoll introduces a novel form of transactions called *eventual transactions* which ensure that the parallel execution of event handlers accessing mutable global shared state through a controlled mechanism always succeed. Eventual transactions help improve scalability for certain workloads.

(3) *Parallel JavaScript Engine*. JavaScript applications are executed by the TigerQuoll parallel execution engine. Its run-

PPoPP'13, February 23–27, 2013, Shenzhen, China.
Copyright © 2013 ACM 978-1-4503-1922/13/02. . . $10.00

time executes multiple event handler functions in parallel, ensuring a consistent view of the shared memory.

To the best of our knowledge, TigerQuoll is the first parallel JavaScript engine supporting mutable shared state. Furthermore, we are the first introducing eventual transactions for software transactional memory.

This paper is structured as follows. Section 2 introduces the main design goals of TigerQuoll. Section 3 introduces the event-based API, while Section 4 describes how events can be used to build high-level parallel constructs. Section 5 provides a detailed description of the TigerQuoll memory model. Section 6 provides details about the architecture of the TigerQuoll engine. Section 7 presents an initial performance evaluation of the engine, whose results are discussed in Section 8. Section 9 presents related work, and Section 10 concludes.

2. Parallelizing JavaScript

Introducing parallelism in a language that has not been designed for it is a challenging task. Our solution preserves compatibility with existing JavaScript applications allowing developers to write parallel code adopting a programming style familiar with the one of single-threaded sequential asynchronous programming. In more detail, TigerQuoll has been designed to meet the following requirements:

(1) **Backward compatibility**. Existing applications designed to run on single-threaded JavaScript engines shall be fully compatible with TigerQuoll. To guarantee strict compatibility the sequential semantics shall be enforced by the runtime, which needs to execute existing legacy code strictly sequentially (i.e., in the same thread).

(2) **Forward compatibility**. Applications developed for parallel execution on the TigerQuoll parallel engine shall also be executable by existing single-threaded JavaScript engines (at the cost of losing the performance benefits coming from parallelism).

(3) **Implicit parallel entities**. The complexity of the programming model of JavaScript shall not be increased by adding explicit parallel entities such as threads or processes. Therefore, no coordination mechanisms (e.g., locks, barriers, or message queues) need to be exposed to the developer.

(4) **Event-based concurrency**. The only concurrency control mechanism supported by the programming model shall be the one already present in single-threaded JavaScript, that is, event-based concurrency. Asynchronous event-based programming is therefore extended (through a novel API) to express parallel computations.

(5) **Shared memory space**. To preserve the existing single-threaded JavaScript programming model, a mutable global state shall be supported by the programming model. However, the developer shall not be responsible for ensuring thread-safety and liveness. Instead, the programming model shall automatically guarantee atomicity, consistency, and isolation. Such properties shall be guaranteed transparently, as the developer shall not be in charge of adding explicit atomic blocks.

The main difference of the TigerQuoll programming model over existing solutions for parallel JavaScript is that it is conveniently compatible with the programming style JavaScript developers are already familiar with, not requiring developers to reason in terms of either share-nothing parallelism (as in WebWorkers [1]) or immutable shared state (as in Intel's River Trail [19] or Parallel Closures [23]). Indeed, the TigerQuoll programming model allows developers to write parallel JavaScript applications adopting a parallelism model with mutable shared-memory and asynchronous event-based concurrency, allowing applications to transparently benefit from the presence of multiple cores.

3. The Programming Model of TigerQuoll

The programming model of TigerQuoll is based on event production and consumption. Every JavaScript object in TigerQuoll can produce and consume events. The resulting event-based model is a set of low-level APIs which can be directly used for parallel programming, or as basic building blocks for developing convenient higher-level parallel primitives. This is discussed in the next section, where common high-level parallel constructs are illustrated.

3.1 Core Event API

The two main primitives for event-based programming in the TigerQuoll API are `on` and `emit`. Using the `on` primitive, any object can specify an event handler callback function to be associated with any event. Events are specified using strings (called event labels). The `emit` primitive is used to notify the runtime that an event has happened. It is a non-blocking primitive which immediately returns. The `on` primitive is memory-less, meaning that it is not triggered by any event emitted before the callback was registered. Events can also be associated with parameters, like in this example callback registration:

```
obj.on('connection', function(fd) {
    // open the 'fd' descriptor and handle the request
});
```

which can be triggered by the following event emission:

```
// somewhere in the connection handling code...
obj.emit('connection', 33)
```

The two pieces of code correspond to how incoming connections are handled by a JavaScript server (e.g., the Node.JS socket server). An object responsible for accepting the incoming request (`obj`) is registered to listen for an incoming connection using `on`. When the runtime receives a new connection on a listening socket the `'connection'` event is emitted, and the socket descriptor on which the connection has been accepted is passed to the event handler. The callback associated with the `'connection'` event is then invoked with the actual value of the `fd` variable as argument (i.e., `33` in the example). Multiple calls to `emit` correspond to multiple invocations of the callback function handling the event.

To ensure that a callback is executed only once, and is then immediately unregistered, the API provides the `once()` primitive:

```
1  obj.once('fireAndForget', function() {
2      // do something...
3  })
4  obj.emit('fireAndForget')
5  obj.emit('fireAnotherEvent')
6  obj.emit('fireAndForget')
```

The `once()` primitive registers the callback to be executed only once. Hence, the second time `emit` is called with `'fireAndForget'` as argument (line 6) it will not cause the callback to be re-executed again. The `emit` primitive is asynchronous and does not offer any ordering guarantees. As a consequence, the two events (`'fireAndForget'` and `'fireAnotherEvent'`) in the example may be executed in any order.

The dual primitive is `never()`:

```
    obj.never('ev')
```

which is used to remove the callback handler for a specific event identifier.

3.2 Event-based Parallelism

The `on/emit` API is similar to the one found in server-side event-based frameworks *à la* Node.JS as well as many client-side libraries for JavaScript (e.g., JQuery or Async.JS). Differently from such frameworks, event handlers (i.e., callbacks) are executed by the runtime *in parallel*. Consider the following example:

```
    var stats = { tot : 0, found : [] }
    obj.on('data', function(n) {
        if(isPrime(n)) {
            stats.tot++
            stats.found.push(n)
        }
    })

    var inArr = new Array(1, 2, 3, 4, ...)
    for(var i=0; i<inArr.length; i++)
        obj.emit('data', inArr[i])
```

The example corresponds to a simple primes checker. The checker operates on an array of numbers (`inArr`) and asynchronously evaluates for every element whether the given number is prime by calling the `isPrime(n)` function. When a prime number is found, it is saved.

When executed by an event-driven JavaScript framework such as Node.JS, the code above will process all the elements of the array sequentially, because JavaScript is single-threaded by design, and so are the existing JavaScript execution engines. Conversely, the code in the example is executed in parallel by the TigerQuoll runtime, using all the cores in the system. This is made possible by the TigerQuoll engine, which executes multiple event handlers (i.e., the callback for the `'data'` event) in parallel. Furthermore, event handlers are guaranteed to run atomically with respect to other handlers. Therefore, the counter in the example is guaranteed to have a consistent value at the end of the computation. As the whole event emitter is guaranteed to execute atomically, also the array keeping track of all the prime numbers found during the parallel computation will eventually be consistent.

3.3 Event Handler Synchronization

Asynchronous, event-based programming is the default abstraction for dealing with I/O in JavaScript. Thanks to the event-based programming model, a single-threaded application can trigger multiple downloads sequentially and wait for multiple transfers to happen in parallel, thus overlapping the download of multiple remote resources thanks to mechanisms such as `select` or `epoll` (implemented either by the OS Kernel or by the browser). With single-threaded JavaScript, the asynchronous interaction with multiple resources (for instance, multiple ongoing downloads) requires that the single JavaScript thread never blocks.

The ability to execute multiple event handlers asynchronously and in parallel introduces a more stronger need for event synchronization. In such a context it makes sense to provide the developer with a mechanism to automatically synchronize specific event execution with other events completion. Such mechanism could then be exploited to schedule the execution of certain event handlers upon the completion of other events. This is made possible in TigerQuoll through the `waitall` primitive, which can be used to register *event guards*. Consider the following example:

```
    // register an event handler for 'compute'
    obj.on('compute', function() {
        // do something...
    })

    // trigger the 'compute' event N times
    for(var i=0; i<N; i++)
        obj.emit('compute')

    // register an event guard for 'compute'
    obj.waitall('compute', function() {
        // all the 'compute' events have been emitted
        // and all the corresponding event handlers
        // have returned
    })
```

In the code above an event is emitted several times, causing the engine to execute the corresponding callbacks in parallel. The `waitall` primitive is used to register another event handler which is executed *after* all the `compute` events have been processed (i.e., when all the parallel callbacks have been executed and have returned). One important consideration about the `waitall` primitive is that it allows to wait for any number of events emitted by the currently running event handler, thus allowing for event composition.

As for the `on` primitive, `waitall` is memory-less: if the event it is required to wait for has been emitted and executed before a call to `waitall`, the event emission cannot be tracked, and the callback associated with the event guard will not be executed. In particular, event guards are guaranteed to track the event emission of all the events emitted by the currently running event handler. As a consequence, event guards cannot guarantee to track events emitted by multiple event handlers concurrently.

4. High-level Parallelism

The TigerQuoll Core Event API offers a relatively low abstraction level. This low-level substrate can be conveniently used to build high-level mechanisms and constructs that Web developers can use to exploit parallelism in their everyday life. In this section we introduce some of the possible high-level constructs (i.e., asynch, finish, map/reduce) that can be built upon the Core Event API of TigerQuoll.

4.1 Futures and Task Parallelism

Many programming languages feature a notion of future tasks [17]. These can be introduced in JavaScript with the `spawn` method :

```
    // Execute the 'fun' function with argument 'arg'
    // in parallel, and get a future as the result
    var future = spawn(fun, arg)
    // register a callback to get the result
    future.get(function(result) {
        // 'result' contains the return value
        // for 'fun(args)'
    })
```

In the example, the `fun` function is executed asynchronously in parallel, and the result is fetched calling `get`, which will call the given callback with the function return value once its execution has completed.

Another task parallelism construct many languages feature is lightweight tasks for immediate parallel execution. Such tasks are usually spawned through an asynchronous function call, and the runtime provides a way for synchronizing on task completion. For example, in X10 [7] such primitives are called `async` and `finish`. Having such primitives in JavaScript would simplify the way parallel computations can be developed. They can be used as in the following ex-

```
var spawn = function(f, arg) {
    // Create an object to track the state of the async call
    var _A = {}
    // Register the function for async execution
    _A.on('go!', function() {
        // Call the function
        _A.result = f(arg)
    })
    // Register the future callback
    .on('get!', function(cb) {
        cb(_A.result)
    })
    // Fire the event to call the function
    .emit('go!')
    // Return the future object
    return {
        get: function(callback) {
            // Register the callback to be called once the
            // result has been computed, or return it
            if (_A.result)
                return callback(_A.result)
            _A.emit('get!', callback)
    } }
}
```

```
var async = function(fun, a, b) {
    // schedule 'fun' for asynchronous execution
    // using the 'finish' global object as the
    // event emitter
    finish.emit('go', fun, a, b)
}
var finish = function(f) {
    // register a callback for executing functions
    // in parallel using the 'finish' (this) object
    finish.on('go', function(fun, a, b) {
        fun(a, b)
    })
    // execute the function, which will call
    // async multiple times
    f()
    // Return the ondone function
    return {
        ondone : function(callback) {
            // register a wait guard for the
            // 'go' event
            finish.waitall('go', function() {
                callback()
            })
    }}
```

Figure 1. Simplified implementation of `spawn`, `async`, and `finish` using the TigerQuoll API

ample, which applies a function `fun` to all the elements of an array (`data`) in parallel:

```
    var data = [...] // the array to be processed

    var fun = function(index) {
        data[index] = someProcessing(data[index])
    }
    finish(function() {
        for(var i in data)
            async(fun, i)
    }).ondone(function() {
        // since all the parallel tasks have completed,
        // now the data array contains the result
    })
```

As shown in Figure 1, the JavaScript version of `spawn` makes use of the TigerQuoll API by means of an auxiliary object to keep track of the state of the parallel tasks' execution (called `_A`). The `spawn` primitive makes use of the auxiliary object to register an event to trigger the asynchronous invocation of the function (`'go!'`), and returns an object representing the future. When the `get` method of the future is called, a callback is registered which will eventually be called with the result of the asynchronous function execution.

The implementation of the `async` and `finish` primitives uses another shared object (i.e., the `finish` function itself) to associate the emission of all the parallel tasks being executed by `async` with a same event emitter object. In this way, the `waitall` primitive can be used to postpone the `ondone` callback execution until all the tasks have been executed. Support for nested `finish` invocations could be achieved by using a stack of shared objects instead of the `finish` function itself (not shown in the Figure).

4.2 Structured Parallelism

Another example of high-level parallelism constructs which can be built with the event-based API is represented by some very common structured parallelism patterns (a.k.a. Algorithmic Skeletons [9, 24]). Such patterns simplify the way common parallel patterns can be programmed by introducing the notion of structured parallel patterns. Very popular examples of such patterns are Map/Reduce [11] and Scatter/Gather [8].

Building such patterns on top of the TigerQuoll API is straightforward. Consider the following implementation of a MapReduce-like computation for processing all the elements of a given array in parallel:

```
Array.prototype.mapred = function(map, reduce) {
    var obj = {}
    obj.waitall('go', function() {
        var result = reduce(this)
        obj.emit('donePar', result)
    })
    // register an event for parallel processing
    obj.on('go', function(fun, x) {
       fun(x)
    })
    // emit an event for
    for(var v=0; v<this.length; v++)
        obj.emit('go', map, this[v])

    return {
        ondone : function(fincb) {
            obj.on('donePar', fincb)
    }}
}
```

As in common MapReduce computations, the example above applies a given map function to all the elements of a fixed size array, and eventually calls a reduction function returning the result. The event-based parallel MapReduce can be easily invoked in the following way:

```
// declare an array
var a = new Array(1,2,...)

// execute the mapreduce using two arbitrary functions
a.mapred(map, reduce)
.ondone(function(result) {
    // the variable 'result' contains the
    // result of the computation
})
```

5. Shared Memory Model

Unlike existing approaches to parallelize JavaScript which are either based on message-passing (i.e., share-nothing) or on shared but immutable data, TigerQuoll parallel event handlers can access and also modify shared data. Global uncontrolled heap access requires the developer to deal with synchronization mechanisms (e.g., locks or barriers [21]) which often result in complex and error-prone programming techniques. Access to heap objects can be mediated by using atomic blocks, usually implemented through Software Transactional Memory (STM) or Lock Inference, but such approaches would again require the developer to explicitly identify which portion of the code is to be protected with an atomic block, and data races or race conditions would still be possible for unprotected code.

TigerQuoll adopts a radically simplified solution, with the goal of keeping the programming model as close as possible to the one of single-threaded JavaScript. The goal is not to completely hide parallelism from the developer. Instead, the developer must be aware of the fact that event handlers are executed in parallel, but all the data consistency issues are automatically handled by the runtime. To this end, TigerQuoll executes automatically and transparently every parallel event handler shielded behind an STM-mediated atomic block. This has the advantage of giving the developer the impression that every handler has a consistent view of the global shared state as it would have when being executed by a single-threaded execution engine. As opposed to introducing an explicit atomic block construct, we chose to associate atomic semantics with the event handlers, which are already the basic unit of parallelization in TigerQuoll.

5.1 Eventual Transactions

Despite the benefit of keeping the programming model of TigerQuoll as close as possible to the one of single-threaded JavaScript, STMs offer poor performance for certain workload types [18]. Apart from the fixed cost of managing a transaction's metadata, one of the problems with STM's performance lies in the fact that for high-contention workloads transactions are forced to abort frequently, reducing efficiency and wasting computing resources. Although the debate on the maturity of STMs is still ongoing [6, 14], there is some general consensus that STM-based solutions have scalability limits. For certain workloads, TigerQuoll offers the possibility to relax the default transactional isolation allowing to modify shared data using a different, more efficient transactional mechanism. This novel mechanism, called *eventual transactions*, corresponds to a class of transactions which never fail to commit, and always succeed in updating the global state after their execution.

The key consideration for speeding up transactional workloads in TigerQuoll is that some algorithms do not require shared data to be consistent *during* the execution of the transaction, but only *after* commit has happened. For instance, this is the common case when parallel tasks are performing computations on partial results on a shared data structure. This consideration can be used to implement transactions which never fail to commit, as data inconsistencies on the same shared value during the transaction do not mean any incorrect semantics, as long as the transactional system is given a way to solve the conflict when the transaction completes.

Consider the case of a JavaScript word counter, that is, a function to count the frequency of the words within a text. The function consumes a stream of data by tokenizing each input chunk in order to count the number of occurrences of each word. Such functions are very common in server-side applications for buildings systems such as Web crawlers or to generate trending topics for services such as Twitter.

As every event handler is executed atomically and with isolation, there is no way to let event handlers communicate

```
// shared object to collect the statistics
    var stats = {
        total : 0           // total number of words
        word : new Array()  // occurrences of each word
    }
    finish(function() {
// open and scan the input file
        open(url, function(chunk) {
// spawn a parallel task for each chunk
            async(function() {
                var tokens = tokenize(chunk)
// for each token update the statistics
                for(var i=0; i<tokens.length; i++) {
                    var word = tokens[i]
                    if( ! stats.word[word]) {
                        stats.total++
                        stats.word[word] = 0
                    }
                    stats.word[word]++
} }) }) })
.ondone(function() {
// once all chunks have completed return the statistics
    console.log(stats.total)
})
```

Figure 2. Parallel word count function in TigerQuoll

partially computed values. In other words, the result of the execution of any event handler (including eventual transactions) become visible to other handlers only when the handler terminates (and commits its result). This execution semantics can be combined with the `waitall` primitive, which guarantees that a given event handler is executed once a set of other parallel events have been successfully executed. For instance, this is the case for the MapReduce example in Section 4.2 where the `result` variable is always guaranteed to be consistent.

The code in the example (Figure 2) reads from a data stream by opening a URL (which could correspond to a remote Web resource or a file stored locally). The URL is opened through the `open` function, which invokes its callback as soon as a data chunk is available. As the callback uses `async`, the chunks will be processed in parallel. For each chunk, the parallel callbacks will tokenize the string and will update the global shared object containing the statistics (`stats`).

One important consideration about the example is that the `stats` object (more precisely, its fields) is not required to be consistent during the execution of the parallel event handlers, as what really matters for the word counter is to produce a consistent result, i.e., to return a consistent (and correct) value of `stats`. In other words, the fields of the `stats` object need to be consistent only eventually, i.e., after all parallel callbacks have completed.

This property can be explicitly specified by the developer by marking certain fields of an object as *eventual*. In the example this can be done using the `markEventual()` primitive provided by the TigerQuoll API:

```
    // The 'total' field is eventual
    stats.markEventual('total', sum)
    // All the elements of the array are eventual
    stats.word.markEventual('*', sum)
```

Marking a field as eventual tells the runtime not to fail in case it detects an inconsistent value of the field at commit-time. This implies that the runtime needs to know how to deal with inconsistent values. More precisely, it needs to know how to *accumulate* the partial value of an eventual field when committing its value to the global shared state. This

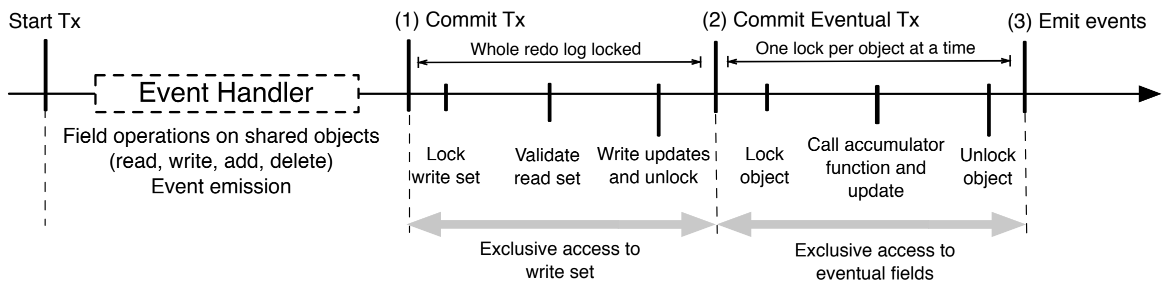
Start Tx
(1) Commit Tx
(2) Commit Eventual Tx
(3) Emit events
Whole redo log locked
One lock per object at a time
Event Handler
Field operations on shared objects (read, write, add, delete) Event emission
Lock write set
Validate read set
Write updates and unlock
Lock object
Call accumulator function and update
Unlock object
Exclusive access to write set
Exclusive access to eventual fields

Figure 3. Overview of the commit phase of the TigerQuoll runtime.

is done by passing another argument to `markEventual()`, called the *accumulator function*. In the example above the accumulator function only has to add the partially computed value of field to the global value. This can be specified as follows:

```
    function sum(global, initial, final) {
        return global+(final-initial)
    }
```

Accumulator functions receive three arguments as input, and return the new value to be stored in the global object. The three input arguments are (1) the value of the global field at the moment the accumulator function is called, (2) the initial value of the field before the parallel event handler was called, and (3) the final value of the field at the end of the execution of the event handler.

In the example, eventual fields are used to count, and therefore the accumulator function corresponds to a simple sum function adding the partial result as evaluated by the event handler to the global value. This is natural for fields with numeric values. Since the TigerQuoll programming model does not constrain the type of fields which can be marked eventual, other kinds of accumulator functions may be specified by the developer. The only constraint for accumulator functions is that they must be side-effect free and they should not access any shared object different from the ones they are passed as arguments.

6. Architecture

In this section we describe the principal design decisions behind the TigerQuoll parallel engine architecture. The engine is a prototype JavaScript engine derived from Mozilla SpiderMonkey [4]. The most relevant changes include modifications to the native implementation of the JavaScript `Object` class to associate every object with the needed transactional metadata. Moreover, the `Object` class has also been modified to enable objects with support for event emission, consumption and synchronization through the `on`, `emit` and `waitall` primitives. Every object also holds a private table of event handlers registration. The TigerQuoll runtime runs multiple threads accessing the same memory heap mediated by an STM layer.

The TigerQuoll runtime features an event-based execution system based on a simple master-worker pattern for processing multiple events in parallel. The system is composed of multiple threads holding a pointer to a shared double-ended queue containing references to event objects. Events are consumed nondeterministically by the threads, therefore allowing for parallelism.

To guarantee compatibility with existing JavaScript applications, the TigerQuoll engine behaves as a standard non-parallel engine as long as the TigerQuoll runtime is not explicitly activated. Once active, JavaScript applications can run parallel tasks only through the event-based API (e.g., through `on`/`emit`) or through higher-level constructs already present in the TigerQuoll runtime (for instance `async`). This ensures that existing applications that use neither the event-based API nor the TigerQuoll runtime will always run sequentially.

To guarantee TigerQuoll applications to run in non-parallel JavaScript engines, the TigerQuoll event-based API can be implemented in pure JavaScript by extending the prototype of the `Object` object. All the event emissions and consumptions can be therefore made asynchronous just by using a global queue shared by all the objects within the same application. Indeed, this is the same execution model as in Node.JS, which handles event consumption and emission in the JavaScript space.

6.1 Transactional Support

The TigerQuoll runtime features a STM with global versioning clock, lazy version management, and commit-time locking [18]. Conflicts are detected both at commit-time and during transactions' execution. The versioning management algorithms implemented by the TigerQuoll STM are similar to the ones of the TL2 STM [12], with the main difference that TigerQuoll features per-field versioning (as opposed to per-object versioning) with per-object locking.

The commit-time locking with redo logs fits well with the event-based design of the TigerQuoll engine, as state changes are made persistent only after the transaction has completed. This also guarantees that read-only transactions (i.e., read-only event handlers) can operate in parallel without any lock acquisition. The per-field version management prevents transactions operating on different fields of the same object from failing because of versioning conflicts.

Transactions are committed in three distinct steps, namely (1) non-eventual fields commit, (2) eventual fields commit, and (3) deferred event emission. The whole process is summarized in Figure 3, while the three phases are described in detail in the following sections.

6.1.1 Non-eventual Fields Commit

Read and write operations on non-eventual fields are mediated through a redo log and two sets tracking field reads and field writes, respectively. The redo log keeps a thread-local copy of global objects as locally computed by each parallel event handler. The redo log is managed with a lazy strategy,

```
// tx struct which holds all the logs
Transaction tx;
// start the transaction
do {
    // event handler execution: the redo_log
    // is created and modified as well as the
    // read set, the write set, the eventual log
    // and the events log
} while( ! Tx_Commit_redo(&tx) );
// redo_log committed: eventual fields can be processed
foreach(JS_OBJECT obj in eventual_log)
    // Get a reference to the global shared object
    JSObject *global = tx->eventual[obj]->global;
    // --- (1) Lock the object --- //
    Lock(&global)
    // scan all its eventual fields
    foreach(EV_FIELD f in global->ev_fields) {
        // Get the accumulator for this field
        jsval accFun = global->acc_fun[f];
        // prepare the arguments for the accumulator
        jsval *argv;
        argv[0] = JS_ReadField(global, f)
        argv[1] = tx->eventual[iter]->delta
        argv[2] = tx->eventual[iter]->snapshot
        // --- (2) Execute the accumulator function --- //
        jsval result = JS_Execute(&accFun, argv)
        // store back the result
        JS_WriteField(global, f, result)
    }
    // Release the lock
    UnLock(&global)
}
```

Figure 4. Eventual transactions commit-phase pseudocode.

meaning that global object fields are copied to the redo log only when accessed for the first time.

Conflicts are resolved using per-field versioning. Each time a transaction is executed, it reads the most recent value of a shared global clock (a 64-bit integer), obtaining a Read Version number (RV) which is used to validate both the read and write sets. Both during the transaction and at commit time, the RV of the transaction is compared against the version of the fields as stored in the corresponding global state field. When a field with a more recent version is found, the transaction is aborted, as the current value of the field has been changed by another transaction in the meanwhile. Modifications to the object structure (i.e., field additions and deletions) are treated as special cases of transactional write operations. At commit-time, the STM (1) acquires all the per-object locks of all the fields in the write set; (2) validates the the read set against the RV; (3) in case of no conflicts (either with lock acquisition or read version management), commits and updates the shared objects writing the new values. (4) Eventually it releases all the locks.

6.1.2 Eventual Fields Commit

Eventual transactions have the property of never failing. They always commit by accumulating data in shared fields marked as eventual through a user-given function called accumulator. This is made possible by operating on a thread-local copy of the value of local fields which is managed by the STM runtime through a separate log, called eventual log.

During an event handler execution, all the accesses of eventual fields are mediated by the eventual log, and no actual access to the global object's fields is performed. Similarly to regular transactions, the eventual log keeps track of all the operations performed on eventual fields. Conversely, the log is not used for validating the consistency of the field neither during the transaction's execution nor at commit-time. As any operation happens on the local copy of every eventual field, any operation on such data structures happens with snapshot isolation from the event handler perspective.

At the end of the event handler execution, the eventual log holds the locally computed value of the eventual field (called the *delta*) plus the initial value of the field, called *snapshot*. Together with the current global value, these two values are then passed to the accumulator function as arguments.

The commit phase for eventual fields (Figure 4) is performed in two steps:

(1) *Locking of objects with eventual fields*. The eventual log is scanned and for every eventual field, a lock is acquired on the corresponding objects. Locks on different objects are not acquired all-at-once, as in the commit phase of the standard STM. Instead, it is safe to acquire only one lock at a time, as eventual fields do not need to guarantee atomicity. The eventual log is sorted so as to acquire only one lock per object, thus allowing to commit all the eventual fields belonging to the same object at the same time.

(2) *Accumulator function execution*. Once the lock is acquired, the accumulator function corresponding to the eventual field is called passing as arguments the current value of the global object, the delta value, and the snapshot value. The value returned by the function is written back to the global value. The lock on the object can be released once all its eventual fields have been updated.

6.1.3 Deferred Event Emission and I/O

In a system with everything mediated by an STM it becomes crucial to properly treat all the deterministic blocking operations which cannot be re-executed in case of commit failure. This is usually the case with I/O operations such as blocking file read/write operations as well as standard output operations. Fortunately, the case for JavaScript is simpler, as all the I/O operations already happen asynchronously. Therefore, all I/O operations in JavaScript naturally play well with the transactional system, as they are already happening outside of event handlers.

The asynchronous nature of JavaScript is of great advantage for including an STM support in the runtime, as the only operation which has to be treated as a special case is event emission. In fact, a failed transaction which emits an event immediately before committing will be re-executed by the runtime and will likely re-emit the same event. This could easily lead to inconsistencies. To avoid this, the TigerQuoll event-based programming model does not assume that events are actually emitted immediately after `emit` is called. The emission of events is asynchronous by design (there is no guarantee that a thread will be ready for executing the corresponding callback at the moment the event is emitted), and therefore event emission can be safely postponed. Hence, during the execution of an event handler, emitted events are buffered in another thread-local log, called event log. Only after the transaction has successfully committed the event log is processed and deferred events are safely emitted for parallel processing.

7. Evaluation

To evaluate the performance of the TigerQuoll engine we have performed two distinct classes of experiments. First, we evaluated the performance of the engine to assess the performance overhead of the TigerQuoll engine compared to the most popular existing share-nothing parallelism solution (i.e., WebWorkers). Second, we evaluated the engine in the context of shared-memory applications, to observe

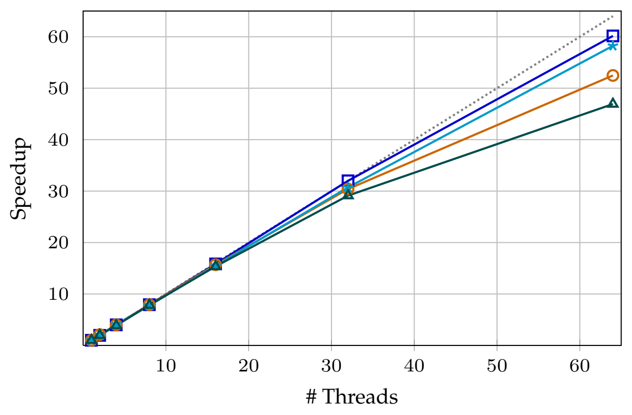

Figure 5. Share-nothing scalability. The graph shows the scalability of TigerQuoll for the Primes (—□—) and Mandelbrot (—○—) benchmarks compared with the equivalent WebWorkers Primes (—★—) and Mandelbrot (—△—) benchmarks.

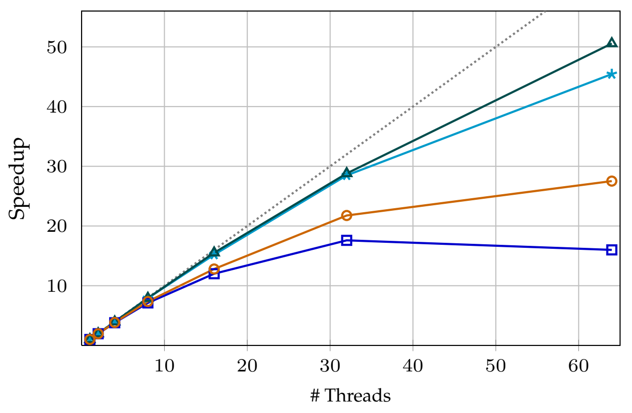

Figure 6. Shared Memory Scalability: Primes checker with shared counter. The graph presents TigerQuoll scalability in case of high contention with regular transactions (—□—) and eventual transactions (—○—), as well as low contention with regular transactions (—★—) and eventual transactions (—△—).

the performance of eventual transactions for high- and low-contention workloads. All the experiments have been executed on a 32 cores AMD-Bulldozer machine with support for 64 parallel (hyper-threaded) threads. The machine has a total of four CPUs connected to four NUMA nodes. All the results presented in this Section are average values computed over five independent runs of each experiment. The standard deviation is negligible.

7.1 Share-nothing Scalability

To assess the performance of the engine in comparison with existing share-nothing solutions, we have parallelized some existing JavaScript benchmarks using TigerQuoll and WebWorkers, and we have measured how the execution time decreases when adding more parallel threads to the engine. Results are depicted in Figure 5.

The algorithms selected for the evaluation are the parallel calculation of a 1024x1024 Mandelbrot Set and a parallel primes checker scanning 10^6 integers looking for prime numbers. As clearly depicted in the figure, the TigerQuoll engine has performance comparable to the ones of WebWorkers, and scales linearly with almost ideal scalability up to the number of physical cores (32) in the system (lines —○— and —□—). This shows that the event-emission, routing and consumption mechanism of the engine as well as its transactional support do not prevent share-nothing applications from scaling. This also means that share-nothing algorithms can be parallelized using the `on`/`emit` primitives of TigerQuoll in addition to using explicit parallel entities such as WebWorkers and message passing coordination.

7.2 Shared Memory Scalability

Of the algorithms presented in the previous section one can be easily modified to become a shared memory algorithm. In fact, the primes number calculator can be modified to use a shared object to keep track of the prime numbers it has found. In more detail, the algorithm implements a traditional divide-and-conquer scheme by partitioning the space of integer numbers to check, and by assigning each parallel event handler a partition of the space for processing. Each handler thus receives an interval to scan and searches for primes in its local partition only, eventually updating the global object with the number of prime numbers it has found once done with its job.

As in many data-parallel computations with shared state, the size of the task assigned to parallel workers is a crucial performance parameter. In fact, tasks with a too small size can easily degrade performance because of data contention, while tasks with an outsized dimension tend to degrade scalability (especially when the tasks are not homogeneous in terms of processing time). To measure the impact of task size in the case of the primes checker we have performed an additional experiment measuring the performance of the algorithm using the shared counter with different task size. Results depicted in Figure 6 describe how with a small task size (10^2 numbers per event) the STM is forced to abort very often (see line —□—, where the STM aborts are on average more than 10% of the total started transactions), while with a bigger task size (10^3 numbers per event) the STM is still able to scale (line —★—, with a failure rate of less than 5%).

Fortunately, this is the classic case in which the partial result of the computation (i.e., updating the counter) is not needed by the parallel task. Therefore, we could mark the field of the shared object counting the number of primes as eventual, and specify that we need an accumulator function which just sums the delta to the global value of the counter. The performance of the TigerQuoll runtime using the eventual counter are depicted in the same figure (lines —○— and —△—). Using eventual fields significantly out-performs the version using regular transactions, since the presence of the eventual field saves the transaction from aborting and restarting.

The impact of contention on shared-memory algorithms can in some cases dramatically affect the performance of an STM system. This is the case for the experiment depicted in Figure 7, where a data intensive workload with high and low contention has been evaluated. The experiment corresponds to the word-counter example presented in Figure 2. In the experiment, the parallel word-counter is given a text file of 4MB to parse. The file contains a variety of equally distributed words which corresponds to the creation of thousands of items on the shared array. The experiment has been executed with two chunk sizes to vary the contention on the shared array. As expected, using regular transactions will not scale, since the abort rate of the transactions is very high as soon as multiple threads are handling events in parallel (lines —□— and —★—). With almost certain probability two parallel event

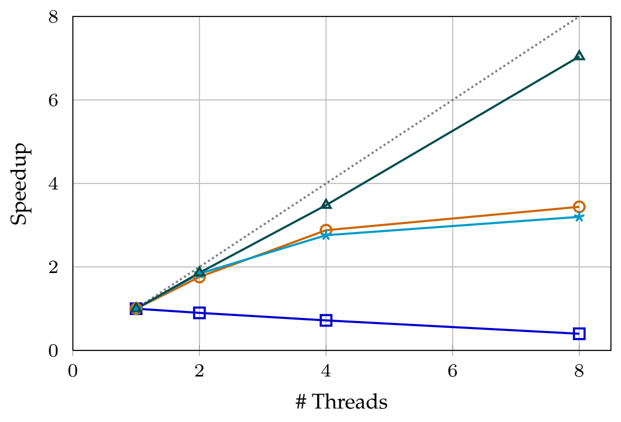

Figure 7. Shared Memory Scalability: word counter. The graph presents TigerQuoll scalability in case of high contention with regular transactions (—□—) and eventual transactions (—○—), as well as low contention with regular transactions (—★—) and eventual transactions (—△—).

handlers will try to create or update the same element of the shared array, and all but one transaction will have to be aborted and re-executed. By marking all the fields of the array as eventual, this effect is mitigated and the system scales when adding more parallel threads (lines —○— and —△—).

8. Discussion

The results from the initial evaluation of our VM confirm that shared-memory parallelism can be safely brought to JavaScript with an event-based model of parallelism. Our approach succeeds in keeping the programming model as close as possible to existing single-threaded JavaScript and in maintaining compatibility with existing applications. These two goals represent key factors for introducing parallelism in the domain of JavaScript, as the language has a very wide and heterogeneous user community spanning from client-side Web application developers to server-side developers.

Our event-based approach to parallelism is not competing with emerging parallel solutions for JavaScript (e.g., WebWorkers and RiverTrail), but instead should be considered as complementary. In particular, our approach does not prevent developers from using such solutions, but offers them an alternative for developing applications using a parallel programming model other than message passing or immutable shared data, still without introducing the complexity of programming with locks, and giving good speedup (ideal speedup, indeed, when falling back to the case of share-nothing computations). Other solutions such as WebWorkers can still be used in all the circumstances in which a master-worker parallelism pattern is more natural to be expressed. However, we think that shared-memory event-based parallelism represents a suitable solution for parallelizing several problems peculiar to the domain of JavaScript, like for instance the real-time parallel processing of data streams from sources such as RESTful services [5] or WebSockets. In terms of programming model, it would be interesting to explore how to combine multiple models, having for instance WebWorkers which internally execute multiple events in parallel.

The design decisions underlying eventual transactions have similar rationale. Our goal is to propose a novel, convenient programming-model abstraction, which provides transactions which never abort, at the cost of ensuring only eventual consistency. As implemented in TigerQuoll, the model is complementary to existing STM-based approaches and allows us to speed-up transactional event handlers when strong consistency is not needed. The metadata overhead for managing eventual fields is equivalent to the overhead of standard transactions, as all the operations on each field have to be tracked. Similarly, the consistency management overhead of eventual fields is similar to the one of transactions, as eventual fields are made consistent by acquiring a lock. Performance benefits come from the fact that the commit phase never forces a transaction to re-execute, therefore eventual transaction's performance is comparable to the one of regular transactions which always commit.

The TigerQuoll engine has been developed and tested with the workloads presented in Section 7, and is being currently improved. As of now, the overhead introduced by the event-based system is negligible. In other words, a single-threaded event-based execution performs almost identically to the equivalent code executed on a unmodified engine. The overhead introduced by STM metadata is proportional to the size of the transaction's logs.

9. Related Work

Several efforts are being directed towards overcoming current limitations of JavaScript concerning its support for parallelism. As part of the HTML5 standardization, WebWorkers [1] offer a simple message-passing abstraction for implementing the Actor model in JavaScript. This technology has been used in [15] to develop an event-based programming model for parallelizing JavaScript applications, which hides WebWorkers from the developer's perspective but still assumes a share-nothing memory model. On the server-side, Cluster [2] is a process-based parallelism library for Node.JS implementing a programming model similar to Actor-based concurrency. On the client-side, River Trail [19] offers an API for developing data-parallel computations by means of automatic compilation of JavaScript code to OpenCL [3] so that parts of the computation is offloaded on the GPU. Only applications using static immutable data structures are supported. Another interesting solution for parallelizing JavaScript is Parallel Closures [23]. The approach attempts to bring fork/join-like parallelism to JavaScript through constructs similar to `async` and `finish`. Differently from TigerQuoll, Parallel Closures operate on immutable (read-only) shared data, and support only fork/join parallel patterns [22].

All of these approaches show the importance and the need for offering simple parallelism support in JavaScript applications. The TigerQuoll engine is based on an alternative, complementary approach, which allows developers to exploit a protected, shared, mutable, global address space. Furthermore, TigerQuoll can express share-nothing parallelism (as with WebWorkers) without messages and explicit parallel entities, and allows to implement any arbitrary data-parallel algorithm over mutable shared-memory. Developers used to asynchronous programming can directly use the TigerQuoll Core Event API, while others can take advantage of the higher level constructs that are built on top of it.

Out of the realm of JavaScript, other languages feature a task-parallelism programming model. Some notable examples are X10 [7], F# [28] and [20]. Each language provides different ways of controlling and interacting with parallel tasks, but none of them is based on explicit event emission and consumption.

Software Transactional Memory is the most prominent way of enabling atomic blocks-based programming. STMs are used in several languages and libraries (e.g., the Akka framework in Scala [26] or [10, 13]). A notable attempt to bring STM-based solutions to JavaScript has been presented in [25] where the authors describe a system for automatic parallelization of existing legacy single-threaded applications through STM-based speculation. The approach differs from TigerQuoll in that our goal is not to parallelize existing applications, but instead to offer a programming model for building new parallel applications.

To the best of our knowledge, we are the first to introduce eventual transactions. Out of the realm of transactions associative and commutative operations are used in some parallel programming languages and models to speed-up certain computations. Piccolo [27], for instance, allows to specify lightweight associative functions to be automatically applied by the runtime when modifying the parallel elements in a shared table.

10. Conclusion

In this paper we introduced TigerQuoll, an execution engine and an event-based API for parallel programming in JavaScript. TigerQuoll features transactional support for programming using a mutable shared memory model through the execution of multiple asynchronous parallel event handlers. TigerQuoll brings an alternative to share-nothing parallelism to JavaScript with a programming model already familiar to JavaScript developers, ensuring compatibility. TigerQuoll features an STM-based protection of shared memory accesses, and features an innovative class of transaction, called eventual transactions, to speed-up high-contention workloads. Our initial evaluation confirms that the engine is able to fully exploit the underlying parallelism in modern multicore machines while preserving the programming model of single-threaded JavaScript.

Acknowledgments

The research presented in this paper has been supported by the Swiss National Science Foundation (SINERGIA grant nr. CRSI22_127386) and by the European Commission (Seventh Framework Programme grant 287746).

References

[1] HTML5 Web Worker API specification draft. URL `http://dev.w3.org/html5/workers/`.

[2] Node.JS: evented programming for networked services in JS. URL `http://www.nodejs.org`.

[3] OpenCL, the standard for parallel programming of GPU. URL `http://developer.amd.com`.

[4] Mozilla SpiderMonkey JS engine. URL `www.mozilla.org/js/spidermonkey/`.

[5] D. Bonetta, P. Achille, C. Pautasso, and W. Binder. S: a scripting language for high-performance RESTful Web services. In *Proc. of PPoPP*, pages 97–106, 2012.

[6] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software Transactional Memory: Why Is It Only a Research Toy? *Queue*, 6(5):46–58, Sept. 2008.

[7] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proc. of OOPSLA*, OOPSLA '05, pages 519–538, 2005.

[8] M. Cole. *Algorithmic skeletons: structured management of parallel computation*. MIT Press, Cambridge, MA, USA, 1991.

[9] M. Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Comput.*, 30(3):389–406, Mar. 2004.

[10] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 336–346, 2006.

[11] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[12] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proceedings of the 20th international conference on Distributed Computing*, DISC'06, pages 194–208, 2006.

[13] A. Dragojević, R. Guerraoui, and M. Kapalka. Stretching transactional memory. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 155–165, 2009.

[14] A. Dragojević, P. Felber, V. Gramoli, and R. Guerraoui. Why STM can be more than a research toy. *Commun. ACM*, 54(4): 70–77, Apr. 2011.

[15] A. Erbad, N. C. Hutchinson, and C. Krasic. DOHA: scalable real-time Web applications through adaptive concurrent execution. In *Proceedings of the 21st international conference on World Wide Web*, WWW '12, pages 161–170, 2012.

[16] D. Flanagan. *JavaScript. The Definitive Guide*. O'Reilly, 5th rev. edition, 2006.

[17] R. H. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, Oct. 1985.

[18] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory*. Morgan and Claypool, 2nd edition, 2010.

[19] S. Herhut, R. L. Hudson, T. Shpeisman, and J. Sreeram. Parallel programming for the Web. In *Proceedings of the 4th USENIX conference on Hot Topics in Parallelism*, HotPar'12, pages 1–6, 2012.

[20] M. Isard and A. Birrell. Automatic mutual exclusion. In *Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, HOTOS'07, pages 3:1–3:6, 2007.

[21] D. Lea. *Concurrent Programming in Java. Second Edition: Design Principles and Patterns*. Addison-Wesley, Boston, MA, USA, 2nd edition, 1999.

[22] D. Lea. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, JAVA '00, pages 36–43, 2000.

[23] N. D. Matsakis. Parallel closures: a new twist on an old idea. In *Proceedings of the 4th USENIX conference on Hot Topics in Parallelism*, HotPar'12, pages 1–6, 2012.

[24] M. D. McCool. Structured parallel programming with deterministic patterns. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, HotPar'10, pages 1–6, 2010.

[25] M. Mehrara, P.-C. Hsu, M. Samadi, and S. A. Mahlke. Dynamic parallelization of JavaScript applications using an ultra-lightweight speculation mechanism. In *17th International Conference on High-Performance Computer Architecture*, HPCA-17, pages 87–98, 2011.

[26] M. Odersky, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, M. Zenger, and et al. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[27] R. Power and J. Li. Piccolo: building fast, distributed programs with partitioned tables. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–14, 2010.

[28] D. Syme, T. Petricek, and D. Lomov. The F# asynchronous programming model. In *Proceedings of the 13th international conference on Practical aspects of declarative languages*, PADL'11, pages 175–189, 2011.