The Parallel Event Loop Model and Runtime

A parallel programming model and runtime system for safe event-based parallel programming

Doctoral Dissertation submitted to the

Faculty of Informatics of the *Università della Svizzera Italiana*

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

presented by

Daniele Bonetta

under the supervision of

Prof. Dr. Cesare Pautasso

September 2014

Dissertation Committee

Prof. Dr. Walter Binder	Università della Svizzera Italiana, Switzerland
Prof. Dr. Mehdi Jazayeri	Università della Svizzera Italiana, Switzerland
Prof. Dr. Nate Nystrom	Università della Svizzera Italiana, Switzerland
Prof. Dr. Pascal Felber	Université de Neuchâtel, Switzerland
Dr. Nicholas D. Matsakis	Mozilla Research, USA

Dissertation accepted on 10 September 2014

Prof. Dr. Cesare Pautasso
Research Advisor
Università della Svizzera Italiana, Switzerland

Prof. Dr. Igor Pivkin, Prof. Dr. Stefan Wolf
PhD Program Directors

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Daniele Bonetta
Lugano, 10 September 2014

Nihil difficile volenti

Fam. Rusca

Abstract

Recent trends in programming models for server-side development have shown an increasing popularity of event-based single-threaded programming models based on the combination of dynamic languages such as JavaScript and event-based runtime systems for asynchronous I/O management such as Node.JS. Reasons for the success of such models are the simplicity of the single-threaded event-based programming model as well as the growing popularity of the Cloud as a deployment platform for Web applications.

Unfortunately, the popularity of single-threaded models comes at the price of performance and scalability, as single-threaded event-based models present limitations when parallel processing is needed, and traditional approaches to concurrency such as threads and locks don't play well with event-based systems.

This Dissertation proposes a programming model and a runtime system to overcome such limitations by enabling single-threaded event-based applications with support for speculative parallel execution. The model, called *Parallel event loop*, has the goal of bringing parallel execution to the domain of single-threaded event-based programming without relaxing the main characteristics of the single-threaded model, and therefore providing developers with the impression of a safe, single-threaded, runtime. Rather than supporting only pure single-threaded programming, however, the parallel event loop can also be used to derive safe, high-level, parallel programming models characterized by a strong compatibility with single-threaded runtimes.

We describe three distinct implementations of speculative runtimes enabling the parallel execution of event-based applications. The first implementation we describe is a pessimistic runtime system based on locks to implement speculative parallelization. The second and the third implementations are based on two distinct optimistic runtimes using software transactional memory. Each of the implementations supports the parallelization of applications written using an asynchronous single-threaded programming style, and each of them enables applications to benefit from parallel execution.

Acknowledgements

It's hard to tell how far from our expectations life can be, when no direction home is set, and nothing but you can drive to the *nächster halt*. I was lucky, though, I was not alone during my journey.

First, I would like to thank Cesare for having spent a good part of his good time teaching me how good research can be done in a good way. Redundancy of good words is not unintentional, as working with him was good, a good honor, and so will be in the future. This Dissertation would not exist without his good advice.

Good, indeed, was also to work at USI, where I had the pleasure to meet many brilliant people. Walter is one of them, and I am very thankful I had the pleasure to work and learn from him. Besides USI, I am also honored I had the pleasure and the privilege to meet amongst the smartest guys on earth during my time at Mozilla Research and Oracle Labs. Both internships have been amongst the most formative events of my life, and I am honored I had such a great opportunity.

I am honored I had in my committee Mehdi, Pascal, Niko, and Nate. I am very thankful for their strong feedback and support, and for having helped me connecting the dots from Pisa to Lugano, across Switzerland and the US, via California.

I am lucky I also had the pleasure and the fun of working with many smart and happy PhD students and researchers at USI. Names I should mention are Achille, Danilo, Domenico, Saeed, Aibek, Philippe, Stephen, and many many more.

Last but not least, a big thank you goes to my family and my old friends. Good they have been there to remind that life is not only about parallelism and concurrency. There still is no answer to my questions. It was good, however, to search.

D

Contents

Figures

Tables

Part I

Prologue

Chapter 1

Introduction

The Web is rapidly becoming a mature application-hosting platform. Technologies such as WebRTC [28] and WebSockets [25] promise to further sustain this trend by allowing Web clients to produce, consume, and elaborate data. As a consequence, computing-intensive applications are required to run both on the client and on the server, increasing the need for parallel processing in Web applications and services. The trend is further confirmed (and influenced) by the abundance of multicore machines in all the tiers of the Web stack, from clients, to servers, through middlewares. Examples of services and Web applications for which the need for parallelism is becoming urgent and challenging are RESTful Web services [78], and in general all services nowadays populating the Web (including HTTP-based streaming services [77] and WebSocket services [25] for real-time Web applications [136]). Other examples that promise to become relevant in the near future are all applications connected to an abundant production and consumptions of data, e.g., pervasive Web applications [27], and so-called Big Data [142] applications. As a consequence of this need, new parallel programming models for Web-based applications start to emerge. Notable examples can be found in particular in the domain of client-side programming, where technologies such as RiverTrail [94, 93] promise to reduce the gap between client-side developers and parallel execution via a clean data-parallel programming model. Beyond data parallelism, however, current solutions for parallel programming for Web applications lack support for structured parallelism [126], and developers willing to implement efficient high-performance applications must instead rely on their own experience, and implement ad-hoc solutions based on share-nothing actor-based [31] models.

In this dissertation we describe how some of the current limitations can be solved by increasing the level of abstraction of parallel programming models for the Web by relying on simple asynchronous event-based programming and a parallel execution engine.

1.1 Event-based programming, parallelism, and the Web

Since the introduction of the first parallel machines, parallel programming has always been considered a complex field in which only expert programmers could play. Only recently [96] the abundance of parallel hardware infrastructures in the form of multi-core machines has revealed the urgent need for novel abstractions and models to deal with parallel programming [113]. In the same historical period, another phenomenon has influenced the development of programming models and frameworks: the Web. As a matter of fact, the impressive growth of Web technologies has revealed some limitations of existing programming models, feeding the demand for novel tools and abstractions to develop dynamic, rich, and high-performance Web applications.

In this context, the need for novel ways to develop high-performance software to be quickly exposed on the Web has been particularly clear in the context of Web applications, where significant efforts have been spent on the research and development of novel solutions (examples are Google's Dart [22], and ASM.JS [2], to cite a few). Parallelism, of course, is another way to achieve high-performance (orthogonal to single-threaded performance optimization). The need for novel ways to develop high-performance Web applications is particularly noticeable on the server side, where parallel execution would be beneficial in two distinct but very related aspects, namely, to increase the service throughput (1), and to optimize service latency (2). For what concerns throughput optimization, server-side applications have to face the *natural* parallelism deriving from the presence of multiple clients performing requests concurrently. In this context, the presence of many concurrent clients can be exploited to process multiple independent requests in parallel. Concerning latency optimization, parallel processing can be exploited to reduce the execution time of *a single* client request, for instance by parallelizing some CPU-bound operations that are required in order to produce the client response. In this Dissertation we will address both classes of parallelism, focusing on server-side systems relying on event-based concurrency and the event loop [33].

Event-based concurrency is a concurrency control model in which concurrent interactions are modeled as asynchronous events processed by a single thread of execution. Events are usually generated by external entities (e.g., an I/O runtime system) which usually provide a callback associated with the event (to be executed once the event has been processed). The single-threaded nature makes of event-based concurrency a very attractive model, as developers do not have to deal with data races and synchronization, and can make strong assumptions about shared state consistency. However, the simplicity of the model comes at the price of scalability for workloads that do require CPU-intensive operations, given the single-threaded nature of the runtime.

Of particular success in the domain of server side development is Node.JS [15, 146], a JavaScript networking framework for the development of event-based Web services. Node.JS is a combination of a very efficient HTTP server and a high-performance

JavaScript execution engine (Google's V8 [6]), relying on an event-based concurrency model. The tight integration of the V8 engine with the HTTP server makes of Node.JS a very convenient solution for Web services development, as -among other benefits- it enables Web developers with a single language for both the client and the server. Moreover, Node.JS (as JavaScript) is single-threaded, and service developers can implement services without having to worry about concurrency management, as the JavaScript code is guaranteed to be executed always by one thread, while the HTTP server is responsible for handling multiple requests in parallel. So as long as the JavaScript code implementing the service business logic is not a bottleneck, the Node.JS runtime can handle concurrent requests with very high throughput. Conversely, when the service is in charge of performing some CPU-bound computation, the single-threaded event loop of Node.JS represents a limitation, preventing the service from scaling or ensuring acceptable latency. Indeed, event-based concurrency also presents some drawbacks (with respect to multi-threading). In particular, the single-threaded nature of the event loop system makes it hard to scale event based systems on multicore machines without relaxing one of the core properties of such systems (i.e., data-races-free shared state, and scalability).

In this dissertation we discuss how existing single-threaded event-based systems such as Node.JS can be extended and modified to enable safe parallel programming.

1.2 JavaScript as a programming language for parallel programming

JavaScript is the most popular programming language featuring a single-threaded event-based programming model both on its client-side implementation (where the browser runs the event loop) and on its server-side counterpart (where the I/O substrate is the event loop). This dissertation is therefore mostly focused on JavaScript, and on JavaScript-based event loop frameworks.

JavaScript is single-threaded, not designed for parallel execution, and so are existing JavaScript execution engines. The evolution of the language has highly influenced the design of such engines, pushing implementers towards single-threaded implementations that are more suitable to optimize due to the lack of concurrency and therefore the lack of synchronization. From an historical perspective, the language has become popular as a scripting language for the generation of dynamic Web content, and early JavaScript execution engines were designed as simple bytecode interpreters, implemented without having any performance goal. The diffusion of technologies such as HTML5 [29], and the subsequent wide adoption of Web applications such as GMail and Facebook, has changed the nature of the language, making of JavaScript the most popular language for the development of client-side applications. One of the most evident consequences of such evolution has been the considerable research efforts [84] targeting JavaScript engines in the recent years. Such efforts have lead to a notable performance improvement, and existing execution engines such as Google V8 and Mozilla SpiderMonkey all feature high-performance runtime techniques such as polymorphic inline caches [99] and Just-in-time compilation [32].

Notwithstanding the performance improvements already obtained, very little research has been done in the field of parallelism for JavaScript. Consequently, existing state-of-the-art JavaScript engines are all single-threaded. Only recently, technologies such as WebWorkers [8] and RiverTrail [93] have been proposed to enable JavaScript applications with some support for parallel programming. However, both approaches are somehow influenced by the single-threaded architecture of existing engines, and both approaches present limitations in the context of event-based concurrency.

WebWorkers, for instance, is a share-nothing message-based programming model for JavaScript which relies on the mere replication of multiple instances of the JavaScript execution engine. The programming model requires the developer to explicitly start an arbitrary number of parallel entities, and to explicitly and manually coordinate them using message passing. Despite being well suited for many parallel algorithms and applications, actor-like programming cannot be considered a general solution for JavaScript, as it may result complex to adopt for JavaScript developers, and introduces portability issues (for instance, it is impossible for the developer to predict the exact number of parallel entities to start, as clients can be very heterogeneous). Moreover, the mere replication of the JavaScript engine execution context has negative effects in terms of resource utilization (e.g., memory footprint), and contradicts one of the key advantages of event-based concurrency, e.g., shared state. To overcome the limitations of the WebWorkers model, Mozilla and Intel have recently started researching on an experimental data-parallel programming model based on the RiverTrail model [93][1] in the Firefox browser. As opposite to WebWorkers, RiverTrail supports an elegant notion of *temporary immutable* state, which can be shared in parallel to implement data-parallel operations (mostly array-based). The model also supports non-read-only state, at the cost of parallel execution: with the goal of not relaxing the shared state nature of JavaScript, the RiverTrail runtime *speculatively* tries to execute tasks in parallel[2], and pragmatically performs the actual computation in parallel only if the tasks can be safely parallelized, i.e., they are side-effects-free. When a task is not suitable for parallel processing, its execution is not aborted, and is anyway carried out by a single thread. The model represents a significant step towards safe structured parallelism for JavaScript, but it is not suitable for event-based programming, due to its *blocking* nature: whenever a parallel computation is attempted, the execution flow of the application blocks until the parallel execution has completed. This would correspond to a severe degradation of the throughput of the event-based service, in particular in the context of long-running computations. Moreover, the parallel API of

[1]And its successive evolution into the Strawman ECMA specification draft [21]

[2]In this Dissertation we adopt the expression "speculative parallelization" commonly used in the domain of self-parallelizing runtime systems [137, 34].

RiverTrail is at the moment limited at data-parallel applications.

In this dissertation we argue in favor of a different, more generic, event-based model in which JavaScript event handler functions are the core unit of parallelization, and developers are free to execute any valid function (including those with side effects) potentially in parallel, accessing (and potentially updating) any JavaScript object.

1.3 Initial overview of the Parallel event loop

This Dissertation introduces the Parallel Event Loop model and runtime (PEV). At a very high level, the Parallel Event Loop is a combination of a parallel runtime for the safe execution of event handlers in parallel, and an event-based programming model supporting the scheduling of event handlers with multiple orderings (differently from a single-threaded event loop in which events are always processed with FIFO ordering). Consider the example of a JavaScript server-side application written using an event-based programming style depicted in Figure 1.1. The example presents a simplified Web crawler scanning a potentially infinite set of Web pages[3], looking for some string patterns[4]. Each time a page is downloaded, it is parsed by the crawler to look for new URLs to follow, and the number of occurrences for each pattern is counted using a global shared object (i.e., `patternsFound`).

The application is a valid Node.JS service. On a common JavaScript engine, parallelism would be limited at I/O operations, and the crawler would run on a single thread (that is, on a single-threaded event loop), parsing pages as they arrive, updating the shared objects as soon as any of the required patterns is found. In a PEV system, the same unmodified source code would potentially consume multiple web pages fully in parallel, without any noticeable difference, except from the speedup deriving from parallel execution. At the end of its execution, the crawler run using the PEV will produce the same result as the one produced by the single-threaded execution.

This is made possible by the PEV thanks to its execution runtime, which attempts to process events in parallel, speculatively. As not all the crawled web pages will contain the pattern, multiple web pages can safely be crawled in parallel. In ideal conditions (i.e., when events can be efficiently parallelized), the speculation scheme implemented by the PEV leads to an automatic parallelization of the application.

In addition to the advantages coming from parallel speculative execution, another benefit of the PEV runtime is to be found in its unmodified programming model, as it allows developers to write applications *as if* they were developing for a standard non-parallel event-based runtime such as Node.JS.

[3]Additional code to limit the depth of the Crawler has been omitted for simplicity.

[4]In the example, `USI`, `INF`, and `Lugano`

```javascript
var http = require('http');

// start page
var startURL = 'http://www.usi.ch/';

// Global object to store the final values
var patterns = { 'USI':0, 'INF':0, 'Lugano':0 };

// Utility function to crawl a Web page
function crawl(URL) {
  http.get(URL, function(res) {
    // Get the HTML body of the document
    var htmlBody = res.body;

    // Get all the URLs in the page
    var URLs = parseURLs(htmlBody);

    // Count all the occurrences of the patterns
    var occurrences = countOccurrences(patterns, htmlBody);

    // Update global counters
    for (var pattern in occurrences) {
      if(occurrences[pattern] > 0) {
        patterns[pattern] += occurrences[pattern];
      }
    }

    // Recursive crawl URLs
    for(var url in URLs) {
      crawl(URLs[url]);
    }
  });
}

// Start crawling from the first page
crawl(startURL);
```

Figure 1.1. Simple event-based application in JavaScript

1.4 Thesis statement

The goal of our research is to provide Web developers with novel instruments to ease the development of high-performance services and applications, overcoming the limitations of current event-based frameworks. We conduct our research in two related research areas (i.e., programming models and runtime systems) with the aim of providing a synergistic approach for the advancement of the current state-of-the-art. We can identify the following two general claims the Dissertation will argue:

(1) *Single-threaded event-based runtime systems can be safely extended to support multi-threaded shared-memory parallelism. There is no need to introduce explicit parallel entities such as threads or processes where event-based programming is supported, and there is no need to change the programming model.*

(2) *By enabling the safe parallelization of the event-based runtime, event-based programming can be conveniently adopted to derive safe structured parallel programming models and abstractions, still being compatible with single-threaded systems. When the application offers opportunities for parallel execution, the parallel event loop can effectively exploit multiple cores running event handlers in parallel.*

1.5 Contributions

This Dissertation contributes to advancing the state of the art in the field of event-based programming in the following aspects:

- Event-based speculative parallel programming. The Dissertation introduces the PEV model, a programming model and runtime system allowing event-based programming to be exploited also in the domain of parallel programming. The resulting model is not a model for explicit parallel programming, but rather a model that a speculative runtime can attempt to efficiently parallelize.

- The PEV API brings out-of-order parallel execution to the JavaScript event-based applicative domain, enabling a new class of server-side applications such as CPU-intensive services.

- The PEV Runtime can be implemented in different ways, relying on any form of speculative parallelization that can optimistically or pessimistically execute multiple event handlers in parallel. In this Dissertation we demonstrate how three existing speculative runtimes (a runtime based on locks and two runtimes based on Software Transactional Memory) can be extended and embedded in a language execution runtime to implement the model.

- The PEV runtime implementations presented in this dissertation also contribute advancing the state of the art in the VM research area, as we describe the architecture of speculation runtimes for event-based JavaScript.

1.6 Summary and outline

The Dissertation is structured as follows:

- Chapter 2 presents background information about event-based programming and JavaScript. In particular, the chapter introduces core concepts such as the event loop model of concurrency as well as one of its most popular implementations, that is, Node.JS.

- Chapter 3 introduces the Parallel Event Loop model and its event emission API. The model extends the single-threaded event loop with an additional event emission API allowing events to be executed with multiple orderings. The relaxed execution of events coupled with a safe programming model are used to derive higher-level programming models for parallel computing in event-based runtimes.

- Chapter 4 shows how the event-based API of the PEV can be used to build high-level parallel programming models that can benefit from the parallel execution of event handlers. The programming models described in the chapter are both safe and potentially parallelizable by a speculative runtime system.

- Chapter 5 describes the requirements for a speculative runtime implementing the PEV API, presenting three possible implementations called Optimistic, Pessimistic, and Hybrid PEV. Such implementations implement partially or completely the programming model introduced in Chapter 3, and are implemented using both pessimistic and optimistic speculative runtimes. Pessimistic runtimes are based on mutual exclusion using locks, while optimistic ones rely on Software Transactional Memory.

- Chapter 6 introduces *Node.Scala*, an HTTP server based on a Pessimistic PEV runtime. Node.Scala targets the Scala language and runs on the JVM platform. The chapter presents the system implementation and performance.

- Chapter 7 introduces *TigerQuoll*, an optimistic PEV runtime. TigerQuoll targets the JavaScript language and is based on the SpiderMonkey JavaScript execution engine. A Software Transactional Memory runtime is embedded in the language runtime to enable speculative parallelization. The chapter presents the system implementation and performance.

- Chapter 8 introduces *Truffle.PEV*, two implementations of a PEV runtime based on an Optimistic and on a Hybrid runtime. The two runtimes are implemented targeting the JVM runtime using the Truffle framework. The chapter presents the system implementation and performance.

- This Dissertation covers research topics from multiple fields. Related work can be found from programming languages to runtime systems research, through Web services and service-oriented systems, up to software transactional memory runtimes and spec- ulative runtimes in general. Chapter 9 discusses relevant related work.

- Chapter 10 concludes this Dissertation and identifies future research directions.

Chapter 2

Background: single-threaded event loop

The event loop is a popular software design pattern for the development of so-called *reactive* applications, that is, applications that respond to external *events*. Examples of such applications are graphical user interfaces, and any class of services, including OS-level services and Web services. In this chapter we give an overview of event-based systems, with a particular focus on event-based server systems and their corresponding programming models in JavaScript.

2.1 Event-based concurrency

Event-based concurrency is a concurrency control model based on event emission and consumption. Historically opposed to the most common multi-threaded paradigm [53], event-based systems have gained notable attention in the context of service-oriented computing because of their simple but scalable design. In an event-based system, every concurrent interaction is modeled as an *event*, which is processed by a single thread of execution, called the *event loop*. The event loop has an associated *event queue*, which is used to schedule the execution of multiple pending events. Every event has an associated *event handler* function. Every event handler can access and modify a global memory space, which is shared with other event handlers. Being the event loop a single thread of execution, the event-based concurrency model presents the following properties:

- *Safe shared state.* Every event handler comes along with an associated state, which can be shared between multiple handlers. Since all the events are processed by the same thread of execution, the shared state is always guaranteed to be consistent, and data races are not possible (that is, concurrent modifications on the state from other threads never happen).

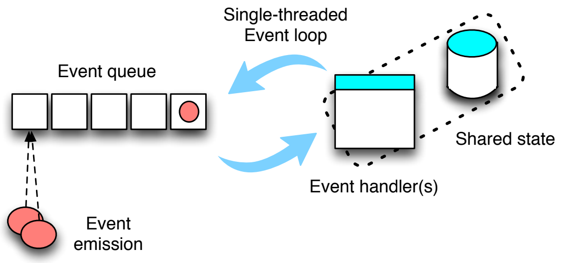


Figure 2.1. Event generation and execution in a single-threaded event loop.

- *Nonblocking processing.* Since all the events are processed by the same single thread of execution, each event handler must avoid long-running computations. A time-consuming event handler (e.g., a long CPU-intensive operation) would prevent other events from being processed, thus blocking the system. As a consequence of this design, blocking operations (for instance, I/O) are not supported by the runtime, and every operation must be asynchronous and nonblocking.

- *Ordered execution.* Events are added to the event queue with FIFO policy. This ensures that events produced with a given ordering are consumed (that is, executed) with the same ordering. Event ordering can be leveraged to chain successive events in order to split long-running computations, as well as to implement other forms of nonblocking computations[1].

The high-level architecture of an event-based system is depicted in Figure 2.1. Events are usually produced by external sources such as I/O operations (emitted by the operating system) or by the application interacting with the event loop (e.g., a GUI application controlling mouse movements). Every event handler is executed sequentially, and handlers have full control over the heap space of the process. Event handlers can invoke functions synchronously (by using the stack) or asynchronously, by emitting new events. This is depicted in Figure 2.2.

Event loop concurrency has some desirable characteristics for concurrent programming. Among others, the most important property of event-based systems is that data races are avoided *by design* allowing developers to make stronger assumptions on how each instruction is interleaved when concurrent requests are in the system. More formally, each event handler is guaranteed to run *atomically* with respect to other event handlers, and atomicity is naturally enforced by the single-threaded event loop.

Event-based concurrency also presents some drawbacks (with respect to multithreading). First, the single-threaded nature of the event loop system makes it hard to

[1]For instance, in GUI programming, the *mouseDown* event handler (generated when the user clicks on an icon) can be assumed to always run before the *mouseUp* event (generated when the button is released).

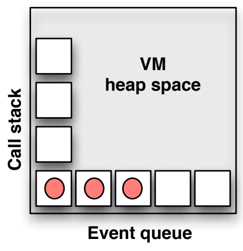


Figure 2.2. Event-based execution and shared memory in Node.JS and in the Browser.

scale event based systems on multicore machines without relaxing one of the properties above. In particular, the simplest way of scaling event-based systems on multicore machines is by replicating the event loop on multiple independent share-nothing processes. This has the drawback of *splitting* the memory space of event handlers between multiple processes, thus preventing event handlers from sharing a common memory space[2].

2.2 Event-based server architectures

Static HTTP servers are among the most common applications relying on single-threaded event-based concurrency. Services published on the Web need to guarantee high throughput and acceptable communication latency while facing fluctuating client workloads. To handle high peaks of concurrent client connections, several engineering and research efforts have focused on Web server design [49]. Of the proposed solutions, event-driven servers [61, 135] have proven to be very scalable, and their popularity has spread among different languages and platforms, thanks to the simple and efficient runtime architecture of the event loop [118, 115]. Servers of this class are based on the ability offered by modern operating systems to interact asynchronously (through mechanisms such as Linux's `epoll` [65]), and on the possibility to treat client requests as collections of events. Following the approach proper of event-based systems, in event-driven servers each I/O operation is considered an event with an associated handler. Successive events are enqueued for sequential processing in an I/O event queue, and processed by the infinite event loop. The event loop allows the server to process concurrent connections (often nondeterministically) by automatically partitioning the time slots assigned to the processing of each request, thus augment-

[2]Moreover, approaches based on process-level replication have other drawbacks with respect to memory footprint and service latency, as the communication between multiple processes can represent a source of performance degradation due to the need for exchanging -via memory copy- data structures (e.g., session data) between multiple processes.

ing the number of concurrent requests handled by the server through time-sharing. In this way, sequential request processing is overlapped with parallel I/O-bound operations, maximizing throughput and guaranteeing fairness between clients. Thanks to the event loop model, servers can process thousands of concurrent requests using a very limited number of processes (usually, one process per core on multicore machines). Being the server static, the lack of any integration with any dynamic language runtime makes the parallelization of such services trivial [135].

2.3 Event-based Frameworks and Web Programming

Of particular success in the domain of Web development are the so-called event-based frameworks, i.e., frameworks based on an event-based runtime systems for request handling and response processing. Such frameworks allow the developer to specify the service's semantics as a set of asynchronous callbacks to be executed upon specific events; a model which results particularly convenient when embedded in managed runtimes for languages such as JavaScript, Python, or Scala (popular examples of such frameworks are Node.JS [15], Akka [1], and Python Twisted [106]). Unlike other thread-based solutions, event-based frameworks have the advantage of not requiring the developer to deal with synchronization primitives such as locks or barriers, as a single thread (the event loop thread) is capable of handling a high number of concurrent connections. Furthermore, the asynchronous event-based programming style of such frameworks has influenced the programming model of client-side Web applications, as every Web application nowadays is using asynchronous event-based programming in JavaScript and HTML5 (i.e., AJAX [79]).

Event-based frameworks have recently become very popular, and several companies have started offering cloud-based PaaS hosting for event-based service development platforms (examples are Microsoft Azure [14] and Heroku [7]). However, despite their success, event-based frameworks still present limitations which could prevent them from becoming a mature solution for developing high-performance Web applications. In particular, the lack of structured parallelism forces the developer to implement complex ad-hoc solutions by hand. Such unstructured approaches often result in cumbersome solutions which are hard to maintain [126]. Some of the limitations of current programming models can be described as follows:

1. *event loop replication*: the event-based execution model offers only partial solutions to exploit parallel machines, since it relies on a single-threaded infinite loop. Such centralized architecture limits parallelism, since it is possible to exploit parallel machines only by replicating multiple processes using some Inter-Process Communication (IPC) mechanism for coordination. This approach prevents event-based applications from exploiting the shared memory present in multicore machines. Moreover, callbacks could have interdependencies, and cannot therefore be executed on multiple cores (indeed, the sole possibility to exploit multiple cores is by replicating the entire

event loop process). The common solution is to bind each specific connection to a specific core, preventing connections from exploiting per-request parallelism (i.e., to start multiple parallel processes to handle a single client request, using for instance more complex patterns such as, e.g., MapReduce [64] for generating the response).

2. *Shared state*: In single-threaded event loop concurrency event handlers can exploit a common shared memory space (that is, some data shared between multiple concurrent client requests). However, this becomes impossible when running multiple parallel event loop processes. The common solution to this issue is to use an external service to manage the state (e.g., Memcached [12]). However, such solution presents other limitations when the shared state is immediately needed by the application's logic. For instance, it is not always efficient to use an in-memory database to handle the temporary result of a parallel computation before serving the computation's result to the client.

Another consideration more specific to server-side event-based systems can be made considering the nature of server-side architectures. In particular, one of the main peculiarities of server-side parallelism over "traditional" parallel programming relies on the actual source of parallelism. In fact, in server-side development there is a *natural* source of parallelism represented by the abundant presence of multiple concurrent client requests which should be handled in parallel. Client requests could have interdependencies (for instance in stateful services), but in the great majority of cases each client request does not require the service business logic to use explicit parallelism (for instance using threads). This implicitly means that the logic of each request handler could be written using a plain sequential style, or an event-based style, but no explicit parallel programming abstractions need to be used for generating the client response while ensuring scalability.

We can call this peculiar characteristic of server-side development *natural parallelism* of Web services. Current existing frameworks do not fully exploit this source of parallelism, and instead require the developer to either manually manage parallelism (for instance by replicating processes or starting threads) or to rely on existing limited solutions usually offered by the cloud provider (for instance, several cloud providers allow users to specify the number of parallel replicas of a service that should be started automatically, at the cost of limiting the class of applications that can be potentially hosted in the cloud [109]).

2.4 Event-based programming models

The peculiarities of event-driven systems have promoted several programming models relying on event loop concurrency. Of particular interest in the scope of this Dissertation are all frameworks and libraries which allow Web services to be scripted and/or embedded within other language runtimes. Examples of programming models based or relying on the event loop include libraries and frameworks (e.g., Vert.X [26], Python

Twisted [106] or Java NIO [35]), as well as VM-level integrations such as Node.JS. Existing approaches can be divided into two main categories, namely programming models that explicitly require the developer to produce and consume events, and models that abstract the event loop by means of other higher-level concepts. In this section we give a brief overview of the two approaches, and we argue in favor of high-level ones.

2.4.1 Explicit event emission

Explicit loop manipulation programming requires the developer to explicitly produce and consume events in the form of callback functions. This can be done by explicitly adding a function to the global event queue, using a primitive called `async`[3]:

```
// Main event emission primitive for the single-threaded loop
async( Function to be added to the event queue ,  Optional arguments );
```

The primitive is nonblocking, and returns immediately after having added a function to the event queue. The queue is thread-safe, as other functions may be concurrently added by the underlying operating system. The primitive is the main mechanism used by event-based systems to introduce asynchronous execution, but does not provide any mechanism for relating functions with events. This is usually achieved with two higher-level core primitives, namely `on` and `emit`:

```
// Core event emission primitive
emit( Event label ,  Event arguments );
```

```
// Core callback registration primitive
on( Event label ,  Callback );
```

The `on` primitive can be used to specify one or more event functions (i.e., callbacks) to be associated with a specific event. In this Dissertation we adopt the convention that events are named using strings (called *event labels*), but other ways of specifying event identifiers also exist. We also adopt the convention that event emission and handling can be bound with existing object instances, and therefore expressions like `obj.emit`/`obj.on` mean that an event is being emitted and consumed only if the given object instance has registered a callback for the given event label. This is conceptually equivalent to having a callback table private to each object instance, but other design decisions can also be adopted (for instance, a globally shared callback table).

The `emit` primitive is used to notify the runtime that an event has happened, and its callback can be added to the event queue. The primitive is internally implemented using `async`, as depicted in Figure 2.3.

Events can also be associated with arguments. This is also presented in Figure 2.3, where at line 10 an event emitter is used to produce and consume events. The example corresponds to how incoming connections are handled by an event-based socket

[3]In other runtimes the `async` primitive can be called differently, e.g., `nextTick` in Node.JS

```
Object.prototype.emit = function(label, args) {
  for(var callback in this.callbacks[label]) {
    async(this.callbacks[label][callback], args);
  }
}
Object.prototype.on = function(label, callback) {
  this.callbacks[label].push(callback);
}

var obj = {};

obj.on('connection', function(fd) {
  // open the file descriptor 33 and handle the request
});

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

Figure 2.3. Usage and implementation of the `emit` primitive.

server like Node.JS. 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 file 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` will result in multiple invocations of the callback function handling the event.

Event ordering ensures that multiple events emitted in a specific order will be executed in the same order they are produced. As a consequence, the following code depicted in Figure 2.4 will always result in the same console output.

The event emission primitives are low-level basic building blocks that can be used to build higher-level abstractions. Other primitives to wait for a specific event to happen, to deregister event handlers, as well as to trigger multiple events upon certain conditions (for instance using a pub/sub pattern [38]) exist. In the next Section we will discuss how such low-level primitives can be used as simple blocks for building other more convenient high-level abstractions.

2.4.2 Implicit loop manipulation and Node.JS

Although many libraries for explicit loop manipulation exist, explicit loop programming is usually considered a complex programming model, specific for low-level system development[4]. To overcome the complexity of the explicit interaction with the event loop, other high-level models have been proposed.

[4]The name of one of the most popular event-based frameworks, i.e., Python Twisted [106] is a perfect indicator for the complexity that certain event-based applications tend to have.

```
obj.on('data', function(data) {
  console.log('the data is: ' + data);
});
obj.emit('data', 1);
obj.emit('data', 2);
obj.emit('data', 3);

// expected console output:
// the data is: 1
// the data is: 2
// the data is: 3
```

Figure 2.4. Ordered event emission using `emit`.

One of the most popular of such frameworks is Node.JS. Node.JS is a programming framework for the development of Web services in which the event loop is hidden behind a simpler programming abstraction, which allows the developer to treat event-driven programming as a set of callback function invocations, taking advantage of JavaScript's anonymous functions. Since the event loop is run by a single thread, while all I/O-bound operations are carried out by the OS, the developer only writes the sequential code to be executed for each event within each callback, without worrying about concurrency issues. Consider the following *hello world* web service written in Node.JS JavaScript:

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(1337, '127.0.0.1');
console.log('Server running at http://127.0.0.1:1337/');
```

The example corresponds to the source code of a very minimal yet scalable web service. No explicit loop manipulation is done by the user, who just has to provide a callback (in the form of an anonymous function, at line 2) to be executed for each incoming request. The Node.JS runtime is internally relying on explicit loop manipulation, as presented in Figure 2.5. The code in the example corresponds to the source code for the HTTP server of Node.JS, which is written in JavaScript itself. The code in this example registers multiple event handlers to handle the multiple phases of an incoming request handling. Since incoming requests may be sliced by the OS runtime into multiple *data chunks* (line 9), an event handler is registered to accumulate multiple chunks. Once the incoming data has been fully received, an explicit event emission is used (i.e., `newRequest`) to eventually invoke the user-provided callback.

Event-based programming coupled with the high performance of the V8 Engine makes of Node.JS one of the Web frameworks with the highest scalability in terms of concurrent connections handling [111]. Its programming model based on asynchronous callbacks fits well with the event loop as every I/O-event corresponds to a

```
http.prototype.createServer = function(callback) {
  // Create a new server instance
  var server = new runtime.HttpServer();
  // Event handler for any incoming connection
  server.on('connection', function(socket) {
    // create a buffer to store incoming data
    var buffer = runtime.createInputBuffer();
    // register an handler to accumulate incoming data
    socket.on('data', function(data) {
      if(data != '\n\r') {
        // accumulate data chunks
        buffer.push(data);
      } else {
        // once the data has been received, notify
        server.emit('newRequest', buffer);
      }
    });
  });
  // register an event handler to invoke the callback
  server.on('newRequest', function(data) {
      var request = runtime.createRequestObject(data);
      var response = runtime.createRssponseObject(data);
      // call the user-provided callback
      callback(request, response);
  });
}
```

Figure 2.5. Implementation of an event-based HTTP server

function callback invocation. As with event-driven systems, concurrent client requests are processed in parallel, overlapping I/O-bound operations with the execution of callbacks in the event loop. The event loop is implemented in a single process, while all the I/O-bound operations are carried out by the OS, thus, the developer has only to specify the sequential code to be executed for each event within each callback. Despite of the power of the event loop and the simplified concurrency management of the programming model, frameworks like Node.JS still present some limitations preventing them to exploit modern multicore machines.

2.5 Limitations of single-threaded event loop systems

When it comes to parallel execution, event-driven frameworks like Node.JS may be limited by their runtime system in at least two distinct aspects, namely (1) the impossibility of sharing a common memory space among processes, and (2) the difficulty of building high throughput services that need to use blocking function calls.

Concerning the first limitation, common event-based programming frameworks are not designed to express thread-level parallelism, thus the only way of exploiting multiple cores is by replicating the service process. This approach forces the developer to adopt parallelization strategies based on master-worker patterns (e.g., WebWork-

```
 1 var SIZE = 100000;
 2 // Size of the subset to be returned
 3 var SUBSET = 10000;
 4 // Init the shared read-only array
 5 var A = new Array();
 6 for(var i=0; i<SIZE; i++) {
 7   A.push(Math.floor(Math.random()*10000));
 8 }
 9 // Create the service and start it
10 var http = require('http');
11 http.createServer(function (req, res) {
12   // Get a subset of the array
13   var x = new Array();
14   for(var i=0;i<SUBSET;i++) {
15     var id = Math.floor(Math.random()*SIZE);
16     x.push(A[id]);
17   }
18   // Sort it
19   x.sort();
20   // Return the sorted array
21   res.end('sorted:' + x);
22 }).listen(1337, '127.0.0.1');
```

SUBSET	Throughput (msg/s)	Latency (ms)
10^2	2865.4	3.4
10^3	1799.3	6.9
10^4	253.8	40.3
10^5	23.3	421.2

Service performance for different values of SUBSET

Figure 2.6. Array sorting microbenchmark in Node.JS.

ers [8]), which however require a share-nothing architecture to preserve the semantics of the event loop. Whenever multiple processes need to share state (e.g., to implement a stateful Web service), the data needs to be stored into a database or in an external in-memory repository providing the necessary concurrency control.

Concerning the second limitation, event-based programming requires the developer to deeply understand the event loop runtime architecture and to write services with non-blocking mechanisms so as to break down long-running operations into multiple processing steps.

As an example, consider the Web service in Figure 2.6. The code in the example corresponds to a simple stateful service holding an in-memory data structure (the array `A`, at line 5) which is "shared" between multiple concurrent clients. Each time a new client request arrives the service simply extracts a subset of the array and copies it to a new per-request object (the array `x`, at line 13). Once the subset array has been copied, the new array is sorted by using the `sort` JavaScript builtin function, and the resulting sorted array is sent back to the client. The service corresponds to a simple yet significant microbenchmark showing how the two limitations introduced above may affect the Node.JS model:

- Long-running computations: sorting the array corresponds to a small yet significant operation that can significantly affect the performance of the service. This is shown in the table presenting the performance of the service for different values of the `SUBSET` variable (which corresponds to the size of the subset of the array to be copied)[5]. As the table clearly shows, an even small size of the array is enough

[5]The microbenchmark is executed on two distinct machines using a recent version of Node.JS

to affect the performance of the service in a very relevant way. In particular, the throughput drops and many client requests fail because of a timeout. Requests that do not timeout are affected as well, as the average latency of the service also grows.

- Shared state: to reduce the slowdown introduced by the sorting operation, the only solution for current event-based systems would be to offload the sorting operation to another event loop running in another process. Not having the ability to safely share state among processes, however, the state would need to be either replicated (that is, cloned into multiple processes), or to be sent to other processes as needed (e.g., by using an external database). Both approaches may result in a waste of space (that is, increased memory footprint) and might affect the service's latency. Moreover, from a programming model point of view the event-based system will lose part of its appeal because of the introduction of message passing and explicit coordination between processes.

In the next sections of the Dissertation we will discuss how the PEV model can be used to solve or attenuate the two issues without changing the single-threaded programming model.

(v0.10.28) with the `ab` HTTP client using with the following configuration: `ab -r -n 10000 -c 10 http://127.0.0.1:1337/`.

Part II

Programming Model

Chapter 3

The parallel event loop model

In this Dissertation we argue that the single-threaded event loop model can be safely extended to exploit the inner parallelism of shared-memory multicore machines. To this end, a new programming model and runtime system, called the *parallel event loop*, is introduced. A novel API allows developers to interact with the event loop to schedule and synchronize event execution. The API can be used to expose to higher levels of abstraction several parallel programming models. The parallel event loop does not modify the original semantics of the single-threaded loop, but enriches it with support for out-of-order execution. As a direct consequence, programs developed for a traditional single-threaded event loop can still safely be run in the parallel event loop, without modifications. At the same time, programs explicitly developed for the parallel event loop can still run on single-threaded loops (without any semantical difference). This chapter presents the main characteristics of the parallel event loop model, and presents examples of how it can be used in the domain of server-side programming.

3.1 The design of the parallel event loop

The *Parallel event loop* (PEV) is a programming model and a runtime system supporting the execution of event handlers concurrently and *potentially* in parallel. Unlike with the single-threaded event loop (SEL), event processing is not limited to sequential First In First Out (FIFO) processing (i.e., one by one, as events are added to the event queue), and the system also provides safe mechanisms for scheduling and consuming events with other orderings.

The intuitive motivation for the PEV model is that from a programming model point of view event handlers are often independent (e.g., they process different client requests), and thus might offer chances for parallelization. The target developer for the PEV is any developer already using a single-threaded event loop for the development of Web applications. The model is not designed to introduce support for high-performance computing in the single-threaded event loop domain: parallel programming is often complex and scaling applications up to thousand of cores requires developers to master advanced programming techniques [95] aimed at dealing with complex issues such as locality, cache coherency, etc. Rather, the goal is to *nicely* and transparently introduce *opportunities* for parallel processing in the single-threaded model, with the goal of keeping the model simple and safe.

At high level, the PEV is a model for *speculative* parallel processing with the following characteristics:

- *Implicit speculative execution.* The model does not expose any explicit parallel entity such as actors, processes, or threads, nor any explicit blocking synchronization primitive such as locks, barriers, countdown latches, etc. Given the lack of any explicit construct for parallel execution, the developer does not have any explicit way of scheduling a computation for parallel execution. The developer can only specify the application's logic, and the runtime will speculatively try to execute portions of it in parallel. The developer is however *aware* of the fact that the runtime will try to speculate on the potential for parallelization of the application.

- *Stateful computation.* Speculative parallelization implies that the developer is allowed to develop applications *as if* he was using a single-threaded event loop system. This includes supporting side effects and shared state between callbacks. Emerging stateless programming models for event-based frameworks such as functional reactive programming [128] are nicely parallelizable by the PEV, while other workloads with globally shared state might still offer opportunities for parallel execution.

- *Simple sequential semantics.* The main programming model advantage of single-threaded event-based systems is that the parallel processing of I/O is hidden behind the convenient single-threaded processing of callbacks. This has the advantage of saving scalability for I/O-based workloads still preserving the simplicity of a single-threaded race-free programming model. The PEV extends this model to CPU-bound applications by enforcing the *impression* of a single-threaded runtime.

At its core, the parallel event loop can be seen as the combination of an event-based API and a runtime system supporting the parallel execution of event handlers:

- *Parallel emission API*. The PEV extends the core API of the single-threaded event loop with an additional event emission API. The additional API can be used to schedule events for execution with policies other than FIFO. Combined with the core (single-threaded) event emission API, the PEV offers an API that can be

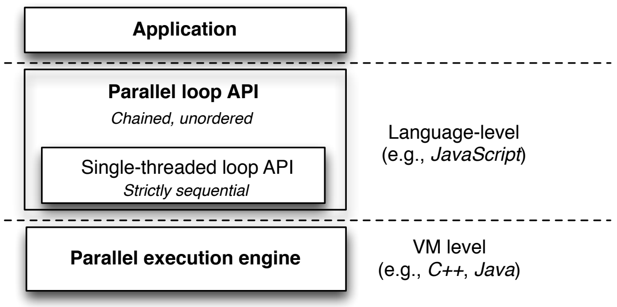


Figure 3.1. Parallel event loop API and runtime overview.

used to schedule events for ordered and unordered execution. In the PEV model the developer does not have the power to specify which task (i.e., which event handler) has to be executed in parallel, but only the order in which they shall be processed.

By combining ordered and unordered events, the system can support several parallel (implicit and explicit) programming models. The original event-based API of the single-threaded event loop can be considered as an (ordered) subset of the PEV API, and is fully supported by the PEV runtime in order to provide backward compatibility.

- *Parallel runtime*. The PEV runtime system supports the concurrent execution of multiple event handlers, in parallel, when possible. The runtime ensures the safe execution of event handlers, and enforces the correct execution order of events, as indicated by the event emission PEV API.

Beyond the benefits deriving from parallel execution, the PEV is also designed to have strict interoperability with single-threaded event loops. In particular, when using only the core single-threaded event emission API, the PEV is equivalent to a single-threaded loop runtime, offering the following compatibility properties:

- *Event ordering*. The PEV executes events ensuring semantical equivalence with the single-threaded loop, when the core sequential API is used. As a consequence, there is no noticeable difference (in terms of execution semantics) between an application running on a PEV system and one executed by a SEL.

- *Memory model compatibility*. Every event executed by the PEV can access and modify any memory location, including the ones potentially shared with other concurrent events[1]. This is the default model for a SEL, where only a single event is running at a time, and data races are not possible.

[1]According to the memory model and scoping of the target language, e.g., JavaScript.

By enabling the parallel execution of event handlers, the above requirements also correspond to design constraints, which the PEV runtime enforces in the following way:

- *Asynchronous execution*. All the callbacks shall be processed asynchronously, that is, no blocking callback scheduling is supported. In a single-threaded event loop, the application must never stop waiting for some interaction with external resources such as the OS or the user interface. The reason is that suspending the single main thread on some external interactions would prevent other event handlers from being executed, thus harming performance. As a consequence, event handlers are usually small, and any interaction with external resources is nonblocking and asynchronous. In the context of a parallel event loop runtime, the system would still be able to progress while some handlers are waiting for some computations to complete. However, keeping a thread busy waiting for interactions with external resources would nevertheless result in an inefficient use of the system's resources, as new threads would have to be executed when all the existing threads would block. Consequently, the PEV must embrace the same asynchronous nonblocking abstraction as in the single-threaded loop. This design is also motivated by the need of being fully compatible with a single-threaded model.

- *Execution order*. Events emitted using FIFO ordering have to *appear* as if they were executed by a single event loop. To improve performance, events can potentially be executed by the PEV runtime speculatively out-of-order (i.e., as soon as possible). When speculative execution of ordered events is supported, however, events are automatically reordered by the runtime.

- *Memory protection*. To enforce the same memory model of the single-threaded loop, parallel events that might generate side effects (potentially visible to other events) must be executed in *isolation*, not exposing any observable side effects to other events until they complete their execution. Moreover, the execution of event handlers must be *atomic*, ensuring that all the modifications to the shared state will be visible to other concurrent handlers only at the point in time when the event has completed.

By enforcing these constraints, the PEV runtime guarantees that applications developed using only the core sequential API will have the exact same semantics as if they where running on a single-threaded runtime, but might potentially benefit from parallel execution. Conversely, applications developed using the extended (parallel) API could potentially benefit from parallel execution, without relaxing the properties of event-based programming. The API compatibility scheme for the PEV model is summarized in Figure 3.1.

	Application developed using the single-threaded loop API.	*Application developed using the parallel loop API.*
Running on a single-threaded runtime	Native support.	The single-threaded runtime will execute all the event handlers, with no noticeable speedup.
Running on the parallel event loop runtime	The PEV runtime might still try to execute handlers in parallel, always enforcing the correct semantics.	The PEV runtime will schedule events in parallel as much possible.

Table 3.1. API compatibility of the PEV.

3.2 Implicit parallelism and event ordering

In the PEV model, programs developed targeting a SEL system might potentially be executed in parallel by the runtime system, eventually enforcing the same semantics of the equivalent sequential execution. In the parallel programming literature such systems are usually called *implicitly parallel* systems [80], meaning that they provide developers with the impression of a single-threaded runtime system in which parallelization is done automatically. According to this definition, the PEV can be classified as an implicitly parallel runtime system. However, the compatibility requirement on FIFO-ordered events (needed to enforce the single-threaded loop compatibility) is not a strict requirement for *every* event handler. Indeed, certain events of specific classes can be processed with ordering others than FIFO.

In addition to FIFO event ordering, the PEV introduces two other classes of event handlers. The following three distinct classes of events are supported:

- *Globally strictly ordered events* (FIFO). Events that have to be executed respecting the same order they were added to the queue. Events of this class are the sole events supported by SEL runtimes.
- *Chained ordering*. Events that have to be processed only *after* a specific event has been processed, but not necessarily after *all* the other events in the queue. Events of this class have a logic dependence with other events, but can be executed out-of-order with respect to unrelated events.
- *No ordering (unordered)*. Events that do not need to be processed with any ordering, as they do not depend on any other event. Events of this class can be processed as soon as possible.

The three classes of events are defined in Figure 3.2. The PEV runtime system can speculatively attempt to parallelize the execution of all of the three classes of events,

Event class	Semantics
Strictly ordered	Every event that is selected for execution at instant T_i and needs to wait for an event already scheduled at instant T_{i-1} to complete before being executed. All side effects produced by the event at T_{i-1} are visible to the event T_i.
Chained	An event that is selected for execution at instant T_i and has to wait for *some* events at instant T_{i-n}, for an arbitrary n with $n < i$.
Unordered	An event that is selected for execution at instant T_i, and does not have to wait for any other event before being safely scheduled.

Table 3.2. Event classes supported by the PEV.

meaning that also strictly ordered events might potentially be executed by multiple threads. This is depicted in Figure 3.2. The Figure presents the typical scenario for an event-based server-side framework like Node.JS. The service has received two incoming concurrent connections, and is ready for processing them by having in its event queue all the events that have been triggered by the underlying operating system event emission substrate. A single-threaded event loop will process the events in the queue one by one, thus respecting the ordering in which events were emitted (a). When running in the PEV, however, events can be consumed by the runtime in different orders. When events are added to the queue with strict ordering, events are executed (potentially by more than one thread) enforcing the same ordering of single-threaded execution (b). Assuming the two connections can be processed in parallel, independently (this is often the case with stateless Web services), events can be consumed in parallel, just by assigning each thread a specific connection. This is equivalent to identifying a relation between events, which imposes that events belonging to distinct connections must be processed after events belonging to the same connection. This can be obtained by using chained events (c). Note that this approach to request handling is conceptually different from having two independent processes processing the two requests, as all the event handlers share the same memory space; indeed, this is equivalent to having a single process accepting two requests concurrently, in parallel. Finally, events respecting unordered processing (d) will respect an execution schema in which all events are executed as soon as possible[2].

In a SEL system, the single-threaded nature of the runtime (and its centralized

[2]For this example, this might correspond to a potential violation of the semantics of the original Web service; still, unordered processing might be perfectly legal in other domains, for example when consuming unordered streams of elements coming from multiple distinct data sources.

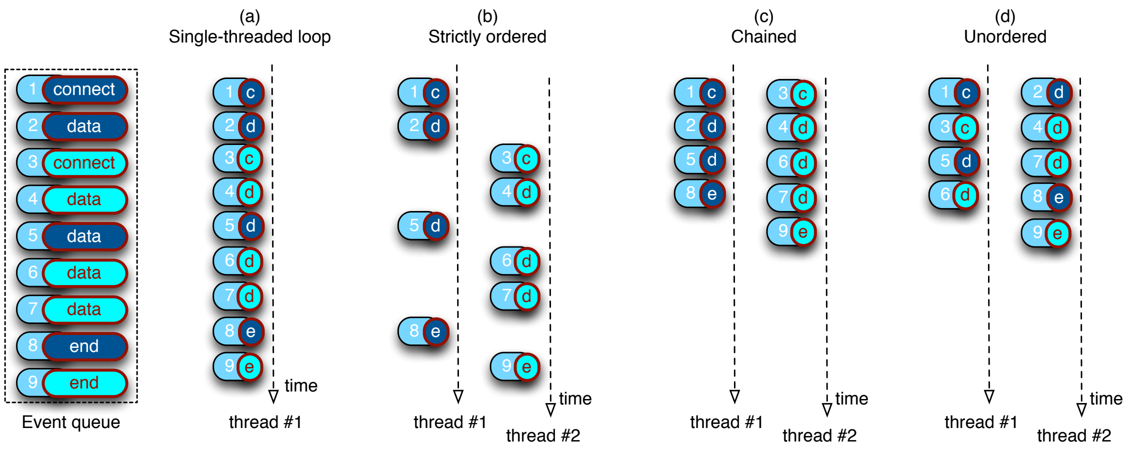


Figure 3.2. Parallel event processing and ordering.

event queue) naturally enforce strict event processing. The two other classes are also supported by the single-threaded loop by means of *flattening*, as not having to respect a specific order also means that *any* ordering is a valid execution order, including the strictly sequential one (that is, the same order events are scheduled).

Event scheduling different than FIFO do not result particularly useful in the context of a single-threaded event loop; however, assuming that distinct event classes can be safely executed concurrently, events scheduled with different policies might potentially benefit from parallelization, resulting in an increased overall throughput. Following this intuition, the PEV system treats chained and unordered events as two distinct classes, which can directly benefit from parallel execution.

This also implies a form of *forward compatibility*, as programs explicitly developed for the PEV will also run on a single-threaded runtime, loosing any potential performance improvement deriving from parallel execution, but without any noticeable difference in terms of application semantics.

In the following section we describe the core event emission API of the PEV model, and we discuss how the API can be used to build higher-level parallel programming models.

3.3 Core parallel event emission API

The core interface for asynchronous execution in the single-threaded event loop as introduced in the previous chapter is represented by the `async` primitive. The primitive can be used to schedule the asynchronous execution of a callback function together with an arbitrary number of arguments, as depicted in Figure 3.3. The PEV extends the event emission API with two additional primitives, namely *asyncChained* and *async-*

```
// Execute the given callback asynchronously
async(  Callback ,  Arguments );

// Execute the given callback as soon as possible for the given target
asyncNow( Event target ,  Callback ,  Arguments );

// Execute the given callback using chained ordering for the given target
asyncChained( Event target ,  Callback ,  Arguments );
```

Figure 3.3. PEV core event emission API

Now, also depicted in the Figure.

The two primitives can be used to schedule an event callback for execution with different execution order. The primitives accept a parameter, called *event target*, which can be used to *bind* events to a specific object instance. Event targets can be considered as *logical* event queues, and can be used to make event handlers to run concurrently. The two primitives differ in the way they interact with the event target, as well as in the way they schedule callbacks execution:

- Callbacks scheduled using the `asyncChained` primitive will respect strictly sequential ordering with respect to other events emitted using the same event target, and will have to wait for *all* the other events scheduled on the same target. As an example, events with target A scheduled at time instant T_i will be executed only once all the other events with the same event target (scheduled at time T_n with $n < i$) will have completed. Events with targets different than A do not need to wait for such events before being executed.

- Callbacks scheduled using the `asyncNow` primitive will be executed as soon as possible with respect to other event classes (including unordered events of the same targets). Such events, however, might have to synchronize with chained events with the same event target (if any). In this case, events will be executed as soon as all the chained events with the same target have completed.

Event targets can be specified using any object instance. Using a global object instance, global ordering can be obtained. As a convention, we use the `"global"` string literal as the *global event target*. Callbacks scheduled using the `"global"` target will have global execution ordering, and all the events emitted using this target will be forced to respect it. By using the global target, the `async` primitive of the SEL can be considered equivalent to the following primitive:

```
// SEL Async defined using the PEV asyncChained primitive
async = function(callback, args) {
  asyncChained('global', callback, args);
}
```

The API can be used to directly schedule callbacks with different execution orders. Consider the code samples presented in Figure 3.4, and the corresponding execution schedule. The primitives `asyncNow` and `asyncChained` can be combined to obtain partial scheduling of events, as described in Figure 3.4 (d). In particular, unordered events sharing an event target with chained ones will have to wait for all the previous chained events to complete before executing. Similarly, chained events will have to wait for all the unordered events present in the system.

Thanks to the extended API, callbacks can be scheduled in the PEV to obtain different execution orderings. The two event emission primitives have to be considered core, low-level, tools for building higher-level abstractions for asynchronous parallel programming. In the following sections the primitives are used as building blocks for an extended event emission API which will then be used as the core API for high-level, structured, parallel programming models.

3.3.1 Shared memory and atomicity

Event handlers are executed by the PEV runtime as atomic and isolated tasks. The execution is automatically enforced by the runtime, which implements specific mechanisms to ensure atomicity and isolation. As a result of the automatic runtime management of concurrent access, event handlers always appear as if they were executed on a single thread. Consider the following example:

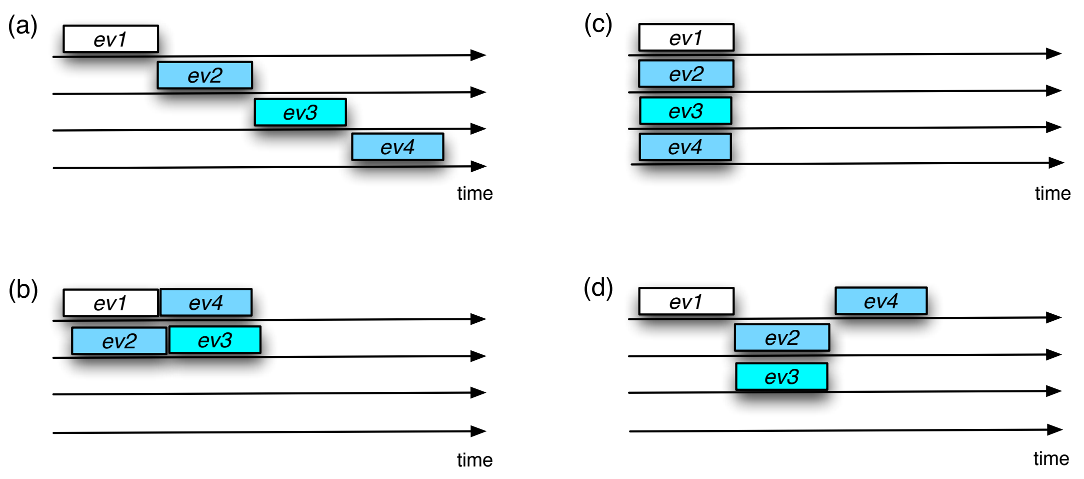
```
// an object in the scope of the two functions, and therefore potentially shared
var shared = 0;

function f1() { shared++; }
function f2() { shared--; }

asyncNow('global', f1);
asyncNow('global', f2);
```

Notwithstanding the fact that both functions `f1` and `f2` are allowed to run concurrently, the final value of the `shared` variable will always be zero, as both event handlers will execute atomically.

3.4 Strictly ordered events

To ensure full backward compatibility with applications developed for the single-threaded event loop model, the PEV model must provide an implementation of the `emit` primitive. The primitive can be supported in the PEV model just by using the `asyncChained` primitive with global event target:

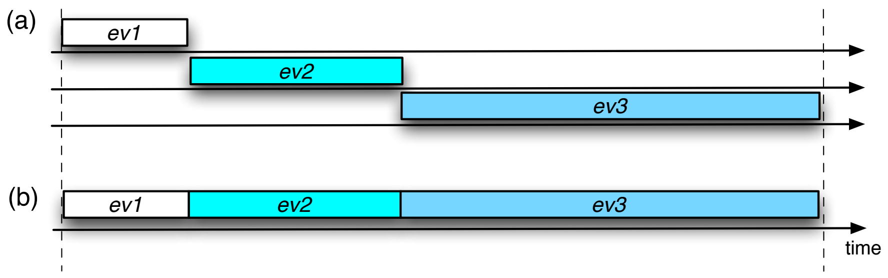
```
Object.prototype.emit = function(label, arguments) {
  for(var callback in this.callbacks[label]) {
    asyncChained('global', this.callbacks[label][callback], arguments);
  }
}
```



```
function ev1() { ... }
function ev2() { ... }
function ev3() { ... }
function ev4() { ... }

// (a) Strictly sequential, global
asyncChained('global', ev1);
asyncChained('global', ev2);
asyncChained('global', ev3);
asyncChained('global', ev4);

// (b) Chained (different event targets run in parallel)
var objA = {}; var objB = {}
asyncChained(objA, ev1);
asyncChained(objB, ev2);
asyncChained(objA, ev4);
asyncChained(objB, ev3);

// (c) Unordered
asyncNow('global', ev1);
asyncNow('global', ev2);
asyncNow('global', ev3);
asyncNow('global', ev4);

// (d) Partially ordered
asyncChained('global', ev1);
asyncNow('global',  ev2);
asyncNow('global',  ev3);
asyncChained('global', ev4);
```

Figure 3.4. Execution of multiple event handlers using `asyncNow` and `asyncChained`.

This ensures that every event emission made through the `emit` primitive will belong to the same global target, and will use FIFO ordering. This is demonstrated in the code presented in Figure 3.5. The three event handlers in the example simply perform



```
var obj = {};
// register three event handlers for three distinct events
obj.on('ev1', function() {
  var r = computeNthFibonacciNumber(10);
  print('event 1 result: ' + r);
});
obj.on('ev2', function() {
  var r = computeNthFibonacciNumber(20);
  print('event 2 result: ' + r);
});
obj.on('ev3', function() {
  var r = computeNthFibonacciNumber(30);
  print('event 3 result: ' + r);
});

// emit an event instance for each handler.
obj.emit('ev1'); obj.emit('ev2'); obj.emit('ev3');
```

Figure 3.5. Ordered event processing with the Parallel Event Loop (a) and a single-threaded event loop (b).

some arbitrary CPU-bound computation (in the example, calculate the Nth Fibonacci sequence number) and then print the result on the system's standard output console. Figure 3.5 depicts the execution of the three event handlers in both a single-threaded event loop system and one of the potential execution schemes in the PEV runtime. The three events are processed sequentially, in order (they might potentially run in different threads). Since the PEV supports globally shared state, there is no difference between the two execution schemes in terms of memory model. The only noticeable difference is that the PEV runtime might concurrently process event handlers of other classes (e.g., unordered ones) in other threads.

In both a single-threaded loop and the PEV, the three events are pushed accordingly into the event queue, and handlers are executed with the expected ordering. Eventually, the console will always show the following output for both the PEV-based execution and the SEL-based one:

```
event 3 result: 832040
event 2 result: 6765
event 1 result: 55
```

3.5 Relaxed ordering of events

The strictly ordered execution of events supported by the PEV runtime guarantees compatibility with the single-threaded loop model. Events with chained ordering as well as unordered events can be treated by the PEV runtime with a different, potentially parallel, execution policy.

3.5.1 Chained execution

Event targets can be leveraged to combine multiple events in a convenient way, to *chain* event emissions. In particular, it is possible to combine strictly sequential emissions with chained and unordered emissions, to obtain complex execution patterns. This can be made possible by introducing the `emitChained` primitive:

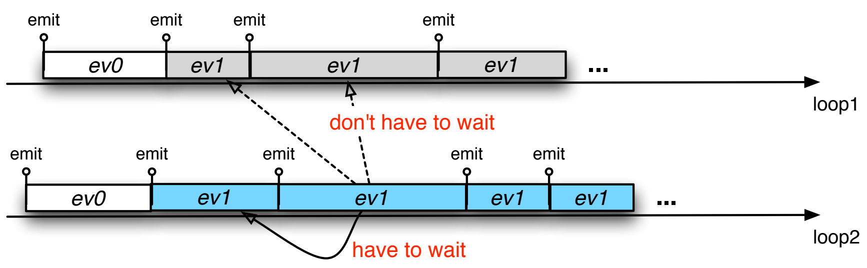
```
Object.prototype.emitChained = function(label, arguments) {
  // The object instance is the event target
  for(var callback in this.callbacks[label])
      asyncChained(this, this.callbacks[label][callback], arguments);
}
```

By using the current object instance as the target of the event emission, the `emitChained` primitive has the important property that all the events emitted against the `this` target will synchronize *only with their object instance*, and will ignore event handlers with a different target (including global ones). By exploiting this property, events can be *chained* between multiple event emissions. The `emitChained` function can be used to chain all the events that will be generated after its first invocation using the other primitives previously discussed. In the simplest case, this can be used to emulate an execution model similar to the replication of two event loops, as shown in the example depicted in Figure 3.6. The emission of the event `ev0` through `emitChained` causes the corresponding event handler to run immediately. As `emitChained` has chained ordering, it acts as the *root* event for all the other events emitted by its handler. The handler in the example will emit a potentially infinite set of events of type `ev1`, which will run with sequential ordering. However, such events will not have to wait for the other `ev1` events generated by the execution of the second `ev0` event, as they will belong to a distinct target.

3.5.2 Unordered execution

Another event emission primitive can be introduced to support the parallel execution of event handlers out-of-order, `emitNow`:

```
Object.prototype.emitNow = function(label, arguments) {
  // The object instance is the event target
  for(var callback in this.callbacks[label])
      asyncNow(this, this.callbacks[label][callback], arguments);
}
```

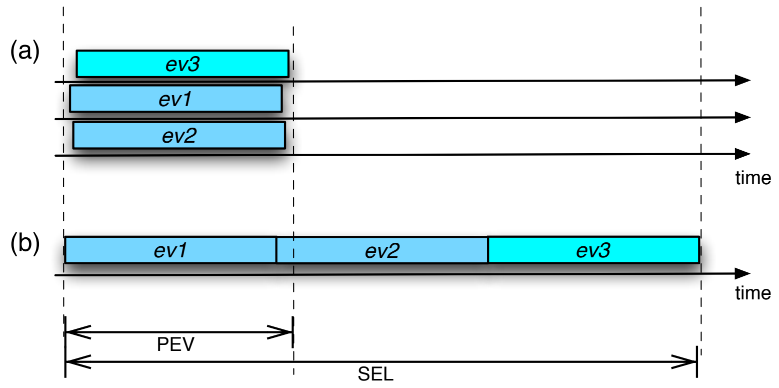


```
var loop1 = {};
var loop2 = {};

function start() {
  // callbacks are executed using "apply", passing as "this" the current target.
  this.emitChained('ev1');
}
function loop() {
  if(some_condition) {
    this.emitChained('ev1');
  }
}

loop1.on('ev0', start);
loop1.on('ev1', loop);

loop2.on('ev0', start);
loop2.on('ev1', loop);

loop1.emitChained('ev0');
loop2.emitChained('ev0');
```

Figure 3.6. Chained events. Events of type `ev1` have to synchronize with events belonging to the same event target (i.e., `ev0`), but don't have to respect strictly sequential ordering.

The `emitNow` primitive schedules an event handler for immediate execution. Consider the example depicted in Figure 3.7, corresponding to a modified version of the example in the previous section. By using `emitNow`, all the events are executed out-of-order. In other words, events of type `ev1` will not have to wait for events `ev2` to complete, and so on.

Events emitted using `emitNow` do not have to synchronize between each other. Still, they have to synchronize with other events emitted against the same event target, thus, they can be safely combined with events generated using the `emitChained` primitive. Consider the example of event emission combining strictly sequential events and chained event emission, depicted in Figure 3.8. Events emitted with `emitNow` have to wait for event `ev0` before being allowed to complete. Similarly, event `ev3` have to synchronize with events `ev1` and `ev2`. The execution scheme for this last example is



```
var obj = {};
obj.on('ev1', function() {
  var r = computeNthFibonacciNumber(10);
  print('event 1 result: ' + r);
});
obj.on('ev2', function() {
  var r = computeNthFibonacciNumber(10);
  print('event 2 result: ' + r);
});
obj.on('ev3', function() {
  var r = computeNthFibonacciNumber(10);
  print('event 3 result: ' + r);
});
obj.emitNow('ev1');
obj.emitNow('ev2');
obj.emitNow('ev3');
```

Figure 3.7. Event emission using `emitNow` running on a PEV and on a SEL system

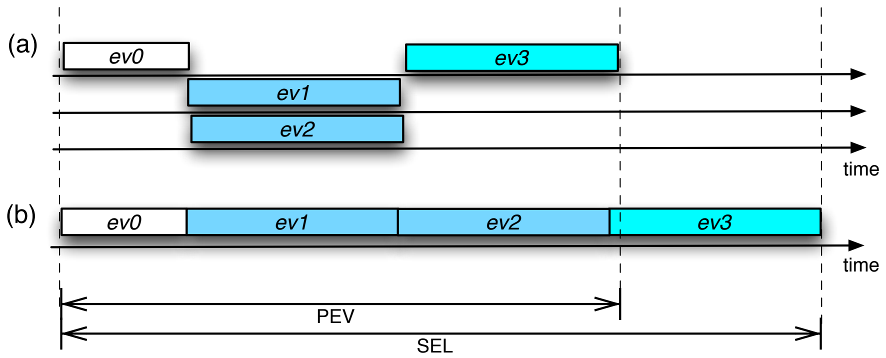
also depicted in Figure 3.8.

3.5.3 Globally unordered events

Similarly to chained events, another primitive function can be introduced to schedule events for full out-of-order execution. This can be done with the `emitUnordered` primitive:

```
Object.prototype.emitUnordered = function(label, arguments) {
  // Using global target
  for(var callback in this.callbacks[label])
      asyncNow('globalUnordered', this.callbacks[label][callback], arguments);
}
```

The primitive can be used to schedule events with global unordered scheduling: events generated with `emitUnordered` will not have to synchronize with any event in the system, including globally ordered events. The following code example presented



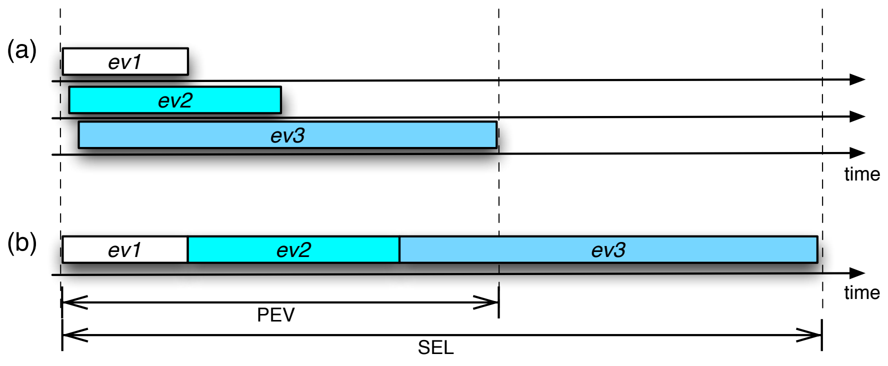
```
var obj = {};
obj.on('ev0', function() {
  var r = computeNthFibonacciNumber(10);
  print('event 0 result: ' + r);
});
obj.on('ev1', function() {
  var r = computeNthFibonacciNumber(10);
  print('event 1 result: ' + r);
});
obj.on('ev2', function() {
  var r = computeNthFibonacciNumber(10);
  print('event 2 result: ' + r);
});
obj.on('ev3', function() {
  var r = computeNthFibonacciNumber(10);
  print('event 3 result: ' + r);
});
obj.emitChained('ev0');
obj.emitNow('ev1');
obj.emitNow('ev2');
obj.emitChained('ev3');
```

Figure 3.8. Event emission combining `emitNow` and `emitChained` running on a PEV (a) and on a SEL (b).

in Figure 3.9 demonstrates this primitive by registering three event handlers and executing them. Differently from the execution on a single-threaded loop, the events will be scheduled using a non-predictable order, and retired as soon as possible, always potentially in parallel, without any need for synchronization.

3.6 Utility functions

In addition to the `async` primitives, the PEV model provides two utility functions that can be used to wait for some specific events to be executed. Such utility functions can be used to listen for some event handlers to be executed, and can be used to compose multiple event executions. As with every other event-based API, the primitives are



```
var obj = {};
// register three event handlers for three distinct events
obj.on('ev1', function() {
    var r = computeNthFibonacciNumber(10);
    print('event 1 result: ' + r);
});
obj.on('ev2', function() {
    var r = computeNthFibonacciNumber(20);
    print('event 2 result: ' + r);
});
obj.on('ev3', function() {
    var r = computeNthFibonacciNumber(30);
    print('event 3 result: ' + r);
});
// emit an event instance for each handler.
obj.emitUnordered('ev1');
obj.emitUnordered('ev2');
obj.emitUnordered('ev3');
```

Figure 3.9. Parallel event processing using `emitUnordered` on a PEV (a) and on a SEL (b).

nonblocking[3]. The first of the two functions is called `waitN` and has the following signature:

```
waitN([event target], [emission instances], [callback function]);
```

The primitive can be used to register a callback function to be executed *after* the event handler associated with a specific event has been executed on a specific event target for a given number of times. The callback is executed only once, and this is an example of its usage:

```
var obj = {}
obj.on('foo', function(n) {
  print('foo '+n);
});
```

[3]In the worst case, the registered callback will never be executed.

```
obj.waitN('foo', 2, function() {
  print('foo was emitted and consumed 2 times')
});
for(var i=1; i<6; i++)
  obj.emit('foo',i);
```

The anonymous function passed to `waitN` as an argument is called after two event emissions have been completed, and this is the output generated by this code snippet:

```
foo 1
foo 2
foo was emitted and consumed 2 times
foo 3
foo 4
foo 5
```

A second utility function provided by the PEV model is `waitEach`, which has behavior similar to `waitN` with the difference that its callback is re-triggerable. Consider the following example:

```
var obj = {};

obj.on('foo', function(n) {
  print('foo '+n);
});

obj.waitEach('foo', 2, function() {
  print('foo was emitted and consumed again 2 times');
});

obj.emit('foo',1);
obj.emit('foo',2);
obj.emit('foo',3);
obj.emit('foo',4);
obj.emit('foo',5);
```

The function passed to `waitEach` is called every time two event emissions have been completed, and this is the output generated by this code snippet:

```
foo 1
foo 2
foo was emitted and consumed again 2 times
foo 3
foo 4
foo was emitted and consumed again 2 times
foo 5
```

The callback function of the two utility functions has an important property: it is called having as an argument an array composed of all the results of the invocations of the event emission callbacks. Consider the following example:

```
var obj = {};

obj.on('foo', function(n) {
  print(n);
  return n+1;
});
```

```
obj.waitEach('foo', 2, function(results) {
  print('foo was emitted 2 times, with result '+result);
});

obj.emit('foo',1);
obj.emit('foo',2);
```

which will produce the following output:

```
foo 1
foo 2
foo was emitted 2 times, with result [2,3]
```

By exploiting the `results` array, the two utility functions can be conveniently used to synchronize between multiple event emissions, as well as to implement complex synchronization patterns and intermediate computation steps. Being the PEV model compatible with a single-threaded event loop, both functions can also be implemented in a SEL model. This is an equivalent implementation of the `waitN` function for a single-threaded runtime:

```
Object.prototype.waitN = function(label, emissions, callback) {
  this.results = [];
  var self = this;
  var originalCallback = this.handlers[label];
  this.on(label, function() {
    var result = originalCallback();
    self.results.push(result);
    if(self.results.length === emissions) {
      callback(self.result);
    }
  });
}
```

A similar semantics is also provided in the domain of single-threaded JavaScript in the form of ECMA6 promises [119].

3.7 Programming model overview

The PEV event-based programming model is to be considered a low-level instrument for deterministic and nondeterministic scheduling of events. Differently from other low-level concurrency control mechanisms such as locks and mutexes, however, the core event emission API of the PEV enforces atomicity between event handlers at the runtime level, thus avoiding data races. Complexity, as often happens, comes along with the advantage of expressiveness: in Chapter 4 we will show how the low-level API can be used to build higher-level programming models hiding the complexity of event emission behind structured programming models, including implicitly parallel programming models and structured skeletal-based models. As a reference, Table 3.3 summarizes the core API introduced in this chapter.

Method	Event scheduling
`asyncChained('global', ...)`	Schedules an event handler function with global ordering.
`asyncNow('globalUnordered', ...)`	Schedules an event handler for immediate execution.
`asyncChained(obj, ...)`	Schedules an event handler with chained ordering, synchronizing only with previously emitted events with the same event target. When multiple targets are used in the PEV, events of this class will not have to synchronize with other events from other event targets.
`asyncNow(obj, ...)`	Schedules an event handler for immediate execution with respect to other events of the current event target. Callbacks might have to synchronize with chained events of the same event target.
`waitN(...)`	Registers an ad-hoc event handler which is called after a given event handler is executed for fixed number of times, collecting the return values of each invocation.
`waitEach(...)`	Re-triggerable version of `waitN`.
`obj.emit(...)`	Emits an event with global sequential ordering. Fully compatible with the single-threaded `emit` primitive.
`obj.emitUnordered(...)`	Emits an event with global unordered scheduling. The event runs immediately and might be executed in parallel. It does not have to synchronize with any other event at all.
`obj.emitNow(...)`	Schedules an event for immediate execution. The event runs immediately and might be executed in parallel with other events. It might need to synchronize with other events emitted with the same event target.
`obj.emitChained(...)`	Emits an event chained with other events that have been emitted with the same event target. Events emitted using of distinct event targets are consumed in parallel.

Table 3.3. PEV event emission API summary.

Event #2	Event emission #1					
	`asyncChained` *Target: global*	`asyncChained` *Target: A*	`asyncChained` *Target: B*	`asyncNow` *Target: global*	`asyncNow` *Target: A*	`asyncNow` *Target: B*
`asyncChained` *Target: global*	$ev_1 \rightarrow ev_2$	*undef*	*undef*	$ev_1 \rightarrow ev_2$	*undef*	*undef*
`asyncChained` *Target: A*	*undef*	$ev_1 \rightarrow ev_2$	*undef*	*undef*	$ev_1 \rightarrow ev_2$	*undef*
`asyncChained` *Target: B*	*undef*	*undef*	$ev_1 \rightarrow ev_2$	*undef*	*undef*	$ev_1 \rightarrow ev_2$
`asyncNow` *Target: global*	$ev_1 \rightarrow ev_2$	*undef*	*undef*	*undef* *	*undef*	*undef*
`asyncNow` *Target: A*	*undef*	$ev_1 \rightarrow ev_2$	*undef*	*undef*	*undef*	*undef*
`asyncNow` *Target: B*	*undef*	*undef*	$ev_1 \rightarrow ev_2$	*undef*	*undef*	*undef*

Table 3.4. PEV happened-before relation for different event emissions. The cell marked with the $*$ symbol corresponds to the event emission discussed in the example.

3.7.1 Happened-before relation

The classes of events supported by the PEV model enable the out-of-order execution of event handlers. Some event classes have a strict ordering guarantee, whereas other event handlers do not. It is important to specify what classes of event handlers support deterministic (ordered) execution. To this end, we can define a notion of *happened-before* relation for event classes. The happened-before relation for the PEV is depicted in Table 3.4. The table presents the happened-before relation between two events scheduled by the same event handler. For example, consider the following two events scheduled using the `asyncNow` primitive[4]:

```
// Event emission #1, with target 'Global'
asyncNow('global', emission1);
// Event emission #2, with target 'Global'
asyncNow('global', emission2);
```

Depending on the event target and on the core API function used to schedule the event handler (i.e., `asyncNow` and `asyncChained`) the PEV model guarantees a different happened-before relation and semantics. The happened-before relation is specified in the Table by using the symbol "→", with the following meaning:

$ev_1 \rightarrow ev_2$: All the write operations to any object performed by the event handler#1 are visible (i.e., happened-before) from any read operation to the same object instance performed by event handler#2.

[4]The example event emission corresponds to the cell marked with the symbol $*$ in Table 3.4.

undef : There is no happen-before relation, and event handler#2 might or might not see *all* the write operations performed by event handler#1 at the moment it is executed (as the PEV model enforces atomicity).

For clarity, the table presents the happened-before relation with respect to three distinct event targets, i.e., "global", "A", and "B". In principle, however, the "global" event target is just like any other event target, with the only difference that it is the one already being used by single-threaded event loop runtimes.

Chapter 4

Case studies

The PEV API is a low-level tool for asynchronous event scheduling and synchronization. The goal of the PEV is to give a way to combine asynchronous event consumption with the advantages of the impression of a single-threaded programming model, to exploit multicore shared-memory machines. The parallel event emission API is a low-level instrument that can be used to schedule and concatenate multiple events, spanning from FIFO scheduling down to out-of-order execution. Because of its level of abstraction, the API is not intended for direct use, and in particular is not supposed to be used by application developers. Conversely, the API can be used as a layer for the development of more convenient higher-level programming models. In this Chapter we present some case studies relying on the PEV model and its API to build more advanced high-level programming model abstractions.

4.1 Event-based parallel programming

Event-based systems use the asynchronous execution of callback functions as the core programming metaphor to model the interactions with concurrent events. By relaxing the order in which callbacks can be processed, opportunities for parallel processing naturally increase, as the runtime system does not have to incorporate synchronization points into its core execution logic. Moreover, by enforcing a consistent memory model (characterized by atomicity and isolation), it is possible to further increase the abstraction of programming models that can be supported by the PEV system. According to a popular classification of parallel programming models [80], we can identify the following two main categories:

- *Implicitly parallel* models. Programming models that do not require any sort of manual parallelization by the developer, and in which the parallelization is completely and automatically carried out by the runtime system. In these models the developer can freely reason as if no concurrency was present in the system.

- *Explicitly parallel* models. Programming models that require (in multiple and very diverse ways) the intervention of the developer in order to introduce parallel execution in the application.

The PEV API can be used to derive programming models lying in both categories. Examples of implicit parallel programming models are all programming models that somehow support the automatic parallelization of an application either because of some properties of some algorithms or of some application domain (e.g., *natural parallelism*. as defined in Chapter 2), where parallelism can be naturally found, as in the case of stateless services (e.g., stateless HTTP dynamic content providers), or the case for array-based computations [126]. For all such cases the developer can freely develop the applications *as if* the application only had to accept one request at a time, without having to deal with parallelization and synchronization. The runtime system can then freely parallelize (through process replication) the service as many times as required. Similar automatic parallelization techniques can also be introduced in the context of more complex scenarios in which there is still some sort of *natural* parallelism which can be automatically exploited by the runtime (for instance through speculation).

Examples of explicit parallel programming models are many, and span from thread-level to process-level parallelism, from distributed systems to more structured approaches. Rather than giving a unified definition for such models, we can classify explicit models according to one of their main properties, that is, the use of an explicit *parallel entity*, i.e., the use of a well-defined component responsible for the execution of a parallel task in the application. By adopting this categorization, we can identify two general classes of explicitly parallel programming models:

- *Unstructured* models (relying on some *explicit parallel entities*) are all the models requiring the developer to deal with the creation and the coordination of multiple explicit parallel entities. Examples of parallel entities are threads, actors [31], agents, and processes. All such programming models are usually considered advanced models, as they usually require the developer to deal with deadlocks, data races, race conditions, or complex message-based coordination patterns. Moreover, the lack of any structured approach to parallelization might result in a suboptimal utilization of resources, as developers can hardly implement scalable applications without any structured approach[1]

- *Structured* models in which parallel entities are hidden behind more high-level, structured, forms of parallelism. Examples of such programming models are all the models belonging to the class of algorithmic skeletons [56], usually identified by the class of parallel patterns [126, 139] they use. Popular examples of

[1]As an example, identifying the grain of the task to be parallelized, as well as the ideal number of threads or processes to be started is often non-trivial. Moreover, models like the Actor model that usually are considered more simple than threads still present relevant limitations when shared state is needed by the computation, e.g., in the domain of Graph processing.

such models are MapReduce-like data-parallel computing models [64], as well as more generic scatter/gather models. Other examples of such models are parallel DSLs for processing peculiar data structures such as graphs (e.g., Green-Marl [100], or signal collect [143]).

Thanks to its speculative runtime and its parallel API, the PEV model can be employed in multiple scenarios, covering a wide spectrum of programming models from implicitly parallel ones to structured, more scalable, ones. In this chapter we present some relevant high-level applications of the model. In particular, the following case studies are considered:

- Implicit parallelism in the context of Node.JS applications and the actor model. The PEV can be used to replace the single-threaded runtime of Node.JS, enabling the speculative automatic parallelization of Node.JS applications. Similarly, the Actor model of concurrency is an explicitly parallel model in which single-threaded entities (i.e., actors) are coordinated by means of explicit message passing. Being single-threaded and stateful (with a share-nothing model not allowing to share state between actors), the actor model is perceived as a suboptimal model for parallelism in all scenarios that do not match the share-nothing abstraction [145]. In particular, this is evident when a single centralized stateful actor becomes the bottleneck of an application, as each actor cannot *internally* benefit from parallel processing. The PEV model can be used in the context of actor-like applications to solve this issue in the same way that Node.JS processes are parallelized, i.e., by using the PEV runtime to process multiple requests in parallel, speculatively, without violating the impression of the single-threaded actor model.

- Structured parallelism in the context of a single-threaded language VM. A single-threaded language VM (e.g., a JavaScript engine) having the requirement of never blocking (to ensure responsiveness either to the GUI or to incoming client requests) can benefit from the introduction of the parallel processing of callbacks either in the form of asynchronous data-parallel models, as well as in the more complex form of structured parallel programming. In this domain it is of particular interest the class of reactive applications for the development of stream processing systems. The PEV can be used in such domains for structured parallelization of reactive applications, enabling a structured asynchronous parallel programming model which we call *parallel functional reactive programming*.

4.2 Implicit parallelism in Node.JS

As discussed in Chapter 2, event-driven frameworks like Node.JS are limited by their runtime system at least in two aspects, namely (1) the impossibility of sharing a common memory space among processes (when parallelized), and (2) the difficulty of building high-throughput services using blocking, CPU-intensive, function calls.

Concerning the first limitation, the reason is that common event-based programming frameworks are not designed to express shared memory parallelism, and the only way of exploiting multiple cores is by replicating the service process. This approach forces the developer to adopt parallelization strategies based on master-worker patterns (e.g., WebWorkers [8]), which however require a share-nothing architecture to preserve the memory model of the event loop. Whenever multiple processes need to share state (e.g., to implement a stateful Web service), the data needs to be stored in a database or in a in-memory external repository (e.g., Memcached [12]), providing the necessary concurrency control.

Replication-based approaches present the limitation of inhibiting different processes from sharing a common memory space (with low latency). This could represent a programming model limitation for all the cases in which a non-persistent state is needed by the service application (for instance in the case of latency bound services that need the use of some forms of caching). With replication-based approaches (as in the case of Node.JS) the developer is forced to rely on external, additional, frameworks which however increase the response time of the service.

Concerning the second limitation, event-based programming requires the developer to deeply understand the event loop runtime architecture and to write services with non-blocking mechanisms so as to break down long-running operations into multiple processing steps. As already discussed in Chapter 2, for such services it is fundamental not to halt the loop even with simple processing tasks (e.g., sorting a small array in order to produce an answer). Due to the nature of the event loop, nonblocking processing becomes a requirement for high-throughput services, as potentially any long-lasting function call could result in a CPU-bound operation which could eventually halt the event loop. To deal with blocking method calls which could halt the event loop, the service developer has to make a massive and extensive usage of asynchronous functions and algorithms, and needs to offload any CPU-bound operation to another process[2]. Unfortunately, such mechanisms usually involve the adoption of programming techniques which increase the complexity of developing even simple services. Moreover, while non-blocking techniques help increasing the throughput of a service, they might also increase the latency of the responses. As a consequence, services often need to be developed by carefully balancing synchronous and asynchronous operations.

To solve the above two limitations without altering the single-threaded programming model, chained event emission can be used in the runtime of Node.JS: as soon as a request is received by the Node.JS service, a new event target is created for that request, and all future events belonging to that specific request will be processed with the same event target.

[2]Consider the example of sorting an array discussed in Chapter 2

By adopting this parallelization strategy, the service can be automatically parallelized, having multiple requests processed in parallel without any manual parallelization, as the PEV acts as a *fork* for every incoming request. The speculative runtime of the PEV ensures that requests that will alter some shared state will be processed in a safe way, and provides scalability in the case of stateless services, whereas in the case of stateful services the speculative engine might anyway be able to increase the throughput of the system without having the developer to introduce explicit parallelization.

The simplest Node.JS service is usually presented with the following code:

```
// Create a new service listening on port 8080
http.createService(function(request, response) {
    // Reply back to the client
    response.end('Hello world');
}).listen(8080);
```

The service registers a callback function, which is successively executed upon any incoming client request. The majority of the Node.JS runtime is written in self-hosted JavaScript. The framework-level source code of the previous example is implemented in the following way[3]:

```
this.on('connection', function(socket) {
  // create a buffer to store incoming data
  var buffer = runtime.createInputBuffer();
  var self = this;
  // register an handler to accumulate incoming data
  socket.on('data', function(data) {
    if(data != '\n\r') {
      // accumulate data chunks
      buffer.push(data);
    } else {
      // once the data has been received, notify
      self.emit('newRequest', buffer);
    }
  });
});
```

Thanks to the PEV parallel engine, the Node.JS single-threaded runtime can be converted to an implicitly parallel framework in which each request is potentially processed in parallel. This is depicted in Figure 4.1. The simple parallelization technique does not affect the Node.JS programming model, and therefore does not require the service developer to deal with the manual parallelization of the service. Conversely, the developer can comfortably use the Node.JS framework having the impression of developing the service for a single-threaded execution engine. When multiple concurrent requests will be received, the PEV runtime will attempt to speculatively process them in parallel, leading to an implicitly parallel programming model. Such implicit parallelization approach would also allow for latency-oriented optimizations (rather than throughput-oriented) through parallelization. In fact, a relevant advantage of

[3]The code corresponds to a simplified version of the original Node.JS source code. The characteristics of this example have already been discussed in Chapter 2.

```
// PEV-enabled request handling
this.on('connection', function(socket) {
  // create a buffer to store incoming data
  var buffer = runtime.createInputBuffer();
  // "self" is a per-connection instance
  var self = this;
  // register an handler to accumulate incoming data
  socket.on('data', function(data) {
    if(data != '\n\r') {
      // accumulate data chunks
      buffer.push(data);
    } else {
      // Emit a chained event with per-request target: multiple requests will
      // potentially execute in parallel
      self.emitChained('newRequest', buffer);
    }
  });
});
```

Figure 4.1. Implicitly parallel Node.JS using the PEV event emission API.

PEV over the existing Node.JS engine is represented by the possibility of optimizing new and existing services for latency, and not only for throughput, whereas current approaches to parallelization of Node.JS-based services all rely on the mere replication of the JavaScript execution engine.

4.3 WebWorkers with safe shared state

On single-threaded frameworks like Node.JS the only way to support CPU-bound computations is by offloading the computation to another worker process (WebWorkers [8], in JavaScript), at the cost of losing shared state. This limitation can represent a relevant issue for services featuring in-memory data structures (e.g., a cache) often used to reduce the service's latency. This limitation can be solved in the PEV model by introducing a form of WebWorkers (called `SharedWorkers` in this example) that can safely access and modify shared data structures. Like standard WebWorkers, SharedWorkers can interact through the `onMessage`/`postMessage` API; unlike WebWorkers, they can also safely access shared state. Consider the example of two workers concurrently reading and modifying a shared array, depicted in Figure 4.2[4].

The code in the example describes a possible scenario for safe workers. A Web service needs to perform some computation involving shared state in order to produce the response (e.g., searching in a shared data structure and collecting some statistics, as in the example). A dedicated worker can be used to perform the computation (line 4), while the service can process other requests. If needed, other workers can be

[4]The HTTP server initialization is omitted for brevity.

```javascript
// Init an http server
var http = require('http');

// the shared array
var shared = new Array(...);

// First worker scanning the array
var w1 = new SafeWorker();
// Second worker updating the array
var w2 = new SafeWorker();

w1.onMessage(function(message) {
  var total = 0, max = shared[0], min = shared[0];
  // The array can be accessed by the worker
  for(var i=0; i<shared.length; i++) { |\label{w1}|
    total = total + shared[i];
    max = _max(max, shared[i]);
    min = _min(min, shared[i]);
  }
  message.response.writeHead(200, {"Content-Type": "text/plain"});
  message.response.end(total+"\n"+max+"\n"+min);
})

w2.onMessage(function(message) {
  var id = someHashing(message.data);
  shared[id] = message.data; |\label{w2}|
  message.response.writeHead(200, {"Content-Type": "text/plain"});
  message.response.end(id);
})

// Simple Node.JS service handling requests: GET will read, PUT will update
http.createServer(function(request, response) {
  // Get one of the two workers, depending on the type of request
  var worker;

  // Parse the request details from the client
  var data = parseMessage(request);

  if (request.method == 'GET') worker = w1;
  if (request.method == 'PUT') worker = w2;

  // We send to the worker also the "response" object, to let him reply to the client
  if (worker)
    worker.postMessage({data:data, response:response});
  else
    response.status = 405;
}).listen(port)
```

Figure 4.2. Safe WebWorkers implemented using the PEV.

started with different tasks (e.g., storing new data in the shared array, at line 18). SharedWorkers can be implemented in the PEV using the following event emission mechanism:

```
var SharedWorker = function(receiver) {
  this.on('message', this.onMessageCallback);
  this.receiver = receiver;
}

SharedWorker.prototype.onMessage = function(callback) {
  this.onMessageCallback = callback;
}

SharedWorker.prototype.send = function(message) {
  this.receiver.emitUnordered('message', message);
}
```

Despite having a similar API, SharedWorkers differ from standard WebWorkers in the following aspects:

- By using `emitUnordered` multiple requests to the same Worker might be processed in parallel, and the order in which requests will be processed is not deterministic.
- The message handler of the safe worker is guaranteed to be executed atomically and in isolation, but has access to all the objects in the scope of its event handler at the moment it is created.
- When event handlers can be executed in parallel (that is, assuming unordered or chained events are used by the worker internal implementation) the worker might process multiple requests in parallel. Making the worker equivalent to an actor that self-parallelizes itself when more than a single request is received.
- As with standard WebWorkers, interactions happen via message passing. However, since message-based interactions in the PEV correspond to event emissions, the `on/emit` API allows developers to specify interaction patterns different than $1 : N$ communication. Moreover, send-by-reference can be supported efficiently, when required.
- When Workers have conflicts, however, the two workers cannot offer ideal scalability, as standard workers would. This comes at the price of providing safe shared state.

4.4 Asynchronous tasks and events

The PEV API can be used to derive common task-based programming models. In this section we describe a PEV-based version of some popular task-based programming models, and we discuss their event-based implementation.

4.4.1 Asynchronous safe futures

Many programming languages feature a notion of future tasks [119][5]. Futures that can safely run in parallel have been proposed for Java [153], and can be introduced in JavaScript using event emission. To this end, the `Future` object can be introduced:

```
// Create a future object
var future = new Future(fun)
```

An utility function called `spawn` can be used to schedule the asynchronous safe future:

```
// Execute the 'fun' function with arguments 'args'
// asynchronously and potentially in parallel
future.spawn(args, function(result) {
    // 'result' contains the return value
    // for 'fun(args)'
})
```

In the example, a future object is created using the `Future` constructor, which accepts a function to be executed asynchronously (i.e., `fun`) as an argument. The function can then be scheduled for asynchronous execution through `spawn`. Once the function will have completed, the result will be the argument to the callback function provided as the second argument.

Since there is no ordering requirement for a single future object, the `spawn` primitive can be implemented using the `emitUnordered` primitive. By using the future object instance as its event target, the future is guaranteed not to have to synchronize with other event handlers:

```
var Future = function(toRunInParallel) {
  var self = this;
  this.on('go', function(args) {
    var result = toRunInParallel(args);
    self.emitUnordered('done', result);
  });
}

Future.prototype.spawn = function(args, onDone) {
  // register the callback
  this.on('done', onDone);
  // The function 'toRunInParallel' has already been stored in the object instance when
       the future was created
  this.emitUnordered('go', args);
}
```

[5]A specification for future-like objects in JavaScript has also been proposed [5]. However, being JavaScript a single-threaded language, the proposed API can be only used to model the asynchronous interaction with entities external to the main JavaScript event loop such as other services, the Browser, or the Operating System

4.4.2 Task-based parallelism

Task-based parallelism can be supported asynchronously by the PEV runtime. Since event emission is asynchronous, it can be easily composed, meaning that it is possible to emit events from other events, like in the following example of an infinite event emission.

```
var foo = { data : 0 }

foo.on('event', function() {
    this.data++;
    this.emitNow('event');
});
// Trigger the first event emission.
foo.emitNow('event');
```

Recursive event emission can be used to extend the Array object prototype with some asynchronous parallel builtin operations, in the following way:

```
// Create a new array in parallel with the squared root of the input array
[1,2,3].mapPar(function(idx, element) {
  return Math.sqrt(element);
}, function(result) {
  // once completed, 'result' will hold the new array (created potentially in parallel)
})
```

The *mapPar* function can be considered an asynchronous version of the equivalent synchronous RiverTrail's one. The function can be implemented in the PEV by combining chained and unordered event emission, as presented in Figure 4.3[6]:

Since the `map` tasks are expected to run in parallel, the `emitNow` event emission is used. The `chunkDone` event is used to accumulate the partial results as computed by parallel tasks. Events of this type are emitted using `emitChained` as it is very likely that such events will not be able to run in parallel (as they will all modify the same object to accumulate the partial results). The chained event emission is bound to an event target instance (i.e., the array itself) which is private to the `mapPar` computation. This means that two (or more) successive invocations of the function will potentially cause multiple computations to be executed in parallel:

```
var a1 = new Array(...); var a2 = new Array(...);

a1.mapPar(sqrt, onDone);
a2.mapPar(sqrt, onDone);
```

In this example, both computations for `a1` and `a2` will be executed in parallel, and will execute the `onDone` function nondeterministically.

Multiple `mapPar` function calls can be combined even further to derive a parallel tree processing function. Assuming a tree is expressed as a combination of arrays (or JSON objects), a function called `visitTree` can be introduced to perform parallel

[6]For simplicity, we assume the array to be divisible by a factor of 2. Therefore, the source code does not show bound checks, overflow management, etc.

```
doInParallel = function(inArray, sequentialKernel, threshold, onDone) {
  function doSequentialKernel(from, to) {
    for(var idx = from; idx<to; idx++) {
      sequentialKernel(idx, inArray[idx]);
    }
  };
  var self = this;
  var result = { finalResult : [], missing:this.length };
  this.on('task', function(from, to) {
    if((from-to) < threshold) {
      var chunkResult = doSequentialKernel(from, to);
      result.emitChained('chunk', chunkResult, from, to);
    } else {
      self.emitNow('task', from, (from+to)/2);
      self.emitNow('task', (from+to)/2, to);
    }
  });
  result.on('chunk', function(chunkResult, from, to) {
    for(var i=from; i<to; i++) {
      this.finalResult[i] = chunkResult[i];
    }
    if(result.missing-- == 1) {
      onDone(this.finalResult);
    }
  });
  this.emitNow('task', 0, this.length);
}

Array.prototype.asyncMap = function(mapFun, onDone) {
  // The threshold can be a factor of the cores in the system and the size of the array
  var threshold = someFactorOf(System.CPU(), this.length);
  doInParallel(this, mapFun, threshold, onDone);
}
```

Figure 4.3. Implementation of an asynchronous MapReduce API using the PEV.

tree traversal. Like with the `mapPar` function, `visitTree` can be implemented using unordered event emission. A reference implementation for such `visitTree` function with an example of usage is presented in Figure 4.4.

4.5 Parallel functional reactive programming

Reactive applications are applications that have to be deployed within time-varying contexts in which any external stimuli might influence the application's behavior. Functional reactive programming [131, 127, 73, 60] (FRP) is a programming model for the development of reactive applications based on pure (i.e., side-effect-free) functions and events. FRP allows developers to model systems that must respond to input over time in a simple and composable way. The primary concepts of FRP are *signals* (also called behaviors, that is, time-varying values) and *events*, corresponding to collections

```javascript
var treeVisitor = {};
treeVisitor.on('visit', function(nodes, index, visitor, done) {
    if (index < nodes.length) {
        visitTree(nodes[index], visitor, function () {
            visitNodes(nodes, index+1, visitor, done);
        });
    } else {
        // Optional "done" function, not used in this example
        done();
    }
});

function visitTree(tree, visitor, done) {
    if (Array.isArray(tree)) {
        // The parallel computation is triggered here
        for(var i=0; i < tree.length; i++) {
            treeVisitor.emitNow('visit', tree, i, visitor, done);
        }
    } else {
        visitor(tree, done);
    }
}

// Traverse the given tree in parallel
visitTree([[1,3,[2]],[6,3],[18,10,83]], function(node) { console.log(node); });
```

Figure 4.4. A parallel tree traversal API expressed in the PEV.

of instantaneous values, or time-value pairs. Events are used to model any form of interaction with external entities (i.e., a mouse click), and behaviors are used to *react* to the external event producing a change in the system's state. Among the different implementations of FRP-inspired models that have been proposed in the literature, reactive models have become popular in languages featuring some notion of a builtin event loop. Examples of languages for which reactive programming frameworks exist are JavaRX [19], Akka's Scala [1], C# [144], and of course JavaScript [17]. In many of the libraries for FRP in JavaScript (e.g., Rx.JS [13]), a reactive application is presented using examples like the following:

```javascript
var mousedrag = mousedown.selectMany(function (md) {
  // calculate offsets when mouse down
  var startX = md.offsetX, startY = md.offsetY;
  // Calculate delta with mousemove until mouseup
  return mousemove.select(function (mm) {
    (mm.preventDefault) ? mm.preventDefault() : event.returnValue = false;
      return { left: mm.clientX - startX, top: mm.clientY - startY };
  }).takeUntil(mouseup);
});
// Update position
subscription = mousedrag.subscribe(function (pos) {
  dragTarget.style.top = pos.top + 'px'; dragTarget.style.left = pos.left + 'px';
});
```

The simple example is used to implement a drag-and-drop function for controlling the mouse in JavaScript using FRP. The code does not include any explicit use of additional data structures to keep track of the mouse position, and makes use of a simple *reactive* abstraction which assigns to every event (e.g., "mouse move") a function to be invoked. Events are then considered streams to be subscribed (using `subscribe`) and observed (using `select`). Examples like this implicitly rely on the fact that events are strongly executed one after the other, as it would not make sense to start executing the *mouse up* event (associated with the end of the drag-and-drop) before all the *mouse move* events have been consumed (or even before the event *mouse down* has been processed). Interestingly, this is not always the case with reactive programming, and relaxed event processing can be introduced also in this domain. Beyond the simple drag-and-drop example, reactive applications are commonly used for the development of services that have to deal with streams of data. An example of reactive data processing is represented in the following code snippet, showing how a data file can be retrieved from the web and processed chunk-by-chunk as data is received.

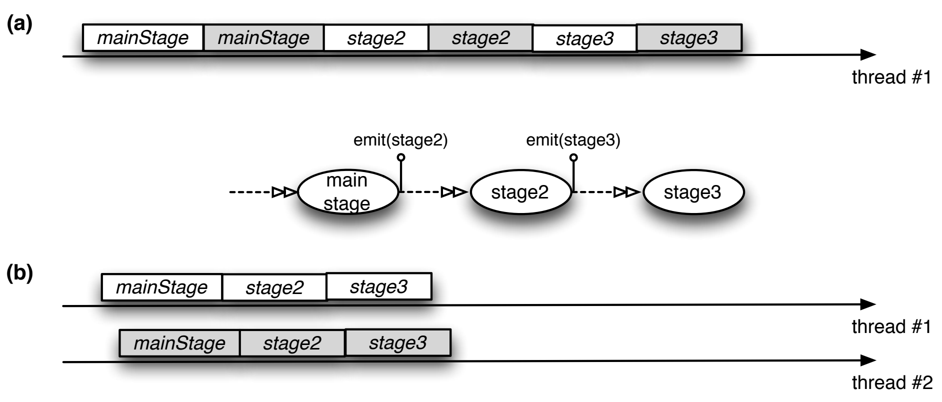
```
var maximum = 0;
http.get('http://yahoo.finance.com/stocks?AAPL\&from=..to=..')
  .asStream(function(chunk) return parseInt(chunk.split(" ")))
  .map(function(number) {
    if(number > maximum) {
      maximum = chunk;
      return maximum;
    }
  })
  .subscribe(function(max) {
    print('got a new local maximum '+max);
  });
```

Conceptually, the `asStream` function causes an event emission for every new data chunk received by the HTTP query, and each emission will cause the `map` function to be executed. In a PEV system, the `map` function will automatically start being executed in parallel. For every element returned by the `map` function, the successive callback (registered using `subscribe`) is called. As the callback is called only wheb the `map` callback returns a value, it can be used to filter elements.

Infinite streams can be processed in a similar way. Of course, this is particularly convenient when the process of consuming data also involves some CPU-intensive computation. Consider the following example:

```
var s1 = open("http://www.twitter.com/...");
var s2 = open("http://www.facebook.com/...");
join([s1, s2]).asStream()
     .map(function(data) {return process(data)})
     .onDone(function(data) {print(data)})
```

In a single-threaded loop model the example above would be affected by a lack of parallelization, preventing the loop from accepting other requests as long as the main thread is performing the processing phase. This is however not the case in a PEV



```
function Pipeline() {
  this.stages = [];
  this.firstStage = {};
}

Pipeline.prototype.addStage = function(fun) {
  var newStage = {};

  newStage.on('element', function(data) {
    var result = fun(data);
    if (this.nextStage) {
      this.nextStage.emitNow('element', result);
    }
  });

  var prevStage = this.stages.length;

  if (prevStage == 0) {
    this.firstStage = newStage;
  } else {
    this.stages[prevStage - 1].nextStage = newStage;
  }
  this.stages.push(newStage);
  return this;
}

Pipeline.prototype.open = function(url) {
  var pipeline = this;
  Stream.open(url).onData(function(streamElement) {
    // Each element of the stream is sent to the first stage in the pipeline
    pipeline.firstStage.emit('element', streamElement);
  });
  return this;
}

// Example usage:
var pipe = new Pipeline()
pipe.addStage(mainStage)
  .addStage(stage2)
  .addStage(stage3)
  .open('http://www.some.data.service.com/');
```

Figure 4.5. A software pipeline with three stages.

system, where a chained event emission of events can be used to spawn the concurrent processing of a data chunk as soon as it is received. It is important to note that the event loop is not blocked, and therefore other reactive applications can still run in the meantime. Similarly to this example, other continuous streaming processing applications could be written in the following way:

```
var bigDocument = "http://en.wikipedia.com/...";
var wordDistribution = [];

open(bigDocument).asStream()
  .map(function(chunk) {
    updateTable(wordDistribution, chunk.split(" "));
  });
```

In this example a simple map-reduce computation is used to count the distribution of words in a table. Differently from the common (distributed) implementation of the map-reduce job, the example is making use of a shared data structure which is updated in-place.

4.5.1 Software pipelining

Relaxed events can be conveniently combined to build dataflow-like event-based computation models that can automatically take advantage of the PEV parallel execution model.

Let's assume that a given computation could be expressed as a sequence of three distinct stages (that is, three operations that have to be executed sequentially), and that the computation can be exposed using the following reactive API:

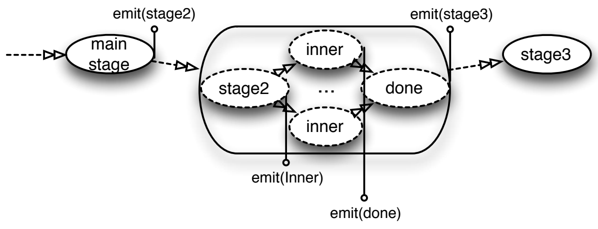
```
// Define a new pipeline
var pipeline = new Pipeline();
// Define the pipeline
pipeline.addStage(function(element) { return mainStage(element); })
        .addStage(function(element) { return stage2(element); })
        .addStage(function(element) { return stage3(element); });
// Start the pipeline with a streaming source
pipeline.open('http://www.some.streaming.service.com/');
```

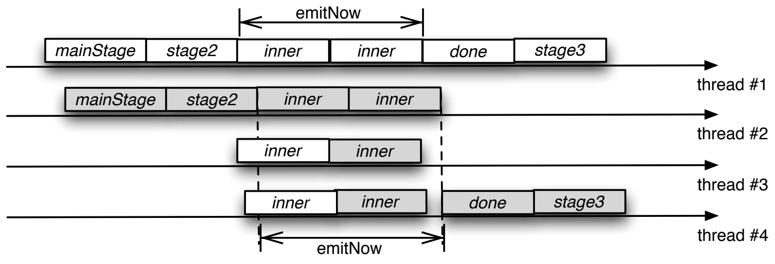
The three stages can be modeled as three distinct event handlers, as depicted in Figure 4.5[7].

The three stages are combined in a chain of multiple events, which are interconnected using unordered event emission. Every time a new element enters the pipeline (that is, a new stream element is received) an event emitter corresponding to a stage in the pipeline emits an event using `emitNow`.

On a single-threaded event loop the natural ordering of event processing will result in a sequential execution of each event handler, which will emit new events while

[7]For simplicity, we assume the pipeline to be *balanced*, i.e., a pipeline in which each stage has the same execution time (on average). Balanced pipelines are commonly found in many streaming computations [55].





```
Pipeline.prototype.addParStage = function(fun) {
  var newStage = {};
  if(Array.isArray(data)) {
    var stage = this; var scatterStage = {};
    scatterStage.on('data', function(elem) {
      var parallelStage = {};
      var results = [];
      parallelStage.on('inner', function(elem) {
        // Store each result in the array
        result[i] = fun(elem);
      });
      parallelStage.on('done', function(elem) {
        // Result contains the final result
        if(stage.nextStage)
          stage.nextStage.emit('data', result);
      });
      for(var i=0; i<elem.length; i++)
        parallelStage.emitNow('inner', elem[i]);
      // Chained emission ensures that the 'done' event
      // will be executed after all the 'inner' handlers
      parallelStage.emitChained('done');
    });
    newStage.on('element', function(data) {
      scatterStage.emit('data', data);
    });
  } else
    newStage.on('element', function(data) {
      var result = fun(data);
      if (this.nextStage)
        this.nextStage.emit('data', result);
    });
  var prevStage = this.stages.length;
  if (prevStage == 0) {
    this.firstStage = newStage;
  } else {
    this.stages[prevStage - 1].nextStage = newStage;
  }
  this.stages.push(newStage);
  return this;
}
```

Figure 4.6. A software pipeline with one of the three stages internally parallelized using a task farm.

completing its execution. By using a dedicated event emitter per each stage in the pipeline, however, the same application executed using the PEV will process multiple stream elements in parallel, as depicted in Figure 4.5. The parallel event loop automatically schedules events as soon as they enter the pipeline, eventually leading to a full utilization of the computing resources in the system (2 cores, in this example). As long as stages do not conflict, the efficiency of the system is guaranteed to be optimal as the pipeline is balanced [55].

Balanced pipelines usually correspond to an ideal subclass of all the possible cases. More frequently, one or more stages in a pipeline have an higher execution time, thus resulting in a bottleneck for the whole pipeline. A common solution to bottlenecks on a single stage is *inner* parallelism, that is, the parallel execution of the internal computation carried out by a single stage. Assuming the bottleneck is identified in the second stage and assuming its computation can be parallelized just by splitting its input into multiple elements (i.e., it operates on an array-like data structure), the pipeline object can be extended with a *parallel* stage. The stage can be implemented using a combination of `emitNow` and `emitChained`, and internally consumes events in parallel, as depicted in Figure 4.6.

This parallelized version would lead to an automatic parallelization of the pipeline.

By introducing unordered events (through `emitNow`) in the second stage, the pipeline is able to perform the internal computation for the stage in parallel. Chained ordering ensures that all the events of type 'done' will execute just waiting for their predecessor to complete, without having to wait for their peers (that is, they all terminate as soon as possible). This execution scheme ensures full parallelism for the given pipeline (in the case of non-conflicting event handlers). The emission of events with chained ordering to the next stage will cause the pipeline to restore the global ordering once all the chained events (for a single `element` value) have been processed.
As discussed, the pipeline can be exposed to the PEV developer using the following reactive-like API, hiding the complexity of the event-based implementation. A fixed three stages pipeline with an internal parallel stage (called a TaskFarm [55]) would have the following usage:

```
function preprocess(element) { return someFunctionOf(element); }
function postprocess(element) { return someFunctionOf(element); }
function taskFarm(element) { return someFunctionOf(element); }

var pipe = new TaskFarmPipeLine();

pipe.addStage(preprocess)
    .addParStage(taskFarm)
    .addStage(postprocess)

pipe.open('http://www.some.streaming.service.com/');
```

Every time a new element from the incoming stream will be emitted, the pipeline will automatically start processing events in parallel. Since some stages might access shared state or have side effects, the PEV could also be used to obtain a programming model combining data-flow programming and side effects similar to the one presented in [82, 83].

Part III

Speculative Runtime

Chapter 5

The parallel event loop runtime

The parallel event loop API is not bound to a specific runtime architecture. In this chapter we introduce three distinct implementations of a PEV runtime system, and we discuss the characteristics of each implementation. Each implementation is based on some notion of event *scheduling* and *speculative execution*. Scheduling is used by the runtime to ensure the correct event execution order as prescribed by the PEV API, while speculative execution is used to overlap as much as possible parallel event execution, always ensuring safety. The runtimes presented in this chapter are based on pessimistic (lock-based) and optimistic (STM-based) speculative parallelization.

5.1 Runtime system overview

Any runtime supporting the PEV API can be seen as a fixed-size thread pool executor service processing only a specific class of executable tasks, that is, event handlers. Differently from popular executor services (e.g., the Java ForkJoin Pool [110]), tasks submitted to the parallel event loop for execution might have to respect a constrained execution order. Moreover, event handlers always have to be transparently executed atomically and in isolation.

In this chapter we describe three implementations of runtimes supporting partially or completely the PEV API. The high-level execution scheme for the three PEV runtime systems can be introduced with the scheme depicted in Figure 5.1.

Events are produced either by the I/O substrate (e.g., by the Operating System) or by the event emission API (a), causing one or more event handlers to be scheduled for execution (b). Depending on the type of the event (e.g., chained vs. unordered) and on other runtime parameters (e.g., event target) events are submitted for direct execution to a specific thread. Right before execution, any event handler is automatically modified with specific *runtime barriers* implementing the runtime system enabling the safe parallel execution of the handler (c). Depending on the parallel runtime, some event

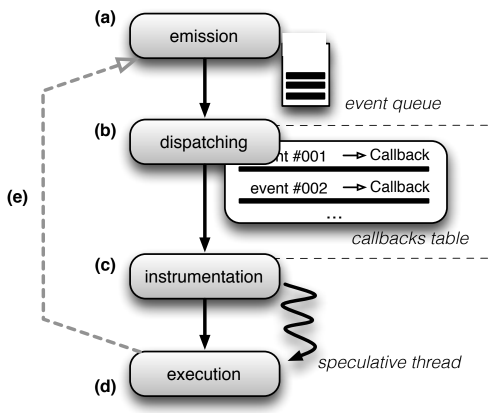


Figure 5.1. Parallel Event Loop general execution scheme.

handlers can be modified ahead-of-time (e.g., at parsing time), and -for frequently used handlers- modified call targets can be cached. Once an event handler is selected for execution, the worker thread executes it by means of a *speculation* engine which ensures safe parallel execution. Based on the speculation policy and on other runtime-level information, the engine might execute the handler (usually in parallel), or might abort the handler execution (d) and re-execute it in the same thread or on a different one. Once the event handler has finally been executed (e), its side effects (including new events that it might have emitted) become visible to other event handlers. The speculation engine ensures that side effects publication is consistent with the correct event scheduling[1].

There are two crucial aspects of each event handler execution, namely *ordering* and *safe execution*:

- *Event ordering*. The correct event execution order has to be enforced by the PEV runtime. The correct execution ordering is enforced in several moments during event execution. In particular, every worker thread makes use of various information (e.g., shared metadata, queues policies) to take local decisions on which event handler to execute.

- *Safe execution*. Once selected for parallel execution, every handler might run in parallel with other handlers, with potential conflicting access to shared object instances. To prevent inconsistent access to shared data, data races and race conditions, every event handler has to be executed atomically and in isolation. To this end, handlers are executed through a *speculative runtime* ensuring such

[1]In other words, the engine takes care of publication safety, also with respect to the system memory model

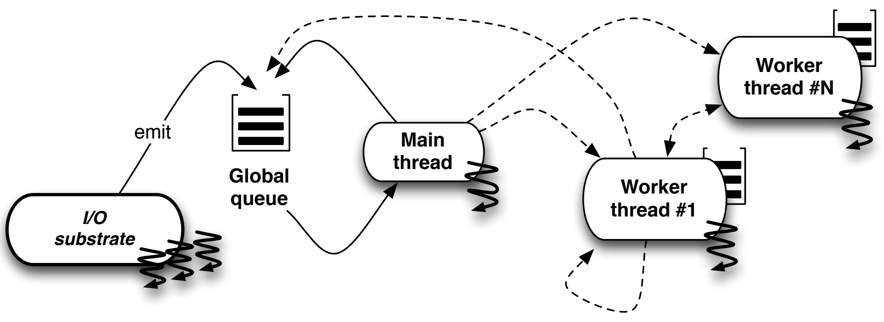


Figure 5.2. Parallel Event Loop runtime high-level architecture. Continuous arrows between the main thread and the I/O substrate indicate that all I/O events are executed by the main thread. Dashed arrows indicate that relaxed event handlers may be executed by other workers in parallel.

properties. The speculative runtime is a runtime component that tries to execute handlers in parallel, taking care of conflict resolution between concurrent handlers. Speculation can be implemented using multiple techniques.

Depending on the PEV runtime, the two operations can be implemented in different ways. All the implementations presented in this Chapter feature a runtime system that is embedded within the language runtime (i.e., the language VM or engine), and share the same high-level system architecture, which is depicted in Figure 5.2. The following main components can be identified in the system:

- *I/O substrate.* Runtime layer dealing with I/O operations, responsible for scheduling callbacks from any event source executed outside of the language runtime (e.g., the Operating System). For compatibility with single-threaded loops, events emitted by this runtime are always globally ordered events.

- *Global event queue.* The main event queue for I/O operations. The queue is always used by the I/O substrate to emit globally ordered events. Depending on the PEV runtime, the queue can also be used to support other event types.

- *Worker (and main) threads.* Threads responsible for modifying event handlers and for executing them in parallel ensuring isolation and atomicity. The number of threads is fixed and is usually close to the number of hardware threads available in the system. A privileged thread called the *main* thread is usually responsible for processing event handlers emitted by the I/O substrate through the global queue. Depending on the PEV implementation, the main thread can also be responsible for processing other event types.

- *Thread-local event queue*. Every thread also manages a thread-private queue. The queue usually contains event handlers that are ready for processing. Depending on the PEV implementation, thread-local queues can also be leveraged to enforce local ordering as well as to implement load balancing via work stealing [110]. In order to enforce the correct ordering, each runtime implements work stealing only for event handlers with relaxed concurrency.

The need for a clean separation between the main thread a worker threads is also motivated by the fact that the PEV system has to ensure compatibility with the single-threaded event loop.

5.2 Speculation engines

Event scheduling and speculative execution can be implemented using several approaches. In this Dissertation we describe and analyze the following three distinct implementations:

- *Pessimistic runtime*. An implementation based on pessimistic lock-based concurrency control. Fine grained locking and a global reader-writer lock are used to enforce atomicity and isolation. Scheduling is obtained by exploiting queues, without using metadata. Globally ordered events are processed by the main thread only, while unordered and chained events are processed by worker threads. Different event targets can be processed in parallel, while two unordered events belonging to the same event target cannot benefit from per-target parallelization (excluding the global target). The main performance characteristics of this implementation are its reduced execution overhead, balanced with good scalability for read-only and side-effects-free (pure) workloads.

- *Optimistic runtime*. An implementation based on a fully-optimistic approach in which every callback is executed in a dedicated software transaction [91], thus providing atomicity and isolation. Additional metadata for validation and commit-time synchronization is used to enforce the correct event execution ordering, and a "transactions everywhere" approach gives the system the ability to execute speculatively any type of event handlers, including globally ordered ones. This implementation presents an increased runtime overhead, but also offers better scalability for high-contention workloads.

- *Hybrid runtime*. An implementation combining aspects of the previous two approached by mixing optimistic worker threads with a pessimistic main thread. With this runtime, all the ordered event handlers belonging to the `global` event target are executed by a main thread using very *light* runtime barriers. All other event handlers (including out-of-order handlers belonging to the `global` event

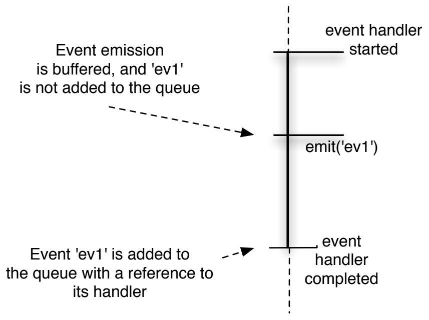

event handler started
Event emission is buffered, and 'ev1' is not added to the queue
emit('ev1')
Event 'ev1' is added to the queue with a reference to its handler
event handler completed

Figure 5.3. Buffered event emission in a single-threaded event loop.

target) are executed by the workers. Event handlers executed by the main thread always commit, reducing the overhead (for single-threaded execution) while increasing the probability of conflicts with other non-global event handlers. Event handlers (executed by workers) that are found to have an high abort rate are re-scheduled to the main thread, thus preventing the system from degrading its performance when the speculation is not effective. This speculation engine offers a limited execution overhead with respect to non-modified single-threaded event loops, still offering good scalability for read-dominated workloads. The speculative runtime is based on the FastLane STM algorithm [151] but other STM algorithms featuring a notion of *irrevocable transaction* [154] might be supported as well. As with the Optimistic system, additional commit-time reordering and synchronization is implemented to enforce correct event ordering.

Each of the PEV systems presented in this chapter has been designed to meet the following goals:

- *Single-threaded loop performance*. When running only globally ordered events (as with a single-threaded loop) the system shall provide acceptable execution overhead.
- *Server-side workloads scalability*. Being the PEV mainly targeted at server-side workloads, a PEV system shall be scalable for workloads that are common in the server-side domain. In particular, read-only workloads and stateless workloads.
- *Speculation effectiveness*. The engine shall be able to improve the single-threaded execution overhead any time the workload (i.e., the handlers pending in the queue) allows for potential parallel execution.

The three implementations discussed in this Chapter share some common components and aspects. In the following subsections we describe their main characteristics.

5.2.1 Event emission

In a single-threaded event loop system, events emitted by the I/O substrate or by the main thread are added to the global event queue as soon as they are produced. Being the runtime single-threaded, such events will start being consumed only once the event handler currently running will have returned. This is equivalent to buffering all event emissions as they happen, and flushing the buffer once the current event handler completes (see Figure 5.3).

The PEV runtime adopts the same buffered event emission model. The main reason is that event buffering and flushing is compatible with any STM-based speculation engine, and events emitted by any parallel event handler (running in a transaction) can be actually emitted only once the transaction is allowed to commit. In other words, event emission is considered a special class of side effects that does not need to be validated, and has the need to become public only *after* the current event handler has completed.

Depending on the ordering requirements of events that are emitted, the event loop implements distinct execution strategies. However, event scheduling always implements the following high-level scheduling rules:

- Since globally ordered events must appear as if they were executed in the emission order, any side effect produced by an event handler must be visible to successive events, and must not interfere with handlers executed previously. Speculation can be used to overlap multiple ordered event executions, but the order in which side effects are made public must be respected.
- As with strictly ordered events, chained ones must be executed in the order they are emitted. Differently, multiple chained events can be interleaved, meaning that side effects produced by events of two distinct event targets can potentially be seen by other events even out-of-order.
- Unordered events can be executed as soon as possible. Given the absence of any ordering, the system can arbitrarily re-schedule event handlers in the most convenient way.

The above scheduling rules are implemented independently from the speculation runtime.

5.2.2 Hashed event emission for chained events

Depending on the PEV runtime, event emission can be implemented very differently for different event types. Usually, globally ordered events are added to the global queue, while chained events are added to worker threads. With the goal of reducing synchronization costs and increasing locality, each PEV runtime adopts a simple scheduling policy for chained event handlers, which we call *hashed emission*: every event target is bound to a specific worker thread, and every time a chained event belonging to that event target is emitted it is scheduled by the runtime into the local queue of the same worker thread. In this way, the local queue of the worker thread can be used to enforce the ordered execution of chained event handlers. Assigning event targets to threads is done using a simple hashing function relying on the fact that the number of threads in the PEV is fixed, and every worker thread can be identified with a unique id.

5.2.3 Event emission and I/O

The asynchronous nature of the event loop is of great advantage for including a speculation engine in the runtime, as the only operation that needs to be treated as a special case is event emission. In fact, a failed speculative execution (e.g., an aborted 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 PEV API model does not assume that events are actually emitted immediately when any of the `async` functions 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 after commit-time. Hence, during the execution of an event handler, emitted events are buffered in a thread-local log. Only after the handler has successfully completed, the event log is processed and deferred events are safely emitted for parallel processing.

In a system with everything mediated by a speculation engine it becomes crucial to properly handle all the deterministic blocking operations which cannot be re-executed in case of a speculation failure. This is usually the case with I/O operations such as blocking I/O operations, as well as standard output operations. Fortunately, the case for the event loop is simpler, as no blocking operation is usually supported (and I/O operations already happen asynchronously, that is, after event handlers have been executed). Therefore, all I/O operations in the PEV play well with speculative systems, as they are already happening *outside* of event handlers, in the I/O substrate. As an example, consider the following event handler:

```
obj.on('dataFromIOSubstrate', function(data) {
  // Do something with 'data', potentially in parallel...
  console.log(data);
});
```

The `dataFromIOSubstrate` event handler receives some data from the I/O substrate, and is executed by a speculation engine. Assuming optimistic execution using an STM runtime, the handler might be aborted and re-executed multiple times during its execution. Aborting the handler multiple times, however, does not involve any additional operation from the I/O substrate to manage the `data` buffer, as it was already emitted by a previous event (that successfully committed). In other words, any input data to an event handler does not need to be re-created in case of aborts and re-execution.

On the other hand, any data created by the event handler has to be buffered and published only in case of successful execution of the handler. This is the case for the standard output operation in the example. In general, output operations can be considered a special case of asynchronous I/O operations: they can be buffered during the speculative execution of the event handler, and be concretely sent to the console output or the I/O substrate once the event handler successfully completes. Event ordering can be used to obtain deterministic console output when needed (using globally ordered events).

5.2.4 Bailout and worst-case scenario

In some cases the speculation might not be effective, or might not be possible. This could be due to practical implementation issues (e.g., some system-level builtin operations that cannot be parallelized), or for performance reasons (e.g., for contention management). Every PEV runtime discussed in this Dissertation features a notion of *irrevocable execution* that can be used to enforce the mutual exclusion of an event handler with respect to other handlers running in parallel. Such notion of irrevocable execution is implemented differently depending on the PEV runtime, but is semantically equivalent to acquiring a global lock (for the irrevocable event handler) to prevent any other handler from running concurrently. Each PEV runtime implements this "worst-case" solution in a different way, and implements different policies for deciding which event handler has to be executed using this modality.

5.2.5 Common data structures

Independently from the speculative engine implementation, every PEV runtime discussed in this Dissertation features some data structures to be shared between worker threads. Every event handler in the system implements the following basic class[2]:

```java
abstract class Event {
  // Unique event id
  private final long eventSequenceNumber;

  // Event target
  private final EventTarget eventTarget;

  // Executable handler
  private final Runnable eventHandler;
}
```

[2]We adopt the convention of expressing runtime components in Java.

At the moment it is emitted, every event instance is assigned a unique sequence number id. Events not requiring any ordering (e.g., globally unordered events) are assigned a special id for immediate execution. The event id is a unique identifier assigned increasingly by the current event target, which is used by the runtime to ensure the correct event execution order:

```
abstract class EventTarget {
  // Generator of sequence ids
  private final AtomicLong nextEventSequence;

  // Sequence number of the next committing event
  private volatile long nextRetiredSequenceNumber;

  public long getNextSequenceNumber() {
    return nextEventSequence.incrementAndGet();
  }
}
```

Another sequence number (i.e., `nextRetiredSequenceNumber`) is used to enforce the correct ordering between events. Being the sequence number a per-target field, two event handlers belonging to two distinct event targets do not need to synchronize.

Events are always emitted by either the I/O substrate (e.g., as a consequence of a new socket connection), or by another retired event handler. Depending on the runtime implementation, event emission might require some ad-hoc scheduling (for instance to bind a specific event target to a specific thread). This implies that distinct implementations of the PEV might require multiple queues to be used to store event instances. Every implementation, however, always features at least a single centralized shared queue which is used to store events emitted in the `global` target by the I/O substrate. In addition to the global shared queue, every worker thread in the system features a private event queue, used to store events that have to be processed by a specific thread.

5.3 Pessimistic PEV runtime

The first of the three PEV runtimes described in this Chapter is the *Pessimistic* runtime. In this implementation, the main thread processes all the events that have to respect global ordering (generated using `emit`), while events belonging to other targets (both chained and unordered ones) are processed by worker threads, and are directly emitted into the queue of the worker thread responsible for a given event target using hashed emission. Worker threads are also used as helper threads that attempt to run globally unordered event handlers concurrently, often providing good scalability when such events are read-only or purely functional.

Event ordering is ensured in the Pessimistic runtime by the natural ordering guaranteed by global and local queues. Safe event execution is enforced through fine-grained locking, while a global shared lock is used to support irrevocable event handlers. Fine-grained locking is used to ensure atomicity in the execution of the event handler via the following scheme:

- At the moment an event handler is to be scheduled, a static analysis phase its performed to identify the set of object instances that will be accessed by the event handler.
- If the analysis can determine that the handler will access shared data using read-only operations, a readers-writer lock will be acquired protecting the object instance. Hashing is used to avoid allocating a lock instance for every object instance, as well as to re-use locks.
- If the analysis can determine that the handler will potentially modify shared data, a write lock is acquired.

The analysis can either fail or succeed. In case of success, the set of locks required to protect the *entire* event handler are acquired at the first read/write operation. To this end, the read/write barriers of the event handler implement the lock acquisition respecting global ordering.

In case of unsuccessful analysis, a global lock is used to ensure atomicity. The global lock acts as a readers-writer lock that can be acquired by multiple threads for read operations, and can be upgraded (to writer) by one thread only. The shared lock implements the following schema:

- The lock is acquired in read-only mode by every thread not holding it before executing an event handler with successful analysis.
- When an event handler requires global exclusive access, the lock is upgraded to writer.

In the Pessimistic PEV, writing event handlers operating on the same shared object contribute only for a minimal part to the performance of the application, and the system does not offer good scalability for workloads having mixed contention on shared resources. However, the system already offers good scalability for read-dominated workloads and workloads dominated by pure functions. Moreover, the fine-grain locking mechanism can offer acceptable scalability for workloads modifying non-contented objects.

By using the global lock and the main event loop thread, the system has the following characteristics:

- The main thread has very light runtime barriers, and runs at almost the same speed of a single-threaded event loop.

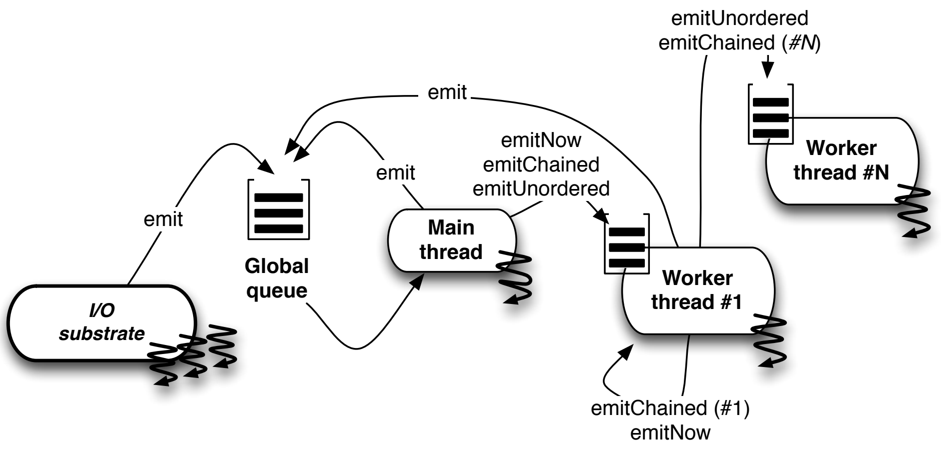


Figure 5.4. Pessimistic routing scheme.

- Worker threads have very light barriers as well, and contribute to the overall performance of the system when the workload is dominated by event handlers that are pure or read-only.
- The queuing scheme ensures that any workload dominated by chained or globally unordered events can benefit from parallel execution.

The main characteristic of this implementation is that no form of logging is needed to execute multiple events in parallel, and event handlers are never aborted. The disadvantage of the implementation is of course that it can efficiently support only a subset of all the existing single-threaded applications, that is, applications written using a read-only immutable or pure (side-effect-free) event handlers.

In the worst-case, this implementation is expected to have performance close to the ones of a single-threaded runtime. Conversely, when the application presents opportunities for parallelization, it can safely run multiple handlers in parallel, always enforcing the same semantics of an equivalent single-threaded runtime.

5.3.1 Scheduling and speculation algorithm

An overview of the scheduling scheme adopted by the Pessimistic PEV runtime is depicted in Figure 5.4. Each thread of the runtime has to pass through the following operations:

- *Fetching and Code modification*. A new event handler is retrieved from the global queue or from the thread-local queue and is modified with the required runtime barriers accordingly. Depending on the event target, a scheduling decision is made.

		Main thread	*Worker threads*
`emit`	*In*	Received from global queue	Cannot receive
	Out	Sent to global queue	Sent to global queue
`emitChained`	*In*	Cannot receive	From local queue (emitted by worker itself)
	Out	To hashed worker	To local queue, if same hash. Else, to hashed worker
`emitNow`	*In*	Cannot receive	From local queue
	Out	To random worker	To local queue
`emitUnordered`	*In*	Cannot receive	From local queue
	Out	To random worker	To random worker
Barriers		Same for main and workers (lock-based)	
Out-Of-order speculation		Not supported	
Irrevocability		Default	

Table 5.1. Pessimistic runtime summary.

- *Execution*. The handler is executed. The runtime barriers ensure that the correct fine-grained locks will be acquired, if any. When the analysis cannot determine a safe locking strategy, the global lock is acquired.

- *Commit*. Once the handler has terminated, it will release the acquired locks (lock acquisition and release happen using a fixed ordering, to avoid deadlocks). Before that, the event handler emits the events emitted locally (i.e., buffered) during its execution. This ensures that read-only event handlers can emit events only when it is safe to do so. Considering event emission at the same level of other side effects would imply that every event handler has to acquire the write lock before committing. Depending on the event target, the new events are added to the thread-local queue or to the globally shared one.

All the events generated by the I/O substrate are added to the global queue, which is responsible for holding all the globally ordered events in the system. The queue is implemented as a concurrent single-consumer/multiple-producers queue [86], and can accept events from the main thread and from workers as well. The main thread is responsible for executing all the globally ordered events emitted by the I/O substrate, and for dispatching other events emitted using the PEV API. The reason for this design is that in the absence of event emission using the extended API the main thread will correspond to a single-threaded loop (with the exception of locking and runtime barriers). Once dispatched, chained events are assigned (using hashed emission) to a worker thread, and will always be processed by the same thread, thus enforcing ordered execution through the local queue. Hashed emission is also used by other worker threads, that will add event handlers directly to thread-local event queues. To allow this, also thread-local queues are concurrent single-consumer/multiple-producer queues. Unordered events emitted by the main thread are randomly executed by a worker thread, while unordered events emitted by a worker thread are directly processed by the worker itself, that is, are added to its local queue. This implies that only unordered event emissions with the global target will be processed in parallel.

A summary of the execution policy is depicted in Table 5.1. This implementation of a PEV runtime corresponds to a simple lightweight implementation targeting mostly chained events, and does not support advanced features such as work stealing between multiple workers and out-of-order execution of ordered events.

The Pessimistic runtime for the PEV runtime shares some similarities with the TML algorithm [63]. The most significant difference is the usage of fine-grained locking when possible, and the fact that event handlers never abort.

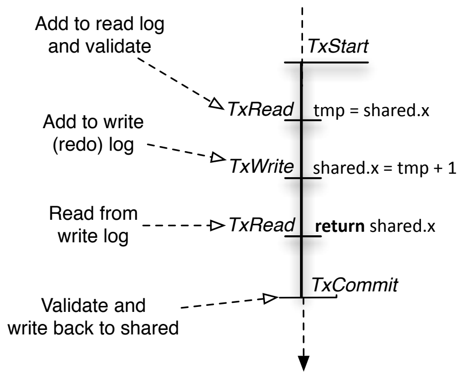
5.4 Optimistic runtime

The second PEV implementation is represented by the *Optimistic* runtime. This implementation is based on the execution of all the event handlers through a software transactional memory (STM) runtime. The STM runtime does not make any difference between unordered events, globally ordered events, and chained events, meaning that all the event handlers can be executed by any available worker thread, and there is no specific thread bound to a specific event target. As with the Pessimistic runtime, hashing is used to schedule events on specific worker threads. However, the presence of work stealing in the runtime makes it possible for event handlers to be executed by any thread.

In addition to work stealing, ordered and chained event handlers can also benefit from an additional speculation technique implemented by the runtime, allowing ordered events to be safely executed out-of-order.

5.4.1 STM-based speculation

The STM-based runtime executes every event handler in a dedicated software transaction, and is implemented by extending the TL2 algorithm [66, 69] for ordered execution of transactions. The TL2 algorithm is a well-known STM algorithm for the generic implementation of software transactions, meaning that it does not make any strong assumption on the higher-level API. At very high level, a software transaction in TL2 operates as an isolated context of execution performing any operation in a "sandboxed" environment, not exposing to other concurrent transactions or threads its modifications to shared objects until it is allowed to do so. Event handlers executed in



```
// This object is created somewhere else in the code, and is potentially shared
var shared = { x : 0 }

// an event handler using the object
asyncNow(shared, function() {
  var tmp = shared.x;
  // ...
  shared.x = tmp + 1;
  // ...
  return shared.x;
});
```

Figure 5.5. Example of STM-based speculative execution.

a software transaction behave similarly. Consider the example depicted in Figure 5.5. The execution of the event handler scheduled using `asyncNow` operates as follows:

- *TxStart* barrier: The event handler is started. A shared global counter is used to keep track of the progress of the transaction, and is read at the moment the event handler starts.

- *TxRead* barrier: At the moment the shared object is read, the shared counter is used to *validate* the operation, that is, to verify if in the meantime some concurrent transaction modified `shared`. Some additional logging is needed to keep track of the operation at commit time. This operation can of course cause the transaction to abort in case of failed validation.

- *TxWrite* barrier: When the object is modified, no real modifications to the actual object are performed. Rather, the transaction-local modification (that is, the value computed locally by the transaction) is stored in a transaction-private log, called *redo*-log. This log will be used later on to perform the actual modifications to the object in case of successful execution.

- *TxRead* barrier: The next time the object is read, the transaction has to return the value that was previously modified, avoiding reading from the actual shared object.

- *TxCommit* barrier: Eventually, if the transaction was not aborted during its execution, the values present in the redo-log are *committed*, that is, are written to the shared object. Some additional validation is needed to ensure that all the values read during the execution of the transaction are still consistent, and synchronization is needed to ensure that the commit phase is atomic and visible to other transactions.

The TL2 algorithm has been chosen since it is one of the most studied STM algorithms. Other algorithms with similar characteristics and with different performance characteristics [91] could be supported as well. In particular, any STM system with the following desirable characteristics can be potentially adapted to be executed in the context of a PEV runtime:

- Strong atomicity [91] is not needed for the PEV runtime, as no user code can ever be executed outside of a transaction.

- Opacity [91] is a desirable property, as it is not possible to predict all the possible conflicts that might be generated by two conflicting transactions, and zombie transactions [91] cannot be tolerated by the system.

- Irrevocable transactions are desirable in order to implement contention management strategies aimed at speeding up frequently aborting transactions as well as to run transactions making use of some non-parallelizable operations as discussed in Section 5.2.4.

5.4.2 Runtime barriers

Depending on whether an event handler needs to wait for some other events (to enforce ordering) or not, this PEV runtime modifies event handlers with two different commit-time barriers:

- *Unordered commit barrier*. Event handlers that do not require any ordering use standard STM commit barriers.

- *Ordered commit barrier*. Event handlers that require synchronization with respect to some event targets (including globally ordered events) execute a special commit barrier that can be used to implement *commit-time reordering*. This barrier makes it safe to run ordered events out-of-order, speculatively.

The event handler modification performed by the Optimistic runtime operates as follows:

- Since every event handler is executed as a standalone software transaction, every event handler callback is extended with the necessary barriers for starting the software transaction (i.e., `TxStart`).
- Every function call in the event callback is modified with a dynamic dispatch mechanism that will replace the actual call target with a new call target implementing the proper runtime barriers. This ensures that all needed runtime barriers will propagate through the execution of the event handler. Caching can be used to make this operation inexpensive for frequently-used functions.
- Every operation potentially involving access to shared data is replaced with STM read or write barriers (`TxRead`/`TxWrite`). Examples of such operations are property access, element access, frame variables, level variables, and other language-specific operations such as the `Arguments` array in JavaScript.
- Event handlers using operations that are known to be unsafe for parallelization, (e.g., the `eval` construct in JavaScript) are marked as not parallelizable, and will be executed as irrevocable transactions. If the unsafe operation is detected at runtime, the transaction is aborted and re-started as irrevocable.

In the following subsections the ordered and unordered executions are discussed in detail.

5.4.3 Events scheduling

The Optimistic PEV runtime implements a different, simpler, scheduling policy than the Pessimistic one. The high-level description of the scheduling approach for this runtime is depicted in Figure 5.6.

In the Optimistic runtime, both the main thread and worker threads are responsible for processing any type of event, regardless of their event target. The goal for this design is providing a fair scheduling not biased towards the main thread. As with the Pessimistic runtime, the main thread is the only thread receiving events from the global queue (generated by the I/O substrate); differently, the main thread does not use the global queue for global ordered events, meaning that calls to `emit` from the main thread will not cause events to be added to the global queue, but rather to one of the worker's queues, selected randomly. Globally ordered events are considered at the same level of other event types, and are therefore processed by workers. Like with the Pessimistic runtime, chained events are assigned to a specific worker using hashing. Differently, every unordered event emission -including unordered events with a non-global target- are potentially processed in parallel by the PEV runtime, as worker threads implement work stealing. In particular, every worker has a concurrent double-ended queue used by worker threads to route chained events and to steal tasks using an approach similar to the one of the Java ForkJoin Pool [110]. Differently from

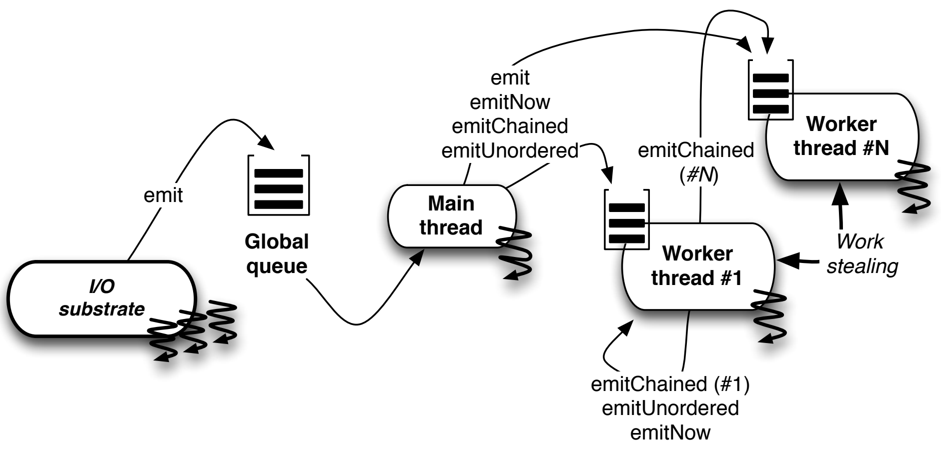


Figure 5.6. Optimistic routing scheme.

the ForkJoin pool, however, tasks emitted by a worker to its own queue are always added on top of the queue, and therefore the worker thread always processes oldest tasks first[3]. Following the same approach, idle threads that attempt to steal work from a worker thread will do so by trying to steal tasks from the top of the queue. Independently of the type of event that was acquired (ordered vs. unordered) the thief thread will immediately attempt execution. In this way, also ordered event handlers will potentially run in parallel. As will be discussed in the next section, a special commit-time STM barrier will make sure that this kind of out-of-order speculation does not violate the semantics of the PEV API for ordered event emission. A summary of the Optimistic PEV runtime is presented in Table 5.2.

Shared metadata and unordered commit barriers

The Optimistic PEV runtime implemented using the TL2 algorithm uses event targets to implement event processing with the correct order (event targets rely on the structure described in Section 5.2.5). Every event instance holds a reference to its corresponding event target object, which is usually the object that emitted the event instance. Depending on the event emission type, the event is modified with distinct TM barriers, as follows:

- Globally unordered events are executed directly, with barriers implementing the standard TL2 algorithm. Depending on the global target, they might be assigned to any available thread, or to a subset of threads.

[3]Conversely, when a thread in the ForkJoinPool attempts to process tasks from its own queue, usually tries to process youngest task first.

		Main thread	*Worker threads*
`emit`	*In*	Received from Global queue	from Main thread or work stealing
	Out	Sent to random worker	To local queue (might be stolen). Commit-time barrier enforces ordering.
`emitChained`	*In*	Cannot receive	From local queue (emitted by the worker itself) or work stealing
	Out	To hashed worker	To local queue (might be stolen). Commit-time barrier enforces ordering.
`emitNow`	*In*	Cannot receive	From local queue (emitted by the worker itself) or work stealing
	Out	To random worker	To local queue (might be stolen)
`emitUnordered`	*In*	Cannot receive	From local queue (emitted by the worker itself) or work stealing
	Out	To random worker	To local queue (might be stolen)

Barriers	Same for main and workers (STM)
Out-Of-order speculation	Supported, for every event type including `emit`
Irrevocability	Via global lock acquisition

Table 5.2. Optimistic speculation summary.

- Other types of events require different runtime barriers that make use of a unique sequence id. The id is assigned to every event instance, and is compared against the `nextCommittingEvent` of the corresponding event target to impose the correct event execution order for multiple event handlers. Event IDs are guaranteed to be unique on a per-target basis.

Unordered events use standard STM barriers, and are executed by some worker threads as soon as possible. Their execution loop is presented in Figure 5.7, and corresponds to a common loop for software transactions, extended with events buffering and delayed emission. As common with STM-based runtimes, the software transaction runs in an infinite loop that will exit only when the transaction will have successfully committed. While executing, no other thread will be able to see the modifications

```java
// The infinite event loop for this worker thread
while (true) {
  boolean commit = false;
  // Get the event from the queue
  Event event = localQueue.pop();
  do {
    try {
      // Try to run the event handler in a software transaction
      TxStart(event);
      commit = TxCommit(event);
    } catch (TxAbortAndRestartException e) {
      // If aborted, restart
      event.localEmitBuffer.clear();
      continue;
    }
  } while (!commit);
  for(Event e : event.localEmitBuffer) {
    // send the event to the correct queue according to scheduling rules
    schedule(e);
  }
}
```

Figure 5.7. Event handlers execution loop using an STM-based runtime.

done by the transaction (as it is running in isolation). In case of early abort or commit-time abort (e.g., because of conflicts with other concurrent transactions) an exception is thrown that will force the transaction to invalidate its transactional logs, and the transaction will be re-started. The `workerCommit` method for unordered events simply compares the transaction-local logs against the global metadata in order to validate the transaction, and in case of successful validation publishes the values as computed by the transaction.

5.4.4 Ordered events speculation

By wrapping every event in a software transaction, the side effects produced by every transaction become public in a serializable way. This implies that it is potentially possible to schedule the execution of every event handler out-of-order (including ordered ones), as long as the side effects of an event committing at time t_i is not causing conflicts with events at t_{i+n} that might still be running in the system.

By relaxing this time constraint for what concerns the entire transaction execution only to the transaction commit-time, it is possible to run event handlers that have been scheduled for ordered execution using an unordered execution policy, imposing the correct execution ordering only on transaction validation. The Optimistic PEV runtime can support this type of out-of-order speculation for ordered events -that is, globally ordered and with chained ordering-, which can be implemented in the following way (see Figure 5.8):

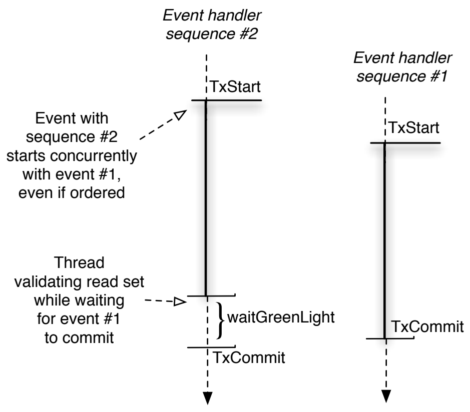

Event handler sequence #2
Event handler sequence #1
TxStart
TxStart
Event with sequence #2 starts concurrently with event #1, even if ordered
Thread validating read set while waiting for event #1 to commit
waitGreenLight
TxCommit
TxCommit

Figure 5.8. Out-of-order execution of ordered events.

- Once selected for execution, every ordered event handler gets a unique identifier, from its event target.

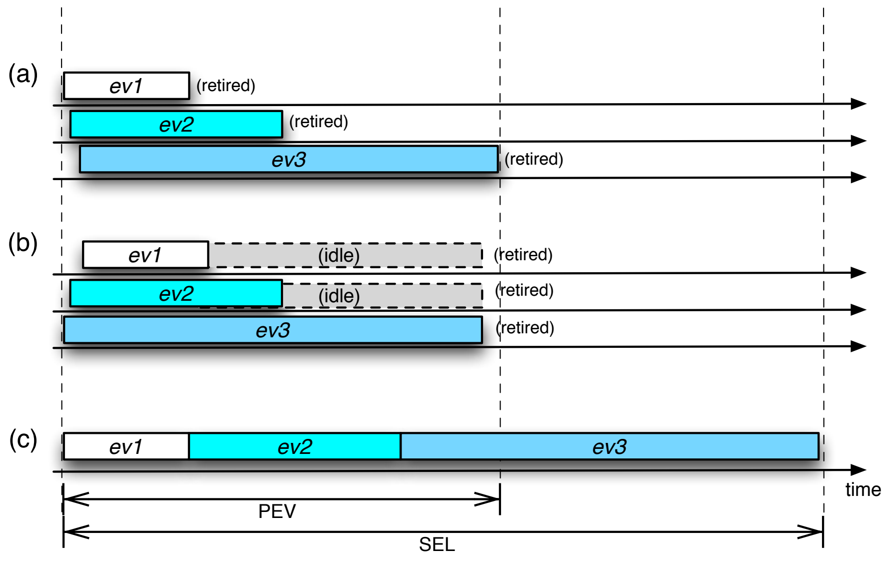
- Once retrieved from the event queue, any ordered event is started as soon as possible, without having to wait for its predecessors (potentially running in another thread) to commit.

- Every event handler, however, is not allowed to commit until all of its predecessors in the same target have successfully committed.

- Once allowed to commit, the read set of the out-of-order event handler is validated, and in case of success the commit operation can be executed. In case of failure, the transaction can be re-executed.

An exemplification of the effects of the speculative execution is depicted in Figure 5.9, where a comparison of the execution of three events in the PEV model is compared against the single-threaded loop execution.

As an example, consider the code example presented in Figure 5.9 (already presented in Chapter 4). According to the specification of the `emit` function, the three event handlers have to be executed sequentially, in the same order they are produced. As the three handlers simply perform some arbitrary CPU-bound computation (in the example, calculate the Nth Fibonacci sequence number), and then print the result on the system's standard output console, there is no interdependency between handlers in terms of conflicting access to shared data structures. Figure 5.9 (c) corresponds to the execution of the three event handlers in both a single-threaded event loop system and on a Pessimistic PEV. In both executions, the three events are pushed accordingly into the event queue, and handlers are executed with the expected ordering. Eventually,



```
var obj = {};
// register three event handlers for three distinct events
obj.on('ev1', function{
  var r = computeNthFibonacciNumber(10);
  print('event 1 result: ' + r);
});
obj.on('ev2', function{
  var r = computeNthFibonacciNumber(20);
  print('event 2 result: ' + r);
});
obj.on('ev3', function{
  var r = computeNthFibonacciNumber(30);
  print('event 3 result: ' + r);
});
// emit an event instance for each handler.
obj.emit('ev1'); obj.emit('ev2'); obj.emit('ev3');
```

Figure 5.9. Parallel event processing with the Optimistic PEV (a, b) and a single-threaded event loop or a Pessimistic PEV (c).

the console will always show the following output for both the PEV-based execution and the SEL-based one:

```
event 1 result: 55
event 2 result: 6765
event 3 result: 832040
```

The Pessimistic PEV cannot take any advantage from the parallel nature of the three event handlers, as they are executed one by one in the main thread, and no worker thread is involved (this would also be the scenario for chained events).

Conversely, the Optimistic runtime can adopt a more aggressive approach and can try to execute transactions out-of-order as soon as a worker thread can offer some work to do. With such approach, the PEV will speculatively try to run the three event handlers in parallel, as soon as they are pushed into the event queue. This eventually results in the automatic parallelization of event processing. All the events in the example modify a shared resource (the standard output), and generate side effects. However, the side effects generated by each event handler do not depend on prior handlers' effects, and thus none of the three event handlers will have conflicts while validating, nor will see partial results as computed by concurrent handlers (lazy conflict detection). Still, the PEV runtime has to make sure that events are retired with the correct ordering. In other words, the runtime has to make sure that events commit in the correct order. This also implies that for each event handler, side effects will be *made public* with the correct event ordering, and events will wait until they are allowed to make any change to the shared resource (that is, writing to the standard output).

Strict ordering implies that events emitted in a given order will impact the total execution time on the PEV. This is the case for the following event emission:

```
// emit an event instance for each handler.
emit('ev3'); emit('ev2'); emit('ev1');
```

In this case (also described in Figure 5.9/b), the PEV will schedule all the events for parallel execution, however, the last event (that is, `ev1`) will not be allowed to complete until all of its predecessors will have completed. This means that at the moment it will try to commit it will wait for other events, wasting computing resources. Despite the inefficient execution, also in this case parallelization leads to improved execution. The execution loop for ordered events can be described with the following pseudocode:

```
// The infinite event loop for this worker thread
while (true) {
    do {
      try {
        // Try to run the event handler in a software transaction
        TxStart(event);
        waitForCommitGreenLightAndValidate(event);
        commit = TxCommit(event);
      } catch (TxAbortAndRestartException e) {
        // If aborted, continue
        continue;
      }
    } while (!commit);
}
```

The event handler immediately starts its execution, and pauses execution until it is allowed to commit. This is done in the `waitForCommitGreenLight` method:

```
void waitForCommitGreenLight(Event event) {
  EventTarget target = event.getTarget();
  long eventSeq = event.getSequenceNumber();
  // Active wait
  while(target.nextRetiredSequenceNumber.get() != eventSeq)
    validate();
}
```

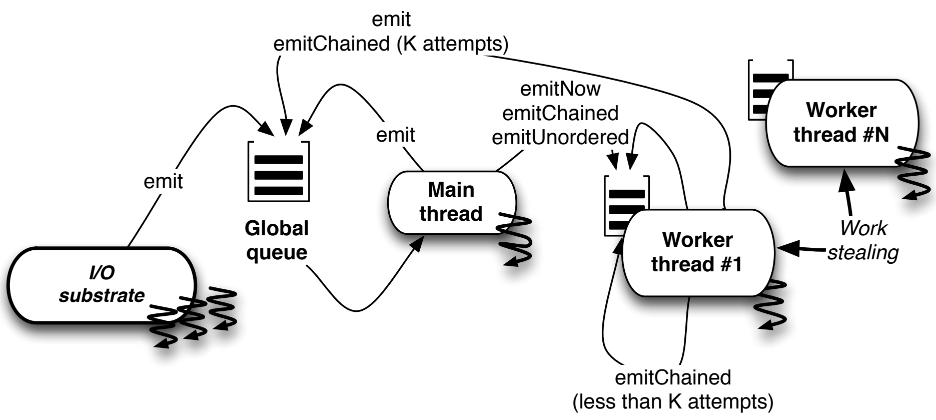


Figure 5.10. Hybrid runtime routing scheme.

The event handler waits until the sequence number of the next ordered event allowed to commit for the given event class matches its sequence number. Since sequence numbers are unique and per-target, events with different event target will not interfere. At the moment the sequence number matches with the one of the next event handler allowed to commit, the method returns and the event handler can perform the final validation. Since event sequence numbers are unique, it is always guaranteed that the event will be the only one allowed to commit even in the case of conflicts with other event handlers, and the event handler will never abort because of a conflict with event handlers with higher commit sequence. Event handlers might still fail because of conflicts with event handlers belonging to other event targets (or unordered ones). The runtime does not privilege ordered events against unordered ones.

5.5 Hybrid runtime

The third implementation of the PEV runtime is represented by the Hybrid runtime. This implementation's goal is to reduce the overhead for ordered events (as such events are the most common ones with single-threaded event loop runtimes) still offering good scalability for other workloads.

This hybrid PEV system is based on the principle of *irrevocable transactions* [154], transactions that are always allowed to commit, and therefore do not have to validate their metadata at commit time. This runtime system can be considered an hybrid approach between the Pessimistic and the Optimistic runtimes, as it reduces the overhead for the main thread by employing a dedicated barrier for event handlers it executes, and makes use of optimistic speculation through a transactional memory to execute event handlers in the other threads.

5.5.1 STM-based speculation

The STM runtime algorithm implemented by this PEV runtime is based on the FastLane [151] algorithm, a software transactional memory runtime explicitly designed to reduce the execution overhead of STM systems by reducing the runtime barriers cost of one of the threads in charge of executing the transactions. Differently from the approach in the Optimistic runtime (where worker threads had the same type of transactional barriers except for the commit one), in FastLane two distinct types of barriers are used:

- *Main* thread: only the write operations need to implement runtime barriers. No logging is needed (nor for reads or writes), and both the `TxStart` and `TxCommit` barriers simply need to acquire a globally shared lock.

- *Worker* threads: other threads have runtime barriers similar to the ones of TL2.

Being the main thread in the system relying on very light barriers, a low execution overhead can be provided by the runtime for the majority of the workloads. By not having logs, the main thread cannot abort transactions, and therefore its transactions are always irrevocable. The advantage of having a fast, low-overhead main thread, comes at the price of scalability, as the irrevocable transaction might force other concurrent transactions to abort more often. Moreover, the need to acquire the global lock at the beginning of every irrevocable transaction has the effect of reducing the efficiency of worker threads. On the other hand, as long as transactions do not conflict, nor abort they can still contribute to the total throughput of the application.

5.5.2 Scheduling

The general scheduling strategy of the hybrid runtime is a combination of the scheduling strategy of the Pessimistic runtime and the one of the Optimistic runtime. Like with the Pessimistic PEV, every globally ordered event handler is executed by the main thread, including the ones emitted by the main thread and by other worker threads. In this way the main thread can offer low execution overhead for operations that are dominated by global events. As events different than globally ordered are emitted, however, they are sent by the main thread to worker threads. Like with the other runtimes, hashing is used to schedule chained events, and like with the Optimistic runtime system, worker threads can perform work stealing. Tasks, however, can be stolen by idle threads only if they are unordered and can be executed in parallel. For chained events, the Hybrid runtime implements another execution strategy:

- Chained events are emitted either by the main thread or by a worker thread, and are added to the local queue of the proper (hashed) worker thread.

		Main thread	*Worker threads*
`emit`	*In*	Received from Global queue	Cannot receive
	Out	To Global queue	To Global queue
`emitChained`	*In*	From main queue (After K attempts)	From local queue (emitted by worker itself) or from work stealing.
	Out	To hashed worker	To local queue or hashed worker
`emitNow`	*In*	From main queue (After K attempts)	From local queue (emitted by worker itself) or from work stealing.
	Out	To random worker	To local queue (might be stolen)
`emitUnordered`	*In*	From main queue (After K attempts)	From local queue (emitted by worker itself) or from work stealing.
	Out	To random worker	To local queue (might be stolen)
Barriers		FastLane Main	FastLane Worker
Out-Of-order speculation		Only for chained and unordered events (no `emit`)	
Irrevocability		Via main thread	

Table 5.3. Hybrid PEV speculation summary.

- Once selected for execution, the event is run by the worker in a transaction. In case of failure, the transaction is not re-executed until it can commit, as in the Optimistic runtime. Conversely, after a number of K failures, the event handler is added to the global queue, and will therefore be executed by the main thread. The number of failures is a constant factor equivalent to the number of available cores in the system.

By adopting this simple contention management strategy, the execution overhead of conflicting event handlers is attenuated by the main thread. Since chained events already have to be executed following a fixed order, they will be executed in the main thread without violating the semantics of the PEV API, and at the same time the worker thread will have more opportunities for executing other tasks. In other words, the main thread is used to reduce contention on certain event handlers.

Work stealing is still possible between worker threads, but will affect only unordered and chained events. An overview of the scheduling policies of the Hybrid PEV runtime is presented in Table 5.3.

5.5.3 Runtime barriers

As discussed, the runtime adopts the following two types of runtime barriers:

- *Main thread.* Globally ordered events are executed only by the main thread of the PEV system, and are modified with light barriers that do not require transactional metadata or perform validation and additional bookkeeping. Only one of such events is allowed at time, and such events always commit. In our Java-based FastLane implementation, the barrier corresponds to a single volatile write on every write operation (no barriers are used for reads).

- *Worker threads.* All the other event handlers running in worker threads are modified to run with more complex STM barriers. Unordered events have a specific commit-time barrier that allows them to validate their metadata as soon as possible, whereas transactions with chained ordering have to wait for other transactions of their same target to complete before committing.

The FastLane STM algorithm is extended to support commit time reordering and synchronization for ordered events, similarly to what is already supported in the Optimistic runtime. This is achieved in the following way:

- *Main thread.* Event handlers scheduled for execution by the main thread cannot be stolen (no out-of-order speculative execution).

- *Worker threads.* Tasks ready for execution in the local queue of worker threads can be acquired by other idle threads for out-of-order execution. Differently than what implemented in the Optimistic runtime, however, out-of-order speculative execution is implemented attempting to execute the task only for a fixed number of times, and after a number of failed attempts the task is sent to the global queue and will be executed by the main thread.

As the main thread always commits, the out-of-order execution of ordered tasks in the Hybrid PEV runtime cannot contribute to the overall system throughput for a relevant factor. However, workloads characterized by non-conflicting event handlers might still benefit from out-of-order execution when chained events are employed.

As the main thread always commits, it is very unlikely that workers could improve the system's throughput with conflicting event handlers.

5.6 Summary

In this Chapter we introduced the design and the main runtime characteristics of three runtime systems implementing the PEV model. Each of the three systems described in this Chapter comes along with its own peculiarities and limitations, offering different performance characteristics. The rationale behind the design of multiple runtimes lies in the fact that for certain workloads latency could be preferred over throughput, and vice versa. The PEV programming model and its event emission API ensures that applications developed targeting the PEV can be redeployed in any of the described runtimes, as well as on a plain single-threaded event-based system.

5.6.1 Overview and Limitations

Each of the PEV runtimes introduced in this chapter has different characteristics. The following general considerations can be made regarding performance as well as limitations:

- The Pessimistic model does not offer good scalability for highly-contented workloads and cannot parallelize unordered events with target different than global, However, it requires very lightweight barriers. The model can thus be already beneficial when the PEV is employed to run *well parallelizable* applications, still allowing developers to program with side effects and the impression of a single-threaded runtime.

- The Optimistic and Hybrid STM-based runtimes attempt to increase the level of parallelism by leveraging work stealing and scheduling. Speculative execution of ordered event handler, is also implemented by the two runtimes, at the cost of a more heavy runtime overhead. The models can be employed with workloads relying on the full set of API offered by the PEV model so as to benefit from the fairness between multiple events processing.

- The Hybrid model can be seen as a compromise between the two models, giving low overhead and execution latency for applications dominated by the core single-threaded API, still supporting the parallel execution of event handlers as well as the speculative execution of every type of event (including globally ordered ones).

5.6.2 Implementations overview

The PEV model can be implemented using different runtime designs. The choice depends on whether the system should support the full set of events or only a subset, and the performance goals for the speculative runtime (e.g., latency vs. throughput). In the next Chapters we will describe three implementations of PEV-based runtimes, namely, Node.Scala, TigerQuoll, and Truffle.PEV. Each implementation has specific runtime implementation details and peculiarities:

- Chapter 6, *Node.Scala*. An HTTP server and programming framework based on a PEV system supporting only globally ordered and chained events for Scala and the JVM. The main applicative domain for Node.Scala is server-side computing for stateless and stateful (read-dominated) workloads that also need to perform CPU-bound computations. The PEV runtime is based on the Pessimistic speculative system described in Section 5.3.

- Chapter 7, *TigerQuoll*. A language execution engine with an embedded PEV system supporting unordered events for JavaScript based on SpiderMonkey [20]. The main applicative domain for TigerQuoll is CPU-bound computations in JavaScript, for applications that cannot halt the event loop. The PEV runtime is implemented with an Optimistic speculative system as discussed in Section 5.4 and supports only the global event target (with both ordered and unordered events).
- Chapter 8, *Truffle.PEV*. A language execution engine for JavaScript with a PEV system supporting the full PEV API implemented on a Truffle-based execution engine for JavaScript [157]. The main applicative domain for this implementation is server-side applications with mixed read-only and CPU-bound workloads. Two versions of the runtime have been implemented to test both the Optimistic and the Hybrid PEV as described in Section 5.4 and Section 5.5.

In the following chapters we provide a detailed description of the three implementations, with a performance evaluation for each of the runtimes.

Chapter 6

Node.Scala

The first of the PEV-based runtimes described in this Dissertation is Node.Scala [40]. Node.Scala is an event-based HTTP Web server for the JVM platform bringing a programming model similar to the one of Node.JS to the Scala language, featuring implicit automatic parallelization of the event-based service. Using Node.Scala, services can be developed using a plain single-threaded event loop model and requests are handled and consumed concurrently, when possible, still enforcing correctness and thread safety [86][1].

Node.Scala is implemented without any VM-level modification, and relies on bytecode rewriting to enforce thread-safety and automatic parallelization. Reader/writer locking is used to implement a Pessimistic PEV runtime system, as described in Section 5.3.

6.1 Programming model for service development

Node.Scala is a server-side framework for HTTP services development. The programming model of Node.Scala is similar to the one of Node.JS, as it features a programming model based on asynchronous callback invocations. Thanks to the PEV runtime, blocking method calls can be executed directly from the main event loop without blocking the service, and concurrent requests running on different threads can safely share state. The goal of the framework is to let developers write services using the same assumptions (single-process event loop) made on the Node.JS platform, while automatically and safely carrying out the parallelization to fully exploit multicore machines. This has the effect of freeing the developer from dealing with the issues of concurrent programming, while keeping all the benefits of the asynchronous programming model

[1]Scala has been chosen as a target language for the speculative runtime due to its support for anonymous functions. Other JVM-based languages could be supported in a similar way, including lambdas in Java 8.

```scala
def fiboS(n: Int): Int = n match {
  case 0 | 1 => n
  case _ => fiboS(n-1) + fiboS(n-2)
}
val cache = new NsHashMap[Int,Int]()
val server = new NsHttpServer(8080)
server.start( connection => // 1st callback
{
  val n = connection.req.query("n").asInstanceOf[Int]
  if( cache.contains(n) )
    connection.res.end("result: " + cache.get(n) )
  else
    server.nextTick( => // 2nd callback
    {
      val result = fiboS( n )
      cache.put (n, result)
      connection.res.end("result: " + result )
    })
})
```

Figure 6.1. Stateful Web Service in Node.Scala.

with implicit parallelism, overlapping I/O and CPU-bound operations, and lock-free synchronization.

The two distinguishing aspects of Node.Scala compared to single-threaded event-based alternatives such as Node.JS are the possibility of using globally shared stateful objects, and the possibility of using blocking time-consuming CPU-bound calls in any event handler without affecting the overall service latency and throughput. An example of a simple Node.Scala Web service (Fig. 6.1) computing the *n*-th Fibonacci sequence number can be used to describe these two aspects. The stateful object (`cache`, of type `NsHashMap`) is used as a cache to store the values of previously computed requests. To perform the computation, a simple blocking function call (`fiboS`) is used. The service makes also use of two callback functions. As in Node.JS, the first callback represents the main entry point for the service, that is, the callback function that will be triggered for every new client connection. The callback is passed as an argument to the `start()` method (implemented in the `NsHttpServer` class). The second callback used in the example is the argument to the `nextTick` method, which registers the callback to perform the actual computation and to update the cache. The `nextTick` method is the Node.Scala equivalent of `asyncChained`.

Each callback is invoked by the Node.Scala runtime whenever the corresponding data is available. For instance, as a consequence of a client connection, an HTTP request, or a filesystem access, the runtime system emits an event, which is put into the event queue. The event will then be taken from the queue by one of the threads running the event loop, which will invoke the corresponding callback function with the received data passed as an argument. In this way, when a new client request is received, the runtime calls the first user-defined callback function passing the `connection` object as argument. The object (created by the runtime) can be accessed by all other nested callbacks, and holds all the details of the incoming request (`connection.req`), as well as the runtime object for generating the client response message (`connection.res`).

The service is stateful since the first callback uses an object with global scoping, `cache`, which is not local to a specific client request (to a specific callback), but is global and thus potentially shared among all parallel threads running the event loop. Node.Scala enables services to safely share state through a specific library of common data structures, which can be used by the runtime system to automatically synchronize multiple concurrent callbacks accessing shared data using the Pessimistic runtime.

The second callback calls a synchronous CPU-bound method. In common event loop frameworks such a blocking call would result in a temporary interruption of the event loop, as discussed in Chapter 2. The parallel runtime system of Node.Scala overcomes this limitation in its architecture by using multiple event processing threads and a Pessimistic parallel event loop. Therefore, blocking synchronous calls do not have a negative impact on Node.Scala service performance as they would have in traditional frameworks. Consequently, programmers can focus on developing the service business logic without having to employ more complex programming techniques to obtain scalability.

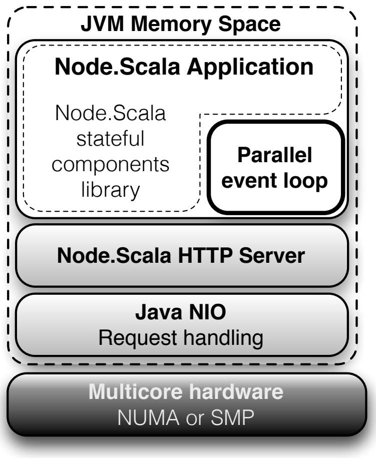
6.2 System architecture

Node.Scala uses a single JVM process with multiple threads to execute a Web service, granting shared memory access to the threads running the parallel event loops. As illustrated in Fig. 6.2 (b), the request processing pipeline consists of tree stages: (1) handling, (2) processing, and (3) completion.

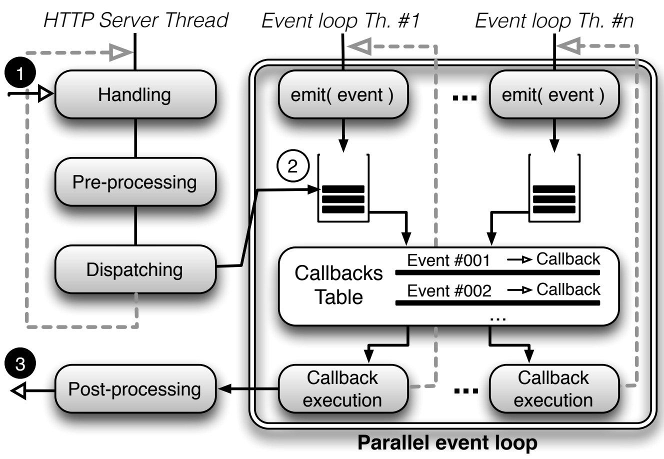
Request handling. Incoming HTTP connections are handled by a dedicated server thread, which pre-processes the request header and emits an event to the parallel event loop to notify a new request has arrived. The operations performed by the HTTP server thread are implemented using the Java New I/O (NIO) API for asynchronous I/O data processing.

Request processing. Multiple event loop threads concurrently process events generated by incoming requests. In particular, each event loop thread removes an event from its local event queue, accesses the callback table associated with that event type, and executes the registered callbacks. New events generated by the execution of a callback are inserted into the local event queue of the processing thread. This mechanism ensures that all the events generated by a specific request are processed sequentially, enforcing ordering for chained events. The callback table is automatically updated each time the execution flow encounters the declaration of a new callback function (see lines 7 and 13 in Fig. 6.1).

Request completion. Responses are buffered using the `end` method. Once all events generated by a request are processed, the system replies to the client using the HTTP server thread, which also performs some post-processing tasks (e.g., generating



(a) High-level Architecture



(b) The Parallel event loop

Figure 6.2. Overview of Node.Scala.

the correct HTTP response headers and eventually closing the socket connection).

6.2.1 Implementation

Like any PEV runtime, Node.Scala Web services are guaranteed to be thread-safe. To this end, the runtime distinguishes between three types of requests: *stateful exclusive*, *stateful non-exclusive*, and *stateless*. This classification depends on the type of accesses to shared variables[2]. If the processing of a request can trigger the execution of a callback that writes to at least one shared variable, the request is considered stateful exclusive. Similarly, if the processing of a request can result in at least one read access to a global variable, the request is considered stateful non-exclusive. All other requests are considered stateless. As a consequence, a stateful exclusive request cannot be processed in parallel with other stateful requests. Conversely, multiple stateful non-exclusive requests can be executed in parallel as long as no stateful exclusive requests are being processed. Finally, stateless requests can be executed in parallel with any other stateless and stateful request. All the three different classes of callbacks are guaranteed to be safe by the locking scheme of the Pessimistic PEV system.

To identify where the synchronization lock has to be acquired and released, Node.Scala intercepts class loading by means of the `java.lang.Instrument` API and performs load-time analysis of the bytecodes of each callback. Each user-defined callback is parsed by Node.Scala to track accesses to global variables. To speedup the analysis, a special class of components that can be safely shared between callbacks is

[2]Accesses to final values are not considered for the classification of requests.

provided, and methods of classes from this library are marked with two custom annotations: `@Exclusive` and `@Nonexclusive`.

Each time the analysis classifies a new callback as performing stateful exclusive or stateful non-exclusive operations, its bytecode is manipulated to inject read and write barriers acquiring the necessary read (i.e., `ReadLock`) and write (i.e., `WriteLock`) locks in order to ensure thread safety. The locking scheme follows the readers-writer pattern of the Pessimistic PEV runtime described in Chapter 5. Lock acquisition instructions are injected at every possible first acquisition of a shared data structure for each callback, while lock release operations are injected at the end of every callback. Therefore, the entire body is guarded by the necessary locks. In case of failure in acquiring the lock, the event loop thread can delay the execution of the callback of a specific request, as the chained event processing model does not impose global ordering for all the events in the system. Thus, the system can continue processing events generated by different requests without breaking the programming model and without halting the service. After a predefined number of failed attempts, exponential backoff is used to guarantee progress of a stalled thread and avoid starvation.

In the worst-case scenario (i.e., every callback performs exclusive callbacks requiring exclusive access to shared state) only a single event loop thread can execute a single request at any given time. Given the very low footprint of the runtime barriers, this ensures very low overhead. In this case, the performance of the service is comparable to the one of single-process, event-based frameworks. In all the other cases, Node.Scala can safely parallelize the execution of callbacks, taking advantage of all available cores to increase throughput, as illustrated in the following section.

6.3 Performance evaluation

To assess the performance of the Node.Scala runtime, we have implemented a Web cache service similar to the one presented in Fig. 6.1. Instead of the simple Fibonacci function, we used a mix of different computations from the set of CPU-bound benchmarks of the SciMark 2.0[3] suite, a well-known collection of scientific computing workloads. The service has been implemented using blocking function calls only, and both stateless and stateful services performance have been evaluated.

The machine hosting the service is a Dell PowerEdge M915 with four AMD Opteron 6282 SE 2.6 GHz CPUs and 128 GB RAM. Each CPU consists of 8 dual-thread modules, for a total of 32 modules and 64 hardware threads. Since threads on the same module share access to some functional units (e.g., early pipeline stages and the FPUs), the throughput of Node.Scala is expected to scale linearly until 32 event loop threads with ideal workload. The system runs Ubuntu GNU/Linux 11.10 64-bit, kernel 3.0.0-15, and Oracle's JDK 1.7.0_2 Hotspot Server VM (64-bit).

[3]`http://math.nist.gov/scimark2/`

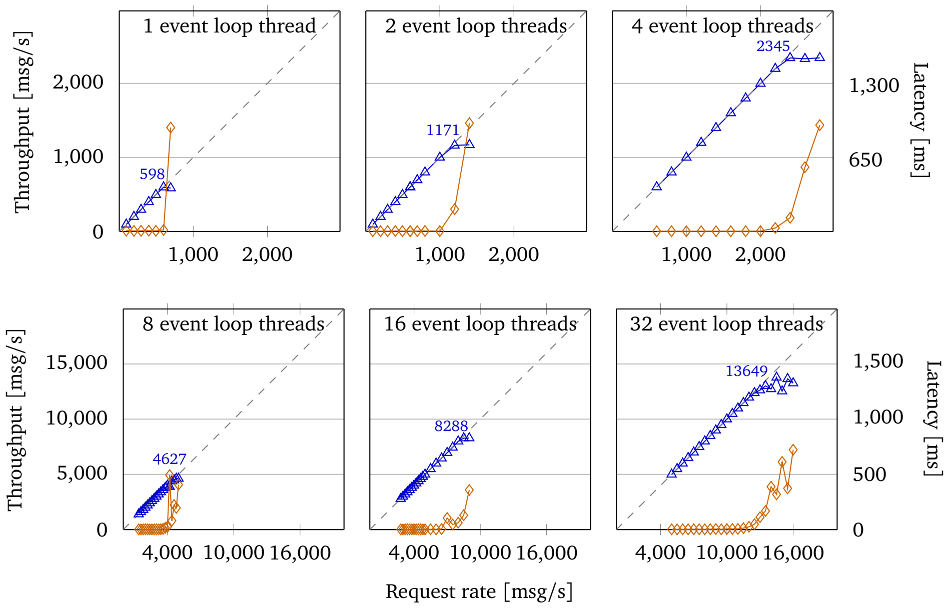


Figure 6.3. Stateless service: throughput (—△—) and latency (—◇—) depending on the arrival rate and the number of event loop threads. The dashed reference line (- - -) indicates linear scalability.

The runtime performance of Node.Scala is measured using a separate machine, connected with a dedicated gigabit network connection. We use httperf-0.9.0[4] to generate high amounts of HTTP requests and compute statistics about throughput and latency of responses. For each experiment, we report average values of five tests with a minimum duration of one minute and a timeout of 5 seconds. Requests not processed within the timeout are dropped by the client and not considered for the computation of the throughput.

6.3.1 Stateless services

To evaluate the performance of the Node.Scala runtime with stateless requests (i.e., with callbacks neither modifying nor accessing any global state), we have disabled the caching mechanism in the evaluated service.

Fig. 6.3 illustrates the variation of throughput and latency of responses depending on the request rate and on the number of event loop threads. The experiment with a single event loop thread resembles the configuration of common single-threaded event-driven frameworks for Web services, such as Node.JS. In this case, the throughput matches the request rate until a value of 600 requests per second. During this interval, the latency remains below 10ms. Afterwards, the system saturates because the single event loop thread cannot process more requests per unit time. As a consequence, the

[4] `http://code.google.com/p/httperf/`

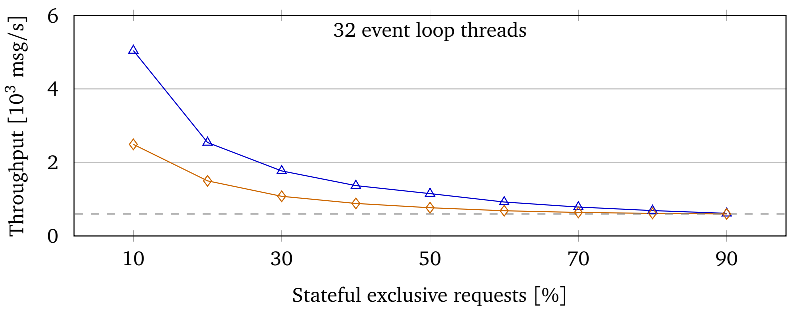


Figure 6.4. Stateful services: throughput of SciMarkSf1 (—△—) and SciMarkSf2 (—◇—) depending on the percentage of stateful exclusive requests. The reference line (- - -) refers to the throughput achievable using a single event loop thread.

throughput curve flattens and the latency rapidly increases to more than one second.

Experiments with larger amounts of threads follow a similar behavior: the latency remains small as long as the system is not saturated, and it rapidly increases afterwards. The peak throughput measured at the saturation point scales almost linearly with the number of event loop threads, up to a value of 13600 msg/s with 32 threads. This confirms the ability of Node.Scala to take advantage of all available CPU cores to improve the throughput of stateless Web services. Our experiments also confirm that the parallel runtime of Node.Scala allows the developer to use blocking function calls without any performance degradation.

6.3.2 Stateful services

To evaluate the performance of stateful services, we enabled the caching mechanism of the Node.Scala service used for the evaluation, and we have tested it with two different workloads. The first one (called `SciMarkSf1`) makes an extensive use of the caching mechanism, forcing the runtime to execute either exclusive or non-exclusive callbacks. The second one, (called `SciMarkSf2`) uses the caching mechanism only to store new data. Therefore, the second workload requires the runtime to process both exclusive and stateless callbacks.

The goal of both workloads is to assess the performance of the service in the worst possible cases, i.e., when the service is intensively using a single common shared object.

Fig. 6.4 reports the peak throughput of the two considered Web services, executed with 32 event loop threads, depending on the amount of stateful exclusive requests. As reference, we plot a line corresponding to the performance with a single event loop thread. When the number of stateful exclusive requests is high, performance is comparable to those of traditional, single-threaded, event-driven programming frameworks. However, when this number is smaller, Node.Scala can effectively take advantage of available cores to achieve better throughput. The difference with the non-modified single-threaded benchmark presented in the picture (less than 3%) also shows that the

Pessimistic PEV runtime has very little runtime overhead, still supporting shared state when needed (at the cost of pessimistic synchronization).

Chapter 7

TigerQuoll

The second PEV runtime described in this Dissertation is the TigerQuoll parallel runtime [41]. TigerQuoll is a JavaScript execution engine (based on Mozilla SpiderMonkey) supporting the execution of event handlers in parallel via an Optimistic PEV system based on the TL2 STM algorithm [66]. The engine supports only globally ordered and unordered events, and the core event emission API of the PEV is used to build high-level parallel constructs such as MapReduce, as discussed in Chapter 4.

7.1 Runtime system architecture

The TigerQuoll engine is a prototype JavaScript engine derived from Mozilla SpiderMonkey [20]. The TigerQuoll runtime features an event-based execution system based on a master/worker pattern for processing multiple events in parallel: a main thread carries out the execution of ordered events, while worker threads support the execution of unordered events in parallel. Both event handlers are executed with the same runtime barriers enabling STM-based speculation. The system is composed of multiple threads holding a pointer to a shared double-ended queue containing references to event objects. Globally ordered events are processed by the main JavaScript engine thread, while unordered events are consumed nondeterministically by the threads, therefore allowing for parallelism. Under speculative execution, each event handler is run and re-executed until completion, and no commit-time reordering is implemented. A runtime mechanism called *runtime switching* is used to mediate between parallel and sequential execution.

The PEV event emission API is directly exposed to the developer, and the PEV runtime in TigerQuoll is not used as a speculative engine for the parallelization of existing JavaScript applications. Rather, the event emission API can be used to build high-level programming models based on asynchronous event emission and shared state.

The event-based model of TigerQuoll in JavaScript to parallelism is not competing with emerging parallel solutions for JavaScript (e.g., WebWorkers and RiverTrail), but instead should be considered as a complementary one. In particular, the 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 read-only data parallelism, still without introducing the complexity of explicit parallel programming models. The programming model of TigerQuoll is an asynchronous event-based programming model with safe access to shared state. 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 HTTP-based services [78] or WebSockets, where a blocking fork/join model such as RiverTrail could affect the overall service latency[1]. 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 TigerQuoll engine has been developed and tested with multiple workloads. The overhead introduced by the event-based system is negligible, while the overhead imposed by the STM runtime depends on the measured workload. The overhead introduced by STM metadata is proportional to the size of the transaction's logs. When the parallelization is not possible (e.g., because of primitive operations that are not safe to be executed in parallel), the event handler is executed in the main thread, without any runtime barrier.

7.1.1 VM level modifications

The most relevant changes to the SpiderMonkey VM include modifications to the native implementation of the JavaScript `JSObject` class to associate every object instance with the needed transactional metadata. Moreover, the `JSObject` class has also been modified to enable objects with support for event emission, consumption and synchronization through the `on` and `emit` primitives, all based on `async`. The TigerQuoll runtime runs multiple threads accessing the same memory heap mediated by an STM layer. In its experimental implementation, the engine supports the fully transparent STM-based speculation over the event loop only for property access on objects shared between multiple functions' scope. Every property access is mediated by the STM-based runtime by modifying the `Proxy` mechanism of SpiderMonkey, a VM-internal mechanism used to implement JavaScript Harmony Proxies [4, 59]. By exploiting this mechanism, the virtual machine intercepts each data access and redirects it to the

[1]RiverTrail blocking operations could of course be offloaded to another worker, at the cost of losing immutable shared state, i.e., one of its most attractive peculiarities.

thread-local transactional copy of shared objects. As a consequence, each operation such as field read and write, but also field add or delete are mediated by the STM. The TigerQuoll engine also features support for transactions operating on fields using a different transactional semantics called *eventual fields*. Such fields are treated differently by the STM runtime, which manages two distinct logs.

7.1.2 Global runtime switch

To guarantee compatibility with existing JavaScript applications, as well as to enforce ordered execution, the TigerQuoll engine behaves as a standard non-parallel engine as long as the TigerQuoll runtime is not explicitly activated via unordered event emission. Once unordered events are emitted, the main thread stops processing ordered events, and waits until all unordered events (running in worker threads) have completed. This mechanism, called global runtime switch, ensures that existing applications that use neither the event-based API nor the TigerQuoll runtime will always run sequentially. Furthermore, this ensures that the application is always running either using parallel workers or using the main threaded, ensuring progress and never halting the application (e.g., never blocking the service). The user has no control over the switching mechanism, which is transparent and simply mediates between the parallel event loop and the single-threaded one.

As discussed, in order 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.

7.2 Transactional support

The TigerQuoll runtime features a STM with global versioning, lazy version management, and commit-time locking [91]. Conflicts are detected both at commit-time and during transactions' execution. The STM algorithm implemented by the TigerQuoll runtime is TL2 [66], and TigerQuoll features per-property versioning (as opposed to per-object versioning), and supports a relaxed semantics for a class of transactional operations called *eventual fields*. 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. The per-field version management prevents transactions operating on different fields of the same object from failing because of versioning conflicts.

```
// 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
    asyncNow(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 7.1. Parallel word count function in TigerQuoll, using the high-level task parallelism library presented in Chapter 4.

7.2.1 Eventual fields in transactions (eventual transactions)

For certain workloads, TigerQuoll offers the possibility to relax the default transactional isolation allowing to modify shared data using a different transactional semantics. This 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 applications do not require shared data to be consistent *during* the execution of the event handler, 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 in isolation, there is no way to let event handlers communicate 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).

The code in the example (Figure 7.1) 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 potentially 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 or validation 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 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.

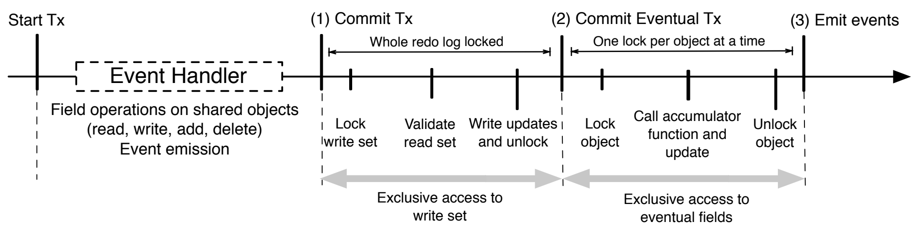

Start Tx
(1) Commit Tx
(2) Commit Eventual Tx
(3) Emit events
Whole redo log locked
One lock per object at a time
Event Handler
Field operations on shared objects (read, write, add, delete) Event emission
Lock write set
Validate read set
Write updates and unlock
Lock object
Call accumulator function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

Figure 7.2. Overview of the commit phase of the TigerQuoll runtime.

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.

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 7.2, while the three phases are described in detail in the following sections.

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 managed by each parallel event handler. The redo log is managed with a lazy strategy, 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.

```
// 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 7.3. Eventual transactions commit-phase pseudocode.

Eventual Fields Commit

The design goal for eventual fields is to propose a 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.

Transactions using only eventual fields 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 the 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 [91] 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 7.3) 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.

7.3 Performance 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 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 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.

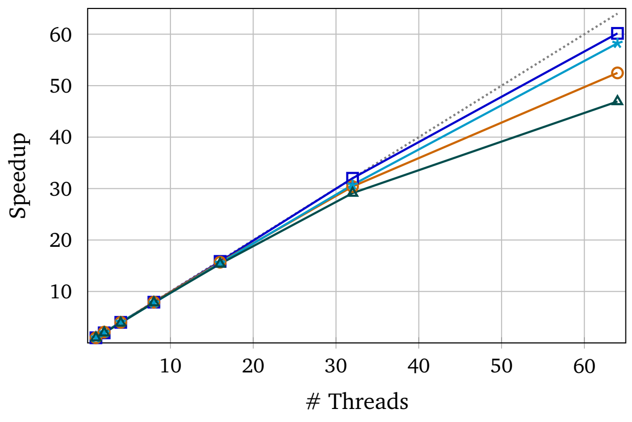


Figure 7.4. Share-nothing scalability. The graph shows the scalability of TigerQuoll for the Primes (—□—) and Mandelbrot (—○—) benchmarks compared with the equivalent WebWorkers Primes (—∗—) and Mandelbrot (—▲—) benchmarks. Speedup is relative to a TigerQuoll engine running with 1 worker thread.

7.3.1 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 7.4.

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.

7.3.2 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

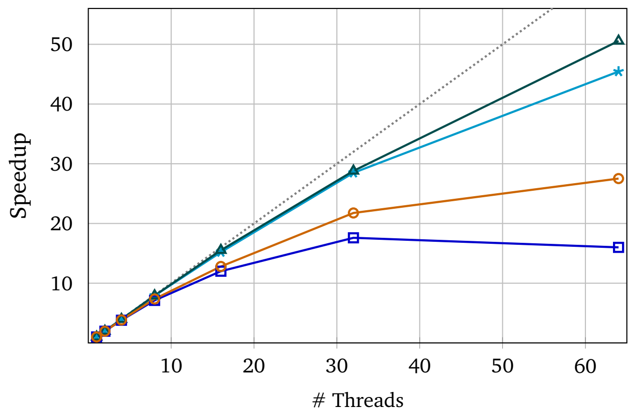


Figure 7.5. Shared Memory Scalability: Primes checker with shared counter. The graph presents TigerQuoll scalability in case of high contention with regular transactions (—□—) and eventual transactions (—○—), as well as low contention with regular transactions (—*—) and eventual transactions (—▲—). Speedup is relative to a TigerQuoll engine running with 1 worker thread.

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 contention, while tasks with an out-sized 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 7.5 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 30% 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 re-starting.

The impact of contention on shared-memory algorithms can in some cases dramat-

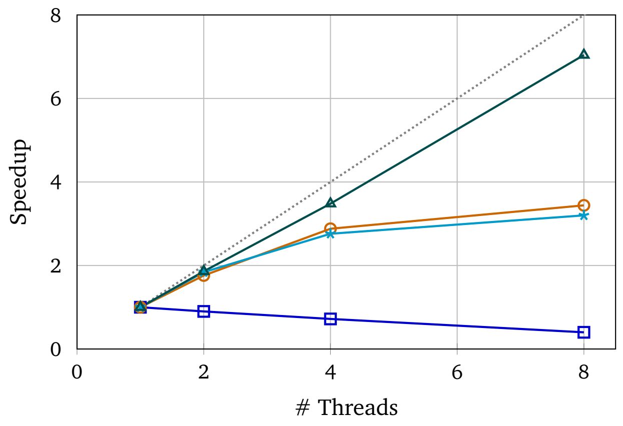


Figure 7.6. Shared Memory Scalability: word counter. The graph presents TigerQuoll scalability in case of high contention with regular transactions (—□—) and eventual transactions (—○—), as well as low contention with regular transactions (—★—) and eventual transactions (—▲—). Speedup is relative to a TigerQuoll engine running with 1 worker thread.

ically affect the performance of an STM system. This is the case for the experiment depicted in Figure 7.6, where a data-intensive workload with high and low contention has been evaluated. The experiment corresponds to the word-counter example presented in Figure 7.1. In the experiment, the parallel MapReduce 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 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 —▲—).

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.

Chapter 8

Truffle.PEV

The third implementation of a PEV system presented in this Dissertation is Truffle.PEV, a JavaScript engine derived from the Truffle JavaScript research engine [157] implementing an Optimistic speculative system and an Hybrid system. The engine has been implemented specifically targeting server-side applications and JavaScript, with the goal of providing a complete implementation of a PEV-based system. In the following subsections we first introduce Truffle as a platform for building high-performance engines for dynamic languages based on the JVM, and we then describe two implementations of the PEV runtime supporting the full set of events as introduced in Chapter 3.

8.1 Truffle and Graal

Truffle [157] is an open-source language implementation framework based on the Java Virtual Machine (JVM), which enables the development of high-performance Abstract Syntax Tree (AST) interpreters for multiple languages. The system presented in this section is derived from the Truffle-based JavaScript engine, which has performance comparable to the ones of leading JavaScript engines such as V8 and SpiderMonkey[1].

Truffle is a VM construction framework based on an high-performance AST interpreter that can be automatically compiled by an optimizing compiler (i.e., the Graal [157, 72] JIT compiler) to highly-specialized machine code. The compiler exploits the structure of the AST interpreter, and compiles the AST (using partial evaluation [81, 57]) to generate code. This approach enables a variety of language implementations to exploit the same optimizing compiler (that is, the same JIT compiler). Each implementation consists of a language-specific AST interpreter [102, 156], and the compiler is reused for all languages. The Truffle.PEV engine consists of a thread-

[1]The Truffle.PEV runtime has been implemented in the context of a collaboration with Oracle Labs and the VM research team.

safe AST interpreter for JavaScript extended to support the parallel execution of multiple functions using a Software Transactional Memory runtime for speculation.

The Truffle.PEV engine is derived from the Truffle.JS single-threaded JavaScript Truffle-based execution engine. The single-threaded engine achieves high-performance from a combination of techniques:

- Each AST node eagerly rewrites itself with an optimized version. Node rewriting specializes the AST for the actual types used, and can result in the elision of unnecessary generality, e.g., boxing and complex dispatch.

- The AST interpreter is automatically compiled by Graal. Compilation by automatic partial evaluation leads to highly optimized machine code without the need for writing a language-specific dynamic compiler.

- De-optimization from machine code back to the AST interpreter handles speculation failures. As long as deoptimization is infrequent, the optimized AST interpreter can itself be optimized by Graal.

Like other existing engines, the Truffle-based JavaScript engine is a single-threaded engine with no support for parallel execution. We extended the AST interpreter with two PEV-based runtime systems to enable parallel speculative execution of event handlers. Differently from modifying existing engines (e.g., SpiderMonkey and TigerQuoll), Truffle allows any change to the engine to be applied directly at the AST level. As the AST is partially evaluated and then compiled in an automatic way, any modification to the AST interpreter is automatically and implicitly compiled by the Graal partial evaluator. This implies that the STM system for the JavaScript engine in Truffle is itself compiled by the Graal compiler, and is thus optimized like any other runtime component of the JavaScript engine. Thanks to this approach, all the barriers implementing the Truffle.PEV speculative runtime are compiled by the JIT compiler, greatly reducing the runtime overhead.

8.2 Runtime Overview

We implemented two distinct versions of Truffle-based PEV engines. The first engine runs an Optimistic system based on the TL2 algorithm [66], while the second one runs an Hybrid PEV system relying on the FastLane algorithm [151]. Both systems are implemented in Truffle by means of AST rewriting, meaning that the original AST of the application to be modified for parallel execution is dynamically extended with runtime barriers enabling parallel execution. The two runtime systems implemented in Truffle support all event classes as introduced in Chapter 3.

At the API level, each Truffle-based PEV system supports a programming model similar to the one of TigerQuoll, with the main difference that the systems support all the types of events discussed in Chapter 5.

The following two speculative runtime systems have been implemented:

- Optimistic system. The first implementation is an Optimistic system as described in Chapter 5. As opposite to the implementation in TigerQuoll, the engine is always enabled, and all events (including ordered ones) are executed by the STM runtime. The STM runtime is based on the TL2 algorithm, extended with commit-time reordering in order to support the speculative parallel execution of ordered events.

- Hybrid STM. The second implementation is an Hybrid PEV runtime as described in Chapter 5, based on the FastLane algorithm [151], also supporting the speculative execution of ordered event handlers.

Both runtime systems feature automatic and dynamic modification of the AST of the functions that have to be executed. When a JavaScript function is to be executed by the STM runtime, its AST is modified on-the-fly with special AST nodes implementing the STM barriers. Examples of AST nodes replaced with STM-enabled ones include, for instance, read and write access to properties, array elements, and level variables accessed from closures. Functions are modified with nodes implementing different aspects of the STM algorithm (e.g., property read and write, function calls, transaction start, commit, etc.), and each node is responsible for implementing a specific STM operation, without introducing any limitation on the existing Truffle runtime, which can self-optimize the AST in the same way it would do with a non-modified AST interpreter. The runtime barriers are designed to be compatible with all the Truffle optimizations. In particular, STM barriers can benefit from partial evaluation like any other Truffle AST node, and are therefore compiled to very optimized machine code.

8.2.1 TM Metadata speculative management

The Truffle approach based on the automatic optimization of AST interpreters allows for some optimizations of the two STM runtimes that help reducing the overhead of the speculative runtime. In particular, the Truffle.PEV runtime features the following two classes optimizations:

- Log elision: with the goal of reducing unnecessary logging, the Truffle.PEV runtimes trace objects that are allocated during the current transaction, and therefore are not visible to other transactions until commit time. This information is exploited by the STM runtime to avoid logging objects allocated in the scope of the current transaction, and the Graal JIT compiler can thus remove all the STM barriers for such object instances. Similarly, the Truffle runtime speculates

```java
public class TxReadPropertyNode {

  private final TxRuntime tx;
  private final String propName;
  private final ObjectDescriptor desc;

  @Override
  public Object execute(VirtualFrame frame) {
    JSObject target = frame.getObject(desc);
    if(target.isTxPrivate()) {
      return target.getProperty(propName);
    } else {
      JSObject txLocal = tx.getWriteLogEntry(target);
      if(txLocal.hasProperty(propName)) {
        return txLocal.getProperty(propName);
      } else {
        tx.preReadBarrier();
        Object sharedValue = target.getProperty(propName);
        // throws TxAbortException
        tx.postReadBarrier();
        tx.addToReadSet(txLocal);
        return sharedValue;
      }
    }
  }
}
```

Figure 8.1. Read barrier for JavaScript object properties implemented as a Truffle AST node.

on certain objects (in particular, the JavaScript global object) assuming they are read-only: as long as this assumption holds, the STM barriers for such objects simply avoid logging, as the object is considered immutable. At the moment one transaction tries to invalidate such assumption, all the transactions are aborted, and a version of the transaction using all the correct STM barriers (not relying anymore on the read-only speculative assumption) is executed.

- Log optimization: when log elision cannot be applied (that is, for object instances that have to be logged to enforce atomicity), the Truffle.PEV runtimes still implement a per-barrier optimization that can reduce the logging overhead, exploiting a key feature of dynamic languages, that is, object shapes [99].

Object shapes are a runtime technique that is used in common JavaScript engines (as well as in other dynamic language runtimes) to optimize the cost for the dynamic resolution of object fields 8 in dynamic objects. Being JavaScript a dynamically typed language, every object can dynamically change the number of its properties, with the type of each property being itself mutable during runtime. To deal with such dynamic nature, object shapes (often also called *Hidden classes*) are data structures keeping track of the "class" of each object at runtime, and are the key data structure behind several optimizations in modern engines. Consider the following example:

```
// obj has shape S0=[] (empty shape)
var obj = {}
// obj's shape is replaced with S1=[x:int]
obj.x = 3
// obj's shape mutates again to S2=[x:int,y:string]
obj.y = 'foo'
```

At runtime, the object changes its shape three times. First -like with every new object- the object is assigned an empty shape. After the first property is added, the shape is replaced with a different one (S1). Similarly, when a second property is added the shape is replaced with a new one, i.e., S2. Since objects with the same shape always have the same internal object layout, shapes can be efficiently used to replace expensive property lookup operations with optimized ones (i.e., using a fixed offset rather than a dynamic lookup mechanism). In this simple example, for instance, the read operation accessing objects with shape S2 is always guaranteed to have the y property (of type String) at fixed offset. Therefore, the property can be directly accessed, without having to query the object's shape to get the property location. Shapes are used to replace generic operations with specialized -and more efficient- ones.

In each of the Truffle.PEV-based implementations, a mechanism similar to object shapes is used by the TM runtime to reduce the overhead of the transactional log. Consider the following example of a callback executed by the PEV engine in a transaction:

```
// obj.x has to be added to the read log. Before the log was empty: []
var x = obj.x
// obj.x is also modified. The write log is modified, and the new value of
// 'x' is stored in a redo log, at offset 0.
obj.x++
// nothing to do here: the object is already logged, just return the
// value of obj.x as in the redo log.
return obj.x
```

Each of the above operations happens at a fixed overhead, and performs the same operations on the transaction-local metadata. In particular, for the above example the following operations are always re-executed:

- First read: add the object to the log, at offset 0.
- Second read (at x++): do nothing, just validate.
- First write: add the object to the write set, at offset 0. Store the locally computed value of x in the redo log, at offset 0.
- Last read: do nothing, just validate and return the value from the redo log at offset 0.

By exploiting the fact that for many transactional operations metadata are always modified with a fixed offset, the Truffle.PEV performs an optimization on the log's access patterns, and injects in the machine code fixed offsets for reading and writing directly from the logs. This class of optimization, called *transactional shapes* helps reducing the runtime overhead, and is based on the speculative assumption that event handlers performing certain operations will most likely have the same access pattern when interacting with metadata. A similar optimization is also used by the engine (in both implementations) to speculate on some operations for their locality: when possible, the runtime barrier for such operations is just a simple assumption check, which does not require any TM barrier.

8.3 Performance Evaluation

The Truffle.PEV engine has been evaluated using different workloads and use cases with the goal of assessing the performance of the two PEV implementations in the context of different workloads. To this end, the evaluation has been focused on workloads belonging to the following categories:

- *Read-dominated workloads*. The Truffle.PEV engine's main target is server-side computing, where workloads are often stateless or read-only. In this context the PEV programming model can be used to automatically speculatively parallelize the execution of the service, as discussed.

- *CPU-intensive workloads*. Having the possibility to execute event handlers in parallel opens up the opportunity for executing new, previously unsupported, workloads that can mix shared data and CPU-bound computations in JavaScript. Since the PEV supports shared state, such benchmarks are *not* purely-functional, side-effects-free benchmarks. Rather, they make use of some shared state during the computation, with a reduced number of conflicting event handlers.

- *Data-intensive workloads*. The speculative runtime allows the engine to execute in parallel also workloads that make intensive use of shared state with conflicting accesses. Data-intensive workloads have also been used to assess the effectiveness of the runtime in the context of conflicting event handlers and non-parallelizable workloads.

The PEV runtime is expected to expose good scalability with the first two classes of benchmarks. Depending on the STM runtime and on the scheduling policies, however, the two runtimes are expected to behave differently, with the Optimistic runtime offering better scalability at the cost of higher runtime overhead. Concerning the third class of experiments, the Hybrid PEV runtime is expected to offer better latency, as it relies on an STM algorithm designed to limit the overhead.

The performance evaluation has been conducted measuring the following aspects:

- Barriers and runtime overhead: the two PEV runtimes implement different runtime barriers. The selected benchmarks have been executed comparing the performance of single-threaded instances of PEV runtimes running with the required runtime barriers against a non-modified single-threaded version. Both measurements also include other runtime-level overhead that are shared by each of the runtimes, e.g., event emission.
- Scalability: each benchmark is executed with an increasing number of parallel threads to assess the ability of the runtime to exploit the resources of the system as they are added.
- Reordering overhead: ordered, chained, and unordered events are exposed as distinct classes of events in the PEV model. Therefore, they should not be considered equivalent, but rather alternative solutions: when event ordering is needed by the semantics of the application, ordered events should be used; when event ordering is not needed (or can be relaxed) other classes of events should be preferred. Despite this programming model distinction, we modified some of the benchmarks by introducing different event classes with the goal of measuring the overhead of executing events as ordered or unordered.

The above aspects (combined) give an idea of the characteristics of the two Truffle.PEV runtimes, and can be used to identify under which circumstances a runtime is preferable over the other.

A set of new and existing workloads has been used for the evaluation. All the benchmarks have been ported to JavaScript and *adapted* to comply with the event-based programming model of the event loop. In particular, all the benchmarks have been converted to resemble the form of event-based services for which each request triggers the execution of one or more event handlers.

Using this benchmarking approach implies that every benchmark will also make use of event emission, buffering, and dispatching, and the results of the performance evaluation will also be affected by all the components involved in the PEV system, including the shared queue. To avoid measuring the overhead introduced by the I/O substrate, I/O requests have been simulated by directly adding events to the global queue.

Because of the event-based benchmarking approach, it was not possible to do a full port of some popular STM benchmarks (e.g., [129]) to the PEV model, mostly because of the impossibility of adapting them to the JavaScript language[2].

Every experiment presented in this section has been executed on a Dell I329 server machine with 128GB or RAM running 4 Intel Nehalem i7 CPUs for a total of 24 cores (with hyper-threading and dynamic frequency switching disabled to reduce nondeterminism). The machine has a NUMA-cc architecture. All the experiments have been

[2]For instance, some benchmarks such as STAMP [129] explicitly rely on the presence of threads and on the ability to interleave transactional with non-transactional code.

executed multiple times, and we report the average peak performance of the different PEV runtimes. Peak performance is obtained after a warmup time that is big enough to let the Graal VM JIT compiler to optimize the executed code. Warmup time is omitted since we focus on benchmarks deployed as services.

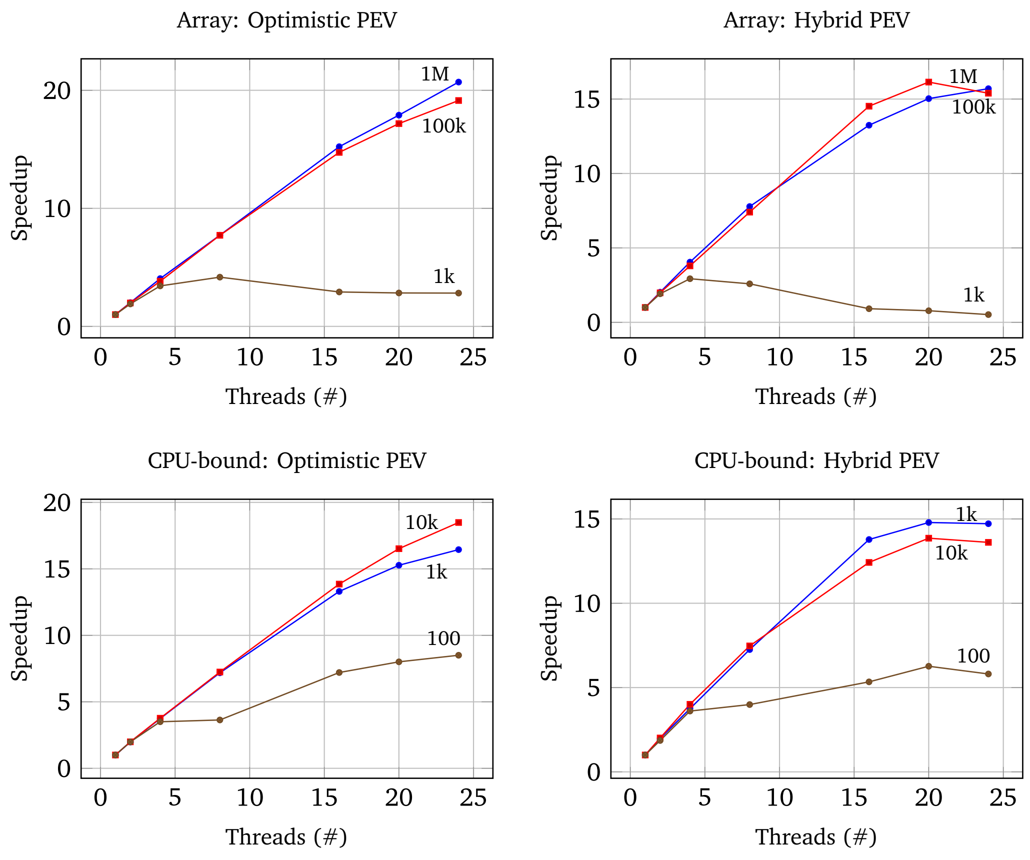


Figure 8.2. Truffle.PEV microbenchmarks. Speedup is relative to a Truffle.PEV engine running with 1 worker thread. Each experiment is executed with a different size of the input parameter: 1k (10^3), 10k (10^4), and 1M (10^6).

A total of nine different benchmarks (plus some microbenchmarks) have been adapted to run with the Truffle.PEV engine.

8.3.1 Event emission

The first set of microbenchmarks has been designed to assess the overhead derived from event emission in the context of different PEV runtimes, as well as to have an estimate of the maximum performance of the PEV system with ideal workloads (i.e., workloads that the PEV engine can easily parallelize). The microbenchmarks consist of two distinct classes of experiments aimed at assessing the baseline performance for each PEV runtime system:

- *Pure stateless*. This benchmark consists of executing a growing number of event handlers that simply keep the CPU busy with a transaction-local CPU-intensive operation, that is, the creation, allocation, and sorting of an array not escaping the event handler's scope, and that therefore can be considered private by the PEV STM runtime, which will remove all the TM barriers for accessing it. The goal of this benchmark is to measure the maximum throughput of the PEV engine when the only shared state is represented by PEV runtime itself (e.g., by the global event queue).
- *Read-intensive*. This benchmark has the goal of assessing the performance of the PEV engines with read-only workloads. For this class of benchmarks, To this end, a shared immutable data structure is allocated on the service's heap, and is accessed by multiple event handlers in parallel. The benchmark also gives an indication of the effect of the impact of the memory bus in the machine considered for evaluation.

To increase the potential for parallel execution, all the benchmarks of this section use unordered events only. The results are depicted Figure 8.2. For each benchmark the execution time is presented with an increasing input parameter size. For the CPU-bound benchmarks the input corresponds to the average execution time for a single event handler (with an increasing size of the array to be sorted), whereas for the read-intensive benchmarks it corresponds to the size of the shared data structure, implemented as a single JavaScript array object instance. The CPU-intensive microbenchmark shows that the PEV system can scale almost linearly with a relatively small size of the array to be sorted. Contention can become a problem for both PEV runtimes when the size of the computation to be executed in parallel is too small (i.e., 1k). The microbenchmarks also show that the two STM runtimes (i.e., TL2 and FastLane) have different scalability characteristics. In general, TL2 scales better than FastLane in both workload types.

8.3.2 CPU-intensive workloads

To assess the performance of the engine with CPU-bound workloads, some CPU-intensive workloads have been adapted to use the event-emission API of the PEV. The benchmarks selected for evaluation are the following:

- Primes. A brute-force prime number calculator. The benchmark calculates all the prime numbers for a given range, storing the results in a shared data structure. Unordered events only are used in this benchmark, and only a single event target is used.

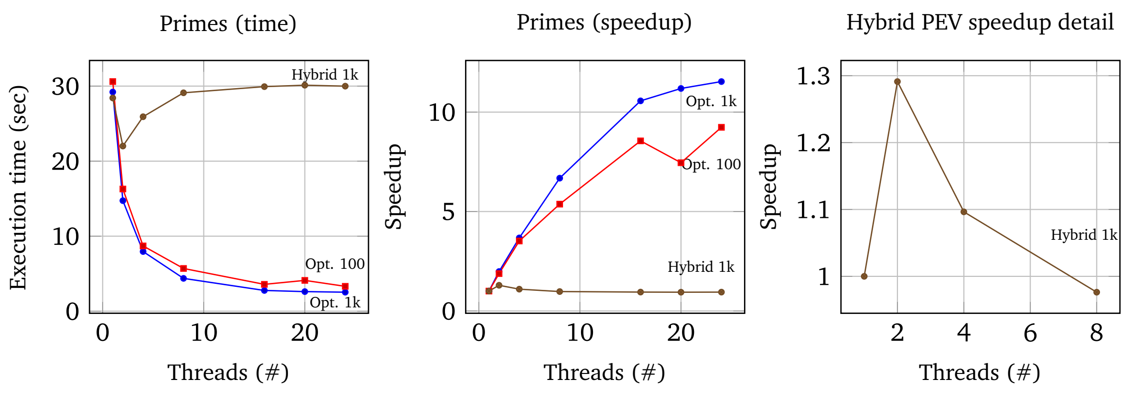


Figure 8.3. Primes benchmark: Optimistic vs. Hybrid PEV. Speedup is relative to the corresponding Truffle.PEV engine running with 1 worker thread.

- Mandelbrot. A mandelbrot set calculator. The resulting Mandelbrot set is stored in a shared data structure. Also with this benchmark, unordered events are used and there is one event target.

- Matrix multiplication. A matrix multiplication benchmark derived from an STM benchmark, adapted to single-threaded event emission. The two input matrices are accessed read-only by the benchmark, and results are stored on a third shared matrix. This benchmark makes use of unordered events to perform the parallel multiplication, and chained (ordered) events to assemble the final result.

- PageRank. The popular PageRank algorithm [43] adapted to use event emission. The benchmark performs a map-reduce-like computation where each map computation is modeled ad an unordered event that does not modify shared state (i.e., it is a read-only handler). Another version of the same benchmark modified to use shared state also on the map operation is discussed in the next section. The benchmark makes use of a combination of chained and unordered events.

Performance data for each CPU-bound benchmark with different workloads is depicted in Figures 8.3 and 8.4. The Primes benchmark shows that with CPU-intensive computations that involve shared state only for a few objects the TL2-based runtime can offer very good scalability, as the event handlers conflict very rarely. Conversely, the FastLane-based algorithm cannot offer the same scalability. This is partially, expected, as the FastLane algorithm is designed to scale only with a limited number of threads. Another reason for the FastLane-based implementation, however, is workload-specific, and depends on the fact that the Primes benchmark always modifies the same data structure. The main thread (with higher priority) will therefore almost always cause the other event handlers to abort. Performance are similar with the Mandelbrot benchmark (which also involves a single shared data structure), but are different with

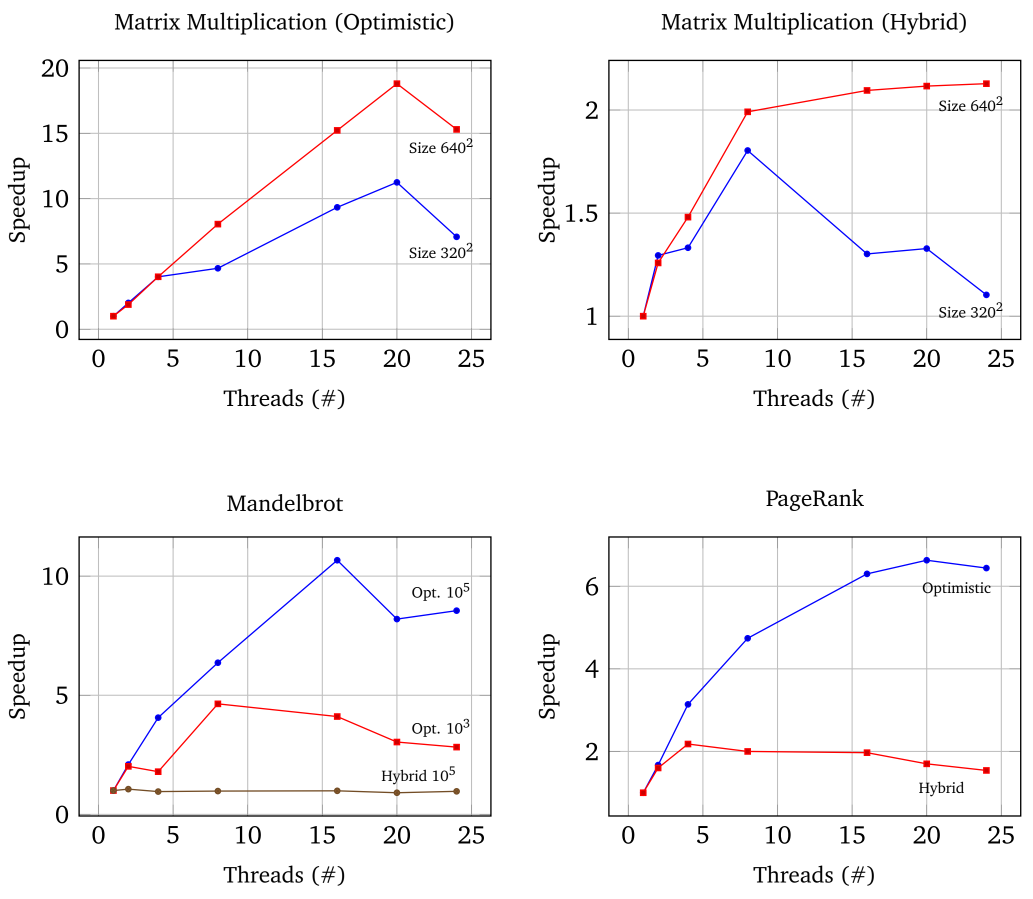


Figure 8.4. CPU-bound workloads: Matrix multiplication, Mandelbrot and PageRank. Speedup is relative to the corresponding Truffle.PEV engine running with 1 worker thread.

the other two benchmarks (Matrix and PageRank in Figure 8.4), where the workload mixes access to different data structures. In this case, both PEV runtimes can offer good scalability. In general, the Optimistic runtime is able to expose a better scalability, since all event handlers have the same opportunity of committing. This is particularly clear with the Primes and Matrix benchmarks, where the engine has almost ideal speedup for certain workload sizes. Conversely, the hybrid STM cannot expose high scalability for such workloads, as the main thread always modifies the shared data structure with higher priority, often forcing other transactions to abort and restart. The effect is attenuated in benchmarks where most of the computation performed by parallel event handlers are not conflicting (as operating on distinct or read-only object instances).

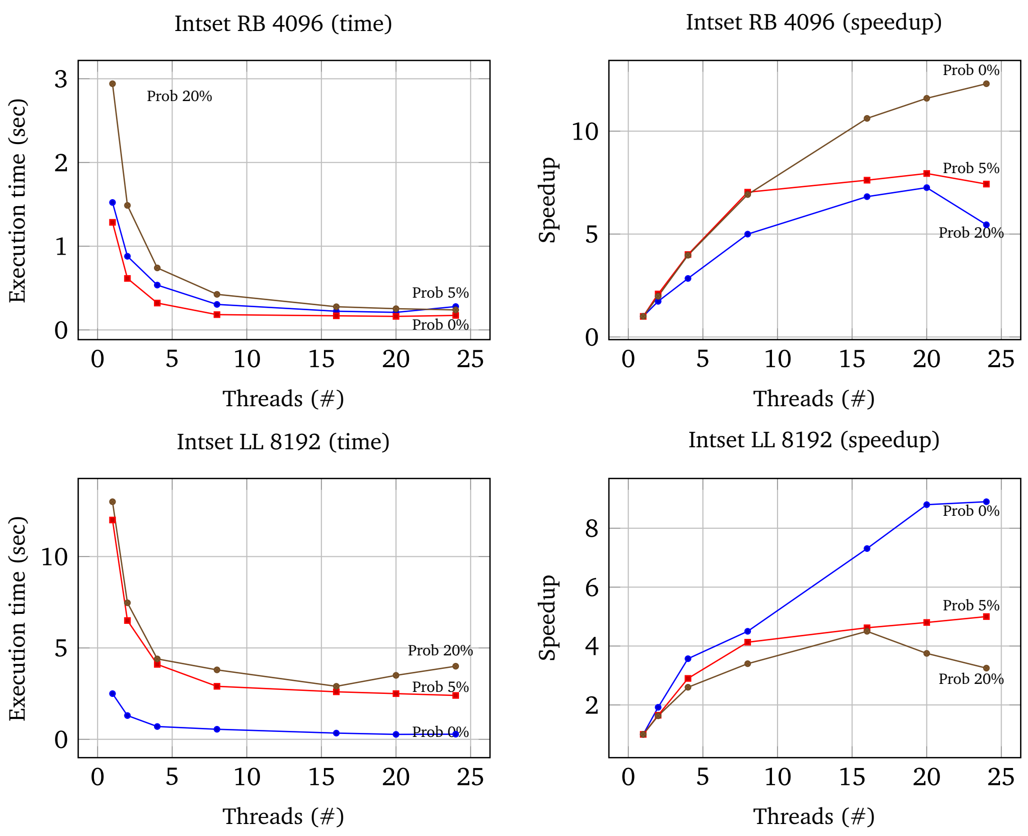


Figure 8.5. IntSet benchmark with the Optimistic PEV. Speedup is relative to a Truffle.PEV engine running with 1 worker thread.

8.3.3 Read-intensive and data-intensive workloads

To assess the performance of the STM runtimes executed by the PEV system, we also adapted some data-intensive server-side and STM benchmarks to the event-based model. The following benchmarks have been considered for evaluation:

- Intset. The service consists of a set of integer numbers implemented with different data structures (i.e., a linked list and a red-black tree). The service is queried with an increasing number of write requests (modify or delete, with write probability going from 5% to 20%). The benchmark simulates a service cache, and therefore makes use of different event target (one per client request). Each event target might issue a number of unordered events.

- Bank. The benchmark simulates a bank Web service mediating online payments. The benchmark is based on unordered events.

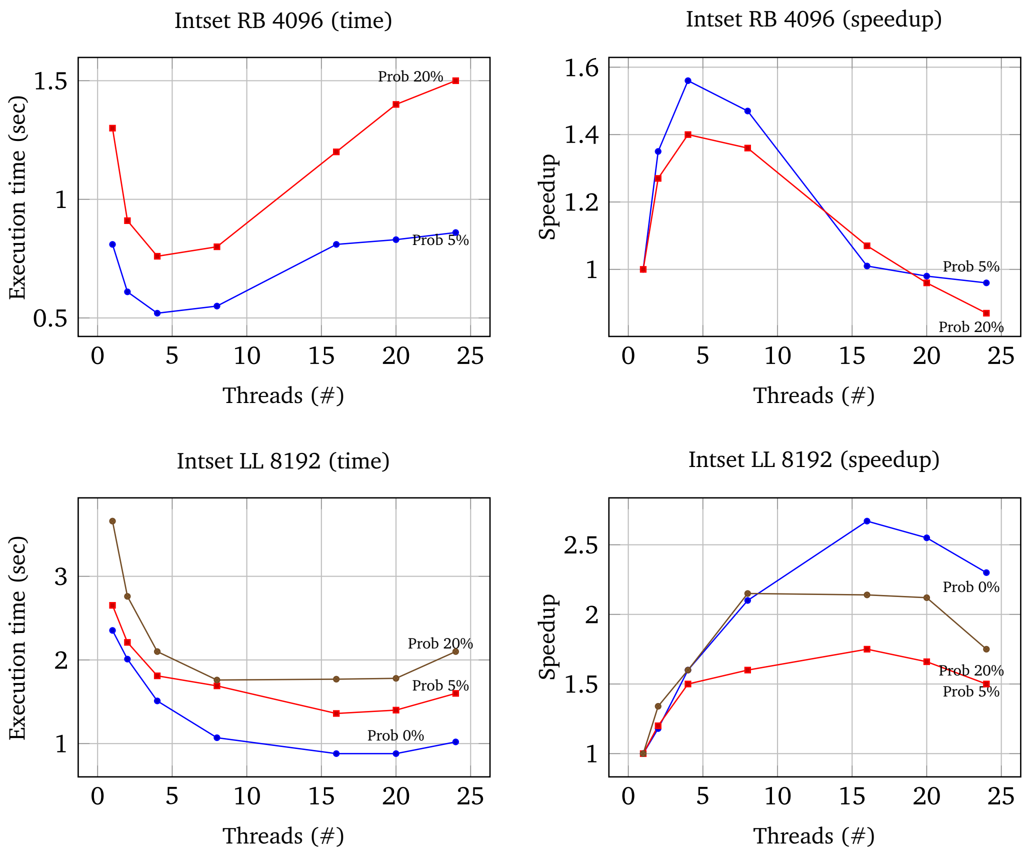


Figure 8.6. IntSet benchmark with the Hybrid PEV. Speedup is relative to a Truffle.PEV engine running with 1 worker thread.

- WordCount. The benchmark simulates a service hosting some static files, and performing some queries over the files. Since every client request requires the service to scan multiple files, each file correspond to one or more event emissions, and therefore multiple files are parsed in parallel for every single client request. The service makes use of a shared data structure to accumulate the temporary state needed in order to produce the request, and is therefore a write-intensive benchmark. The benchmark is based on unordered and chained events, using only the global event target.

- PageRank (with side effects). The benchmark implements the PageRank algorithm presented in the previous section, with the main difference that each "map" task updates a shared data structure in place to store the temporary data structures thus increasing the amount of conflicting event handlers. This version of the benchmark corresponds to a more JavaScript-like way of writing sequential

CPU-oriented applications, as side effects are used to store temporary results.

The performance data for the IntSet benchmark is presented in Figure 8.5 and 8.6. As expected, the Optimistic system features a better scalability compared to the hybrid system. The two figures also show the absolute execution time for the benchmarks (corresponding to the average execution time for performing a fixed number of operations on the data structure). The execution time shows clearly that the two runtime systems offer a trade-off between scalability (i.e., throughput) and latency. With a low number of threads, the Hybrid runtime can serve client requests with a latency which is up to 4 times lower than the one offered by the Optimistic runtime, and is very close to the one of the non-modified single-threaded runtime. Conversely, the Optimistic runtime offers higher throughput also for data-intensive computations. The performance data for the other benchmarks (in Figure 8.7 and Figure 8.8, and Figure 8.9) seem to confirm this trend, always having the Optimistic system exposing better scalability with an increasing number of requests at the cost of response latency. In general, the benchmarks indicate that when the workload involves data-intensive operations with no CPU-intensive operations of any type the two runtime systems offer different performance characteristics, with the TL2-based one offering better scalability for data-intensive computations at the cost of execution latency, whereas the FastLane-based one can offer almost the same execution latency of a single-threaded event loop, but can scale only up to 3x times the execution time of a single-threaded loop.

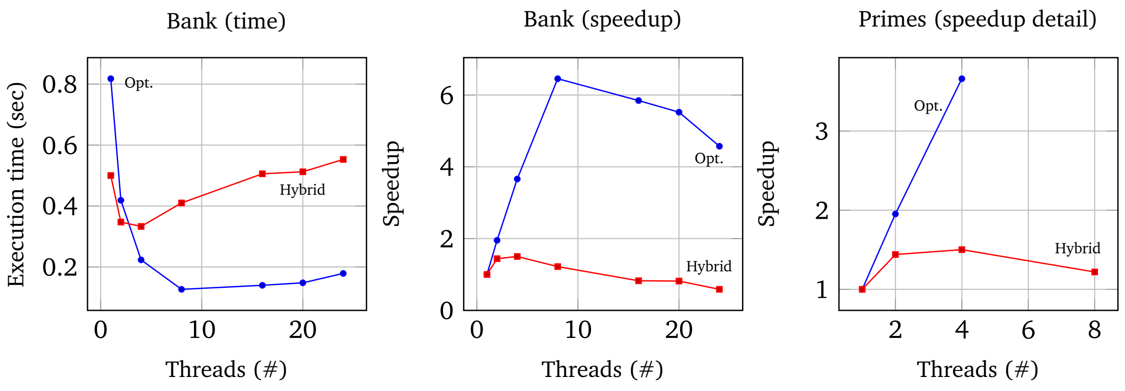


Figure 8.7. Bank benchmark. Speedup is relative to the corresponding Truffle.PEV engine running with 1 worker thread.

8.3.4 Runtime overhead

In order to assess the runtime overhead of runtime barriers, all the benchmarks introduced in the previous sections have been executed running the Truffle.PEV runtimes with a single thread only (i.e., the main thread). All the single-threaded executions

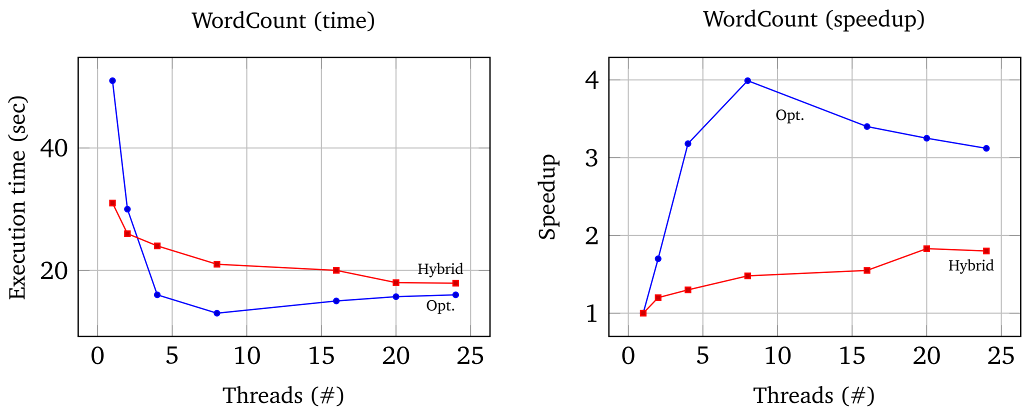


Figure 8.8. Truffle.PEV WordCount. Speedup is relative to the corresponding Truffle.PEV engine running with 1 worker thread.

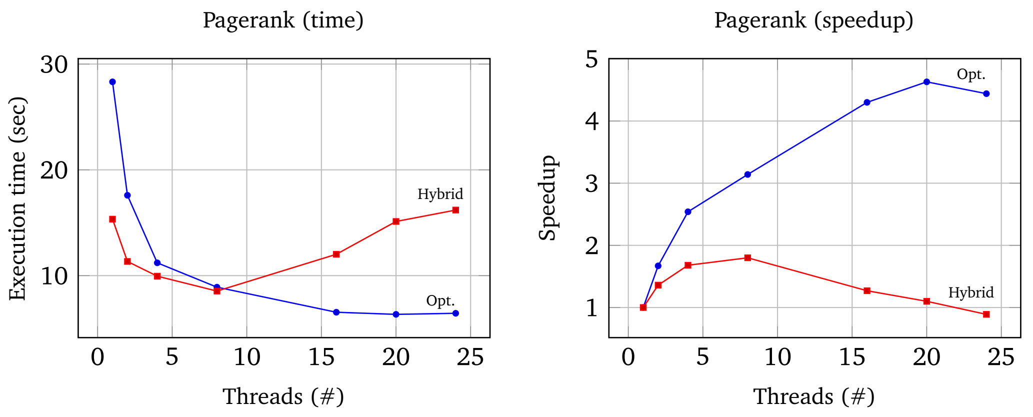


Figure 8.9. Truffle.PEV Pagerank. Speedup is relative to the corresponding Truffle.PEV engine running with 1 worker thread.

have then been compared against a non-modified single-threaded execution of the benchmarks. In this way, it is possible to measure the cost of the runtime barriers for the two PEV runtimes. Results are described in Table 8.1. As a reference, the experiments have also been executed using the Nashorn [10] JavaScript engine included in the recent release of the JDK1.8. Nashorn is a state-of-the-art JavaScript runtime for the JVM, and is included in the evaluation as a reference for the non-modified Truffle JavaScript engine. Other engines such as V8 and SpiderMonkey tend to perform better than Nashorn, but are not based on the JVM runtime.

As depicted in the Table, the overhead for the benchmarks considered is in line with the runtime overhead of STM-based runtimes, going from 1.15% up to a factor of

	Average	LL 8192,Prob.0	LL 8192,Prob.5	LL 8192,Prob.20	RB 4096,Prob.0	RB 4096,Prob.5	RB 4096,Prob.20	Mandelbrot	Primes	Bank	Wordcount	Pagerank	Matrix
Not-modified	1	-	-	-	-	-	-	-	-	-	-	-	-
Optimistic	3.25	1.16	3.88	3.75	1.2	5.31	4.88	2.86	1.11	3.43	4.43	2.65	4.31
Hybrid	1.28	1.16	1.33	1.25	1.5	1.31	1.36	1.07	1.01	1.75	1.23	1.16	1.13
Nashorn	1.95	1.26	1.39	1.25	3.5	2.81	3.23	1.05	2.78	2.12	2.22	1.12	0.75

Table 8.1. PEV runtime overhead (% slowdown factor).

4 times the non-modified execution time.

As expected, the Hybrid runtime is able to expose a lower overhead, and such runtime is always faster than the Optimistic one. In general, however, it has to be considered that for server-side applications the overhead of the Optimistic runtime might still be acceptable, as it allows the service to serve clients in parallel.

8.3.5 Chained and ordered events

All the benchmarks presented in the previous sections use a mix of ordered and unordered event emission, usually with unordered events to trigger the execution of some computation in parallel. To assess the performance of the two PEV runtimes with event handlers involving a more complex interaction with the different classes of events they support, we modified some of the benchmarks of the previous section and we measured the scalability with an increasing amount of ordered tasks for a same given event target.

As we did for the previous section, we also modified the microbenchmarks of the previous section to measure the systems in ideal conditions. For each experiment we executed the microbenchmarks with two distinct event classes, one with ordered event ordering and another with out-of-order execution. We then measured the performance of the PEV engines with an increasing amount of ordered tasks. As defined by the PEV programming model, each ordered task has to wait for its predecessors before execution; an increasingly number of ordered tasks is therefore expected to reduce the throughput of the engine, as event handlers must be executed sequentially. Moreover, an increasing number of ordered tasks might introduce additional overhead due to out-of-order speculative event execution, as executing such event handlers in parallel might reduce the ability of the PEV system to process other unordered events (if any). This is however balanced by the ability to speedup ordered event execution when event handlers do not conflict.

The results for the two classes of microbenchmarks discussed are depicted in Figure 8.10. To show the performance degradation caused by commit-time reordering and avoid measuring other aspects such as the contention on the global queue, we executed the experiment on the ideal case, where both systems can offer ideal scalability. In gen-

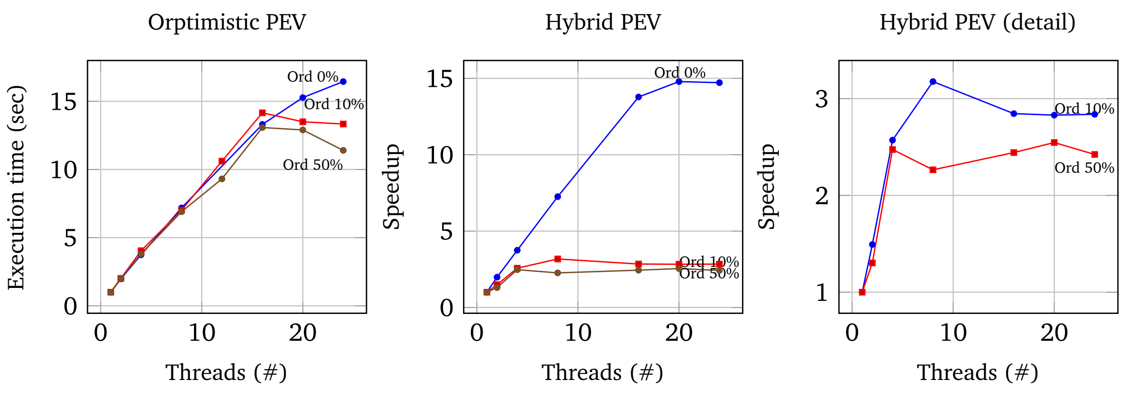


Figure 8.10. Array microbenchmark (1M) with an increasing number of chained events. Speedup is relative to a Truffle.PEV engine running with 1 thread.

eral, the microbenchmarks show that with an increasing amount of (non-conflicting) event handlers that have to wait for their predecessor before committing the overall system's performance are affected very differently, depending on the PEV implementation. For the Optimistic runtime, a small number of ordered event handlers does not affect scalability until the number of parallel workers reaches a certain number. This is partially because the Optimistic runtime also executes ordered handlers in parallel, speculatively. Since event handlers do not conflict, the runtime can effectively execute them in parallel. Things are different with the Hybrid runtime, in which ordered tasks are executed by the main thread, and only a subset of the worker threads attempts to speculatively execute them out-of-order. This speculative approach offers scalability for non-ordered handlers only up to a factor of 2x, and shows that the Hybrid PEV can speculatively execute ordered events out-of-order only for a limited factor.

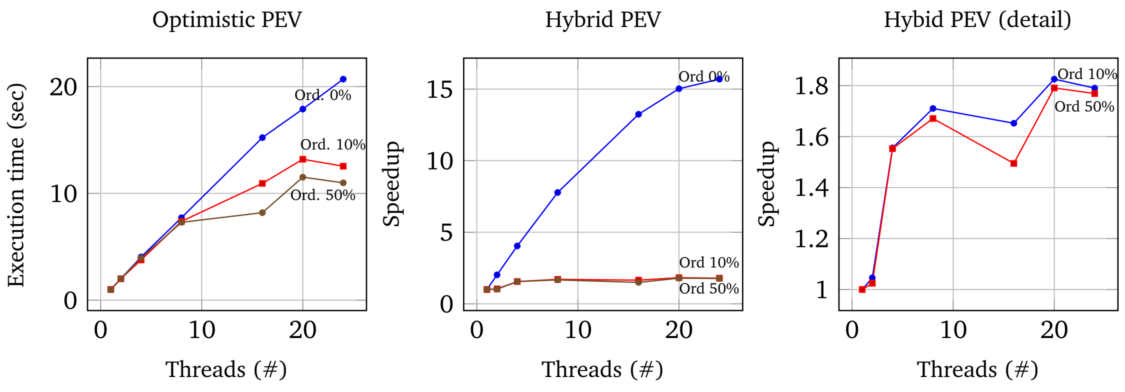


Figure 8.11. CPU-intensive microbenchmark (1M) with an increasing number of chained events. Speedup is relative to a Truffle.PEV engine running with 1 thread.

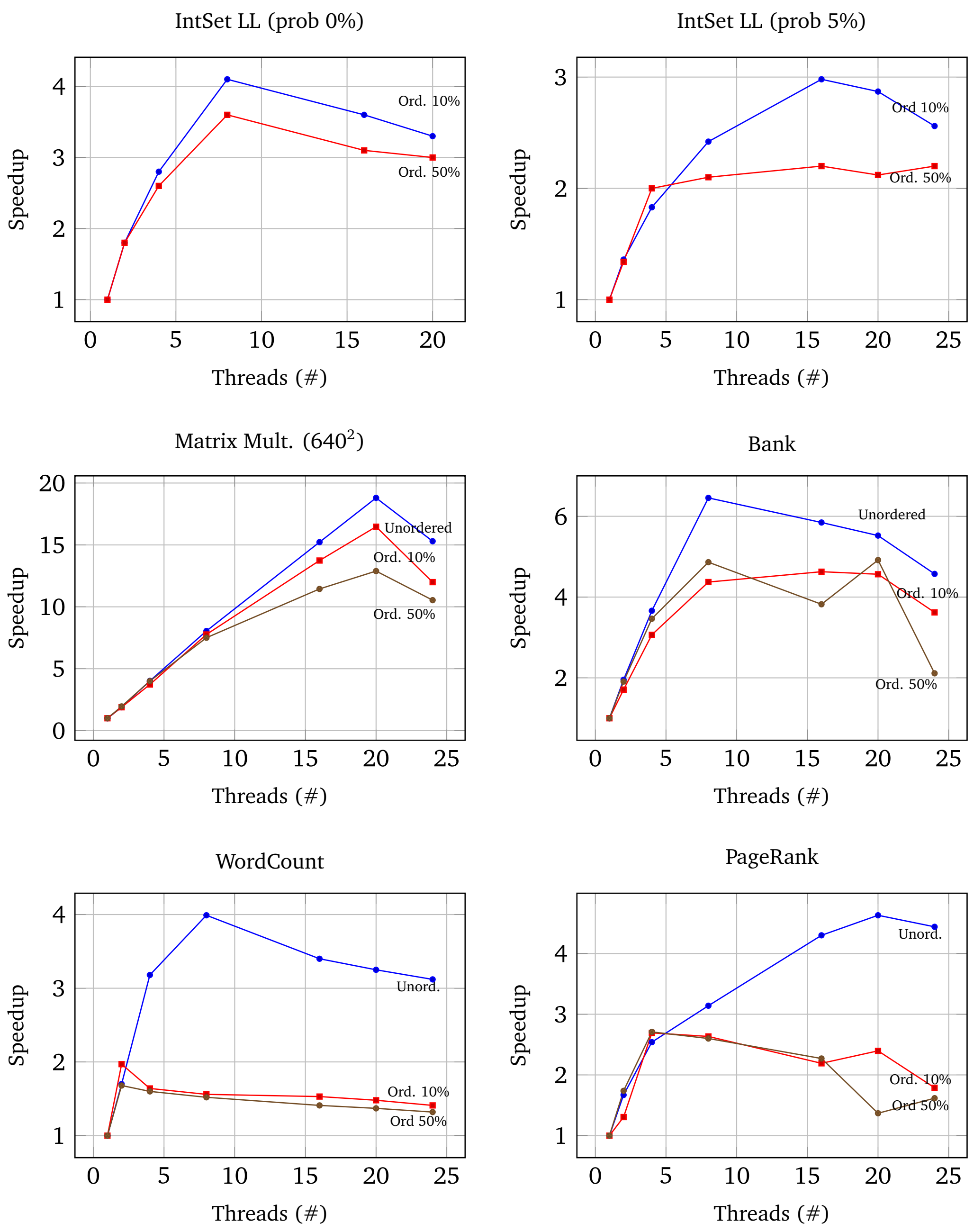


Figure 8.12. Truffle.PEV Mixed chained and unordered workloads. Speedup is relative to an Optimistic runtime running with 1 worker thread.

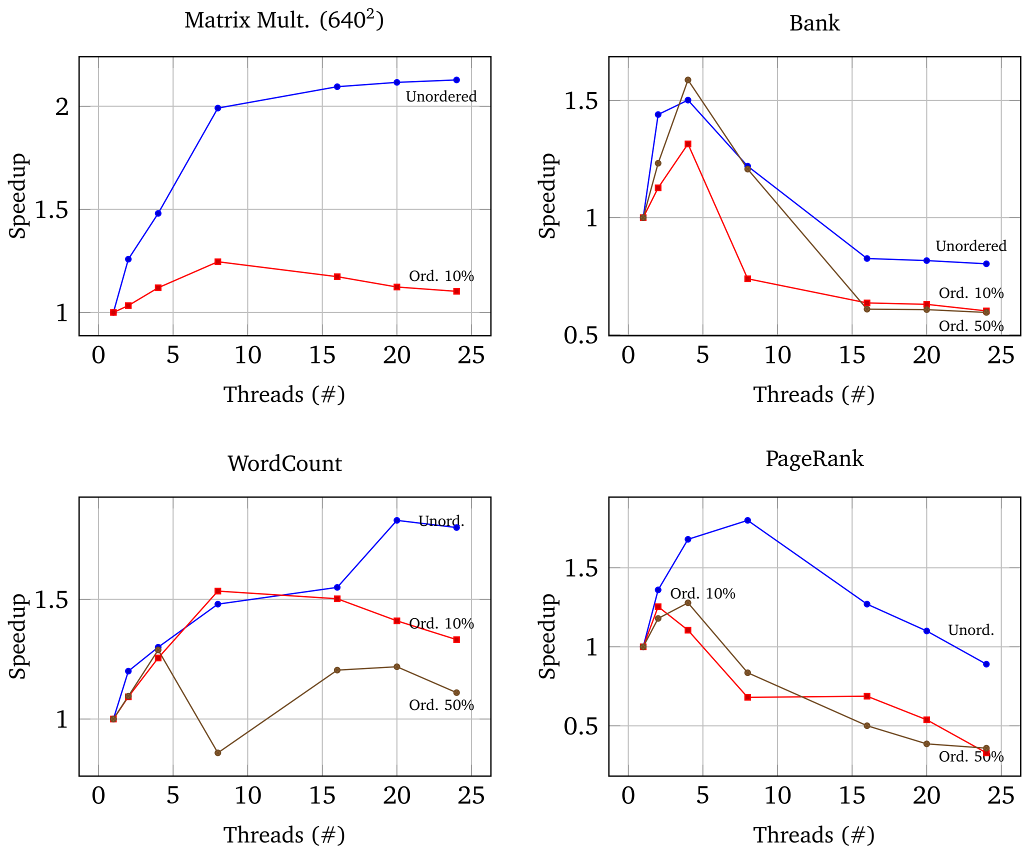


Figure 8.13. Truffle.PEV Mixed chained and unordered workloads. Speedup is relative to an Hybrid runtime running with 1 worker thread.

We also executed a subset of the benchmarks presented in the previous sections with the same approach, i.e., with ordered events. Results for the Optimistic runtime are depicted in Figure 8.12, while the same experiments using the Hybrid PEV are presented in Figure 8.13. For both runtimes we present data only for experiments in which the PEV was able to improve its performance by using speculative parallelization.

For what concerns the Optimistic runtime, the benchmarks confirm that a limited number of ordered event handlers affects the system performance by a factor that is roughly proportional to their number. In particular, when the workload is using a limited number of ordered event emissions (e.g., 10%) most of the benchmarks offer performance close to the one of unordered event emission, thus confirming the effectiveness of the speculative parallelization performed by the PEV runtime. When the number of event handlers that have to be executed respecting global ordering becomes significant (i.e., 50%) the Optimistic PEV runtime can still offer some scalability.

For what concerns the Hybrid runtime, the benchmarks are in line with the microbenchmarks, as the increasing number of ordered events limits the scalability of the system by a considerable factor. In all the considered benchmarks, however, the engine can still offer some speedup over single-threaded execution. Considering that the Hybrid system's focus is runtime overhead this confirms that also with ordered events the engine can still offer some improvement.

8.3.6 Summary

The two PEV runtimes based on the Truffle framework introduced in this Chapter offer different performance characteristics, with the Optimistic runtime almost always outperforming the Hybrid one in terms of scalability, and -dually- the Hybrid one almost always offering a lower overhead. In all the considered cases the Optimistic system offers a better scalability with mixed ordered and unordered events, even presenting good scalability for benchmarks featuring ordered events that do not conflict, leading to an effective parallelization of the workload (e.g., matrix multiplication and intset). In general, when the workload offers some opportunities for parallel execution, either because event handlers do not conflict or because unordered event emission is used to implement certain computations (or both) the two engines can improve the execution time of the event-based application through parallel execution. In particular, the Optimistic PEV runtime can offer very good speedup when the workload is nicely parallelizable, and still offers opportunities for parallel execution when data is contented between event handlers. When contention is high, the Hybrid PEV should be preferred over the Optimistic one, as it offers better execution latency. For pathological cases (e.g., applications in which every event handler cannot be executed in parallel), a single-threaded event loop could be preferred over the parallel event loop: thanks to the programming-model compatibility enforced by the PEV API, it is always possible to deploy applications developed with unordered execution in single-threaded runtimes.

Part IV

Epilogue

Chapter 9

Related Work

This Dissertation covers research topics from multiple fields. Related work can be found from programming languages to runtime systems research, through Web services and service-oriented systems, up to software transactional memory runtimes and speculative runtimes in general. In this chapter we give an overview of the related research work which has not been already covered in other chapters of this Dissertation.

9.1 Server runtime systems

In the past decades, significant engineering and research efforts have focused on Web server design [49]. Server runtime architectures can be roughly classified into three categories [135]: one-process/thread-per-connection, event-based, and hybrid [89, 116]. The first category corresponds to servers in which each connection is assigned a single thread/process which handles the request and generates the appropriate response. This class of services is usually characterized by a good response latency, but presents scalability issues in terms of concurrent connections handling. Also, instantiating a thread per connection has an impact on memory utilization. The second category of runtime design corresponds to servers in which every I/O-related operation is associated with a callback which is executed upon the emission of a specific runtime event. Events are enqueued in a specific queue and are processed by an event loop executed on a single process. This design solution allows a single process to handle multiple connections per thread/process, thus reducing the memory footprint of the server. The third class corresponds to servers designed using a hybrid architecture featuring both threads and event queues. An important example of such architectures is represented by the Staged Event-Driven Architecture (SEDA) [155]. In SEDA the server is composed of multiple processing stages, each of which processes events similarly to an event loop, but using a pool of concurrent threads. For what concerns the first two categories, there also exist several examples of event-based Web servers [133], as well as thread/process-based servers [149].

A long-running debate (e.g., [132, 149]) comparing the merits of the two approaches has been summarized in [135]. In the same paper, an exhaustive evaluation shows that event-based servers yield higher throughput (in the order of 18%) compared to thread-based servers under certain conditions (i.e., for serving static pages).

In the context of hybrid server systems, recent research has attempted to parallelize existing event-based static HTTP servers (i.e., Web servers serving static pages to clients) with techniques to statically or manually identify which callback could be executed in parallel. For instance, the *libasync-mp* library [158] enables event-based servers to execute multiple callbacks on multiple cores. The parallelization approach adopted by the library is based on callback *coloring*: each callback is assigned a color (manually, by the developer), and callbacks with the same colors will be executed on the same core, while callbacks having different colors will be executed on different cores. Coloring-based approaches have shown good performance, and further research has also demonstrated possible optimizations based on work-stealing [85]. The main limitation is that developers must manually indicate which callback should be executed in parallel. To overcome this limitation some static analysis techniques to automate color assignment to callbacks can be used. For instance, Elyze [134] is a static-analysis tool which analyzes the source code of multiple callbacks, and by identifying sets of callbacks sharing the same data can schedule them in parallel. Callbacks coloring is a static, manual, scheduler-based, approach which shares with event targets in the PEV the goal of scaling services when multiple connections can be processed in parallel. Event targets are however a programming model abstraction, and can be employed in contexts other than connection handling. Moreover, coloring-based approaches do not benefit from any dynamic optimization, and therefore might offer only limited scalability if employed at the VM-level as the PEV[1].

9.1.1 Asynchronous programming and event-based services

As discussed, event-based servers [61, 135] have proven to be very scalable, as they are able to handle concurrent requests with a simple and efficient runtime architecture [118, 115]. However, programming services to run on asynchronous event-based servers has always been considered a complex task. Event-based services developed in native languages such as C are subject to the so-called problem of stack-ripping, which forces the developer to manually manage the data needed for processing each asynchronous operation, as successive callbacks (one per event) cannot be called sequentially, and thus cannot benefit from the automatic stack management found in modern languages. Also, the presence of multiple callbacks makes it hard to compose multiple I/O operations based on multiple callbacks.

[1]For instance, in a coloring-based system a callback featuring an *if* block that cannot be statically analyzed must be "pessimistically" considered as unsafe for parallelization, whereas any PEV runtime system will adapt its parallel execution strategy dynamically.

Another limitation is that the event loop forces the developer to use asynchronous callbacks invocation. Such model often requires to use the continuation passing programming style [104] in which either callbacks must be nested or manually managed [108]. Furthermore, continuation passing style might require developers to invert the control flow of the application's logic, for instance requiring to register the event handler for a specific I/O operation before the I/O operation starts. Stack ripping and inversion of control, are outside of the scope of this Dissertation, as several solutions have already been proposed [105].

A proposed solution to the issues of low-level native asynchronous programming is the AC [90] ("Asynchronous C") programming model adopted for implementing the I/O primitives in the Barrelfish OS [36]. With AC any I/O operation is expressed using a synchronous call which is then executed asynchronously. The approach introduces some new constructs (like `async` and `do ... finish`) to the C language and adopts a compiler-based approach (or C-Macros, where possible) to translate synchronous operations into operations which will be executed asynchronously. An alternative approach is represented by *Conch* [107]. Conch is a library for event-based development of network applications which increases the abstraction level for the I/O interaction, thus enabling systems developed using its API to be efficiently executed on both event-based and thread-based runtime systems. Like AC, Conch adopts a compiler-based approach (more precisely, a source-to-source translator), and a library.

Both the programming models of AC and Conch share with the PEV model the notion of callback-based asynchronous programming. Differently from such approaches, the PEV is explicitly designed to be integrated in the (dynamic) language used for the development of the service core business logic, whereas AC/Conch are limited at connection handling and processing. Moreover, none of the approaches above relies on the impression of single-threaded concurrency, and leaves to the developer the management of the parallelization of the service (which has to be carried out using traditional techniques). Finally, other approaches have been proposed to ease the management of event-based programs using static analysis techniques (such as *eel* [58]).

9.1.2 Event-based frameworks

The performance of event-driven architectures has promoted custom programming models for Web service development that rely (explicitly or implicitly) on event loops. Frameworks of this class are based on modern Operating Systems' asynchronous APIs such as `epoll` or `select` [135]. Examples of event-based frameworks include libraries (e.g., Java NIO [35], and Ruby's EventMachine [18]), and language-level integrations such as Node.JS [111]. A similar approach with relevant differences is represented by a JVM-based framework called Vert.x [26]. Vert.x features the same asynchronous programming model of Node.JS, with the main difference that no explicit process-based parallelism can be used, and multiple requests are automatically distributed across multiple share-nothing event loops. Vert.x uses a coloring-based parallelism model for callbacks execution (each new connection is assigned a core which will handle the request until its end, preventing per-request parallelism). Differently from Node.JS, Vert.x processes can share some global data structures between multiple processes. Such data structures must be read-only and cannot be modified, and differently from a PEV model, Vert.x-based services cannot exploit per-request parallelism (i.e., each single request cannot be processed in parallel, unless standard JVM-based models such as Threads and locks are used).

Out of the realm of event-based systems, other approaches have been proposed to deal with asynchronous I/O. For instance, a thread/events hybrid approach has been proposed by Li and Zdancewic's in [117]. Their approach based on Haskell allows to write highly concurrent services using monads as an abstraction for both threads and events. The F# language [144] also features a set of rich asynchronous primitives with elements of type `Async<T>` representing computations which could return before their completion. The asynchronous extensions of F# (and their similar counterpart for the Scala language [1]) enable a reactive-like programming style similar to the one described in Chapter 4 with the PEV. The main difference is that the PEV targets single-threaded runtime systems providing safe parallelism, whereas these asynchronous extensions address more complex runtime systems (such as the CLR or the JVM) and usually do not provide safe parallel execution.

9.1.3 Event emission and Asynchronous task-based programming

The event-based model shares some similarities with the asynchronous execution of tasks supported by popular frameworks such as the Java Fork/Join Pool [110]. Indeed, from a pure runtime perspective any task asynchronous executor service (including the Fork/Join pool) running with a single worker thread is conceptually equivalent to a single-threaded event loop[2]. However, differences with respect to common executor services can be summarized by the following two aspects:

- *Safe execution*. In a PEV model every task is protected against data raced from other tasks by enforcing isolation and atomicity at the runtime level.
- *Ordered execution*. The PEV model enforces ordering via a specific API, whereas an executor service attempts to execute parallel tasks as soon as possible.

Following with the analogy with other parallel programming models for shared-memory concurrency, event handlers resemble the approach of *safe futures* [153], with the main difference that safe futures have to be protected against races with code executed outside of the future (enforcing so-called strong atomicity [91]).

[2]There are, however, still relevant differences in the way an event loop interleaves parallel processing of I/O with sequential execution of callbacks, as in the PEV model every event handler does not execute I/O operations using blocking primitives.

9.1.4 Explicit parallelism models

The one-process/thread-per-connection model can be implemented using several general-purpose parallel programming models. Systems developed using a separate process-per-connection (all threads of execution are in different address spaces) are often called multi-process servers (MP), while shared-memory thread-per-connection architectures (multiple threads within a single address space) are called multi-threaded servers [133] (MT). MP services can be implemented with popular high-performance libraries (e.g., boost [3]), requiring the developer to manage the connection handling phase (for instance, using an embeddable HTTP server as a front-end [11]), and preventing multiple processes from sharing a common memory space. This has the advantage of allowing a more simple deployment on distributed architectures. MT services can be implemented with basically any programming language supporting multi-threading. Despite the advantage of having a common memory space, the coordination and the synchronization between threads has to be manually implemented by the developer. Other, more high-level, solutions for the development of such services exist (e.g., Servlets [9] and RESTlets [120]). However, such models still require manual synchronization.

9.1.5 Actors

Alternative to threads, the Actor model [31] is a concurrency and coordination abstraction based on component isolation and asynchronous message passing. The Actor model lies between high-level implicit parallelism approaches and explicit parallelism models, as it enables to express parallel applications without having to deal with concurrency primitives such as locks or barriers, but it still requires the developer to reason about the relations between multiple parallel entities (and about the number of such entities). Actors are supported in many programming languages, like Scala [112], Erlang [23], Io [24], and are also available through specific libraries [103]. Of particular interest in the context of server-side development is the Akka framework [1]. Akka is an actor-based framework for the development of distributed applications for the JVM (supporting Scala, Java, and other languages). One of Akka's most interesting features is the possibility to target both clusters and multicores with the same abstraction (the actor), allowing the developer to write distributed applications which can be transparently re-deployed on both multicores and clusters. The approach is able to guarantee high scalability. However, the Akka framework still requires the developer to reason in terms of parallel entities, of their number, and about how (and when) their configuration should be adapted for performance tuning. The framework also offers a limited support for STM-based speculative execution (based either on the DeuceSTM bytecode rewriting STM for Scala [88] or on the CCSTM runtime [46]), which still requires the developer to explicitly use atomic blocks or transactional references. The PEV model can be used in a similar way, with the main difference that its runtime as well as the speculative execution of callbacks is transparent to the user. Finally, other similar approaches are [47, 37].

9.2 Speculative runtime systems

In our research we investigate how speculative runtimes could be used to safely execute event-based services on multicores, and in particular to investigate the relation between asynchronous callbacks and atomic operations. Runtimes to execute atomic blocks in parallel can be divided in two main classes, namely pessimistic (lock-based) and optimistic concurrency control.

Lock-based approaches rely on the correct acquisition of all the needed locks to ensure isolation and atomicity. Several approaches in this class are based on a static analysis aimed at identifying for each atomic block the set of locks to be acquired, and the appropriate acquisition order to prevent deadlocks. The static analysis thus tries to identify for each atomic block the potential side effects, and protects the shared data structures accordingly. In [52], for instance, the authors present a compiler-based automatic lock-inference framework. The system is based on a compile-time analysis which identifies the set of locks to be acquired, and a runtime library to perform the lock acquisition. The analysis is able to infer multi-granularity locks, and has shown good performance on a variety of workloads. Many other compiler-based approaches have been proposed [87, 125], with differences in the type of locks acquired (e.g., granularity, read/write locks, etc.), and the way the static analysis identifies shared data access patterns.

Optimistic concurrency control techniques rely on either software or hardware transactional memory [91]. Transactional memories adopt a concurrency model in which operations *appear* to be consistent and to happen atomically and isolated (ACI). To this end, operations are executed in parallel, and in case of any conflict, operations either succeed (and *commit* their result), or fail. In the latter case, transactions are usually aborted and re-executed. Ensuring the ACI properties through a programming model-level abstraction greatly simplifies the way concurrency can be managed by developers [91].

Many software transactional memories (SMT) models and implementations have been proposed [48, 66, 114, 69, 138]. The design of STMs can vary in several aspects, and in general STMs are categorized in the way they manage concurrency, data versioning, and conflict detection [91]. To manage concurrency some STMs adopt a *pessimistic* approach which protects each data modification using locks. This ensures that the TM system can detect and resolve a conflict at the moment it occurs. Other TMs use an *optimistic* approach and log every data modification. Conflicts are detected and resolved after they occur [91]. To manage versioning TMs need to log shared data access. Two general approaches can be identified: *eager* and *lazy* version management. With eager management [130] updates are immediately applied to the shared data, and the TM system needs to hold an *undo* log to rollback operations in case of failure. With lazy versioning, TMs do not modify shared data until they are able to commit. To this end, the system keeps a *re-do* log, which can be used to re-execute the transaction upon failure. Finally, TMs can adopt an *eager* or *lazy* approach also to identify conflicts, detecting the conflict as soon as possible or at commit-time.

Despite the simplicity and the convenience of the atomic block abstraction, maintaining atomicity and isolation requires a computational overhead which often limits TMs' performance (some argue such overhead is too relevant for adopting STMs as a practical solution [50], others do not [68]). To solve these limitations, several approaches have been proposed to optimize conflict detection [66, 67, 141], transactions scheduling [92], and in general contention management [70, 121]. Some other approaches have tried to combine locks and transactions to improve performance [122] through a compiler-based analysis to infer where locks could perform better than transactions [74]. Likewise, run-time techniques have been proposed to dynamically switch between locks and transactions [147], as well as to change the STM algorithm currently in use, in order to obtain more stable performance [152]. A notable software transactional system is the SpecTM [71] transactional memory. SpecTM proposes to trade off the expressibility (and the generality) of traditional transactional memory APIs against performance. In particular, SpecTM identifies a set of data access patterns which could be addressed using a transactional approach with performance comparable to Compare-And-Swap [97] operations. As discussed in Chapter 5, the runtime systems described in this Dissertation are based on the TL2 [66] and the FastLane [151] algorithms.

9.2.1 Commit-time reordering

Reordering speculative tasks (in the form of transactions) at commit time with the goal of enforcing deterministic execution has already been described in [45, 44], where speculative execution is used in the context of an event-processing system[3] to process events out-of-order. Another example of commit time reordering has been described in the context of the IPOT model [150], a programming model providing the developer with a way to deterministically execute transactions with a given ordering. Transactions in IPOT are used as a mechanism to extract parallelism from a sequential application. Differently from the PEV model, the IPOT programming model requires developers to manually *annotate* blocks of sequential code to be executed potentially in parallel, by the STM runtime. Moreover, the PEV model supports events reordering at a logical (programming model) level, which does not necessarily imply that ordered events will really be executed sequentially and/or in parallel. Indeed, some runtime systems presented in this Dissertation (e.g., the Optimistic PEV in Truffle) do not give any guarantee on the actual degree of parallelism of any event execution.

[3]Not to be confused with a single-threaded event loop.

9.2.2 Single-threaded overhead reduction

The problem of the runtime barriers cost for STM systems has been studied from many perspectives, and several solutions have been proposed. One of the PEV runtimes implemented in Truffle.PEV is FastLane [151] which is perhaps the most recent STM algorithm aimed at reducing the overhead for *at least* one (main) thread. Other approaches exist. In [30], for instance, authors propose a model in which one transaction at time is allowed to execute writes directly to shared objects (using an *undo* log) while other transactions use a re-do log, and buffer writes. The goal of this hybrid model is to reduce the overhead of some logging-intensive transactions over other ones. Another approach is represented by the notion of Irrevocable transactions [154] (sometimes also called Inevitable transactions [140]). In both cases, the general idea is that a transaction is marked with highest priority than others, and thus can survive commit-time conflicts with other lower-priority transactions (and handle I/O). In principle, a PEV system could be seen as a system in which the *current* globally ordered event (i.e., the next event which will commit with global target) is an irrevocable event handler.

9.3 Parallel programming for the Web

Notable efforts are being directed towards overcoming current limitations of JavaScript concerning its support for parallelism. As part of the HTML5 standardization, WebWorkers [8] offer a simple message-passing abstraction for implementing the Actor model in JavaScript. This technology has been used in [75] 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 [111] is a process-based parallelism library for Node.JS implementing a programming model similar to Actor-based concurrency. On the client-side, RiverTrail [93] offers an API for developing data-parallel computations by means of automatic compilation of JavaScript functions (which might potentially even be offloaded to an OpenCL [16] runtime, so that parts of the computation is executed on the GPU). Only applications using *temporary* immutable data structures are supported. The model is inspired by a similar solution for the Java platform called Parallel Closures [123]. The approach attempts to bring temporal immutable parallelism to Java through constructs similar to `async` and `finish`. Parallel Closures operate on immutable (read-only) shared data, and support only fork/join parallel patterns. All of these approaches show the importance and the need for simple parallelism support in JS applications, and motivate solutions like the one of the PEV for the server-side nonblocking counterpart.

Out of the realm of JavaScript and client-side development, many languages feature libraries and tools suitable for parallel programming. As an example, task-based parallel programming can be found in languages such as X10 [51], F# [144] and other structured models [101, 54, 139, 148]. Each language provides different ways of controlling and interacting with parallel tasks, but none of them is targeting an inherently single-threaded language such as JavaScript.

9.3.1 Scheduler-based approaches and the S scripting language

Any speculative runtime presented in this Dissertation relies on the assumption that it is not possible to statically assert that an event handler will (or will not) conflict with other events executed concurrently, as it is impossible to make any assumption on the kind of requests a service will receive. This consideration motivates the adoption of speculative runtimes such as the ones using STM.

A radically different approach is represented by scheduling-based systems. In such systems, it is possible to statically identify all the possible conflicts between parallel tasks (usually at compile time) delegating to the runtime only the scheduling of such tasks. Relevant examples of such systems are represented by [98]. Belonging to this same class of runtime systems is S [42], a scripting language for the development of server-side applications in which the developer can only express operations that are pure, stateful, or stateless, and the runtime automatically takes care of the parallelization of operations that can be safely executed in parallel.

Chapter 10

Concluding remarks

In this Dissertation we introduced an API and a runtime system for extending single-threaded event-based programming frameworks with relaxed forms of event ordering. By relaxing the way events are processed by the underlying event loop, the programming model offers potential opportunities for parallel execution. To this end, we have described the implementation of three runtime systems based on some forms of relaxed events processing. Each of the implementations described in this Dissertation is based on some notion of runtime speculation finalized at the execution of events in parallel. Approaches relying on STM-based speculation have been implemented in the context of two existing language runtimes for JavaScript, while an approach based on a more conservative lock-based approach has been described in the context of the Scala language.

The main goal of our research has not been to propose an advanced programming model for experts in parallel programming, but rather to improve an existing programming abstraction (perhaps the most popular one on the Web), providing for a popular single-threaded programming model a minimal set of limited changes enabling parallel speculative execution. The set of changes introduced by the PEV model are *conservative*, meaning that no application written for a single-threaded runtime would be affected by them. The main application domain for the PEV model is server-side development, with a particular focus on the JavaScript language and Node.JS. The peculiar characteristics of the workloads commonly found in such context (e.g., read-only, stateless, I/O-bound) permit to employ the PEV model for the safe parallelization of applications written using a single-threaded event loop programming model. Moreover, the support for safe parallel execution of callbacks opens the door to a new, unexplored, class of parallel applications mixing event-based computing, parallel processing, and safe mutable shared state.

Each of the implementations described in this Dissertation is based on some notion of runtime speculation finalized at the execution of events in parallel. Each of the runtime systems presented in this Dissertation behaves differently with different workloads, and does not correspond to a *generic* solution addressing all the issues proper of the parallel programming research domain. There is no *best solution*, and each runtime system trades runtime overhead against scalability.

Although the research community agrees on the fact that (software) transactional-memory-based approaches have limitations, the research presented in this dissertation suggests that although STM-based systems cannot be considered a generic solution for high scalability (say, thousands of cores), they can still correspond to valid runtime solutions to bring safe parallelism into high-level managed languages based on single-threaded concurrency.

10.1 Future research directions

Providing the developer with the single-threaded strongly ordered event loop *impression* comes at the cost of scalability. We have shown that by relaxing the order in which events can be processed more scalable systems can be built in the context of server-side computing. Still, enforcing atomicity and isolation on a per-event basis has the cost of logging and metadata management. In this dissertation we argued that for certain workload types the overhead might still be acceptable, in particular in the domain of single-threaded languages such as JavaScript, where the only alternative is manual share-nothing parallelization.

Looking forward, language runtimes are moving towards a diffuse cloud-based deployment of services. Dominant trends seem to anticipate a near future in which cloud-based deployment will be the rule, and cloud developers will have to deal with parallelism more than ever. The peculiarities of the Cloud-based deployment, however, will unavoidably make *existing* approaches to high-performance computing such as, e.g., MPI [39] and pthreads unpractical, as the cloud hosting infrastructure might virtualize any OS-level service (and any language runtime), and it thus might even be impossible for a developer to rely on the fact that the number of processes started by his application will map to real OS processes.

A solution might come from models offering the properties of structured parallelism and PGAS memory [51], as such systems give more freedom to the language runtime to optimize the parallelization strategy to adopt. However, such models might need to be re-designed, and adapted to the needs of the Cloud scenario. In such context, the PEV might be extended to support safe and isolated blocks only for specific operations (e.g., only while accessing the PGAS space).

Finally -and more specifically to some of the PEV runtimes presented in this Dissertation- Hybrid HTM/STM-based solutions (such as [124, 62, 76]) might correspond to an open opportunity to improve the performance of the PEV runtimes for certain workload types.

Bibliography

[1] Akka: A Java and Scala framework with Actors, STM and Transactors. http://www.akka.io/.

[2] ASM.JS. http://asmjs.org/spec/latest/.

[3] The Boost library. http://www.boost.org/.

[4] ECMAscript 6 Harmony proxies. http://wiki.ecmascript.org/doku.php?id=harmony:proxies.

[5] ECMAscript 6 specification Draft. http://wiki.ecmascript.org/doku.php?id=harmony:specification_drafts.

[6] Google V8 High-performance JavaScript Engine. https://code.google.com/p/v8/.

[7] Heroku Cloud. http://www.heroku.com/.

[8] HTML5 WebWokers API. http://dev.w3.org/html5/workers/.

[9] Java EE Servlet API. http://docs.oracle.com/javaee/5/api/javax/servlet/Servlet.html.

[10] JDK8 Nashorn JavaScript Engine. http://openjdk.java.net/projects/nashorn/.

[11] The Jetty High-Performance Web Server Project. http://jetty.codehaus.org/.

[12] Memcached: a distributed memory caching system. http://memcached.org.

[13] Microsoft Reactive Extentions. https://github.com/Reactive-Extensions/RxJS.

[14] Microsoft Windows Azure Cloud Platform. http://www.windowsazure.com/.

[15] Node.JS: Evented programming for networked services in JavaScript. http://www.nodejs.org.

[16] OpenCL, the Standard for parallel programming of GPUs. `http://developer.amd.com`.

[17] ReactJS: A JavaScript library for building user interfaces. `http://facebook.github.io/react/`.

[18] Ruby EventMachine. `http://rubyeventmachine.com`.

[19] RxJava: Reactive Java. `https://github.com/ReactiveX/RxJava`.

[20] SpiderMonkey JavaScript Engine. `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey`.

[21] Strawman JS parallelism. `http://wiki.ecmascript.org/doku.php?id=strawman:data_parallelism`.

[22] The Dart language. `http://www.dartlang.org/`.

[23] The Erlang language. `http://www.erlang.org/`.

[24] The Io language. `http://iolanguage.com/`.

[25] The WebSocket API. `http://www.w3.org/TR/websockets/`.

[26] Vert.X. `https://github.com/purplefox/vert.x`.

[27] Web of Things. `http://www.w3.org/community/wot/`.

[28] WebRTC 1.0: Real-time communication between browsers. `http://www.w3.org/TR/webrtc/`.

[29] World Wide Web Consortium - HTML5 Candidate Reccomendation 04. `http://www.w3.org/TR/html5/`.

[30] A.-R. Adl-Tabatabai and A. Welc. Hybrid transactions for low-overhead speculative parallelization, June 5 2012. US Patent 8,195,898.

[31] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

[32] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. A survey of adaptive optimization in virtual machines. In *Processings of the IEEE, 93 (2), 2005. Special Issue on Program Generation, Optimization, and Adaptation*, 2004.

[33] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, May 2014.

[34] D. I. August, J. Huang, S. R. Beard, N. P. Johnson, and T. B. Jablin. Automatically exploiting cross-invocation parallelism using runtime information. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–11, 2013.

[35] J. Bahi, R. Couturier, D. Laiymani, and K. Mazouzi. Java and Asynchronous Iterative Applications: Large Scale Experiments. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–7, 2007.

[36] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP)*, pages 29–44, 2009.

[37] J. R. v. Behren, E. A. Brewer, N. Borisov, M. Chen, M. Welsh, J. MacDonald, J. Lau, and D. E. Culler. Ninja: A framework for network services. In *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference (ATC)*, pages 87–102, 2002.

[38] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. *SIGOPS Oper. Syst. Rev.*, 21(5):123–138, Nov. 1987.

[39] B. Blaise. MPI: A message passing interface, 1993.

[40] D. Bonetta, D. Ansaloni, A. Peternier, C. Pautasso, and W. Binder. Node.scala: Implicit parallel programming for high-performance web services. In *Proceedings of the 18th International Conference on Parallel Processing*, Euro-Par'12, pages 626–637. Springer-Verlag, 2012.

[41] D. Bonetta, W. Binder, and C. Pautasso. TigerQuoll: Parallel event-based javascript. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 251–260, 2013.

[42] D. Bonetta, A. Peternier, C. Pautasso, and W. Binder. S: a scripting language for high-performance restful web services. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, New Orleans, LA, USA, February 2012.

[43] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the Seventh International Conference on World Wide Web (WWW)*, pages 107–117, 1998.

[44] A. Brito. Optimistic parallelization support for event stream processing systems. In *Proceedings of the 5th Middleware Doctoral Symposium*, MDS '08, pages 7–12, 2008.

[45] A. Brito, C. Fetzer, H. Sturzrehm, and P. Felber. Speculative out-of-order event processing with software transaction memory. In *Proceedings of the Second International Conference on Distributed Event-based Systems*, DEBS '08, pages 265–275, New York, NY, USA, 2008. ACM.

[46] N. G. Bronson, H. Chafi, and K. Olukotun. CCSTM: A library-based STM for scala. *def*, 9:10, 2010.

[47] B. Burns, K. Grimaldi, A. Kostadinov, E. D. Berger, and M. D. Corner. Flux: a language for programming high-performance servers. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference (ATC)*, pages 13–13, 2006.

[48] J. a. Cachopo and A. Rito-Silva. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program.*, 63(2):172–185, Dec. 2006.

[49] V. Cardellini, E. Casalicchio, M. Colajanni, and P. S. Yu. The State of the Art in Locally Distributed Web-Server Systems. *ACM Comput. Surv.*, 34:263–311, 2002.

[50] 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.

[51] 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 *Proceedings of OOPSLA*, pages 519–538, 2005.

[52] S. Cherem, T. Chilimbi, and S. Gulwani. Inferring locks for atomic sections. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 304–315, 2008.

[53] G. S. Choi, J.-H. Kim, D. Ersoz, and C. R. Das. A multi-threaded pipelined web server architecture for SMP/SoC machines. In *Proceedings of the 14th International Conference on World Wide Web (WWW)*, pages 730–739, 2005.

[54] W.-C. Chuang, B. Sang, S. Yoo, R. Gu, M. Kulkarni, and C. Killian. Eventwave: Programming model and runtime support for tightly-coupled elastic cloud applications. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC)*, pages 21:1–21:16, 2013.

[55] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, Cambridge, MA, USA, 1991.

[56] M. I. Cole. A Skeletal Approach to the Exploitation of Parallelism. In *Proceedings of the Conference on CONPAR 88*, pages 667–675, 1989.

[57] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 493–501, 1993.

[58] R. Cunningham and E. Kohler. Making events less slippery with eel. In *Proceedings of the 10th conference on Hot Topics in Operating Systems - Volume 10 (HotOS)*, pages 3–3, 2005.

[59] T. V. Cutsem and M. S. Miller. Trustworthy proxies: Virtualizing objects with invariants. In *Proceedings of ECOOP 2013*, 2013.

[60] E. Czaplicki and S. Chong. Asynchronous functional reactive programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 411–422, 2013.

[61] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazires, and R. Morris. Event-Driven Programming for Robust Software. In *Proceedings of the 10th ACM SIGOPS European Workshop (EW)*, pages 186–189, 2002.

[62] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear. Hybrid NOrec: A case study in the effectiveness of best effort hardware transactional memory. *SIGPLAN Not.*, 46(3):39–52, Mar. 2011.

[63] L. Dalessandro, D. Dice, M. Scott, N. Shavit, and M. Spear. Transactional mutex locks. In *Proceedings of the 16th International Euro-Par Conference on Parallel Processing: Part II*, Euro-Par'10, pages 2–13, 2010.

[64] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of OSDI'04*, pages 137–150.

[65] S. Deconinck. *Linux system programming*. Stéphane Deconinck, 2010.

[66] 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. Springer, 2006.

[67] D. Dice and N. Shavit. Understanding tradeoffs in software transactional memory. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 21–33. IEEE Computer Society, 2007.

[68] 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.

[69] 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. ACM, 2009.

[70] A. Dragojević, R. Guerraoui, A. V. Singh, and V. Singh. Preventing versus curing: avoiding conflicts in transactional memories. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, PODC '09, pages 7–16. ACM, 2009.

[71] A. Dragojević and T. Harris. STM in the small: trading generality for performance in software transactional memory. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 1–14. ACM, 2012.

[72] G. Duboscq, T. Würthinger, L. Stadler, C. Wimmer, D. Simon, and H. Mössenböck. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*, VMIL '13, pages 1–10, 2013.

[73] C. Elliott and P. Hudak. Functional reactive animation. In *Proceedings of International Conference on Functional Programming*, pages 263–273, 1997.

[74] M. Emmi, J. S. Fischer, R. Jhala, and R. Majumdar. Lock allocation. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '07, pages 291–296. ACM, 2007.

[75] 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)*, pages 161–170, 2012.

[76] P. Felber, C. Fetzer, P. Marlier, M. Nowack, and T. Riegel. Brief announcement: Hybrid time-based transactional memory. In *Distributed Computing*, pages 124–126. Springer, 2010.

[77] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*. 1999.

[78] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, 2000.

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

[80] V. W. Freeh. A comparison of implicit and explicit parallel programming. *Journal of Parallel and Distributed Computing*, 34(1):50–65, 1996.

[81] Y. Futamura. Partial evaluation of computation process: An approach to a compiler-compiler. *Higher Order Symbol. Comput.*, 12(4):381–391, Dec. 1999.

[82] V. Gajinov, S. Stipic, O. S. Unsal, T. Harris, E. Ayguadé, and A. Cristal. Integrating dataflow abstractions into the shared memory model. In *SBAC-PAD*, pages 243–251, 2012.

[83] V. Gajinov, S. Stipic, O. S. Unsal, T. Harris, E. Ayguadé, and A. Cristal. Supporting stateful tasks in a dataflow graph. In *PACT*, pages 435–436, 2012.

[84] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 465–478, 2009.

[85] F. Gaud, S. Genevès, R. Lachaize, B. Lepers, F. Mottet, G. Muller, and V. Quéma. Efficient workstealing for multicore event-driven systems. In *Proceedings of the 2010 IEEE 30th International Conference on Distributed Computing Systems*, ICDCS '10, pages 516–525. IEEE Computer Society, 2010.

[86] B. Göetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java concurrency in practice*. Addison-Wesley, 2006.

[87] G. Golan-Gueta, N. Bronson, A. Aiken, G. Ramalingam, M. Sagiv, and E. Yahav. Automatic fine-grain locking using shape properties. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 225–242. ACM, 2011.

[88] D. Goodman, B. Khan, S. Khan, C. Kirkham, M. Luján, and I. Watson. Muts: Native scala constructs for software transactional memory. In *Scala Days Workshop, Stanford, Palo Alto, CA, USA*, 2011.

[89] P. Haller and M. Odersky. Actors that Unify Threads and Events. In *Proceedings of the International Conference on Coordination Models and Languages (COORDINATION)*, pages 171–190, 2007.

[90] T. Harris, M. Abadi, R. Isaacs, and R. McIlroy. AC: Composable Asynchronous IO for Native Languages. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 903–920. ACM, 2011.

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

[92] T. Heber, D. Hendler, and A. Suissa. On the impact of serializing contention management on STM performance. *J. Parallel Distrib. Comput.*, 72(6):739–750, June 2012.

[93] 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.

[94] S. Herhut, R. L. Hudson, T. Shpeisman, and J. Sreeram. River Trail: A Path to Parallelism in JavaScript. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, pages 729–744, 2013.

[95] M. Herlihy. The art of multiprocessor programming. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Principles of Distributed Computing*, PODC '06, pages 1–2, 2006.

[96] M. Herlihy and V. Luchangco. Distributed Computing and the Multicore Revolution. *SIGACT News*, 39(1):62–72, Mar. 2008.

[97] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., 2008.

[98] S. T. Heumann, V. S. Adve, and S. Wang. The tasks with effects model for safe concurrency. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 239–250, 2013.

[99] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP'91 European Conference on Object-Oriented Programming*, pages 21–38. Springer, 1991.

[100] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-marl: A DSL for easy and efficient graph analysis. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 349–362, 2012.

[101] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of EuroSys'07*, pages 59–72.

[102] T. Kalibera, P. Maj, F. Morandat, and J. Vitek. A fast abstract syntax tree interpreter for r. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '14, pages 89–102, 2014.

[103] R. K. Karmani, A. Shali, and G. Agha. Actor Frameworks for the JVM Platform: a Comparative Analysis. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 11–20. ACM, 2009.

[104] R. A. Kelsey. A correspondence between continuation passing style and static single assignment form. In *Papers from the 1995 ACM SIGPLAN workshop on Intermediate representations*, IR '95, pages 13–22, 1995.

[105] A. Kennedy. Compiling with continuations, continued. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 177–190, 2007.

[106] K. Kinder. Event-Driven Programming with Twisted and Python. *Linux J.*, 2005.

[107] M. Krohn, E. Kohler, and M. Kaashoek. Simplified Event Programming for Busy Network Applications. In *Proceedings of the 2007 USENIX Annual Technical Conference, Santa Clara, CA, USA*, pages 351–364, 2007.

[108] M. Krohn, E. Kohler, and M. F. Kaashoek. Events can Make Sense. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ATC'07, pages 7:1–7:14. USENIX Association, 2007.

[109] J. R. Larus. Look up!: Your future is in the cloud. *SIGPLAN Not.*, 48(6):1–2, June 2013.

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

[111] R. M. Lerner. At the Forge: Node.JS. *Linux J.*, 2011, 2011.

[112] M. Lesani, M. Odersky, and R. Guerraoui. Concurrent Programming Paradigms, A Comparison in Scala. Technical report, 2009.

[113] B. P. Lester. *The art of parallel programming*. Prentice-Hall, Inc., 1993.

[114] Y. Lev, V. Luchangco, V. J. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Anatomy of a Scalable Software Transactional Memory. In *2009, 4th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT09)*, 2009.

[115] P. Li and E. Wohlstadter. Object-Relational Event Middleware for Web Applications. In *Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research (CASCON)*, pages 215–228, 2011.

[116] P. Li and S. Zdancewic. A Language-based Approach to Unifying Events and Threads. *CIS Department University of Pennsylvania April*, 2006.

[117] P. Li and S. Zdancewic. Combining Events and Threads for Scalable Network Services Implementation and Evaluation of Monadic, Application-level Concurrency Primitives. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 189–199. ACM, 2007.

[118] Z. Li, D. Levy, S. Chen, and J. Zic. Auto-Tune Design and Evaluation on Staged Event-Driven Architecture. In *Proceedings of the 1st Workshop on MOdel Driven Development for Middleware (MODDM)*, pages 1–6, 2006.

[119] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 260–267, 1988.

[120] J. Louvel, V. Templier, and T. Boileau. *RESTlet in Action*. Manning Publications, 2009.

[121] W. Maldonado, P. Marlier, P. Felber, A. Suissa, D. Hendler, A. Fedorova, J. L. Lawall, and G. Muller. Scheduling Support for Transactional Memory Contention Management. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 79–90. ACM, 2010.

[122] S. Mannarswamy, D. R. Chakrabarti, K. Rajan, and S. Saraswati. Compiler Aided Selective Lock Assignment for Improving the Performance of Software Transactional Memory. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 37–46. ACM, 2010.

[123] 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.

[124] A. Matveev and N. Shavit. Reduced hardware transactions: a new approach to hybrid transactional memory. In *Proceedings of the 25th ACM symposium on Parallelism in algorithms and architectures*, pages 11–22. ACM, 2013.

[125] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: Synchronization Inference for Atomic Sections. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 346–358, 2006.

[126] 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.

[127] E. Meijer. Your mouse is a database. *Commun. ACM*, 55(5):66–73, May 2012.

[128] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: a programming language for Ajax applications. In *Proceedings of OOPSLA*, pages 1–20, 2009.

[129] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 35–46. IEEE, 2008.

[130] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based Transactional Memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pages 254–265. Feb 2006.

[131] H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, pages 51–64, 2002.

[132] J. Ousterhout. Why Threads are a Bad Idea (for Most Purposes). In *USENIX Winter Technical Conference*, 1996.

[133] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: an Efficient and Portable Web Server. In *Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC)*, pages 15–15, 1999.

[134] K. Pamnany and J. Jannotti. Elyze: Enabling Safe Parallelism in Event-driven Servers. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '08, pages 15–21. ACM, 2008.

[135] D. Pariag, T. Brecht, A. Harji, P. Buhr, A. Shukla, and D. R. Cheriton. Comparing the Performance of Web Server Architectures. In *Proceedings of the 2nd ACM SIGOPS European Conference on Computer Systems (EuroSys)*, pages 231–243, 2007.

[136] O. Phelan, K. McCarthy, M. Bennett, and B. Smyth. On using the real-time web for news recommendation discovery. In *Proceedings of the 20th international conference companion on World wide web (WWW)*, pages 103–104, 2011.

[137] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August. Speculative parallelization using software multi-threaded transactions. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 65–76, 2010.

[138] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *Proceedings of the 20th international conference on Distributed Computing*, DISC'06, pages 284–298. Springer, 2006.

[139] C. Rodrigues, T. Jablin, A. Dakkak, and W.-M. Hwu. Triolet: A programming system that unifies algorithmic skeleton interfaces for high-performance cluster computing. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 247–258, 2014.

[140] M. Spear, M. Michael, and M. Scott. Inevitability mechanisms for software transactional memory. In *3rd ACM SIGPLAN Workshop on Transactional Computing, New York, NY, USA*, 2008.

[141] M. F. Spear, V. J. Marathe, W. N. Scherer, and M. L. Scott. Conflict Detection and Validation Strategies for Software Transactional Memory. In *Proceedings of the 20th international conference on Distributed Computing (DISC)*, pages 179–193, 2006.

[142] I. Stoica. A berkeley view of big data: algorithms, machines and people. In *UC Berkeley EECS Annual Research Symposium*, 2011.

[143] P. Stutz, A. Bernstein, and W. Cohen. Signal/collect: graph algorithms for the (semantic) web. In *The Semantic Web–ISWC 2010*, pages 764–780. Springer, 2010.

[144] 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. Springer, 2011.

[145] J. Throop. OpenMP: Shared-memory parallelism from the ashes. *Computer*, 32(5):108–109, May 1999.

[146] S. Tilkov and S. Vinoski. Node.js: Using JavaScript to build high-performance network programs. *IEEE Internet Computing*, 14:80–83, November 2010.

[147] T. Usui, R. Behrends, J. Evans, and Y. Smaragdakis. Adaptive locks: Combining transactions and locks for efficient concurrency. *J. Parallel Distrib. Comput.*, 70(10):1009–1023, Oct. 2010.

[148] T. Van Cutsem and W. De Meuter. Event-driven mobile computing with objects., 2010.

[149] R. Von Behren, J. Condit, and E. Brewer. Why Events Are a Bad Idea (for High-Concurrency Servers). In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, page 4, 2003.

[150] C. von Praun, L. Ceze, and C. Caşcaval. Implicit parallelism with ordered transactions. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 79–89, 2007.

[151] J.-T. Wamhoff, C. Fetzer, P. Felber, E. Rivière, and G. Muller. FastLane: Improving performance of software transactional memory for low thread counts. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 113–122, 2013.

[152] Q. Wang, S. Kulkarni, J. Cavazos, and M. Spear. A Transactional Memory with Automatic Performance Tuning. *ACM Trans. Archit. Code Optim.*, 8(4):54:1–54:23, Jan. 2012.

[153] A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 439–453, 2005.

[154] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable transactions and their applications. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 285–296, 2008.

[155] M. Welsh, D. Culler, and E. Brewer. SEDA: an Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 230–243, 2001.

[156] C. Wimmer and S. Brunthaler. ZipPy on Truffle: A fast and simple implementation of Python. In *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '13, pages 17–18, 2013.

[157] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! '13, pages 187–204, New York, NY, USA, 2013. ACM.

[158] N. Zeldovich, E. Yip, F. Dabek, R. T. Morris, D. Mazieres, and F. Kaashoek. Multiprocessor Support for Event-Driven Programs. In *Proceedings of the USENIX Annual Technical Conference (USENIX)*, pages 239–252, 2003.