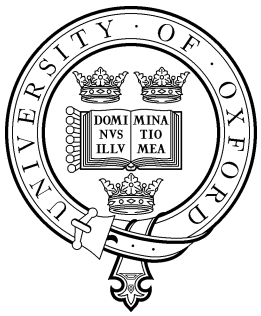
Event-Driven Concurrency in JavaScript



Fedor Sheremetyev
Keble College
University of Oxford

Submitted in partial fulfilment of the requirements for the degree of
Master of Science in Computer Science

September 2011

To Elena, my wife and muse.

Acknowledgements

I am very grateful to my project supervisor Bernard Sufrin for interesting discussions about concurrency and for his insightful comments on the drafts of this dissertation.

I am thankful to Nurzhan Bakibaev, Evgeny Knyazev, and Vladislav Bilyk for reading early drafts and for their helpful comments.

Also I would like to thank Philip Armstrong for granting access to an 8-core machine for testing.

Abstract

Concurrency is one of the most difficult aspects of programming. Yet mastering it becomes increasingly important to take advantage of multiple cores offered by modern processors.

Event-driven approach proved to be successful in the design of scalable I/O-bound systems. However, sequential processing of events does not scale well in compute-bound tasks.

We propose automatic on-line *parallelisation* of event-driven programs with resolution of data conflicts via *transactional memory* as a novel approach to the design of concurrent software.

A prototype of parallelising execution environment for JavaScript demonstrates good scalability of event-driven code in a compute-intensive benchmark.

Contents

List of Figures

Chapter 1

Introduction

Concurrency is considered one of the most difficult aspects of programming but the expansion of multicore processors makes writing concurrent programs increasingly important [54]. Concurrent Programming course suggested that the problem of writing efficient and maintainable concurrent programs is far from being solved and motivated me to do research in this area.

Approaches currently employed in concurrent programming are reviewed in Chapter 2. In Section 2.4 we also describe event-driven programming as a way to structure program execution. Currently, it is used to build scalable I/O-bound systems but is not considered a form of concurrent programming besides separation of I/O from computation.

In Chapter 4 we propose automatic parallelisation of event-driven programs with resolution of conflicts via transactional memory as a form of concurrent programming. Parallelisation maintains simple semantics of event-driven programs yet can take advantage of modern multiprocessors. To our knowledge, this is the first attempt to combine benefits of event-driven programming and transactional memory.

We apply parallelisation to JavaScript programming language as an experiment. JavaScript is convenient for writing event-driven programs thanks to unique combination of its features (reviewed in Chapter 3).

In Chapter 5 we describe implementation of parallelising execution environment. Prototype, built using Google's JavaScript engine V8, demonstrates almost linear scalability of event-driven code in a compute-intensive benchmark.

In Chapter 6 we briefly discuss JavaScript as a concurrent programming language.

Chapter 2

Concurrency and Parallelism

Concurrency is the feature of programming language or, more precisely, feature of its computational model for expressing the possibility of overlapping execution.

Concurrency is often used for the sake of readability and maintainability of the code. Independent activities or tasks whose steps can be interleaved are better described as several concurrent tasks rather than a single sequential one. Concurrency is useful even if the machine is not able to perform several operations simultaneously. On a single-processor system concurrency can be realized via *time sharing*.

Another application of concurrency is related to performance. Programmer can mark as concurrent the steps of a computation which do not interfere with each other. These steps can be then executed simultaneously on multiple processors to complete the computation faster.

Parallelism is the feature of hardware platform which enables simultaneous execution. It is often implemented in the form of multiple processors working on a shared memory — *multiprocessor*.

The topic of this thesis concerns with concurrent constructs in programming languages which help exploit the task parallelism offered by modern multiprocessors.

2.1 Concurrent Execution

Consider internal structure of concurrent tasks. Each task consists of a series of operations which should be performed. We do not use the term "thread" here because tasks can be structured not only sequentially in threads. Active objects and events are examples of different task structures.

Input/output (I/O) operations and access to shared data divide tasks into *steps*. Each step can be performed *uninterruptedly* thus utilising processor in full while

coordination between tasks and I/O introduce delays and may require switching the processor to another task.

There are two main approaches to structuring execution of concurrent tasks. Either steps are organised sequentially, by tasks, and coordinate via modification of shared data. Or steps are organised around shared data and coordinate via messages. Neither of the approaches is inherently better. They are duals of each other [40].

Different task structures emphasize different styles of programming (see [49, page 576]). Depending on the nature of the problem and the requirements, either shared-state or message-passing implementation may be a better fit.

When shared data can be split into independent pieces then message-passing is usually a better choice. If the data is highly coupled or achieving consensus is important then shared-state model fits better.

2.1.1 Ornamental Gardens Problem

To highlight the differences in approaches to concurrency consider the following problem. It was first published by Alan Burns and Geoffrey Davies in 1993 (see [15, page 61]).

> A large ornamental garden, probably formerly the grounds of a British stately residence, is open to members of the public, who must pay an admission fee to view the beautiful collection of roses, shrubs and aquatic plants. Entry is gained by two turnstiles. The management of the gardens want to be able to determine, at any time, the total number of members of the public currently in the grounds. They propose that a computer system should be installed which has connections to each turnstile and a terminal from which the management can ascertain the current total.

A solution to the Ornamental Gardens problem will probably include the following steps.

- Admit people through turnstile. We will consider this an I/O operation, performed by an external device.
- Increment counter value. This can be decomposed into loading current value, calculating and storing new value.
- Read counter value to display it on the terminal.

- Update terminal. This will be considered computationally-intensive operation (e.g. picture should be rasterised to many pixels on a large display).

Figure 2.1 shows two variants of composition of these steps into threads[1]. Circles designate the counter. It can be either shared by threads or owned by a single thread.

When steps belonging to a single task are executed in the same thread (e.g. "Admit People" and "Increment Counter") then the counter can be accessed by several threads simultaneously. Threads need to coordinate somehow to guarantee consistent modification of the counter.

When all steps accessing the counter are executed in the same thread (e.g. "Read Counter" and "Increment Counter") then the counter does not need to be protected. But other threads have to communicate with the thread owning the counter to achieve their goals.

Which solution is better depends on the details of the problem. There are always trade-offs. Many variations of shared-state and message-passing concurrencies are possible. We will consider few of them which are used in practice.

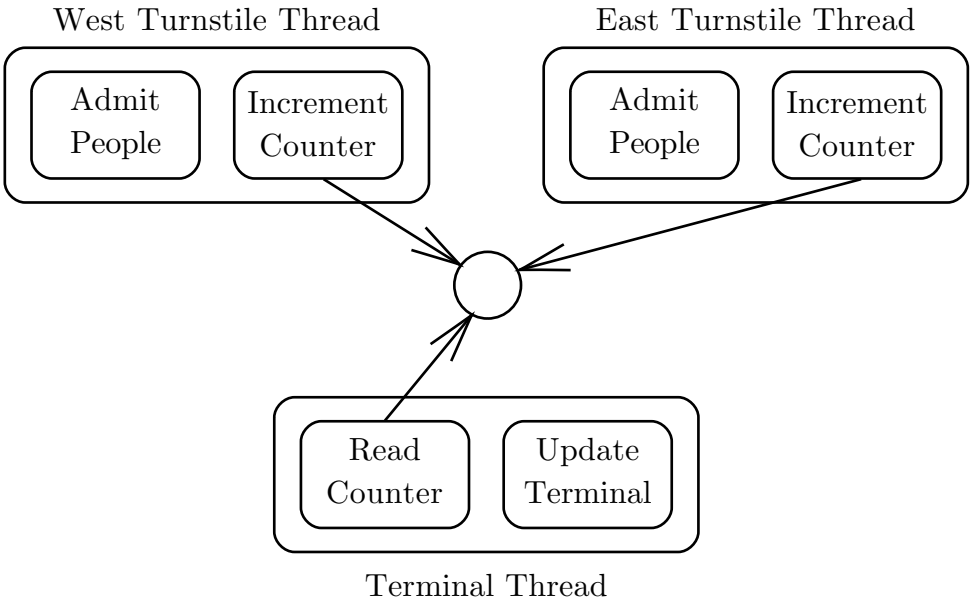
2.2 Shared-State Concurrency

With shared-state concurrency, execution of tasks is structured as a set of threads (usually kernel threads running under pre-emptive OS scheduler). The OS creates for each thread illusion of virtual processor by periodically switching the processor from one thread to another, saving and restoring the *context* of each thread (registers, program pointer etc.). On a multiprocessor several threads may run simultaneously for some period of time.

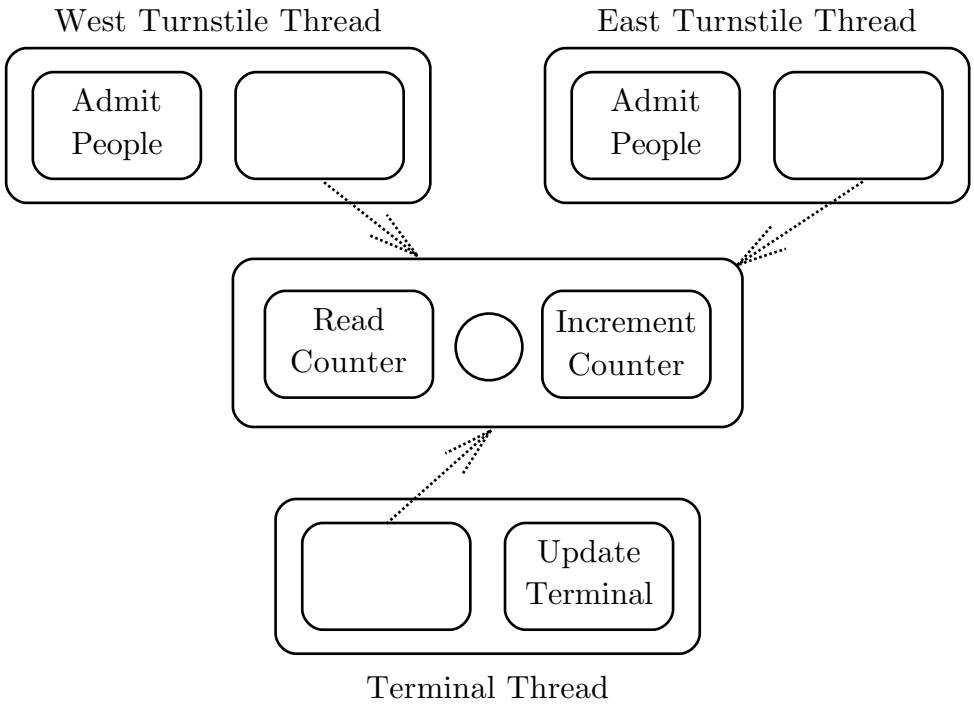
It is possible to manipulate shared memory in concurrent code using the same load and store instructions as in sequential code. But this would introduce race condition — the result of the execution would depend on the interleaving of operations from different threads. Reordering of operations caused by compiler optimisation, processor's out-of-order execution and lack of sequential consistency in caches may cause inconsistent results in programs with data races [10, 14].

To allow consistent execution of concurrent code modern processors provide special atomic instructions: *compare-and-swap* (CAS) or *load-link/store-conditional* (LLSC). Both have the same expressive power [32].

[1]Threads here do not have to be kernel threads. They may well be user-space threads which are multiplexed onto kernel threads by runtime.



(a) Shared State



(b) Message Passing

Figure 2.1: Concurrent Tasks Structure

CAS atomically loads value from memory location, compares it to the provided comparand and writes new value only if the memory value and the comparand are

equal.

LLSC is a pair of instructions. First instruction loads value from memory location. Second instruction stores new value to memory only if there were no updates.

Atomic instructions are slower than load and store but they allow implementation of consistent multi-threaded execution. Atomic instructions are usually available in a programming language either as language constructs or via libraries.

2.2.1 Non-Blocking Algorithms

The lowest level of programming with shared-state concurrency is based directly on atomic instructions. Non-blocking algorithms combine several basic atomic instructions to build higher-level atomic operations.

Figure 2.2 shows a non-blocking implementation of Ornamental Gardens problem in GCC C++ using compiler-supported CAS operation[2].

The operation `__sync_bool_compare_and_swap` atomically stores a new value (second parameter) at a memory location (first parameter) if the value currently held at that location is equal to the provided one (third parameter).

```
1  int counter = 0;
2
3  void* turnstile(char *id) {
4    while (true) {
5      admitSomeone(id);
6
7      while (true) {
8        int old_value = counter;
9        int new_value = old_value + 1;
10       if (__sync_bool_compare_and_swap(&counter,
11             old_value, new_value))
12         break;
13     }
14   }
15 }
16
17 void terminal() {
18   while (true) {
19     int total = counter;
20     updateTerminal(total);
21   }
```

Figure 2.2: Non-Blocking Code in C++

[2]In this and further examples we show only fragments of programs which demonstrate the described concepts. Complete programs have been developed and tested.

Two threads execute `turnstile` function (one for each turnstile) and one thread executes `terminal` function. Both functions operate on a shared global variable `counter`.

The `terminal` thread reads the value of the `counter` without any coordination. We assume that the size of `int` data type is not greater than platform's word size, that data is properly aligned by compiler and that reading an aligned word is atomic. Reading `counter` value does not suspend `terminal` thread.

`turnstile` threads modify the value of the `counter` by repeatedly reading current value and then attempting to store new value. If these operations of two threads are not interleaved then the update of the value succeeds immediately. Otherwise one of the threads updates the value while the other fails in the atomic operation and has to repeat the attempt. Thus `turnstile` threads do not have to wait but may potentially waste processor's time.

With non-blocking algorithms, access to shared data does not divide tasks into steps. Processor remains busy when doing coordination. Several processors can run non-blocking code in parallel without waiting for each other but may need to do extra work to maintain consistency of shared data. The amount of extra work depends on the rate of conflicts.

Non-blocking concurrent algorithms have been developed for various data structures. It is known that most data structures can be implemented wait-free [33]. In practice such constructs are often slower than lock-based implementations.

What is more important, non-blocking algorithms are difficult to reason about. Stack is relatively simple to implement in a non-blocking way because there is only one point of synchronisation — stack top. Queue data structure implementation is already non-trivial. It involves coordinated changes to the both ends of the queue with faster threads helping slower ones [32, page 230].

2.2.2 Locks

Traditional way of writing concurrent code is locking: the execution of a thread blocks until some condition is met. Synchronization primitives have been developed to express various kinds of waiting conditions: mutex, semaphore, barrier, monitor, read-write lock, etc.

Figure 2.3 shows an implementation of Ornamental Gardens problem using Pthread mutex for coordination of access to shared data. Only one `turnstile` thread can be executing the code block between `pthread_mutex_lock` and `pthread_mutex_unlock` calls at the same time. This eliminates conflicting changes on the `counter` variable.

```
1  int counter = 0;
2  pthread_mutex_t mutex;
3
4  void* turnstile(char *id) {
5    while (true) {
6      admitSomeone(id);
7
8      pthread_mutex_lock(&mutex);
9      counter = counter + 1;
10     pthread_mutex_unlock(&mutex);
11   }
12 }
13
14 void terminal() {
15   while (true) {
16     int total = counter;
17     updateTerminal(total);
18   }
19 }
```

Figure 2.3: Lock-Based Code in C++

Locking constructs are usually implemented using atomic memory operations and *spinning* — repeated reads of memory location. In our example, when the mutex is free (denoted by value 0) any thread can *lock* it by atomically setting its value to 1. When another thread attempts to lock the mutex, it repeatedly reads the value of mutex until its value changes to 0 and then tries to set it atomically to 1. Repeated non-atomic reads of a memory location can be served from processor's cache and do not saturate memory bus. When the value is changed by one processor the cache coherence protocol ensures that other processors get the new value.

Waiting for a lock wastes processor time so operating systems usually implement combined approach. They spin for short time then unschedule the thread to let other threads use the processor. Context switching is an expensive operation so the OS tries spinning first.

Tasks are effectively divided into steps by the use of synchronisation primitives. The code between them can be run uninterruptedly. Obtaining locks may result in wasted processor time (while spinning) or in extra thread scheduling overhead (context switch).

The level of parallelism available to the lock-based code highly depends on the pattern of coordination of threads. In the worst case mutual exclusion applied too often can force threads to execute slower than they would run one after another.

It is notoriously difficult to write correct lock-based code. There are many threats to the correctness and reliability: deadlocks, priority inversion, etc. But the major disadvantage is that locking code is not composable. Two perfectly working components can deadlock when used together.

2.2.3 Transactional Memory

A recent approach to shared-state concurrency is *software transactional memory*[3] (STM) [31]. STM allows grouping operations on shared data in transactions thus forming larger atomic operations. Memory transactions are handled in a way similar to database transactions. They are atomic, consistent and isolated (without durability — so called "lightweight transactions").

Changes performed in a transaction are tentative. In case of a conflict between transactions changes can be aborted and transaction restarted. There are different approaches to the management of tentative changes and to the detection of conflicts.

With "eager" approach changes are written directly to the shared memory. Old values are remembered in an *undo-log* and restored if transaction is rolled back. To avoid overlapping changes by several transactions all changed locations are locked until commit. STM implements means to avoid deadlocks (usually via aborting of transactions). Eager approach is more efficient when conflicts are frequent.

With "lazy" approach changes are kept separately in a *redo-log* and written back to shared memory on commit. Without locking it is possible that several transactions have tentative changes to the same memory location. STM must resolve conflicts by delaying or aborting transactions and, at the same time, avoiding livelocks. Lazy approach is more efficient when conflicts are rare.

STMs provide *serialisability* — transactions appear as if they were executed sequentially. This simplifies reasoning about the program because each transaction can be considered in isolation.

Read set and write set consist of all locations correspondingly read or written by a transaction. They can be maintained explicitly or implicitly (e.g. via global clock).

Some STMs provide *opacity* — they guarantee that the read set is always consistent, even if the transaction is aborted. This avoids situations when transaction throws exception or goes into infinite loop due to observation of inconsistent state ("zombie transaction").

[3]We do not consider hardware transactional memory here because a) its semantics is close to STM; b) it is not available in commodity hardware yet.

Management of shared memory can be implemented in STM with different granularity: individual memory locations, memory blocks or objects.

The API for an STM usually includes operations for starting, committing, aborting transactions and for access to memory shared between threads.

The following code fragment (Figure 2.4) shows an example of STM usage in Haskell as applied to the Ornamental Gardens problem.

```
1  turnstile :: String -> TVar Int -> IO ()
2  turnstile id counter = forever $ do
3    admitSomeone id
4    atomically $ do
5      total <- readTVar counter
6      writeTVar counter (total + 1)
7
8  terminal :: TVar Int -> IO ()
9  terminal counter = forever $ do
10   total <- atomically (readTVar counter)
11   updateTerminal total
```

Figure 2.4: Use of STM in Haskell

The `terminal` function and two instances of `turnstile` function are executed in separate threads. Threads repeat execution of tasks' steps which access common transactional variable `counter`. The `turnstile` function encloses two operations on the `counter` (reading current value and writing new value) in a call to `atomically`. This ensures that both operations are executed in a transaction. When two `turnstile` threads simultaneously attempt to modify the `counter`, one them is forced to abort the transaction and repeat it. The `terminal` thread also has to enclose the reading of the `counter` into an atomic block.

The real division of tasks into uninterruptedly executed steps depends on the STM implementation. STM itself can be implemented using locks, wait-free algorithms or combination of both. Synchronisation between transactions may potentially be required on each read or write operation. But there is room for optimisation. Since the synchronisation is not exposed to the programmer the STM can employ sophisticated algorithms to improve processor utilisation. For example, in some STM implementations threads enqueued for commit can help currently committing transaction to complete faster thus increasing throughput [28].

The major advantage of STM is *composability*. One transactional component can be combined with another without breaking consistency.

STM does not guarantee the correctness of the code automatically. It is still necessary to structure the code in terms of transactions correctly. Wrong selection of transaction boundaries may result in incorrect behaviour.

Integration with non-transactional code can be a threat to correctness. The use of blocking I/O operations or synchronisation primitives is also an issue. Blocking one thread can affect transactions in other threads. On the other hand, when most of the code is non-blocking and transactional synchronisation between threads is a problem.

Thread inside a transaction does not see changes to shared data made from other threads. Thus there is no way to send a signal from one thread to another. This problem is usually solved by checking some condition on shared data and forcing transaction to abort if the condition is not met. In this case STM waits for another transaction to make a change to that piece of shared data before restarting the aborted transaction.

STM is an efficient technique for writing concurrent code but it has its own drawbacks and has not become mainstream yet. We had to use Haskell in the example because there is no good implementation of STM for imperative languages.

2.2.4 Data Parallelism

One of the most efficient uses of parallelism is execution of the same instructions on multiple instances of data (*SIMD*). Typical example of this approach is loop parallelisation with independent iterations.

Consider C++ code excerpt in Figure 2.5. It uses OpenMP compiler extensions [18]. The compiler processes `pragma` instruction and emits code to spawn several threads at runtime and distribute load between them.

```
1 int count = 0;
2 #pragma omp parallel for reduction(+:count)
3 for (int i = 2; i < 1000000; i++) {
4   if (is_prime(i)) {
5     count++;
6   }
7 }
```

Figure 2.5: Loop Parallelisation with OpenMP

In the example above it was programmer's responsibility to mark parallelisable loop and to make sure that parallelisation does not cause conflicts. There are other

approaches, including automatic, semi-automatic parallelisation [39] or speculative parallelisation [45].

Loop parallelisation is an efficient technique but it has limited applicability. It is used mostly in scientific calculations which process large amounts of structured data. It is not applicable, for example, to the Ornamental Gardens problem.

Similar means are available in functional programming languages via parallel functional constructs like `parmap` [43]. MapReduce [20] is another example of successful application of data parallelism to large-scale distributed computations.

2.3 Message-Passing Concurrency

Message passing is a popular approach to building distributed systems. With this approach the program is structured as a number of *processes*[4] which do not share data but communicate with each other using messages. Several styles of communication are possible.

2.3.1 Process Calculi

One style of process communication is based on CSP [35] and similar process calculi. In programming languages with CSP-style concurrency processes are usually anonymous but channels of communication are named. Messages are sent synchronously, i.e. sender waits until receiver gets the message.

Figure 2.6 shows an implementation of Ornamental Gardens problem in Go. Go [2] is a new programming language developed by Google in 2009. It features C-like syntax, garbage collection and CSP-style concurrency.

Each presented function can be started in a separate thread — *goroutine*. Goroutines can be implemented either as user threads multiplexed into kernel threads (when the code is compiled with Go compiler) or as real kernel threads (when compiled via `gcc`).

State (variable `total`) is encapsulated in the goroutine executing the `Counter` function. Other goroutines get access to the data by sending and receiving messages over `turnstile` or `terminal` channel. `Counter` thread is repeatedly waiting for a communication on any of two channels. It is ready to send current value when asked by `Terminal` or increment the value when it is requested by `Turnstile`.

[4] Term "process" is traditionally used in message-passing concurrency. It does not correspond to OS processes. The concept of process is often implemented using user-space threads.

```
1  func Counter(turnstile chan int, terminal chan int) {
2    total := 0
3    for {
4      select {
5        case entered := <- turnstile:
6          total += entered
7        case terminal <- total:
8      }
9    }
10 }
11
12 func Turnstile(id string, counter chan<- int) {
13   for {
14     admitSomeone(id)
15     counter <- 1
16   }
17 }
18
19 func Terminal(counter <-chan int) {
20   for {
21     total := <- counter
22     updateTerminal(total)
23   }
24 }
```

Figure 2.6: Message Passing in Go

When, for example, `Turnstile` thread sends a value over `counter` channel (line 15), the value is stored in memory and the thread is suspended. Processor context is then switched to `Counter` thread which was blocked in `select` statement waiting for communication. The `Counter` thread is resumed, value is read from memory (bound to variable `entered`, line 5). After that the `Turnstile` thread can be resumed and continue its execution.

With CSP-style concurrency we have tasks split into steps at the points of channel communication. Processor context may need to be switched after each step but there is room for optimisation of context switching in runtime.

Due to the synchronous communication there is high possibility for deadlock in CSP-style programs. Fortunately, the behaviour of the program often can be modelled using process calculus and verified to eliminate such problems.

2.3.2 Actor Model

The Actor model of concurrency is based on the work by Hewitt et al. [34]. With this approach program is structured as a number of named processes, *actors*, with an

asynchronous channel bound to each actor — *mailbox* for incoming messages.

An implementation of the Ornamental Gardens problem in Erlang is shown in Figure 2.7. Erlang is a garbage-collected functional language with Actor-style concurrency model.

```
1  counter(Total) ->
2    receive
3      { increment } -> counter(Total + 1);
4      { get, From } -> From ! Total, counter(Total)
5    end.
6
7  turnstile(Id, Counter) ->
8    admitSomeone(Id),
9    Counter ! { increment },
10   turnstile(Id, Counter).
11
12 terminal(Counter) ->
13   Counter ! { get, self() },
14   receive
15     Total -> true
16   end,
17   updateTerminal(Total),
18   terminal(Counter).
```

Figure 2.7: Message Passing in Erlang

A separate process is spawned for each function shown in the code. Identifier of the process executing `counter` function is passed to `turnstile` and `terminal` functions as a parameter so that they could communicate with it. All communication is asynchronous. Messages are delivered to process' mailbox where they can be retrieved via pattern-matching. When a process requires a reply message it provides his own identifier (`self()`).

Actors are implemented in Erlang as user threads and the runtime manages execution by multiplexing user threads onto kernel threads. Since there is no data sharing between processes the execution scales very well to multiprocessors and to distributed environment.

Tasks are effectively split into steps at the points of message retrieval. The execution can block if there are no matching messages in the process mailbox.

Deadlocks are still possible in Erlang but are relatively rare because of asynchronous communication.

Message passing (in both variants) is considered to be more robust than shared-state programming. Internally it is implemented using the same basic instructions as shared-state systems but the division of the entire system state into independent pieces helps avoid data races and facilitates compositional programming.

On the other hand, separation of the state makes it more difficult to achieve consensus. A shared-state program could get a global lock and modify all shared data in a consistent way, whereas a message-passing program would need to make a lot of communication to make sure that all pieces of the state are modified consistently.

2.4 Event-Driven Programming

Event handler, or just *event*, is a reaction to device signal or user action, usually implemented as a function. Events are widely used in GUI programming. It is natural to express reactions to user actions in the form of events.

Execution of event-driven programs is managed via *event queue*. Events are added to the queue as the result of external signals (user actions etc.) or explicitly by the program. Events are fetched and processed by *event loop* one by one. Control flow is effectively managed outside of the user code.

Event processing can be implemented in any programming language. Often it takes limited form of "inversion of control" when events are not initiated by the program and only handled. We consider event-driven programming in the broad sense: events can be scheduled by the program itself, e.g. as timer events (scheduled at specific time) or as asynchronous calls (added to the event queue immediately).

Significant difference between thread-based and event-based programs is in the management of *temporary state* (data needed during execution of a task but neither shared nor stored after execution). Stack in thread-based programs maintains temporary state (local variables) automatically and restores it on returning from recursive calls. Event-driven programs have to transfer all required temporary state manually from one event to another because stack is not preserved between events.

Closures and anonymous functions make transfer of temporary state between events convenient. They facilitate continuation-passing style (CPS) programming which is used in writing asynchronous event-driven code (see Section 3.1.4).

Figure 2.8 shows an implementation of Ornamental Gardens problem in event-driven JavaScript. Terminal updating task is implemented with `terminal` function which re-schedules itself as an event on each completion (it is similar to the tail-recursive calls of Erlang implementation). `turnstile` events are re-scheduled as well

but they do counter updating and re-scheduling in a continuation of asynchronous call to `admitSomeone`. `admitSomeone` has to be implemented as an asynchronous operation to avoid blocking execution of other events while I/O operation is being performed.

```
1  var counter = 0;
2
3  function turnstile(id) {
4    admitSomeone(id, function() {
5      counter += 1;
6      setTimeout(function() { turnstile(id); }, 0);
7    });
8  }
9
10 function terminal() {
11   var total = counter;
12   updateTerminal(total);
13   setTimeout(terminal, 0);
14 }
```

Figure 2.8: Event-Driven Code in JavaScript

There is similarity in behaviour between asynchronous message-passing and event scheduling but programs are structured differently. In event-driven programs task steps are not grouped around data. They operate on shared memory.

Events are also different from threads of shared-state concurrency. Steps are not grouped around tasks and access to shared data does not require synchronisation. Events are executed one after another which is equivalent to mutual exclusion on the level of events.

Another significant difference between event-driven code and thread-based code is the management of I/O. In thread-based programs (both with message-passing and with shared-state concurrency) I/O operations are usually expressed as synchronous function calls and implemented by suspending thread until I/O operation is completed. This is not acceptable for event-driven programs since any delay on the event loop thread would delay all events. It is common to implement I/O in event-driven systems asynchronously.

Asynchronous I/O requires providing continuation as a parameter to the function call. Moreover, signature of any function dealing with I/O becomes continuation-based. This syntactic feature draws a line between I/O-bound and non-I/O-bound functions similar to that between pure functions and functions with side effects in

functional programming languages. Exposing this distinction in programming language is important because delays induced by I/O operations are orders of magnitude greater than delays of memory access or synchronisation primitives [19].

In event-driven programs each event is effectively a step of a task which can be executed uninterruptedly. Tasks are divided into steps by explicit scheduling (each new iteration of `terminal`) or by asynchronous I/O calls (call to `admitSomeone` in `turnstile`). I/O operations are usually performed in separate threads managed by runtime. Event-loop thread does not have to do a lot of context switching or spinning. After completing one event it simply takes the next one from the event queue.

Event-driven programs have little overhead. The processor is utilised very efficiently while the event queue is non-empty. Synchronisation may be required for the event queue data structure when I/O threads complete operations but this aspect is managed by the runtime and can be highly optimised.

In the standard model of event-driven programming events are always executed sequentially because there are difficulties with proper coordination of access to shared data. Neither lock-based synchronisation nor message passing can be used because they involve waiting. Any waiting would stall event processing.

Sequential event-driven systems are efficient for I/O-bound tasks [7] but they do not scale to multiprocessors in compute-bound tasks. In Chapter 4 we introduce computational model for concurrency based on combination of parallelised event processing and STM to overcome this difficulty. We apply it to JavaScript programming language which is convenient for writing event-driven programs.

Chapter 3

JavaScript Programming

JavaScript is a scripting programming language developed by Brendan Eich in 1995 for Netscape Navigator 2.0. It was later adopted with slight modifications by other web browsers. In 1997 a standard was developed by ECMA for the core language.

The original design of JavaScript was influenced by Java, Scheme and Self [22]. The result is a language that features mixture of programming styles: imperative, object-oriented [38], functional [53], event-driven [25].

JavaScript is one of the most widely deployed programming languages. It is implemented by all modern web browsers. It is also implemented by several server-side frameworks. All versions of Windows since Windows 98 have JavaScript built-in for scripting (known as "JScript"). Thanks to the latest advances in optimising *just-in-time* (JIT) compilers for JavaScript [41] it also features fast execution [23].

Even though JavaScript is often regarded as a non-professional language, it is possible to develop fairly large client applications in JavaScript. For example, Gmail has reportedly about $443,000$ lines of JavaScript code [37].

3.1 Language Features

Below we describe briefly basic features of JavaScript language which are important for the next chapters.

3.1.1 Objects

Objects in JavaScript are associative arrays with strings as property names. Properties can be accessed using indexing operator ("`[]`") or dot operator ("`.`"). Properties can be added and deleted at runtime.

Typing is dynamic. The applicability of a function to an object is determined by the availability of properties ("duck typing"). Variables can be bound to values of any type.

Memory in JavaScript is managed by a garbage collector (GC). New objects can be created explicitly using `new` keyword, using special notation for objects, arrays or regular expressions, or via definition of a variable in local scope. Unreferenced objects are collected by GC. There are no destructors or finalizers.

3.1.2 Functions

Functions in JavaScript are first-class objects. This makes it possible to implement higher-order functions like `map` [53] or Y combinator [16]. The presence of anonymous functions makes the use of higher-order functions in JavaScript convenient.

There is no inherent distinction between functions and methods of objects. When a function is invoked as a method of an object the implicit formal parameter `this` is bound to the object. Otherwise it is bound to the global context.

Unlike in C or Java, scoping at block level is not available in JavaScript. Function is the level of scope.

3.1.3 Closures

Nested functions can be created inside functions. They can reference variables in the outer function scope. By returning a reference to nested function it is possible to create *closures* in JavaScript.

```
1  function fibonacci() {
2    var a = 0;
3    var b = 1;
4    return function() {
5      var t = a;
6      a = b;
7      b = t + b;
8      return b;
9    }
10 }
11
12 f = fibonacci();
13 console.log(f(), f(), f(), f(), f(), f()); // prints 1 2 3 5 8 13
```

Figure 3.1: Fibonacci Numbers Generation with Closures

The code in Figure 3.1 shows an example of closure in an implementation of Fibonacci numbers generation. `fibonacci()` call yields a closure that references outer function's local variables `a` and `b`. Each successive invocation of the closure modifies the values of these variables and returns the next Fibonacci number.

3.1.4 Continuation-Passing Style

Combination of anonymous functions and closures makes possible the *continuation-passing style* (CPS) which is widely used in asynchronous JavaScript programming. CPS structures execution as a series of nested closures, each used as a continuation parameter in an asynchronous call in outer one.

```
1  function dirSize(dir, callback) {
2    fs.readdir(dir, function(err, files) {
3      if (err)
4        callback(err);
5      if (files.length == 0)
6        callback(null, 0);
7
8      var count = files.length;
9      var size = 0;
10     var errors = [];
11
12     for (var i = 0; i < count; i++)
13       fs.stat(files[i], function(err, stats) {
14         if (err)
15           errors.push(err);
16         else
17           size += stats.size;
18
19         count--;
20         if (count == 0)
21           callback(errors, size);
22       });
23   });
24 }
```

Figure 3.2: Continuation-Passing Style Example

Function `dirSize` presented in Figure 3.2 demonstrates use of CPS. It calculates the sum of file sizes in a directory. `fs.readdir` call receives an anonymous function (continuation) which is invoked on the completion of directory read operation. The continuation can access parameters of the outer function (e.g. `callback`). The calls to `fs.stat` receive an anonymous function as a continuation parameter which refers to the variables one and two levels above it in the nesting structure.

As Adya et al. noted, CPS is effectively an instrument of temporary state management in asynchronous programming similar to the stack in synchronous [11]. Code after a function call has access to the variables defined before the call. Significant difference is that the stack maintains all local variables, while closures transfer only values of explicitly referenced variables thus potentially reducing the amount of maintained data. JavaScript code can use both instruments, usual synchronous calls are available along with CPS.

3.2 Client-Side JavaScript

Most modern web browsers implement JavaScript execution environment. It is actively used in dynamic web pages — HTML documents which change theirs contents and presentation. This technology allows building rich interactive web applications like Google Gmail and Facebook.

Implementations of JavaScript in web browsers follow ECMAScript standard [9]. 5th edition of ECMAScript was published in December 2009 but is not fully implemented yet. Previous, 3rd edition, published in December 1999, is fully implemented by modern browsers.

Execution of JavaScript in web browsers is centred around event loop. Event loop fetches events from event queue and executes them one by one. Scripts referenced on the web page are executed as the first event. There is an API to add more events to the queue.

Events can be reactions to user actions (keyboard, mouse) or callbacks from asynchronous operations or timers. Custom events are added to the queue using `setTimeout` function, often with zero timeout just to force asynchronous execution.

ECMAScript standard does not define event processing. It defines core language only. DOM Level 2 Events specification does define event processing [8] but it does not include description of event loop. And not all browsers follow the specification. There are major differences and inconsistencies in implementation of event processing. For example, an event related to a DOM element (e.g. mouse click) can be delivered from outermost elements to innermost elements in one browser and in opposite direction in another.

Drafts of HTML5 specification and DOM Level 3 Events define behaviour of event loop. HTML5 specification [6] requires event loop to handle one or more task queues as ordered lists of tasks. Task queue is a queue of events from single source (e.g. keyboard, timer, etc.). There is no ordering of task queues.

The `setTimeout` method used for scheduling of custom events does not guarantee execution with exact timeout. Moreover, the standard requires that timeouts less than 4 milliseconds are increased to 4 milliseconds.

For the purpose of this thesis the following characteristics of the event processing are important.

- Events are processed sequentially.
- The order of event processing is not guaranteed. Though future standards may define ordering of events, programmers cannot now rely on this.

Another important characteristic of JavaScript implementation in web browsers is the absence of blocking operations. There is no `sleep` or `wait` function. It is possible to issue synchronous HTTP request but programmers are discouraged from doing so because blocking calls make HTTP page unresponsive[1].

3.3 Server-Side JavaScript

JavaScript is usually implemented in web browsers as a separate component. These components have been used recently to build server-side JavaScript frameworks: Node.js, v8cgi, Juice, etc. There is no standard for server-side JavaScript yet and frameworks implement diverse approaches. We will focus on the most popular one — Node.js [4].

Node.js uses V8, Google's implementation of JavaScript, which is also used by Chrome web browser. The design of Node.js is intended for writing scalable network applications.

The execution of JavaScript in Node.js is also controlled via event loop and the same scheduling mechanisms as in web browsers (`setTimeout`) are supported. Event processing is sequential and the order of event execution is not guaranteed.

Node.js implements asynchronous I/O. Any potentially blocking operation can be scheduled for execution with a callback to be called on completion. This event-driven I/O scales well.

In the next chapter we rely on the described features of the language to propose automatic parallelisation of event-driven programs. This makes JavaScript a promising choice for the development of computationally intensive applications as well.

[1] Most web browsers interrupt events running for too long.

Chapter 4

Automatic Parallelisation

By *parallelisation* we mean taking sequential program code and running parts of it in parallel on a shared memory multiprocessor. Automatic parallelisation can take advantage of computational power of multiprocessors while keeping code simple and straightforward.

As a motivating example, consider simple web-service written in JavaScript that calculates nth prime number (Figure 4.1).

```
1 function serve(request, response) {
2   var n = parseUrl(request.url);
3   var p = nthPrime(n);
4   response.write(p.toString());
5   response.end();
6 };
7
8 var http = require('http');
9 http.createServer(serve).listen(1234);
```

Figure 4.1: HTTP Service in JavaScript

This service can be run with Node.js and requested by any web client at URL `http://servername:1234/?100000`. Function `parseUrl` extracts parameter from the request and function `nthPrime` performs the calculation. Both functions are referentially transparent, i.e. they have no side effects when executed.

Each request from a client results in the creation of a new event which contains a closure around `serve` function with instantiated parameters. Events are placed on the event queue and processed via event loop (see Section 4.1.1 for more details on event processing in JavaScript).

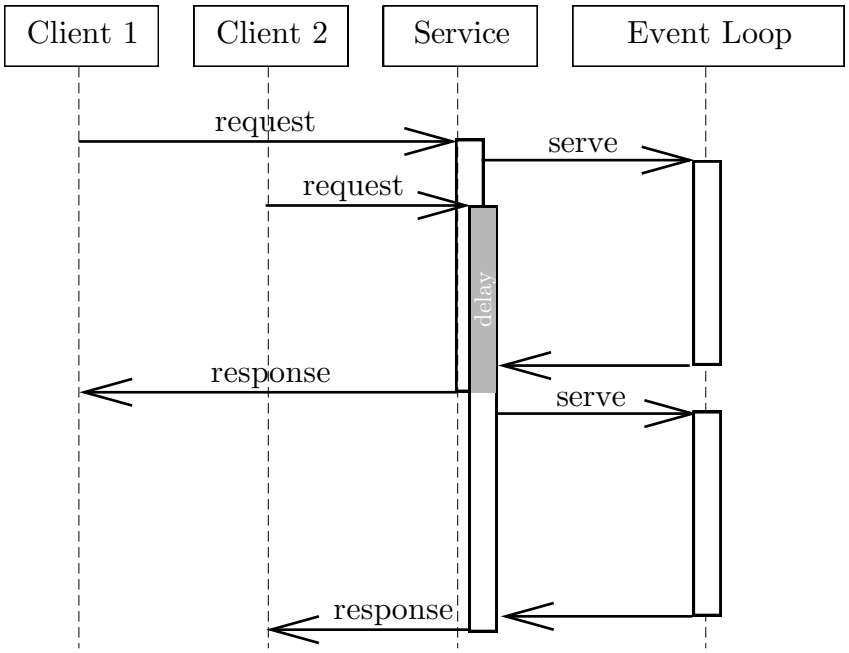


Figure 4.2: Sequential Event Processing

The sequential event processing works well for I/O-bound tasks and often scales better than a multi-threaded implementation [55, 56]. However, single-threaded event loop can become a bottleneck for compute-bound tasks.

The `nthPrime` function in our example is computationally intensive and can take few seconds to be completed for values of `n` as small as 100000. While the function is being executed, all requests from other clients have to be blocked even if the machine has several idle processors. Figure 4.2 shows the delay introduced by sequential processing of requests.

In this thesis we explore the possibility to take advantage of multiprocessors via execution of several events in parallel. The idea is to speculatively execute events under the assumption that they do not interfere with each other. When they do interfere the arising conflicts can be handled by cancelling and re-executing events. Each event effectively corresponds to a transaction.

The proposed out-of-order execution of events is designed to be applicable to the existing JavaScript code[1]. It does not require any modifications of the source code to take advantage of the parallelisation. And, more importantly, parallelisation should not change the semantics of the code (see 4.6.1).

[1]We are interested in server-side code in the first place. Execution of JavaScript in a web browser could also be parallelised.

Other levels of the code (individual statements, loops, functions) could also be parallelised [45] but the level of events in JavaScript is more parallelisable than other levels [24]. Event parallelisation also provides a convenient model for concurrent programming (see Section 4.5).

The following features of JavaScript make it amenable to event parallelisation.

- absence of blocking operations,
- serial event processing semantics,
- unordered event queue.

The non-blocking event-driven code helps utilise processor to the maximum with useful work whenever events are available on the queue. Serial event processing is convenient for reasoning about programs and fits the semantics of transactional execution. The absence of fixed ordering permits flexible scheduling of events thus increasing concurrency.

Parallelisation of event-driven code could be applied to another language but we do not know any suitable alternative. Continuation-passing style (Section 3.1.4) makes writing event-driven code in JavaScript especially convenient.

4.1 Speculative Execution

4.1.1 Default Execution Order

Event loop implemented by all web browsers and server-side JavaScript frameworks is not a part of the ECMAScript specification [9]. W3C HTML5 draft specification partly defines behaviour of event loops [6] whereas previous standards did not.

Different web browsers have different (and often unspecified) event loop implementations. Server-side JavaScript frameworks can introduce their own event handling logic. Most event loops in JavaScript environments work de facto in the following way.

The main script executed by Node.js (or scripts referenced on a web page loaded in a web browser) makes the initial event. The initial event can use the API provided by the environment to schedule other code on the event queue. There are three ways for an event to get to the queue.

1. A method without parameters can be provided to `setTimeout` call with a timeout in milliseconds[2]. It is expected to be executed later, after the currently executing event finishes. The scheduled event does not have to run exactly at the specified timeout but it must not run earlier. Related method `setInterval` schedules an event to run regularly and `process.nextTick` in Node.js is essentially equivalent to `setTimeout` with zero timeout.

2. A method with formal parameters can be provided as a continuation to an asynchronous I/O operation. It is expected to be executed after I/O operation completes. It is the responsibility of the I/O operation to schedule a closure that executes the continuation with actual parameters (see `readdir` call in Figure 3.2 for an example). The closure should be executed as soon as possible. But no guarantees are provided about the order of execution or the magnitude of the delay before the closure is executed (see Section 3.2).

3. An external signal (e.g. user clicking a button on a web page or a connection request in HTTP service) can result in an event scheduled to handle the signal if callback has been set for that signal (for example function `serve` in Figure 4.1). The execution of a callback is similar to I/O operation.

Event queue is processed by event loop as shown in Figure 4.3. Server-side JavaScript engine simply stops when there are no more events to process while web browser waits for user input.

repeat
  event ← *queue.get*
  execute *event*
until *queue* is empty

Figure 4.3: Single-Threaded Event Loop

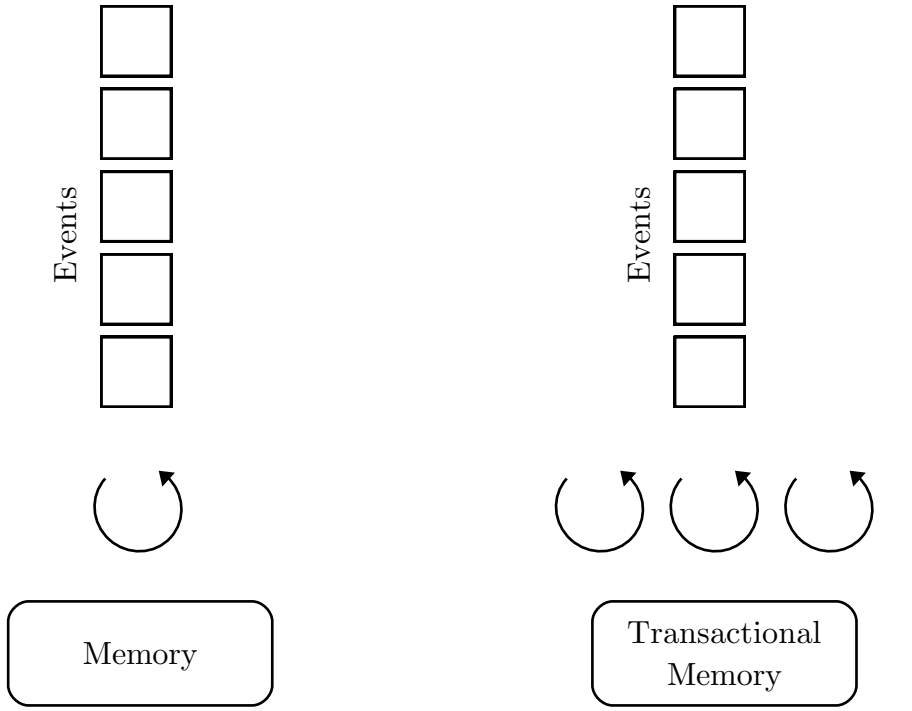
Though events are often processed in FIFO order, this behaviour is not guaranteed. Programmers are discouraged from relying on the order of event processing.

4.1.2 Speculative Parallel Processing

What programmers can rely on is the sequential single-threaded processing of events. Events cannot be interrupted by other events and there is no concurrency in event

[2]Events scheduled via `setTimeout` are often named "callbacks". We use the term "event" for any code executed via event loop.

processing. We are going to change that and introduce concurrency in event processing while keeping the illusion of sequential execution.



(a) Single-Threaded Processing (b) Multi-Threaded Processing

Figure 4.4: Event Loop Parallelisation

Figure 4.4 shows the modifications required for parallelised event processing. We need to start multiple threads to work cooperatively on a common event queue. Each thread should run its own event loop, similar to that in Figure 4.3, but with proper synchronization.

With multi-threaded processing the order in which events are completed depends on the duration of events as well as on the order in which they are started. So the actual order can differ significantly from the order the events would finish if they were processed sequentially in the order of their submission. It is not a problem as long as the code does not rely on the order of event processing (see Section 4.6.1 for the discussion of order-dependent code).

In the example above (Figure 4.1) events are scheduled on HTTP requests from clients. There are no assumptions embedded in the program regarding the order in which requests are served. Provided there is no data shared (which is the case in the example) several requests can be served in parallel without conflicts.

Since the order of events processing is not fixed there is room for optimisation by scheduling events out-of-order. Event loops could gather statistics on the duration of execution for each event and then use it to select the next event to run. Or the event queue can be split between threads with some form of "work stealing" for load balancing [26]. In the simplest case FIFO strategy for each individual event loop can be used.

4.2 Conflict Resolution

Each event in JavaScript is executed with an empty stack. No data is shared between events through the stack. Thus each event loop in the parallel case can have its own JavaScript stack.

However, events can access variables in the global context, which is shared. Parts of local contexts could also be referenced via closures (Section 3.1.3) and shared (for example when several copies of a closure are added to the queue).

Concurrent readings of the same data by two or more events does not break the illusion of independent execution but concurrent modifications of shared data could change the result of computation due to data races. Figure 4.5 shows code fragment where shared `counter` is accessed by two asynchronous events.

```
1 var counter = 0;
2
3 function increment() {
4   counter += 1;
5 }
6
7 setTimeout(increment, 0);
8 setTimeout(increment, 0);
```

Figure 4.5: Conflicting Access to Shared Data

Software transactional memory (STM) introduced by Shavit and Touitou can be used to detect conflicting access to shared data [52]. STM is an approach to memory management that automatically resolves data races (see Section 2.2.3).

STM requires code to be organized as a series of transactions. It guarantees that all changes to shared data in a transaction happen atomically, i.e. either applied or discarded as a whole. Event in JavaScript is a natural unit of transaction. The sequential semantics of order processing fits the serialisation of transactions provided by STM very well.

With STM changes are not final until transaction has been committed. They are either kept in a private copy of shared data and applied on commit ("lazy STM") or are applied directly but rolled back on abort ("eager STM"). In any case events effectively become serialized by their commit time.

To achieve atomicity STM needs to maintain sets of memory locations read or written by each transaction — to apply changes with lazy implementation or to roll back changes with eager implementation. For this to work all read and write operations on shared data in the program need to be instrumented with calls into STM.

Read and write sets can be managed with different granularity. It is most natural to consider JavaScript objects as elements of transactional data.

Figure 4.6 shows the embedding of STM into parallelised event loop.

```
repeat
  event ← queue.get
  repeat
    begin transaction
    execute event within transaction
    if no conflicts then
      commit transaction
    else
      abort transaction
    end if
  until committed
until queue is empty and all other loops are idle
```

Figure 4.6: Event Loop with Transactional Memory

The STM design required for conflict resolution in parallelised event-driven programs is simpler than that directly exposed in a programming language. There is no need for nested transactions because events cannot be nested. There is no need for explicit `retry` operation for synchronisation because semantics of event-driven programs is sequential.

4.3 Garbage Collection

Any object can potentially be shared between events. So the entire heap should be managed by STM. Since the memory in JavaScript is also managed by *garbage collector* (GC) there should be integration between GC and STM. There are two aspects to consider: memory allocation and memory deallocation.

4.3.1 Memory Allocation

Objects in JavaScript are allocated explicitly using `new` keyword or special object notation. They could also be allocated as part of a closure (Section 3.1.3).

New objects are invisible to other events until current transaction commits. So any access to uncommitted new objects can bypass STM and modifications can be performed directly. Immutable objects, e.g. strings in JavaScript, can also be handled without involving STM.

Default implementation of memory allocator in JavaScript is single-threaded but can be made thread-safe by mutual exclusion. An efficient parallel allocator, similar to those used in C++ [27, 46], would be beneficial to multi-threaded event processing.

4.3.2 Memory Deallocation

Deallocation of memory in JavaScript is performed implicitly by garbage collector. Different designs of GC can be used. All practical implementations do reallocations which change addresses of objects. If STM's read and write sets are managed in terms of addresses then relocation invalidates these sets. They need to be updated to refer to the new locations of objects.

This issue can be solved either by updating STM data on each garbage collection or by restricting the GC to time intervals when relocation is safe to perform. It is not clear which approach is better. The topic of efficient integration of STM into GC language does not seem to be well studied. The simplest solution would be to make garbage collections and transactions mutually exclusive.

When the execution of JavaScript is made multi-threaded its memory consumption should grow. Execution is likely to benefit from parallel GC similar to those used in multi-threaded GC languages [42].

4.4 External Code

The ECMAScript specification does not define I/O operations and JavaScript engine implementations do not have them built-in. I/O operations are implemented as external calls into platform-specific code (usually in C++ or Java).

Memory and other resources used by external calls are not managed by JavaScript runtime and cannot be directly controlled by STM. Changes to the resources made by an external call are usually irreversible (e.g. input from keyboard). Thus all

events performing external calls should be run under mutual exclusion and cannot be aborted.

These requirements can be implemented by introducing an exclusive token which is to be obtained before any external call and before commit. The token should be released after commit. There are also STM designs that natively support the notion of irreversible transactions.

Mutual exclusion introduced for all I/O operations may seem to be destroying all concurrency introduced by parallelisation. But it is not, for the following reasons.

- Pure JavaScript code still benefits from parallelisation.
- External calls can be structured to run asynchronously in a separate event thus limiting data conflicts.
- Asynchronous operations can be automatically deferred until commit phase.

Consider code fragment in Figure 4.7 that demonstrates usage of asynchronous I/O in JavaScript.

```
1  var fs = require('fs');
2
3  fs.readFile('/etc/passwd', 'utf-8', function (err, data) {
4    var decoded = decode(data);
5    fs.writeFile('/tmp/decoded', decoded, function(err) {
6      console.log('Finished.');
7    });
8  });
9
10 console.log("File reading scheduled but not executed yet.");
```

Figure 4.7: Asynchronous I/O Example

Invocation of the `readFile` function does not have immediate effect. The callback provided to `readFile` should not be executed until the event with `readFile` call finishes. Decoding of the data and call to `writeFile` happen in a separate event, which is scheduled on the completion of `readFile` operation.

Each event makes a transaction. The call to `readFile` can be rearranged to be executed at the end of the top-level event and can be performed after its commit. This results in a delay introduced into I/O operation but removes the need for the event to hold an exclusive token (see Section 4.6.2 for more details on the correctness of such transformation).

4.5 Event-Driven Concurrency

Parallelisation of event-driven code with automatic resolution of data conflicts introduces a novel computational model for concurrency.

Structuring tasks as a series of events effectively marks these parts as available for overlapping execution under condition of serialisability. STM provides the serialisability guarantee.

Unlike message-passing concurrency, event-driven model does not rely on isolation of shared state. Event scheduling is similar to asynchronous message processing but events have access to the whole shared state without explicit synchronisation.

Event-driven concurrency model is also different from traditional lock-based and non-blocking concurrency. No explicit synchronisation is performed by events. Sequential semantics is equivalent to mutual exclusion on the level of events but is more flexible. It allows simultaneous execution of non-conflicting events.

The model is also different from traditional transactional memory usage (Section 2.2.3) in the following ways.

- There are no explicit threads in event-driven model. Each event is effectively a short-lived thread.
- The management of transactions is not exposed to the programming language. There is no API to explicitly start or commit a transaction.
- There is no synchronisation between concurrent events (usually implemented as `retry` operation in STM). Coordination in event-driven programs is implemented via event scheduling (see Section 6.2 for an example of coordination).

Limiting semantics of STM to the subset covered by event-driven programming allows simple reasoning about programs. Semantically concurrent event-driven programs are equivalent to sequential event-driven programs.

4.6 Correctness

In previous sections we have proposed changes to JavaScript engine design which facilitate parallel execution. Our goal is to make existing JavaScript applications run faster and scale well on multiprocessors. It is important not to break the semantics of the code during optimisation. This section adds more details on the correctness of proposed changes.

4.6.1 Event Ordering

Although writing code that depends on the ordering of event execution in JavaScript is discouraged, it is still possible that such code is written.

Consider code in Figure 4.8. Its behaviour highly depends on the order in which events are executed. If the first event is executed before the second one, then "OK" will be written to the console, otherwise the program will end with an error (because the variable `message` is not defined in the global context).

```
1 setTimeout(function() {
2   message = "OK";
3 }, 0);
4
5 setTimeout(function() {
6   console.log(message);
7 }, 0);
```

Figure 4.8: Order-Dependent Code

Such code is incorrect even without parallelisation because it ignores the non-determinism present in timer scheduling. But parallel out-of-order execution may expose latent errors which did not show up under sequential execution.

One possibility to avoid errors caused by order-dependent code would be automatic detection of dependencies. Techniques developed for dynamic data race detection in multi-threaded programs might be applicable [12]. However, race detection is an NP-hard problem [47] and tools used in practice are unreliable and produce many false positives [51].

Another possibility to prevent order-dependent errors is to provide convenient higher-level language constructs for ordering of events. We consider such constructs in Chapter 6.

4.6.2 Deferred Calls

Deferring asynchronous calls to commit phase would increase concurrency but may change semantics of the code. Consider code fragment in Figure 4.9. If the call to `console.log` is deferred until after second assignment operator then it outputs different value because the value of the parameter passed to `console.log` is changed.

Limiting the application of transformation to calls with immutable parameters should help avoid this issue. Strings are immutable in JavaScript and can be safely used.

```
1 var obj = { val: 'A' };
2 console.log(obj);
3 obj.val = 'B';
```

Figure 4.9: Mutable Parameters in Deferrable Call

On the other hand, it is the semantics of asynchronous calls that defines the moment when the parameters are accessed: synchronously at the initiation of the call or asynchronously. In the later case the code must be resistant to our transformations anyway.

4.7 Efficiency

While our ultimate goal is faster execution, the time frame of this thesis does not allow complete implementation of the proposed ideas. As a first step to the ultimate goal we look for scalability. The next chapter describes the prototype we have built to test scalability of event-driven JavaScript code.

STM is often considered a slow alternative to manual thread synchronisation. However, latest experiments show that good STM can outperform manually tuned locking code [50].

We believe that automatic on-line parallelisation can be made fast enough to outperform single-threaded JavaScript execution on 2 processors with increase in the number of processors giving further improvement. By analogy [29], first implementations of garbage collectors were rather slow but latest versions of Java runtime run comparable to native code written in C++ [1].

Chapter 5

Prototype Implementation

Our prototype of parallelisation runtime is written in C++ and is based on Google's JavaScript engine V8 which is the fastest implementation of JavaScript. The source code of V8 is distributed under New BSD license. There is not much design documentation available on V8 but the source code is well commented.

In this chapter we discuss the design of V8 and the modifications we have made to apply parallelisation ideas from Chapter 4. Our modifications are scattered round V8 code so we do not provide all of them in this thesis. Full modified source code is available on request.

5.1 Google's V8

Most JavaScript engines designed before V8 used interpretation of abstract syntax tree or byte-code as the method of execution. V8 compiles JavaScript into native machine code. The compilation happens *just-in-time* (JIT) when a function is executed for the first time.

All JavaScript objects have semantics of associative containers to which properties can be added or removed dynamically. Property access is the most expensive operation in JavaScript which in general case should be performed by runtime call. To reduce the cost of property access V8 uses in-line caching technique developed by Deutsch and Schiffman for Smalltalk [21]. On first call into runtime the call instruction is replaced with a direct access to the property.

V8 implements an efficient generational garbage collector (GC). It stops the execution of JavaScript to perform collection but performs collections incrementally so pauses are short. The heap is segmented into two parts. During the collection objects are moved from old to new space and all pointers to them are updated.

5.2 Execution Environment

We use V8 in our execution environment (Appendix A.2) to test scalability of parallelised JavaScript. The environment can run multiple event loops working on common event queue. Each event is executed as a transaction on STM-managed heap.

The implementation of execution environment involved

- modification of V8 internals to allow multi-threaded execution,
- interception of read and write operations on objects,
- implementation of STM integrated into V8.

V8 is cross-platform. Our execution environment runs on Windows and Mac OS X with possibility to be ported to Linux.

5.3 Threading Backend

There are several possibilities for execution of multiple event-loops on multiple cores. Our implementation is relying on OS scheduler to arrange spawned threads on separate cores. To avoid unnecessary context switching we do not spawn in our tests more threads than cores available. Each thread runs a copy of event loop working on common event queue.

Threads are created using platform-specific libraries: Pthreads on Mac OS X and Win32 API on Windows. Kernel threads are used as a backend on both platforms. Synchronization is done via platform-specific mutexes. It should be possible to further improve performance by using thread affinity and other lower-level integration with OS.

5.4 Thread State Separation

V8 implementation is inherently single-threaded. To make it ready for multi-threaded execution we had to move JavaScript stack and most of the global data into thread-local storage.

The ability to have several independent copies of JavaScript runtime in single process was implemented in V8 in March 2011 [5]. Each runtime copy was encapsulated in an instance of *isolate* but only one thread was allowed to access an instance of runtime at a time.

Some elements of the runtime state were stored in *thread local storage* (TLS) but most of the state was kept in the isolate. We have moved to TLS other parts of the state which should be private to each thread. Access to shared parts was secured by mutual exclusion.

5.5 Thread-Local Code

Since compiled code can be modified during execution by in-line caching and other dynamic optimisations, threads effectively share not only data but also code. One way to avoid conflicts on code would be to introduce mutual exclusion. However, no matter how fine-grained it had been, it would greatly reduce concurrency due to each function call involving mutex acquisition.

Our solution was to introduce separate private copies of compiled code for each thread. In this case each thread can safely execute and optimise its own copy. The described solution increases memory consumption so we are effectively trading memory for concurrency.

To our knowledge, this is the first documented use of thread-local code. The availability of thread-local JIT-compiled code opens interesting opportunities for optimisation. Parts of thread-specific state traditionally stored in TLS can be embedded in the compiled code and accessed without indirection.

5.6 Commit-Time Invalidation

We use an efficient commit-time invalidation STM designed by Gottschlich et al. [28]. It allows conflict resolution between committing and in-flight transactions by exposing each transaction's read and write sets. Read-only transactions require no commit-time validation at all.

We have implemented a simplified version of commit-time invalidating STM (Appendix A.3). The main differences from Gottschlich et al.'s design are as follows.

- We use an unsophisticated conflict resolution algorithm. Commit-time invalidation makes it possible to delay or cancel committing transaction if that helps other in-flight transactions to commit. Our implementation aborts all conflicting in-flight transactions.

- The use of Bloom filters [13] for read and write sets, suggested by Gottschlich et al., would achieve full opacity[1] in $O(N)$ time, where N is the number of read and write operations. We maintain read and write sets in non-probabilistic data structures (`std::set` and `std::map` which are implemented in C++ STL as self-balancing binary search trees). This increases time to $O(N \log(N))$ but avoids false conflicts which are possible with Bloom filters.

- Gottschlich et al.'s design reduces commit latency by involving transactions queued for commit in helping the currently committing transaction to complete write-backs. We do not implement this optimisation.

The integration of STM into V8 involved instrumentation of object read and write operations in the runtime. Commit-time invalidation requires "lazy STM" implementation. Write operation makes a copy of the object so that all further writes and reads are performed on the copy. Read operation accesses original object if it has not been modified by the current transaction. Object copies are created in V8 heap and can be collected by GC. Copies are written back to the original location in the heap on commit.

5.7 Limitations

The time frame of this thesis does not allow compete implementation though we were able to implement major components. Our prototype has the following limitations which restrict its applicability to arbitrary JavaScript code.

- Garbage collection is disabled. Enabling GC would require implementation of tighter integration between GC and STM (Section 4.3.2). With disabled GC memory allocations are limited by the heap size set at the start of JavaScript runtime. In our testing we used compute-intensive code which did not need a lot of memory allocations.

- In-line caching (Section 5.1) is disabled. To enable it we would need to modify compiler and optimizer to instrument read and write operations in *generated* machine code. Also the management of read and write sets in STM would need to be implemented in assembler. Disabling in-line caching hits the performance dramatically but still allows us to test scalability.

[1] Opacity guarantees that even non-committed transactions do not observe inconsistent state [30].

- External calls are not monitored. In a complete implementation they would require mutual exclusion and deferring of asynchronous calls (Section 4.4). Since our tests are written in (almost) pure JavaScript this limitation does not affect them.

5.8 Evaluation

While our ultimate goal is the increased absolute performance of JavaScript on multiprocessors, in this thesis we approach only first step — scalability.

It would be interesting to test the scalability of JavaScript parallelisation on a real-world application. Unfortunately, the incompleteness of our implementation does not allow that. Most real-world JavaScript applications actively perform memory allocations and use I/O. We would need to enable GC and implement deferred asynchronous calls first.

Specialised JavaScript benchmarks might have been better suited for testing than applications but existing JavaScript benchmarks do not include events. This is a known problem [48].

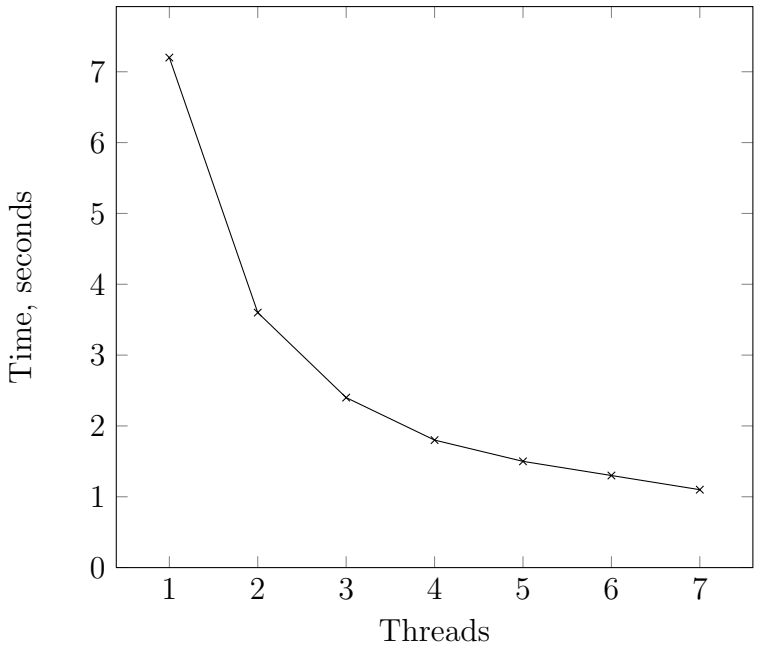
To evaluate scalability we have developed a simple compute-intensive JavaScript benchmark (Appendix A.1). It computes number of primes in range $2 \ldots 10^6$. Correctness of the result is easily verifiable — there are 78498 primes in this range. The range is split into batches. Each of the batches is inspected in a separate event. We used 1000 as default batch size.

We have run this program in our parallel execution environment (Appendix A.2) on an 8-core Xeon multiprocessor under Mac OS X. The results of running the program with different number of threads are shown in Figure 5.1. By limiting the number of threads we effectively limit the number of cores used by the program.

The results show almost linear scalability — 6.5 times speed-up on 7 cores. We cannot draw any conclusion on absolute performance yet because the testing environment runs with V8 optimisations disabled.

It should be noted that the implemented JavaScript program is highly parallelisable. In a real-world application scalability will depend on the amount of data sharing in the application and on the granularity of events.

Figure 5.2 shows dependence between batch size (the range of numbers inspected by one event) and execution time using 2, 4, and 6 cores. The shorter the events the longer total execution time. This is caused by increased overhead of event scheduling and by increased rate of transaction aborts. Shorter transactions in our case have



Threads	Time, seconds
1	7.2
2	3.6
3	2.4
4	1.8
5	1.5
6	1.3
7	1.1

Figure 5.1: Scalability Testing Results

higher chance of being in conflict. Running more transactions in parallel increases conflict rate as well.

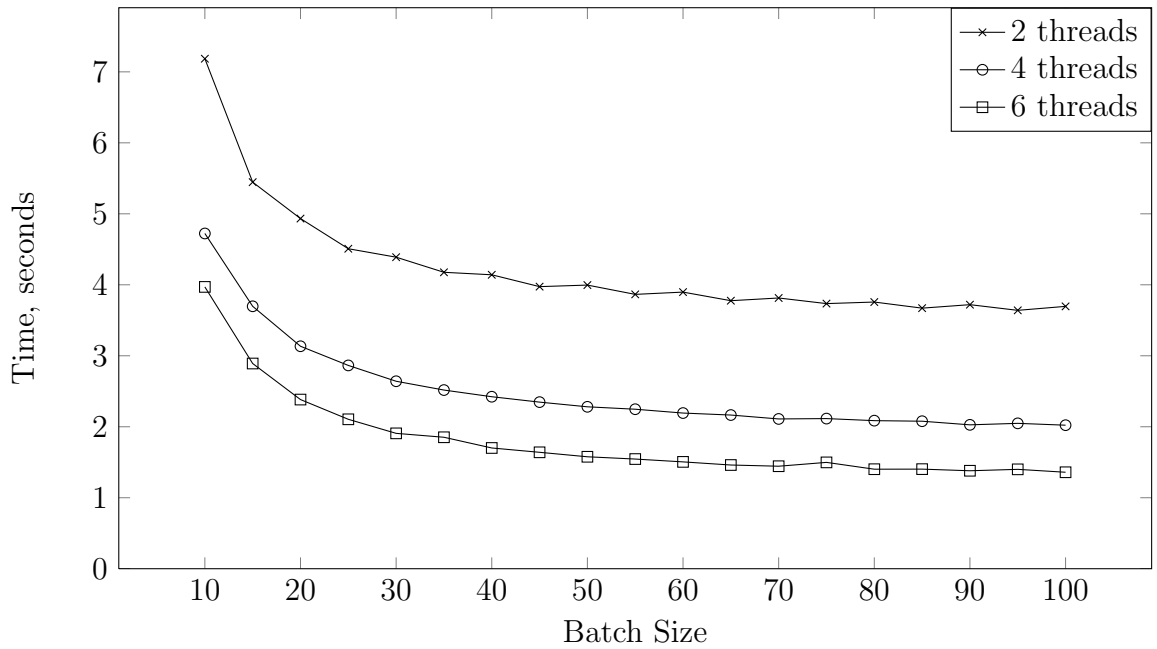


Figure 5.2: Small Batch Size Performance

Figure 5.3 illustrates dependency between batch size and number of aborted transactions. Batch size of 10 numbers results in about 10^5 transactions being aborted

(approximately every event fails once). With batch size larger than 100 numbers the rate of aborts in our tests was close to zero and the time of execution was stable until the size of about 10^5.

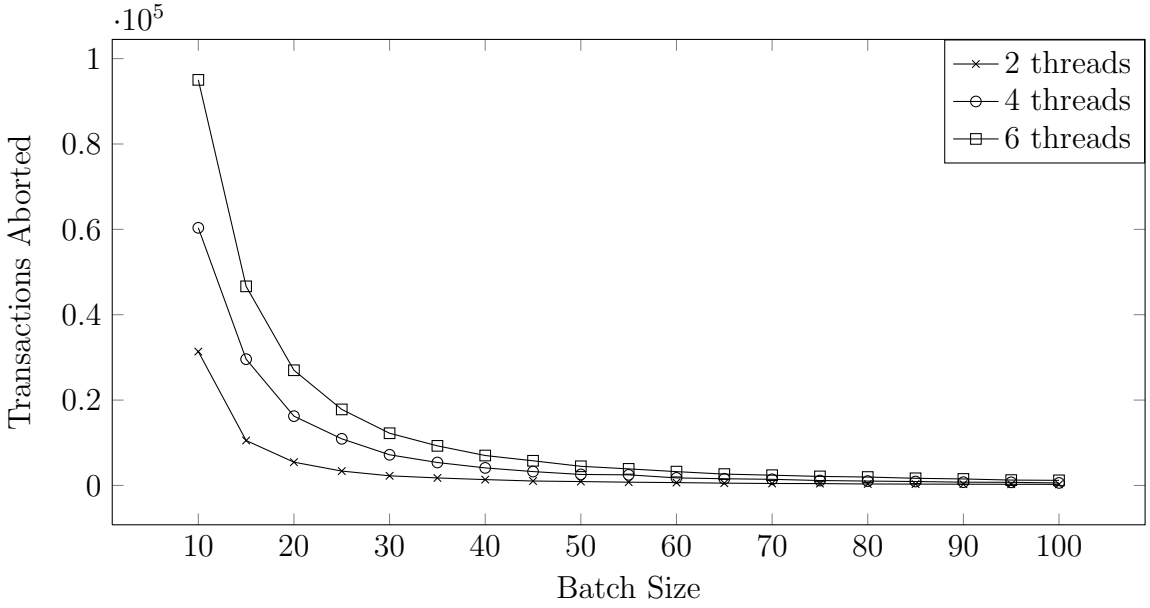


Figure 5.3: Transactions Abort Rate

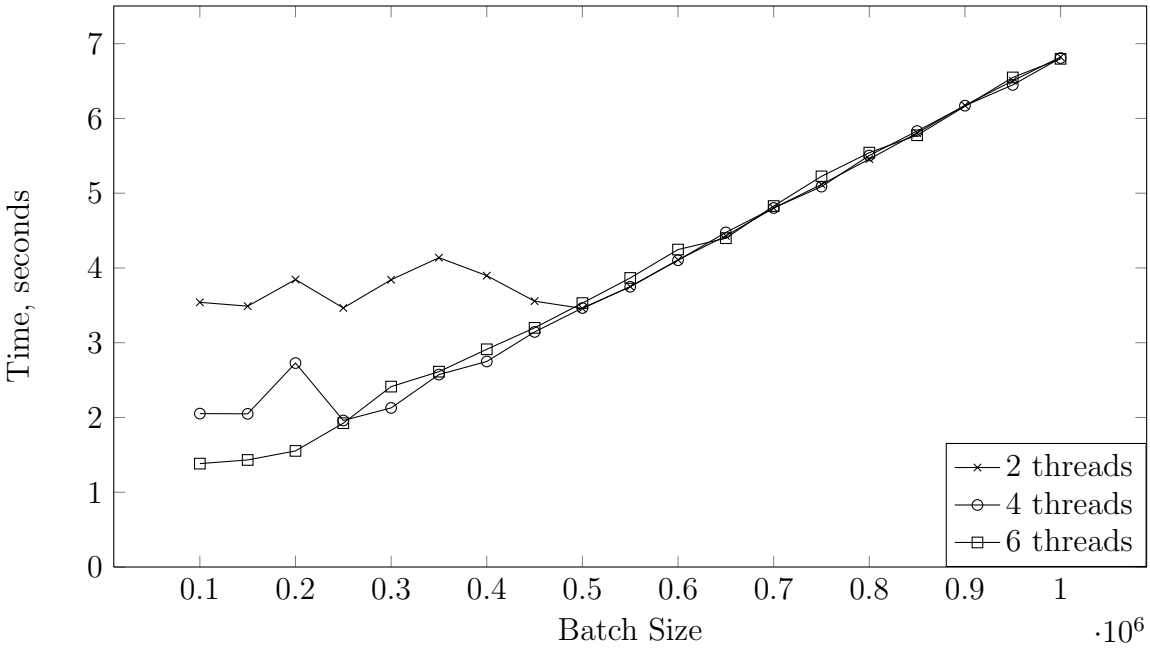


Figure 5.4: Large Batch Size Performance

If the size of batches is too large then it prevents even distribution of computation between cores. Figure 5.4 shows execution time for large batches. With the size

of batch close to 10^6 almost all computation is performed by single core and the execution time is the same for any number of cores. Degradation in our tests starts at batch size of about 2×10^5.

Degradation is not smooth because some of the batch sizes can be distributed evenly. For example, with size 2.5×10^5 we have four events which distribute evenly to four cores — every core has one event to process. With size 2×10^5 there are six events so two of the four cores have to process two events each (4×10^5 numbers).

Chapter 6

Concurrent JavaScript

In Chapter 4 we proposed automatic parallelisation of event-driven JavaScript code. This introduces true concurrent programming in JavaScript. Without parallelisation only asynchronous I/O could be executed concurrently. With parallelisation compute-intensive programs can distribute computation to multiple cores.

Event-based programming is known to be a dual of thread-based programming [40]. But the absence of manual synchronisation makes it more robust than thread-based [17]. We believe that with automatic parallelisation event-driven programming can also be an effective tool in leveraging computational power of modern multiprocessors.

6.1 Writing Parallelisable Code

Event-driven JavaScript programs written in continuation-passing style (Section 3.1.4), which is typical, should parallelise well. To make them even more suitable for parallelisation the following recommendations should be followed.

- Break data-parallel computations into pieces and execute them asynchronously.
- Make checkpoints in long-running computations by splitting them into several asynchronous stages.
- Schedule several tasks at once when possible. Having several events on the queue is crucial for parallelisation.
- Tasks should not be too small because event scheduling has its overhead.
- Minimize coupling between modules. The more data is shared the higher chances of conflict.

Well-structured object-oriented code runs faster in V8 due to automatic inference of classes. In a similar way well-structured concurrent code should be amenable to efficient parallelisation.

6.2 Higher-Level Constructs

To assist in writing parallelisable code libraries with higher-level constructs can be used. There are many of them available for JavaScript.

As an example, consider sequential and parallel execution of two asynchronous operations. It can be written in parallelisable way as shown in Figure 6.1.

```
1  // sequential
2
3  setTimeout(first(
4    function(err) {
5      setTimeout(second(
6        function(err) {
7          done();
8        }), 0);
9    }
10 ), 0);
11
12 // parallel
13
14 var counter = 2;
15
16 setTimeout(first(
17   function() {
18     counter--;
19     if (counter == 0)
20       done();
21   }
22 ), 0);
23
24 setTimeout(second(
25   function() {
26     counter--;
27     if (counter == 0)
28       done();
29   }
30 ), 0);
```

Figure 6.1: Manual Control Flow

The same behaviour can be achieved using *async* library [44] as in Figure 6.2, resulting in more readable and maintainable code.

```
1 var async = require('async');
2
3 // sequential
4 async.series([first, second], done);
5
6 // parallel
7 async.parallel([first, second], done);
```

Figure 6.2: Use of Async Library

6.3 Language Extensions

Asynchronous JavaScript code written in *continuation-passing style* is sometimes considered unreadable due to deep nesting of callbacks. Each asynchronous call requires additional level of nesting. Fragment 6.3 shows a program that asynchronously copies a file and outputs a message on completion. It has two levels of nesting.

```
1 var fs = require('fs');
2
3 fs.readFile('file1.txt', 'utf-8', function(err, data) {
4   fs.writeFile('file2.txt', data, function() {
5     console.log('Done.');
6   });
7 });
```

Figure 6.3: Nested Callbacks in JavaScript

The problem of program readability can be solved by language extensions that automatically unwrap asynchronous calls. Figure 6.4 shows the same code written in Kaffeine, a language based on JavaScript. [3]. The code is more readable and looks synchronous. The "bang" operator `!` marks calls which should be performed asynchronously.

```
1 var fs = require('fs');
2
3 err, data = fs.readFile!('file1.txt', 'utf-8');
4 fs.writeFile!('file2.txt', data);
5 console.log('Done.');
```

Figure 6.4: Asynchronous Calls with Kaffeine

Kaffeine code is compiled into standard JavaScript code and can be executed in any JavaScript environment.

For the calls marked with the bang operator Kaffeine automatically produces asynchronous calls in JavaScript with the rest of the code enclosed in continuations. The code produced from the example in Figure 6.3 is equivalent to the code in Figure 6.3.

Chapter 7

Related Work

Event-driven programming, parallelisation, and transactional memory are active research topics. The following works are related to our topic.

Zeldovich et al. considered parallelisation of event-driven programs [56]. To resolve possible conflicts on shared data they proposed assignment of "colours" to events and simultaneously execution of events with different colours only. Colouring requires manual analysis of the program but the resulting code can run with little parallelisation overhead.

Isard and Birrell proposed automatic mutual exclusion as a new concurrent programming model [36]. In their model program is structured as a set of asynchronous calls. Each asynchronous call is executed in a transaction but can be broken into several atomic fragments by an `yield` construct. Thus any function call can potentially commit current transaction and start a new one. Atomic fragments include guard conditions for coordination of concurrent calls. Failed guard condition results in transaction being aborted and restarted.

Mehrara et al. developed parallelisation engine for JavaScript to exploit loop level parallelism [45]. Their system speculatively parallelises loops and uses software transactional memory for the detection of cross-iteration dependencies. In case of a conflict loop execution is aborted and restarted in sequential mode.

Chapter 8

Conclusions

This thesis takes the first step towards event-driven concurrency. We have proposed automatic parallelisation of event-driven programs using transactional memory as a simple yet scalable computational model for concurrent programming.

Sequential semantics of event-driven programming is a good match for serialisability offered by transactional memory. Events are easier to reason about than most of the concurrent concepts. Parallelisation automatically takes advantage of multiple cores offered by modern processors. Software transactional memory guaranties consistent execution.

As a proof of concept we have built a prototype of parallelisation engine, using JavaScript for testing because of its convenience for event-driven programming. The experiments have demonstrated that compute-bound event-driven code can be automatically parallelised with almost linear scalability on a modern multicore processor.

We believe that event-driven concurrency can be an efficient and reliable alternative to existing approaches in concurrent programming.

Chapter 9

Further Work

We see following interesting and promising directions of further work on the topic of event-driven concurrency.

Formal proof of correctness of program transformations proposed in Chapter 4 should lay ground for further research. Intuition might have been enough for first experiments but for the approach to be used in practical programming more solid ground is required.

The idea of parallelisation of event-driven code can be applied to another, less dynamic than JavaScript, programming language. This may simplify reasoning about programs even further.

There are no realistic benchmarks for event-driven JavaScript available. To measure absolute performance gains of parallelisation it is necessary to develop such benchmarks.

Experimentation with benchmarks will require complete implementation of parallelising execution environment. In particular, garbage collection should be integrated with software transactional memory. The topic of interaction between STM and GC makes an interesting research itself. It does not seem to be studied.

Full-fledged implementation of parallelisation will allow embedding of modified engine into existing server-side JavaScript frameworks to test performance and scalability on real-world applications. This may also bring practical benefits improving the performance of server-side JavaScript programs.

Since event-driven programming closely reflects the asynchronous nature of computer architecture, it would be interesting to investigate deeper integration of event processing into operating system. For example, it might be efficient to bypass the level of threads in OS and schedule processor resources directly in terms of event processing.

Appendix A

Listings

A.1 Scalability Test

```
1  // adapters for execution under Node.js
2  async = typeof async != 'undefined' ? async : process.nextTick;
3  print = typeof print != 'undefined' ? print : console.log;
4
5  function isPrime(n) {
6    for (var i = 2; i*i <= n; i++) {
7      if (n % i == 0)
8        return 0;
9    }
10   return 1;
11 };
12
13 var counters = {
14   processed : 0,
15   primes : 0
16 };
17
18 function inc_primes(n) {
19   counters.primes += n;
20   return counters.primes;
21 }
22
23 function inc_processed(n) {
24   counters.processed += n;
25   return counters.processed;
26 }
27
28 // returns a closure
29 function searchPrimes(first, last) {
30   return function () {
31     var local_count = 0;
32     for (var i = first; i < last; i++) {
33       if (isPrime(i))
```

```
34          local_count++;
35      }
36      var global_count = inc_primes(local_count);
37      if (inc_processed(last - first) == LAST - FIRST)
38        print(global_count + " primes.");
39    }
40  };
41
42  var FIRST = 2;
43  var LAST = 1000000;
44  var BATCH = 1000;
45
46  for (var i = FIRST; i < LAST; i += BATCH) {
47    var first = i;
48    var last = i + BATCH;
49    if (last > LAST)
50      last = LAST;
51    var closure = searchPrimes(first, last);
52    async(closure);
53  };
54
55  // http://primes.utm.edu/howmany.shtml
56  // pi(10000) = 1229
57  // pi(100000) = 9592
58  // pi(1000000) = 78498
59  // pi(10000000) = 664579
```

A.2 Execution Environment

```
1  // we include internal header which includes the public one
2  #include <v8.h>
3
4  // standard library
5  #include <queue>
6  #include <string>
7  #include <fstream>
8
9  using namespace v8;
10
11 v8::internal::Thread::LocalStorageKey thread_name_key =
12   v8::internal::Thread::CreateThreadLocalKey();;
13
14 // reads a file into a v8 string.
15 Handle<String> ReadFile(const char* filename) {
16   std::ifstream in(filename, std::ios_base::in);
17   std::string str;
18   str.assign(std::istreambuf_iterator<char>(in),
19     std::istreambuf_iterator<char>());
20
21   return String::New(str.c_str());
22 }
23
24 // JavaScript function print(value,...)
25 Handle<Value> Print(const Arguments& args)
26 {
27   static v8::internal::Mutex* print_mutex =
28     v8::internal::OS::CreateMutex();
29   v8::internal::ScopedLock print_lock(print_mutex);
30
31   char* thread_name = reinterpret_cast<char*>(
32     v8::internal::Thread::GetExistingThreadLocal(thread_name_key));
33   HandleScope handle_scope;
34   for (int i = 0; i < args.Length(); i++) {
35     if (i > 0) { printf(" "); }
36     String::Utf8Value str(args[i]);
37     printf("[%s] %s", thread_name, static_cast<const char*>(*str));
38   }
39   printf("\n");
40   fflush(stdout);
41   return Undefined();
42 }
43
44 // each event incapsulates a JavaScript closure
45 struct Event {
46   Persistent<Function> Func;
47
48   Event(Handle<Function> func) {
49     Func = Persistent<Function>::New(func);
```

```
50    }
51
52    void Execute() {
53      Func->Call(Func, 0, NULL);
54    }
55
56    ~Event() {
57      Func.Dispose();
58    }
59  };
60
61  std::queue<Event*> event_queue;
62  v8::internal::Atomic32 running_threads = 0;
63  v8::internal::Atomic32 aborted_transactions = 0;
64  v8::internal::Mutex* mutex = v8::internal::OS::CreateMutex();
65
66  // JavaScript function async(function())
67  Handle<Value> Async(const Arguments& args) {
68    v8::internal::ScopedLock mutex_lock(mutex);
69
70    HandleScope handle_scope;
71    Handle<Function> func = Handle<Function>::Cast(args[0]);
72
73    Event* e = new Event(func);
74    event_queue.push(e);
75
76    return Undefined();
77  }
78
79  class WorkerThread : public v8::internal::Thread {
80    Persistent<Context> context_;
81  public:
82    WorkerThread(const char* name, Handle<Context> context) :
83      v8::internal::Thread(reinterpret_cast<v8::internal::Isolate*>(
84        Isolate::GetCurrent()), name) {
85      context_ = Persistent<Context>::New(context);
86    }
87
88    virtual void Run() {
89      SetThreadLocal(thread_name_key, const_cast<char*>(name()));
90
91      // enter isolate
92      Isolate::Scope isolate_scope(
93        reinterpret_cast<Isolate*>(isolate()));
94
95      // enter context
96      Context::Scope context_scope(context_);
97
98      bool active = true;
99      v8::internal::Barrier_AtomicIncrement(&running_threads, 1);
100
101     // loop until queue is empty and others are idle too
102     while (true) {
103       Event* e = NULL;
```

```
104       {
105         v8::internal::ScopedLock mutex_lock(mutex);
106
107         if (active) {
108           // count me out
109           running_threads--;
110           active = false;
111         }
112
113         if (!event_queue.empty()) {
114           e = event_queue.front();
115           event_queue.pop();
116           // count me back in
117           running_threads++;
118           active = true;
119         } else {
120           if (running_threads == 0) {
121             // we are done
122             break;
123           }
124         }
125       }
126
127       if (e != NULL) {
128         // restart transaction until it is successfully committed
129         v8::internal::Transaction* transaction = NULL;
130         while (true) {
131           transaction = isolate()->stm()->StartTransaction();
132
133           HandleScope handle_scope;
134           e->Execute();
135
136           if (isolate()->stm()->CommitTransaction(transaction)) {
137             // printf("[%s] Comitted.\n", name());
138             break;
139           } else {
140             // printf("[%s] Aborted.\n", name());
141             v8::internal::Barrier_AtomicIncrement(
142               &aborted_transactions, 1);
143           }
144         }
145         delete e;
146       }
147     }
148   }
149 };
150
151 int main(int argc, char **argv) {
152   if (argc <= 2) {
153     printf("Usage: mininode <threads> <script.js> <V8 flags>\n");
154     return 1;
155   }
156   int threads = atoi(argv[1]);
157   char* filename = argv[2];
```

```
158
159   const int MAX_THREADS = v8::internal::CoreId::kMaxCores - 1;
160   if (threads < 1 || threads > MAX_THREADS) {
161     printf("Thread number should be between 1 and %d\n", MAX_THREADS);
162     return 1;
163   }
164
165   // disable V8 optimisations
166   char flags[1024] = { 0 };
167   strcat(flags, " --noopt");
168   strcat(flags, " --always_full_compiler");
169   strcat(flags, " --nocrankshaft");
170   strcat(flags, " --debug_code");
171   strcat(flags, " --nocompilation_cache");
172   strcat(flags, " --nouse_ic");
173   V8::SetFlagsFromString(flags, strlen(flags));
174   int argcc = argc - 2;
175   V8::SetFlagsFromCommandLine(&argcc, argv + 2, false);
176
177   V8::Initialize();
178
179   // create a stack-allocated handle scope
180   HandleScope handle_scope;
181
182   // create a template for the global object and set built-ins
183   Handle<ObjectTemplate> global = ObjectTemplate::New();
184   global->Set(String::New("async"), FunctionTemplate::New(Async));
185   global->Set(String::New("print"), FunctionTemplate::New(Print));
186
187   // create a new context
188   Persistent<Context> context = Context::New(NULL, global);
189
190   // enter the context for compiling and running the script
191   Context::Scope context_scope(context);
192
193   // load and run the script
194   Script::New(ReadFile(filename), String::New(filename))->Run();
195
196   int64_t start_time = v8::internal::OS::Ticks();
197
198   // run event loops in worker threads
199   WorkerThread* thread[MAX_THREADS];
200   for (int i = 0; i < threads; i++) {
201     char name[100];
202     sprintf(name, "Worker %d", i);
203     thread[i] = new WorkerThread(name, context);
204     thread[i]->Start();
205   }
206
207   // stop when all threads are idle and the event queue is empty
208   for (int i = 0; i < threads; i++) {
209     thread[i]->Join();
210   }
211
```

```
212   int64_t stop_time = v8::internal::OS::Ticks();
213   int milliseconds = static_cast<int>(stop_time - start_time) / 1000;
214   printf("%d milliseconds.\n", milliseconds);
215   printf("%d threads.\n", threads);
216   printf("%d transactions aborted.\n", aborted_transactions);
217
218   // dispose the persistent context
219   context.Dispose();
220
221   return 0;
222 }
```

A.3 Software Transactional Memory

```
1  #ifndef V8_STM_H_
2  #define V8_STM_H_
3
4  #include "globals.h"
5  #include "list-inl.h"
6
7  namespace v8 {
8  namespace internal {
9
10 // suppose that each thread is scheduled on individual core
11 class CoreId {
12 public:
13   static CoreId Current();
14   static int CurrentInt() { return Current().ToInteger(); }
15   int ToInteger() const { return id_; }
16
17   static int const kMaxCores = 8;
18
19 private:
20   CoreId(int id) : id_(id) { }
21
22   int id_;
23 };
24
25 class Transaction;
26
27 class STM {
28 public:
29   Transaction* StartTransaction();
30   bool CommitTransaction(Transaction* trans);
31
32   JSObject* RedirectRead(JSObject* obj);
33   JSObject* RedirectWrite(JSObject* obj);
34
35 private:
36   DISALLOW_IMPLICIT_CONSTRUCTORS(STM);
37
38   Isolate* isolate_;
39   Mutex* transactions_mutex_;
40   List<Transaction*> transactions_;
41   Mutex* commit_mutex_;
42
43   friend class Isolate;
44 };
45
46 } }  // namespace v8::internal
47
48 #endif  // V8_STM_H_
```

```
1  #include "v8.h"
2  #include "stm.h"
3  #include "isolate.h"
4
5  #include <set>
6  #include <map>
7
8  namespace v8 {
9  namespace internal {
10
11 CoreId CoreId::Current() {
12   int thread_id = ThreadId::Current().ToInteger();
13   ASSERT(thread_id >= 0);
14   ASSERT(thread_id <= kMaxCores);
15   return CoreId(thread_id - 1);
16 }
17
18 class Transaction {
19   typedef std::set<JSObject*> ObjectSet;
20   typedef std::map<JSObject*, JSObject*> ObjectMap;
21
22 public:
23   Transaction(Isolate* isolate) :
24     aborted_(false),
25     isolate_(isolate),
26     mutex_(OS::CreateMutex()) {
27   }
28
29   JSObject* RedirectRead(JSObject* obj) {
30     ASSERT_NOT_NULL(obj);
31
32     // if aborted return NULL to terminate
33     if (aborted_) {
34       return NULL;
35     }
36
37     // lookup in write set and redirect if included
38     ObjectMap::const_iterator it = write_set_.find(obj);
39     if (it != write_set_.end()) {
40       return it->second;
41     }
42
43     // lookup in read set and return if included
44     if (read_set_.find(obj) != read_set_.end()) {
45       return obj;
46     }
47
48     // include in read set and return
49     ScopedLock lock(mutex_);
50     read_set_.insert(obj);
51     return obj;
52   }
53
```

```
54  JSObject* RedirectWrite(JSObject* obj) {
55    ASSERT_NOT_NULL(obj);
56
57    // if aborted return NULL to terminate
58    if (aborted_) {
59      return NULL;
60    }
61
62    // lookup in copy set and return if included
63    if (copy_set_.find(obj) != copy_set_.end()) {
64      return obj;
65    }
66
67    // lookup in write set and return if included
68    ObjectMap::const_iterator it = write_set_.find(obj);
69    if (it != write_set_.end()) {
70      return it->second;
71    }
72
73    // make a copy, include it in write set and return
74    Object* result;
75    { MaybeObject* maybe_result = isolate_->heap()->CopyJSObject(obj);
76      if (!maybe_result->ToObject(&result)) return NULL;
77    }
78    JSObject* copy = JSObject::cast(result);
79    copy_set_.insert(copy);
80
81    ScopedLock lock(mutex_);
82    write_set_.insert(ObjectMap::value_type(obj, copy));
83    return copy;
84  }
85
86  void CommitHeap() {
87    // copy all objects in write set back to their original location
88    Heap* heap = isolate_->heap();
89    for (ObjectMap::const_iterator it = write_set_.begin();
90         it != write_set_.end(); ++it) {
91      Address dst_addr = it->first->address();
92      Address src_addr = it->second->address();
93      heap->CopyBlock(dst_addr, src_addr, it->first->Size());
94    }
95  }
96
97  bool HasConflicts(Transaction* other) {
98    const ObjectMap& other_write_set_ = other->write_set_;
99    for (ObjectMap::const_iterator it = other_write_set_.begin();
100        it != other_write_set_.end(); ++it) {
101     JSObject* obj = it->first;
102     if (read_set_.find(obj) != read_set_.end()) {
103       //printf("RW conflict on "); obj->PrintLn();
104       return true;
105     }
106     if (write_set_.find(obj) != write_set_.end()) {
107       //printf("WW conflict on "); obj->PrintLn();
```

```
108         return true;
109       }
110     }
111     return false;
112   }
113
114   void Lock() { mutex_->Lock(); }
115   void Unlock() { mutex_->Unlock(); }
116
117   void Abort() { aborted_ = true; }
118   bool IsAborted() { return aborted_; }
119
120   void ClearExceptions() {
121     isolate_->clear_pending_exception();
122     isolate_->clear_pending_message();
123   }
124
125 private:
126   volatile bool aborted_;
127   Isolate* isolate_;
128   ObjectSet read_set_;   // objects read by this transaction
129   ObjectMap write_set_;  // objects written by this transaction
130   ObjectSet copy_set_;   // temporary copies we have created
131   Mutex* mutex_;
132 };
133
134 Transaction* STM::StartTransaction() {
135   Transaction* trans = new Transaction(isolate_);
136   isolate_->set_transaction(trans);
137
138   ScopedLock transactions_lock(transactions_mutex_);
139   transactions_.Add(trans);
140   return trans;
141 }
142
143 STM::STM() {
144   transactions_mutex_ = OS::CreateMutex();
145   commit_mutex_ = OS::CreateMutex();
146 }
147
148 bool STM::CommitTransaction(Transaction* trans) {
149   ASSERT_NOT_NULL(trans);
150   ASSERT_EQ(trans, isolate_->get_transaction());
151
152   ScopedLock commit_lock(commit_mutex_);
153   ScopedLock transactions_lock(transactions_mutex_);
154
155   bool comitted = false;
156
157   // if the transaction was aborted then clear exceptions flag
158   // so that it is not transferred to next attempt
159   if (trans->IsAborted()) {
160     trans->ClearExceptions();
161   } else {
```

```
162       // lock all transactions
163       for (int i = 0; i < transactions_.length(); i++) {
164         transactions_[i]->Lock();
165       }
166
167       // intersect write set with other transactions
168       // abort those in conflict
169       for (int i = 0; i < transactions_.length(); i++) {
170         Transaction* t = transactions_[i];
171         if (t == trans) {
172           continue;
173         }
174         if (t->HasConflicts(trans)) {
175           t->Abort();
176         }
177       }
178
179       // copy write set back to the heap
180       trans->CommitHeap();
181
182       // unlock all transactions
183       for (int i = 0; i < transactions_.length(); i++) {
184         transactions_[i]->Unlock();
185       }
186
187       comitted = true;
188     }
189
190     isolate_->set_transaction(NULL);
191
192     ASSERT(transactions_.RemoveElement(trans));
193     delete trans;
194     return comitted;
195   }
196
197   JSObject* STM::RedirectRead(JSObject* obj) {
198     Transaction* trans = isolate_->get_transaction();
199     if (trans == NULL) {
200       return obj;
201     }
202     return trans->RedirectRead(obj);
203   }
204
205   JSObject* STM::RedirectWrite(JSObject* obj) {
206     Transaction* trans = isolate_->get_transaction();
207     if (trans == NULL) {
208       return obj;
209     }
210     return trans->RedirectWrite(obj);
211   }
212
213   } }  // namespace v8::internal
```

Bibliography

[1] The Computer language benchmarks game. http://shootout.alioth.debian.org/.

[2] The Go programming language. http://golang.org/.

[3] Kaffeine: extended Javascript for pros. http://weepy.github.com/kaffeine/.

[4] Node.js: Evented I/O for V8 JavaScript. http://nodejs.org/.

[5] V8 issue 510: Several V8 instances in a process. http://code.google.com/p/v8/issues/detail?id=510.

[6] EventLoops. HTML5. http://www.w3.org/TR/html5/webappapis.html#event-loops.

[7] A little holiday present: 10,000 reqs/sec with Nginx! http://blog.webfaction.com/a-little-holiday-present.

[8] Document Object Model (DOM) level 2 events specification. http://www.w3.org/TR/DOM-Level-2-Events/, 2000.

[9] ECMAScript language specification. http://www.ecma-international.org/publications/standards/Ecma-262.htm, 2009.

[10] Sarita Adve. Data races are evil with no exceptions. *Communications of the ACM*, 53(11):84, November 2010.

[11] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R Douceur. Cooperative task management without manual stack management. In *Proceedings of the 2002 Usenix Annual Technical Conference*, June 2002.

[12] Utpal Banerjee, Brian Bliss, Zhiqiang Ma, and Paul Petersen. A theory of data race detection. In *Proceeding of the 2006 workshop on Parallel and distributed systems: testing and debugging*, pages 69–78. ACM Press, 2006.

[13] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[14] Hans-J. Boehm. How to miscompile programs with "benign" data races. *3rd USENIX Workshop on hot topics in parallelism*, 2011.

[15] Alan Burns and Geoffrey Davies. *Concurrent Programming*. 1993.

[16] Douglas Crockford. The little JavaScripter. `http://www.crockford.com/javascript/little.html`.

[17] Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazières, and Robert Morris. Event-driven programming for robust software. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 186–189. ACM Press, 2002.

[18] Leonardo Dagum and Ramesh Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5 (1):46–55, 1998.

[19] Jeff Dean. Designs, lessons and advice from building large distributed systems. *Keynote from LADIS*, 2009.

[20] Jeffrey Dean and Sanjay Ghemawat. MapReduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72, January 2010.

[21] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 297–302. ACM Press, 1984.

[22] Brendan Eich. Popularity. `http://brendaneich.com/2008/04/popularity/`.

[23] Geoff Flarity. Is JavaScript faster than C? `http://onlinevillage.blogspot.com/2011/03/is-javascript-is-faster-than-c.html`.

[24] Emily Fortuna, Owen Anderson, Luis Ceze, and Susan Eggers. A limit study of JavaScript parallelism. In *IEEE International Symposium on Workload Characterization*, pages 1–10. IEEE, 2010.

[25] Federico Galassi. Event driven JavaScript. `http://www.slideshare.net/fgalassi/event-driven-javascript`.

[26] Fabien Gaud, Sylvain Genevès, Renaud Lachaize, Baptiste Lepers, Fabien Mottet, Gilles Muller, and Vivien Quéma. Efficient workstealing for multicore event-driven systems. In *Proceedings of the IEEE 30th International Conference on Distributed Computing Systems*. IEEE Computer Society, 2010.

[27] Sanjay Ghemawat. TCMalloc: thread-caching malloc. `http://google-perftools.googlecode.com/svn/trunk/doc/tcmalloc.html`.

[28] Justin E Gottschlich, Manish Vachharajani, and Jeremy G. Siek. An efficient software transactional memory using commit-time invalidation. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2010.

[29] Dan Grossman. The transactional memory / garbage collection analogy. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, 2007.

[30] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 175–184. ACM Press, 2008.

[31] Tim Harris, James R. Larus, and Ravi Rajawar. *Transactional Memory*. Second edition, 2010.

[32] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. 2008.

[33] Maurice P Herlihy. Impossibility and universality results for wait-free synchronization. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 276–290. ACM Press, 1988.

[34] Carl Hewitt, Peter Bishop, and Richard Stelger. A universal modular ACTOR formalism for artificial intelligence. *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245, August 1973.

[35] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[36] Michael Isard and Andrew Birrell. Automatic mutual exclusion. *Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, 2007.

[37] Joab Jackson. Google Gmail to harness HTML5. `http://www.macworld.com/article/152344/2010/06/html5_gmail.html`.

[38] Mike Koss. Object oriented programming in JavaScript. `http://mckoss.com/jscript/object.htm`.

[39] Per Larsen, Razya Ladelsky, Jacob Lidman, Sally A. McKee, Sven Karlsson, and Ayal Zaks. Automatic loop parallelization via compiler guided refactoring. *IMM Technical Report*, 2011.

[40] Hugh C Lauer and Roger M Needham. On the duality of operating system structures. *ACM SIGOPS Operating Systems Review*, 13(2):3–19, 1979.

[41] Dave Mandelin. Know your engines: How to make your JavaScript fast. `http://people.mozilla.com/~dmandelin/KnowYourEngines_Velocity2011.pdf`.

[42] Simon Marlow, Tim Harris, Roshan P James, and Simon Peyton Jones. Parallel generational-copying garbage collection with a block-structured heap. In *Proceedings of the 7th international symposium on Memory management*. ACM Request Permissions, June 2008.

[43] Simon Marlow, Simon Peyton Jones, and Satnam Singh. Runtime support for multicore Haskell. *ACM SIGPLAN Notices*, 2009.

[44] Caolan McMahon. Async.js: async utilities for Node and the browser. `https://github.com/caolan/async/`.

[45] Mojtaba Mehrara, Po-Chun Hsu, Mehrzad Samadi, and Scott Mahlke. Dynamic parallelization of JavaScript applications using an ultra-lightweight speculation mechanism. *IEEE 17th International Symposium on High Performance Computer Architecture*, pages 87–98, 2011.

[46] Maged M Michael. Scalable lock-free dynamic memory allocation. *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, 2004.

[47] Robert H B Netzer and Barton P Miller. What are race conditions?: Some issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1(1): 74–88, March 1992.

[48] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin G Zorn. JSMeter: Comparing the behavior of JavaScript benchmarks with real Web applications. In *Proceedings of the 2010 USENIX conference on Web application development*, 2010.

[49] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. 2004.

[50] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L Hudson, Chi Cao Minh, and Benjamin Herzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2006.

[51] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: data race detection in practice. In *Proceedings of the workshop on Binary instrumentation and applications*, pages 62–71. ACM Press, 2009.

[52] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10, 1997.

[53] Oliver Steele. Functional JavaScript. `http://osteele.com/archives/2007/07/functional-javascript`.

[54] Herb Sutter and James Larus. Software and the concurrency revolution. *ACM Queue*, 3(7), 2005.

[55] Felix von Leitner. Scalable network programming. `http://bulk.fefe.de/scalable-networking.pdf`, 2003.

[56] Nickolai Zeldovich, Alexander Yip, Frank Dabek, Robert T Morris, David Mazières, and Frans Kaashoek. Multiprocessor support for event-driven programs. In *Proceedings of the USENIX Annual Technical Conference*, 2003.