Caustic: A Transactional Programming Language

Ashwin Madavan

April 5, 2018

Abstract

Many programming languages provide fundamental abstractions such as locks, semaphores, and monitors to explicitly deal with race conditions. Some, like Rust [7], go a step further and are able to statically detect race conditions between concurrent threads. But none, however, are able guarantee that race conditions will be completely eliminated from distributed systems. Distributed systems form the computing backbone of nearly every major technology from social networks to video streaming, but their intricate complexity coupled with the inability to detect race conditions makes designing them extremely error-prone. Caustic is a programming language for building correct distributed systems. Programs written in Caustic may be distributed arbitrarily without the use of any explicit synchronization, and will never exhibit race conditions.

1 Introduction

Concurrency is *hard*. Concurrency refers to situations in which multiple programs simultaneously modify shared data. Concurrent programs may be run across threads, processes, and, in the case of distributed systems, networks. Concurrency is challenging because it introduces ambiguity in execution order, and it is precisely this ambiguity that causes race conditions. Race conditions are situations in which the execution order of concurrent programs affects the outcome. For example, suppose there exist two programs A and B that each increment a shared counter x. Each program must first read the current value of x in order to write $x + 1$. If B reads *after* A writes, then B reads $x+1$ and writes $x+2$. However, if B reads *before* A writes but after A reads, then both A and B will read x and write $x+1$. This is an example of a race condition, because the value of the counter x depends on the order in which A and B are executed. This race condition may seem relatively benign, but it can have catastrophic consequences in practical systems. Suppose the value of x corresponded to your bank balance. What if your bank determined your balance differently depending on the order in which deposits are made? Race conditions manifest themselves in subtle ways in concurrent systems, and they can often be difficult to detect and challenging to remove.

1.1 Eliminating Race Conditions

Before introducing mechanisms to deal with race conditions, we must first define the necessary and sufficient criteria that any such mechanism must satisfy to correctly mitigate race conditions. Race conditions are a relatively well-studied problem in database literature, and we may draw upon known results from the field to define four properties of race-free programs. [6]

- **Atomic**: Programs are all-or-nothing; they are never partially applied.
- **Consistent**: Programs see the effect of all completed programs.
- **Isolated**: Programs cannot see the effect of in-progress programs.
- **Durable**: The effects of completed programs are permanently visible.

1.2 Locks and Leases

A common approach to dealing with race conditions is locking. Before a program accesses or modifies shared data, it first declares its intentions to other programs by acquiring a lock that it subsequently releases when it is finished with the data. Locks trivially satisfy the ACID criteria, because they require programs to have exclusive ownership before performing any operations on shared data. However, locking introduce new problems.

First, concurrent acquisition of multiple locks can cause a deadlock that prevents the system from making progress. For example, if program A acquires lock x and then attempts to acquire y and program B acquires y and then attempts to acquire x, then neither A nor B can make progress because each is waiting for the other to release their lock. This problem is typically mitigated by imposing a total order on lock acquisition; for any two locks x and y, x will always be acquired before y or y will always be acquired before x.

Second, faulty programs may never release their locks. For example, if program A acquires lock x and subsequently fails, then no other program can ever acquire x. This problem is typically mitigated by using leases. [5] A lease is a lock that is automatically released after a certain amount of time. Programs must be carefully constructed so that they complete their operations on shared data within the lease duration or refresh the lease before it expires.

Third, locking is expensive. Locks must be acquired regardless of whether or not there actually are concurrent operations on shared data, because programs cannot know if there are or will be other programs that want to simultaneously use the data. This significantly degrades performance in situations where contention between programs is low.

Fourth, locking cannot protect against programmer error. Programmers may omit or incorrectly use a lock and thereby introduce race conditions into their program. Locks cannot guarantee they will be used correctly, and, therefore, cannot guarantee that race conditions will be exhaustively eliminated from a program.

1.3 Database Transactions

An alternative approach is to use a transactional database to protect shared data from concurrent modification. There are a number of storage systems that provide ACID transactions in full or in limited capacity. However, each of these storage systems has their own bespoke interface for specifying transactions that are often lacking in functionality and performance. Recent years have marked a proliferation in NoSQL databases that scale well by shedding functionality. These databases were not popularized because of their query languages, they were *in spite* of them. Some, like Cassandra and Aerospike, attempt to mimic the relational semantics of SQL, but they fall short of implementing the entire SQL specification. Others, like MongoDB and DynamoDB, implement entire new query languages. Even SQL is not beyond reproach. SQL lacks a canonical implementation and has ambiguous and unintuitive syntax. [3] Relational databases like MySQL and PostgreSQL that each claim to implement the same SQL specification actually implement incompatible subsets of its functionality. Stack differences between query languages tightly couples storage systems and the programs that are run on them, and makes the choice of database in an application effectively permanent. While transactional storage systems provide the necessary guarantees on which correct distributed systems can be built, their collective lack of a robust and uniform interface makes it all but impossible to design non-trivial applications.

1.4 Optimistic Concurrency

Optimistic concurrency allows multiple programs to simultaneously access, but not modify, shared data without acquiring locks. Each program locally buffers any modifications that it makes and attempts to atomically apply the modifications when it completes conditioned on the data that it accessed remaining unchanged. If any data was modified, the program retries. This conditional update, known as a multi-word compare-and-swap and referred to as a **transaction**, is known to satisfy the ACID criteria and is widely used in a number of software transactional memory systems including Caustic. [11] Optimistic concurrency assumes that contention between programs will be low, because frequent retries can significantly degrade performance.

2 Architecture

Caustic is composed of three components: a **runtime** that executes programs, a **standard library** that simplifies program construction, and a **compiler** that simplifies program syntax. In this section, we'll describe each of these components in detail.

2.1 Runtime

The runtime is a virtual machine that dynamically translates **programs** into transactions. A program is an abstract-syntax tree that is composed of **literals** and **expressions**. A literal is a scalar value of type flag, real, text, and null which correspond to bool, double, string, and null respectively in most C-style languages. An expression is a function that transforms literal arguments into a literal result. Expressions may be chained together arbitrarily to form complex programs. Table 1 enumerates the various expressions supported by the runtime.

2.1.1 Execution

The runtime uses iterative partial evaluation to gradually reduce programs into a single literal result. Modifications to keys are tracked using multiversion concurrency control. Each key is associated with a **revision**, or versioned value, whose version number is incremented each time that its value is changed. A revision A *conflicts with* B if A and B correspond to the same key and the version of A is less than B. Conflict is an asymmetric relation; if A conflicts with B, then B does not conflict with A.

The runtime *executes* programs on **volumes**. A volume is a database that supports *get* and *cas*. Get retrieves the revisions of a set of keys and cas transactionally updates a set of keys if and only if a set of dependent revisions do not conflict with their current revisions in the database. Given correct implementations of get and cas, the runtime executes programs according to the following procedure.

1. **Fetch**: Get all keys that are read or written by the program that have not been fetched before and add the returned revisions to a local snapshot.

2. **Evaluate**: Recursively replace all expressions with literal arguments with their corresponding literal result. For example,

$$add(real(1), sub(real(0), real(2))) \rightarrow real(-1)$$

   The result of all writes is saved to a local buffer and the result of all read expressions is the latest value of the key in the local buffer or snapshot.

3. **Repeat**: Loop until the program is reduced to a single literal. Because all expressions with literal arguments return a literal result, all programs will eventually reduce to a single literal.

4. **Commit**: Cas all keys in the local buffer conditioned on all revisions in the local snapshot. Because programs are executed with snapshot isolation, they are guaranteed to be serializable. Serializability implies

that concurrent execution has the same effect as some sequential execution, and, therefore, that program execution will be robust against race conditions.

2.1.2 Optimizations

First, execution is tail-recursive. Therefore, programs may be composed of arbitrarily many nested expressions without overflowing the stack frame. This also allows the Scala compiler is able to aggressively optimize execution into a tight loop. Second, the runtime batches I/O. Reads are performed simultaneously whenever possible and writes are buffered and simultaneously committed. By batching I/O, the runtime performs a minimal number of operations on the database. This has significant performance implications, because I/O overhead is overwhelmingly the bottleneck by many orders of magnitude. [4]

2.2 Standard Library

The runtime provides native support for an extremely limited subset of the operations that programmers typically rely on to write programs. The standard library supplements the functionality of the runtime by exposing a rich Scala DSL complete with static types, records, math, collections, and control flow.

2.2.1 Typing

The runtime natively supports just four dynamic types: $flag$, *real*, *text*, and *null*. Dynamic versus static typing is a religious debate among programmers. Advocates of dynamic typing often mistakenly believe that type inference and coercive subtyping cannot be provided by a static type system. In fact, they can. Because static type systems are able to detect type inaccuracies at compile-time, they allow programmers to write more concise and correct code. [8] The standard library provides rich static types and features aggressive type inference and subtype polymorphism.

The standard library supports four *Primitive* types. In descending order of precedence, they are: *String*, *Double*, *Int*, *Boolean*. A $Value[+T <: Primitive]$ represents a scalar value. Values are covariant in T; a $Value[Int]$ is a $Value[Double]$, but a $Value[String]$ is not a $Value[Boolean]$. Values may either be *Constant* or *Variable*. A *Constant* corresponds to an immutable quantity, and a *Variable* corresponds to a value that is either stored locally in memory or remotely in the database.

2.2.2 Records

In addition to these *Primitive* types, the standard library also supports references to user-defined types. A $Reference[T]$ is a reference to an object of type T. References use Shapeless to materialize compiler macros that permit the fields of T to be statically manipulated and iterated. A current limitation is that objects cannot be self-referential; an object cannot have a field of its own type.

2.2.3 Math

The runtime natively supports just nine mathematical operations: *add*, *sub*, *mul*, *div*, *pow*, *log*, *floor*, *sin*, and *cos*. However, these primitive operations are sufficient to derive the entire Scala math library using various mathematical identities and Taylor series approximations. The *div*, *log*, *sin*, and *cos* functions can actually be implemented in terms of the other primitive operations; however, native support for them was included in the runtime to improve performance. The standard library provides implementations for the functions enumerated in Table 2.

2.2.4 Collections

The runtime has no native support for collections of key-value pairs. The standard library provides implementations of three fundamental data structures: *List*, *Set*, and *Map*. These collections are mutable, statically-typed, and thread-safe. Collections take care of the messy details of mapping structured data onto a flat namespace, and feature prefetched iteration. A current limitation is that collections may only contain *Primitive* types.

2.2.5 Control Flow

The runtime has native support for control flow operations like *branch*, *cons*, and *repeat*. However, it is syntactically challenging to express these contructs. The standard library uses structural types to provide support for *If*, *While*, *Return*, *Assert*, and *Rollback*. The standard library uses an implicitly available parsing *Context* to track modifications made to variables, references, and collections and to detect when any control flow statements are called.

3 Compiler

The standard library provides additional functionality that is deficient in the runtime to make it easier to construct programs. However, the standard library does not address the syntactic challenges of expressing programs. The compiler translates programs written in the statically-typed, object-oriented Caustic programming language into runtime-compatible programs. For example, consider the following example of a distributed counter written in Caustic. This program may be compiled into a Scala library that may be run without modification on any underlying transactional storage volume and distributed arbitrary without error. The Akka project also provides an implementation of a backend-agnostic distributed counter. Their implementation is almost seven times longer.

```
module caustic.example

/**
 * A count.
 *
 * @param value Current value.
 */
record Total {
  value: Int
}

/**
 * A distributed counter.
 */
service Counter {

 /**
  * Increments the total and returns its current value.
  *
  * @param x Reference to total.
  * @return Current value.
  */
  def increment(x: Total&): Int = {
    if (x.value) x.value += 1 else x.value = 1
    x.value
  }

}
```

3.1 Generation

The compiler uses ANTLR [9] to generate a predicated LL(*) parser from an ANTLR grammar file. The compiler generates code by walking the parse tree and replacing statements and declarations with their equivalent formulation using the standard library. The compiler mangles definitions so that they are lexically scoped. It also performs type inference and checking to determine and verify static types. The compiler is integrated into the uild system and provides a undle that implements syntax highlighting and code completion for most text editors and IDEs.

References

[1] Marcos K. Aguilera, Joshua B. Leners, and Michael Walfish. "Yesquel: Scalable Sql Storage for Web Applications". In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP '15. Monterey, California: ACM, 2015, pp. 245–262. ISBN: 978-1-4503-3834-9. DOI: 10.1145/2815400.2815413. URL: http://doi.acm.org/10.1145/2815400.2815413.

[2] Marcos K. Aguilera et al. "Sinfonia: A New Paradigm for Building Scalable Distributed Systems". In: *SIGOPS Oper. Syst. Rev.* 41.6 (Oct. 2007), pp. 159–174. ISSN: 0163-5980. DOI: 10.1145/1323293.1294278. URL: http://doi.acm.org/10.1145/1323293.1294278.

[3] C. J. Date. "Some Principles of Good Language Design: With Especial Reference to the Design of Database Languages". In: *SIGMOD Rec.* 14.3 (Nov. 1984), pp. 1–7. ISSN: 0163-5808. DOI: 10.1145/984549.984550. URL: http://doi.acm.org/10.1145/984549.984550.

[4] Jeff Dean. *Building Large-Scale Internet Services*. 2010. URL: http://static.googleusercontent.com/media/research.google.com/en//people/jeff/SOCC2010-keynote-slides.pdf (visited on 04/05/2018).

[5] C. Gray and D. Cheriton. “Leases: An Efficient Fault-tolerant Mechanism for Distributed File Cache Consistency”. In: *SIGOPS Oper. Syst. Rev.* 23.5 (Nov. 1989), pp. 202–210. ISSN: 0163-5980. DOI: 10.1145/74851.74870. URL: http://doi.acm.org/10.1145/74851.74870.

[6] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992. ISBN: 1558601902.

[7] Ralf Jung et al. “RustBelt: Securing the Foundations of the Rust Programming Language”. In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017), 66:1–66:34. ISSN: 2475-1421. DOI: 10.1145/3158154. URL: http://doi.acm.org/10.1145/3158154.

[8] Erik Meijer and Peter Drayton. “Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages”. In: *OOPSLA’04 Workshop on Revival of Dynamic Languages*. 2004.

[9] T. J. Parr and R. W. Quong. “ANTLR: A predicated-LL(K) Parser Generator”. In: *Softw. Pract. Exper.* 25.7 (July 1995), pp. 789–810. ISSN: 0038-0644. DOI: 10.1002/spe.4380250705. URL: http://dx.doi.org/10.1002/spe.4380250705.

[10] Christopher J. Rossbach et al. “Dandelion: A Compiler and Runtime for Heterogeneous Systems”. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP ’13. Farminton, Pennsylvania: ACM, 2013, pp. 49–68. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522715. URL: http://doi.acm.org/10.1145/2517349.2522715.

[11] Nir Shavit and Dan Touitou. “Software Transactional Memory”. In: *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC ’95. Ottowa, Ontario, Canada: ACM, 1995, pp. 204–213. ISBN: 0-89791-710-3. DOI: 10.1145/224964.224987. URL: http://doi.acm.org/10.1145/224964.224987.

Table 1: Runtime Expressions

Expression	Description
add(x, y)	Sum of x and y.
both(x, y)	Bitwise AND of x and y.
branch(c, p, f)	Executes p if c is true, or f otherwise.
cons(a, b)	Executes a and then b.
contains(x, y)	Returns whether or not x contains y.
cos(x)	Cosine of x.
div(x, y)	Quotient of x and y.
either(x, y)	Bitwise OR of x and y.
equal(x, y)	Returns whether x and y are equal.
floor(x)	Floor of x.
indexOf(x, y)	Returns the index of the first occurrence of y in x.
length(x)	Returns the number of characters in x.
less(x, y)	Returns whether x is strictly less than y.
load(n)	Loads the value of the variable n.
log(x)	Natural log of x.
matches(x, y)	Returns whether or not x matches the regex pattern y.
mod(x, y)	Remainder of x divided by y.
mul(x, y)	Product of x and y.
negate(x)	Bitwise negation of x.
pow(x, y)	Returns x raised to the power y.
prefetch(k, s)	Reads keys at k/i for i in $[0, s)$.
read(k)	Reads the value of the key k.
repeat(c, b)	Repeatedly executes b while c is true.
rollback(r)	Discards all writes and returns r.
sin(x)	Sine of x.
slice(x, l, h)	Returns the substring of x between $[l, h)$.
store(n, v)	Stores the value v for the variable n.
sub(x, y)	Difference of x and y.
write(k, v)	Writes the value v for the key k.

Table 2: Math Library

Function	Description
abs(x)	Absolute value of x.
acos(x)	Cosine inverse of x.
acot(x)	Cotangent inverse of x.
acsc(x)	Consecant inverse of x.
asec(x)	Secant inverse of x.
asin(x)	Sine inverse of x.
atan(x)	Tangent inverse of x.
ceil(x)	Smallest integer greater than or equal to x.
cos(x)	Cosine of x.
cosh(x)	Hyperbolic cosine of x.
cot(x)	Cotangent of x.
coth(x)	Hyperbolic cotangent of x.
csc(x)	Cosecant of x.
csch(x)	Hyperbolic cosecant of x.
exp(x)	Exponential of x.
floor(x)	Largest integer less than or equal to x.
log(x)	Natural logarithm of x.
log(x, y)	Log base y of x.
log10(x)	Log base 10 of x.
log2(x)	Log base 2 of x.
pow(x, y)	Power of x to the y.
random()	Uniformly random number on [0, 1).
round(x)	Closest integer to x.
round(x, y)	Closest multiple of y to x.
sec(x)	Secant of x.
sech(x)	Hyperbolic secant of x.
signum(x)	Sign of x.
sin(x)	Sine of x.
sinh(x)	Hyperbolic sine of x.
sqrt(x)	Square root of x.
tan(x)	Tangent of x.