High Level Concurrency in C∀

by

Colby Parsons

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2023

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Concurrent programs are notoriously hard to write and even harder to debug. Furthermore concurrent programs must be performant, as the introduction of concurrency into a program is often done to achieve some form of speedup.

This thesis presents a suite of high-level concurrent-language features in the new programming language CFA, all of which are implemented with the aim of improving the performance, productivity, and safety of concurrent programs. CFA is a non-object-oriented programming language that extends C. The foundation for concurrency in CFA was laid by Thierry Delisle [15], who implemented coroutines, user-level threads, and monitors. This thesis builds upon that work and introduces a suite of new concurrent features as its main contribution. The features include a **mutex** statement (similar to a C++ scoped lock or Java **synchronized** statement), Go-like channels and **select** statement, and an actor system. The root ideas behind these features are not new, but the CFA implementations extends the original ideas in performance, productivity, and safety.

Acknowledgements

To begin, I would like to thank my supervisor Professor Peter Buhr. Your guidance, wisdom and support has been invaluable in my learning and development of my research abilities, and in the implementation and writing of this thesis.

Thank you Thierry Delisle for your insight and knowledge regarding all things concurrency. You challenged my ideas and taught me skills that aid in both thinking about and writing concurrent programs.

Thanks to Michael Brooks, Andrew Beach, Fangren Yu, and Jiada Liang. Your work on CV continues to make it the best language it can be, and our discussions in meetings clarified the ideas that make up this thesis.

Finally, this work could not have happened without the financial support of David R. Cheriton School of Computer Science and the corporate partnership with Huawei Ltd.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

CAS *compare-and-set (swap)* 48, 49, 73

DCAS *double compare-and-set (swap)* 49, 74

DWCAS *double-wide (width) compare-and-set (swap)* 49

FCFS *first-come first-served* 23, 24

FIFO *first-in first-out* 23, 24, 34, 35, 42, 45

LIFO *last-in first-out* 23

LL *load linked* 49

QPCAS *queue pointer compare-and-set (swap)* 49, 50

RAII *resource acquisition is initialization* 15, 77

RTTI *run-time type information* 36

SC *store conditional* 49

TOCTOU *time-of-check to time-of-use* 26, 27, 72, 77

Chapter 1

Introduction

Concurrent programs are the wild west of programming because determinism and simple ordering of program operations go out the window. To seize the reins and write performant and safe concurrent code, high-level concurrent-language features are needed. Like any other craftsmen, programmers are only as good as their tools, and concurrent tooling and features are no exception.

This thesis presents a suite of high-level concurrent-language features implemented in the new programming-language C∀. These features aim to improve the performance of concurrent programs, aid in writing safe programs, and assist user productivity by improving the ease of concurrent programming. The groundwork for concurrent features in C∀ was designed and implemented by Thierry Delisle [15], who contributed the threading system, coroutines, monitors and other basic concurrency tools. This thesis builds on top of that foundation by providing a suite of high-level concurrent features. The features include a **mutex** statement, channels, a **waituntil** statement, and an actor system. All of these features exist in other programming languages in some shape or form; however, this thesis extends the original ideas by improving performance, productivity, and safety.

1.1 The Need For Concurrent Features

Concurrent programming has many unique pitfalls that do not appear in sequential programming:

1. Race conditions, where thread orderings can result in arbitrary behaviours, resulting in correctness problems.
2. Livelock, where threads constantly attempt a concurrent operation unsuccessfully, resulting in no progress being made.
3. Starvation, where *some* threads constantly attempt a concurrent operation unsuccessfully, resulting in partial progress being made.
4. Deadlock, where some threads wait for an event that cannot occur, blocking them indefinitely, resulting in no progress being made.

Even with the guiding hand of concurrent tools these pitfalls still catch unwary programmers, but good language support helps significantly to prevent, detect, and mitigate these problems.

1.2 Thesis Overview

Chapter 2 of this thesis aims to familiarize the reader with the language C∀. In this chapter, syntax and features of the C∀ language that appear in this work are discussed. Chapter 3 briefly discusses prior concurrency work in C∀, and how the work in this thesis builds on top of the existing framework. Each remaining chapter introduces an additional C∀ concurrent-language feature, which includes discussing prior related work for the feature, extensions over prior features, and uses benchmarks to compare the performance the feature with corresponding or similar features in other languages and systems.

Chapter 4 discusses the **mutex** statement, a language feature that provides safe and simple lock usage. The **mutex** statement is compared both in terms of safety and performance with similar mechanisms in C++ and Java. Chapter 5 discusses channels, a message passing concurrency primitive that provides for safe synchronous and asynchronous communication among threads. Channels in C∀ are compared to Go's channels, which popularized the use of channels in modern concurrent programs. Chapter 6 discusses the C∀ actor system. An actor system is a close cousin of channels, as it also belongs to the message passing paradigm of concurrency. However, an actor system provides a greater degree of abstraction and ease of scalability, making it useful for a different range of problems than channels. The actor system in C∀ is compared with a variety of other systems on a suite of benchmarks. Chapter 7 discusses the C∀ **waituntil** statement, which provides the ability to synchronize while waiting for a resource, such as acquiring a lock, accessing a future, or writing to a channel. The C∀ **waituntil** statement provides greater flexibility and expressibility than similar features in other languages. All in all, the features presented aim to fill in gaps in the current C∀ concurrent-language support, enabling users to write a wider range of complex concurrent programs with ease.

1.3 Contributions

This work presents the following contributions within each of the additional language features:

1. The **mutex** statement that:
   - provides deadlock-free multiple lock acquisition,
   - clearly denotes lock acquisition and release,
   - and has good performance irrespective of lock ordering.
2. The channel that:
   - achieves comparable performance to Go, the gold standard for concurrent channels,
   - has deadlock detection,
   - introduces easy-to-use exception-based close semantics,
   - and provides toggle-able statistics for performance tuning.
3. The in-memory actor system that:
   - achieves the lowest latency message send of all tested systems,
   - is the first inverted actor system to introduce queue stealing,
   - attains zero-victim-cost stealing through a carefully constructed stealing mechanism,

- gains performance through static-typed message sends, eliminating the need for dynamic dispatch,
- introduces the copy queue, an array-based queue specialized for the actor use-case to minimize calls to the memory allocator,
- has robust detection of six tricky, but common actor programming errors,
- achieves very good performance on a diverse benchmark suite compared to other actor systems,
- and provides toggle-able statistics for performance tuning.

4. The **waituntil** statement that:
   - is the only known polymorphic synchronous multiplexing language feature,
   - provides greater expressibility for waiting conditions than other languages,
   - and achieves comparable performance to similar features in two other languages.

Chapter 2

Introduction to C∀

2.1 Overview

The following serves as an introduction to C∀. C∀ is a layer over C, is transpiled[1] to C, and is largely considered to be an extension of C. Beyond C, it adds productivity features, extended libraries, an advanced type-system, and many control-flow/concurrency constructions. However, C∀ stays true to the C programming style, with most code revolving around **struct**s and routines, and respects the same rules as C. C∀ is not object oriented as it has no notion of this (receiver) and no structures with methods, but supports some object oriented ideas including constructors, destructors, and limited nominal inheritance. While C∀ is rich with interesting features, only the subset pertinent to this work is discussed here.

2.2 References

References in C∀ are similar to references in C++; however C∀ references are *rebindable*, and support multi-level referencing. References in C∀ are a layer of syntactic sugar over pointers to reduce the number of syntactic ref/deref operations needed with pointer usage. Pointers in C∀ differ from C and C++ in their use of 0p instead of C's NULL or C++'s nullptr. References can contain 0p in C∀, which is the equivalent of a null reference. Examples of references are shown in Listing 2.1.

Listing 2.1: Example of C∀ references

```
int i = 2;
int & ref_i = i;                    // declare ref to i
int * ptr_i = &i;                   // ptr to i

// address of ref_i is the same as address of i
assert( &ref_i == ptr_i );

int && ref_ref_i = ref_i;           // can have a ref to a ref
ref_i = 3;                          // set i to 3
int new_i = 4;
```

[1] Source to source translator.

```
// syntax to rebind ref_i (must cancel implicit deref)
&ref_i = &new_i;                        // (&*)ref_i = &new_i; (sets underlying ptr)
```

2.3 Overloading

C∀ routines can be overloaded on parameter type, number of parameters, and *return type*. Variables can also be overloaded on type, meaning that two variables can have the same name so long as they have different types. A routine or variable is disambiguated at each usage site via its type and surrounding expression context. A cast is used to disambiguate any conflicting usage. Examples of overloading are shown in Listing 2.2.

Listing 2.2: Example of C∀ overloading

```
int foo() { sout | "A"; return 0;}
int foo( int bar ) { sout | "B"; return 1; }
int foo( double bar ) { sout | "C"; return 2; }
double foo( double bar ) { sout | "D"; return 3; }
void foo( double bar ) { sout | bar; }

int main() {
    foo();                              // prints A
    foo( 0 );                           // prints B
    int foo = foo( 0.0 );               // prints C
    double foo = foo( 0.0 );            // prints D
    foo( foo );                         // prints 3., where left-hand side of expression is void
}
```

2.4 with Statement

The C∀ **with** statement is for exposing fields of an aggregate type within a scope, allowing field names without qualification. This feature is also implemented in Pascal [33]. It can exist as a stand-alone statement or wrap a routine body to expose aggregate fields. If exposed fields share a name, the type system will attempt to disambiguate them based on type. If the type system is unable to disambiguate the fields then the user must qualify those names to avoid a compilation error. Examples of the **with** statement are shown in Listing 2.3.

Listing 2.3: Example of C∀ **with** statement

```
struct pair {  double x, y;  };
struct triple {  int a, b, c;  };
pair p;

with( p ) {                             // stand-alone with
    p.x = 6.28;  p.y = 1.73;            // long form
      x = 6.28;     y = 1.73;           // short form
}
void foo( triple t, pair p ) with( t, p ) {     // routine with
```

```
    t.a = 1;  t.b = 2;  t.c = 3;  p.x = 3.14;  p.y = 2.71;  // long form
     a = 1;    b = 2;    c = 3;    x = 3.14;    y = 2.71;  // short form
}
```

2.5 Operators

Operators can be overloaded in C∀ with operator routines. Operators in C∀ are named using an operator symbol and '?' to represent operands. Examples of C∀ operators are shown in Listing 2.4.

Listing 2.4: Example of C∀ operators

```
struct coord {
    double x, y, z;
};
coord ++?( coord & c ) with( c ) {          // post increment
    x++;  y++;  z++;
    return c;
}
coord ?<=?( coord op1, coord op2 ) with( op1 ) { // ambiguous with both parameters
    return (x * x + y * y + z * z) <= (op2.x * op2.x + op2.y * op2.y + op2.z * op2.z);
}
```

The operator '|' is used for C∀ stream I/O, similar to how C++, uses operators '<<' and '>>' Listing 2.5.

Listing 2.5: Example of C∀ stream I/O

```
char c;  int i;  double d;
sin  | c | i | d;                    // read into c, i, and d
sout | c | i | d;                    // output c, i, and d
x 27 2.3                             // implicit separation between values and auto newline
```

2.6 Constructors and Destructors

Constructors and destructors in C∀ are special operator routines used for creation and destruction of objects. The default constructor and destructor for a type are called implicitly upon creation and deletion, respectively. Examples of C∀ constructors and destructors are shown in Listing 2.6.

Listing 2.6: Example of C∀ constructors and destructors

```
struct discrete_point {
    int x, y;
};
void ?{}( discrete_point & this ) with(this) { // default constructor
    [x, y] = 0;
}
void ?{}( discrete_point & this, int x, int y ) { // explicit constructor
    this.[x, y] = [x, y];
}
```

```
void ^?{}( discrete_point & this ) with(this) { // destructor
    ?{}( this );                        // reset by calling default constructor
}
int main() {
    discrete_point x, y{};              // implicit call to default ctor, ?{}
    discrete_point s = { 2, -4 }, t{ 4, 2 };  // explicit call to specialized ctor
} // ^t{}, ^s{}, ^y{}, ^x{} implicit calls in reverse allocation order
```

2.7 Polymorphism

C supports limited polymorphism, often requiring users to implement polymorphism using a **void** * (explicit type erasure) approach. C∀ extends C with generalized overloading polymorphism (see Section 2.3, p. 5), as well as parametric polymorphism and limited inclusion polymorphism (nominal inheritance).

2.7.1 Parametric Polymorphism

C∀ provides parametric polymorphism in the form of **forall**, and **trait**s. A **forall** takes in a set of types and a list of constraints. The declarations that follow the **forall** are parameterized over the types listed that satisfy the constraints. A list of **forall** constraints can be refactored into a named **trait** and reused in **forall**s. Examples of C∀ parametric polymorphism are shown in Listing 2.7.

Listing 2.7: Example of C∀ parametric polymorphism

```
// sized() is a trait that means the type has a size
forall( V & | sized(V) )                // type params for trait
trait vector_space {
    // dtor and copy ctor needed in constraints to pass by copy
    void ?{}( V &, V & );               // copy ctor for return
    void ^?{}( V & );                   // dtor
    V ?+?( V, V );                      // vector addition
    V ?*?( int, V );                    // scalar multiplication
};

forall( V & | vector_space( V ) ) {
    V get_inverse( V v1 ) {
        return -1 * v1;                 // can use ?*? routine defined in trait
    }
    V add_and_invert( V v1, V v2 ) {
        return get_inverse( v1 + v2 );  // can use ?+? routine defined in trait
    }
}
struct Vec1 { int x; };
void ?{}( Vec1 & this, Vec1 & other ) { this.x = other.x; }
void ?{}( Vec1 & this, int x ) { this.x = x; }
void ^?{}( Vec1 & this ) {}
Vec1 ?+?( Vec1 v1, Vec1 v2 ) { v1.x += v2.x; return v1; }
Vec1 ?*?( int c, Vec1 v1 ) { v1.x = v1.x * c; return v1; }
```

```
struct Vec2 { int x; int y; };
void ?{}( Vec2 & this, Vec2 & other ) { this.x = other.x; this.y = other.y; }
void ?{}( Vec2 & this, int x ) { this.x = x; this.y = x; }
void ^?{}( Vec2 & this ) {}
Vec2 ?+?( Vec2 v1, Vec2 v2 ) { v1.x += v2.x; v1.y += v2.y; return v1; }
Vec2 ?*?( int c, Vec2 v1 ) { v1.x = v1.x * c; v1.y = v1.y * c; return v1; }

int main() {
    Vec1 v1{ 1 };                       // create Vec1 and call ctor
    Vec2 v2{ 2 };                       // create Vec2 and call ctor
    // can use forall defined routines since types satisfy trait
    add_and_invert( get_inverse( v1 ), v1 );
    add_and_invert( get_inverse( v2 ), v2 );
}
```

2.7.2 Inheritance

Inheritance in C∀ is taken from Plan-9 C's nominal inheritance. In C∀, **struct**s can **inline** another struct type to gain its fields and masquerade as that type. Examples of C∀ nominal inheritance are shown in Listing 2.8.

Listing 2.8: Example of C∀ nominal inheritance

```
struct one_d { double x; };
struct two_d {
    inline one_d;
    double y;
};
struct three_d {
    inline two_d;
    double z;
};
double get_x( one_d & d ){ return d.x; }

struct dog {};
struct dog_food {
    int count;
};
struct pet {
    inline dog;
    inline dog_food;
};
void pet_dog( dog & d ) { sout | "woof"; }
void print_food( dog_food & f ) { sout | f.count; }

int main() {
    one_d x;
    two_d y;
    three_d z;
```

```
    x.x = 1;
    y.x = 2;
    z.x = 3;
    get_x( x );                    // returns 1;
    get_x( y );                    // returns 2;
    get_x( z );                    // returns 3;
    pet p;
    p.count = 5;
    pet_dog( p );                  // prints woof
    print_food( p );               // prints 5
}
```

Chapter 3

Concurrency in C∀

The groundwork for concurrency in C∀ was laid by Thierry Delisle in his Master's Thesis [15]. In that work, he introduced generators, coroutines, monitors, and user-level threading. Not listed in that work were basic concurrency features needed as building blocks, such as locks, futures, and condition variables.

3.1 Threading Model

C∀ provides user-level threading and supports an M:N threading model where M user threads are scheduled on N kernel threads and both M and N can be explicitly set by the programmer. Kernel threads are created by declaring processor objects; user threads are created by declaring a thread objects. Listing 3.1 shows a typical examples of creating a C∀ user-thread type, and then as declaring processor (N) and thread objects (M).

Listing 3.1: Example of C∀ user thread and processor creation

```
thread my_thread {                      // user thread type (like structure)
    ... // arbitrary number of field declarations
};
void main( my_thread & this ) {         // thread start routine
    sout | "Hello threading world";
}
int main() {                            // program starts with a processor (kernel thread)
    processor p[2];                     // add 2 processors = 3 total with starting processor
    {
        my_thread t[2], * t3 = new();   // create 2 stack allocated, 1 dynamic allocated user threads
        ... // execute concurrently
        delete( t3 );                   // wait for t3 to end and deallocate
    } // wait for threads t[0] and t[1] to end and deallocate
} // deallocate additional kernel threads
```

A thread type is are defined using the aggregate kind **thread**. For each thread type, a corresponding main routine must be defined, which is where the thread starts running once when a thread object are is created. The processor declaration adds addition kernel threads alongside the existing processor given to each program. Thus, for N processors, allocate $N - 1$ processors. A thread is implicitly joined at deallocation, either implicitly at block exit for stack allocation or

explicitly at delete for heap allocation. The thread performing the deallocation must wait for the thread to terminate before the deallocation can occur. A thread terminates by returning from the main routine where it starts.

3.2 Existing and New Concurrency Features

C∀ currently provides a suite of concurrency features including futures, locks, condition variables, generators, coroutines, monitors. Examples of these features are omitted as most of them are the same as their counterparts in other languages. It is worthwhile to note that all concurrency features added to C∀ are made to be compatible each other. The laundry list of features above and the ones introduced in this thesis can be used in the same program without issue, and the features are designed to interact in meaningful ways. For example, a thread can inteact with a monitor, which can interact with a coroutine, which can interact with a generator.

Solving concurrent problems requires a diverse toolkit. This work aims to flesh out C∀'s concurrent toolkit to fill in gaps. Futures are used when a one-shot result needs to be safely delivered concurrently, and are especially useful when the receiver needs to block until the result is ready. When multiple values have to be sent, or more synchronization is needed, futures are not powerful enough, which introduces the need for channels. A close cousin of channels is actor systems, which take message passing a step further and go beyond channels to provide a level of abstraction that allows for easy scalability and separation of concerns. The **waituntil** and **mutex** statements provide utilities allowing for easier use of the existing features. All the contributions of this thesis provide the ability to solve concurrent problems that formerly would require a user to either implement a similar feature themselves or construct an ad-hoc solution.

Chapter 4

Mutex Statement

The mutual exclusion problem was introduced by Dijkstra in 1965 [19, 20]: there are several concurrent processes or threads that communicate by shared variables and from time to time need exclusive access to shared resources. A shared resource and code manipulating it form a pairing called a *critical section (CS)*, which is a many-to-one relationship; *e.g.*, if multiple files are being written to by multiple threads, only the pairings of simultaneous writes to the same files are CSs. Regions of code where the thread is not interested in the resource are combined into the *non-critical section (NCS)*.

Exclusive access to a resource is provided by *mutual exclusion (MX)*. MX is implemented by some form of *lock*, where the CS is bracketed by lock procedures acquire and release. Threads execute a loop of the form:

```
loop of thread p:
    NCS;
    acquire( lock );  CS;  release( lock ); // protected critical section with MX
end loop.
```

MX guarantees there is never more than one thread in the CS. MX must also guarantee eventual progress: when there are competing threads attempting access, eventually some competing thread succeeds, *i.e.*, acquires the CS, releases it, and returns to the NCS. A stronger constraint is that every thread that calls acquire eventually succeeds after some reasonable bounded time.

4.1 Monitor

C∀ provides a high-level locking object, called a *monitor*, an elegant and efficient abstraction for providing mutual exclusion and synchronization for shared-memory systems. First proposed by Brinch Hansen [5] and later described and extended by C.A.R. Hoare [27]. Several concurrent programming languages provide monitors as an explicit language construct: *e.g.*, Concurrent Pascal [6], Mesa [51], Turing [30], Modula-3 [4], μC++ [9] and Java [23]. In addition, operating-system kernels and device drivers have a monitor-like structure, although they often use lower-level primitives such as mutex locks or semaphores to manually implement a monitor.

Figure 4.1 shows a C∀ and Java monitor implementing an atomic counter. A *monitor* is a programming technique that implicitly binds mutual exclusion to static function scope by call and return. In contrast, lock mutual exclusion, defined by acquire/release calls, is independent

```
monitor Aint {
    int cnt;
};
int ++?( Aint & mutex m ) { return ++m.cnt; }
int ?=?( Aint & mutex l, int r ) { l.cnt = r; }
int ?=?(int & l, Aint & r) { l = r.cnt; }

int i = 0, j = 0;
Aint x = { 0 }, y = { 0 };    // no mutex
++x;  ++y;                    // mutex
x = 2;  y = i;                // mutex
i = x;  j = y;                // no mutex
```

(a) C∀

```
class Aint {
    private int cnt;
    public Aint( int init ) { cnt = init; }
    synchronized public int inc() { return ++cnt; }
    synchronized public void set( int r ) {cnt = r;}
    public int get() { return cnt; }
}
int i = 0, j = 0;
Aint x = new Aint( 0 ), y = new Aint( 0 );
x.inc();  y.inc();
x.set( 2 );  y.set( i );
i = x.get();  j = y.get();
```

(b) Java

Figure 4.1: Atomic integer counter

of lexical context (analogous to stack versus heap storage allocation). Restricting acquire and release points in a monitor eases programming, comprehension, and maintenance, at a slight cost in flexibility and efficiency. Ultimately, a monitor is implemented using a combination of basic locks and atomic instructions.

Like Java, C∀ monitors have *multi-acquire* (reentrant locking) semantics so the thread in the monitor may acquire it multiple times without deadlock, allowing recursion and calling of other MX functions. For robustness, C∀ monitors ensure the monitor lock is released regardless of how an acquiring function ends, normal or exceptional, and returning a shared variable is safe via copying before the lock is released. Monitor objects can be passed through multiple helper functions without acquiring mutual exclusion, until a designated function associated with the object is called. C∀ functions are designated MX by one or more pointer/reference parameters having qualifier **mutex**. Java members are designated MX with **synchronized**, which applies only to the implicit receiver parameter. In the example above, the increment and setter operations need mutual exclusion, while the read-only getter operation is not MX because reading an integer is atomic.

As stated, the non-object-oriented nature of C∀ monitors allows a function to acquire multiple mutex objects. For example, the bank-transfer problem requires locking two bank accounts to safely debit and credit money between accounts.

```
monitor BankAccount {
    int balance;
};
void deposit( BankAccount & mutex b, int deposit ) with( b ) {
    balance += deposit;
}
void transfer( BankAccount & mutex my, BankAccount & mutex your, int me2you ) {
    deposit( my, -me2you );                    // debit
    deposit( your, me2you );                   // credit
}
```

The C∀ monitor implementation ensures multi-lock acquisition is done in a deadlock-free manner

```
monitor RWlock { ... };
void read( RWlock & rw, ... ) {
    void StartRead( RWlock & mutex rw ) { ... }
    void EndRead( RWlock & mutex rw ) { ... }
    StartRead( rw );
    ... // read without MX
    EndRead( rw );
}
void write( RWlock & mutex rw, ... ) {
    ... // write with MX
}
```

(a) monitor

```
void read( RWlock & rw, ... ) {


    mutex( rw ) { ... }
    ... // read without MX
    mutex{ rw ) { ... }
}
void write( RWlock & mutex rw, ... ) {
    ... // write with MX
}
```

(b) mutex statement

Figure 4.2: Readers writer problem

regardless of the number of MX parameters and monitor arguments via resource ordering. It it important to note that C∀ monitors do not attempt to solve the nested monitor problem [45], as such nested **mutex** statements lose their deadlock-free guarantee.

4.2 mutex statement

Restricting implicit lock acquisition to function entry and exit can be awkward for certain problems. To increase locking flexibility, some languages introduce a mutex statement. Figure 4.2 shows the outline of a reader/writer lock written as a C∀ monitor and mutex statements. (The exact lock implementation is irrelevant.) The read and write functions are called with a reader/writer lock and any arguments to perform reading or writing. The read function is not MX because multiple readers can read simultaneously. MX is acquired within read by calling the (nested) helper functions StartRead and EndRead or executing the mutex statements. Between the calls or statements, reads can execute simultaneous within the body of read. The write function does not require refactoring because writing is a CS. The mutex-statement version is better because it has fewer names, less argument/parameter passing, and can possibly hold MX for a shorter duration.

This work adds a mutex statement to C∀, but generalizes it beyond implicit monitor locks. In detail, the mutex statement has a clause and statement block, similar to a conditional or loop statement. The clause accepts any number of lockable objects (like a C∀ MX function prototype), and locks them for the duration of the statement. The locks are acquired in a deadlock free manner and released regardless of how control-flow exits the statement. The mutex statement provides easy lock usage in the common case of lexically wrapping a CS. Examples of C∀ mutex statement are shown in Listing 4.1.

Listing 4.1: C∀ mutex statement usage

```
owner_lock lock1, lock2, lock3;
mutex( lock2, lock3 ) ...;                  // inline statement
mutex( lock1, lock2, lock3 ) { ... }        // statement block
void transfer( BankAccount & my, BankAccount & your, int me2you ) {
```

```
    ... // check values, no MX
    mutex( my, your ) { // MX is shorter duration that function body
        deposit( my, -me2you );          // debit
        deposit( your, me2you );         // credit
    }
}
```

4.3 Other Languages

There are similar constructs to the mutex statement in other programming languages. Java has a feature called a synchronized statement, which looks like the C∀'s mutex statement, but only accepts a single object in the clause and only handles monitor locks. The C++ standard library has a scoped_lock, which is also similar to the mutex statement. The scoped_lock takes any number of locks in its constructor, and acquires them in a deadlock-free manner. It then releases them when the scoped_lock object is deallocated using *resource acquisition is initialization* (RAII). An example of C++ scoped_lock is shown in Listing 4.2.

Listing 4.2: C++ scoped_lock usage

```
struct BankAccount {
    recursive_mutex m;                   // must be recursive
    int balance = 0;
};
void deposit( BankAccount & b, int deposit ) {
    scoped_lock lock( b.m );             // RAII acquire
    b.balance += deposit;
}                                        // RAII release
void transfer( BankAccount & my, BankAccount & your, int me2you ) {
    scoped_lock lock( my.m, your.m );    // RAII acquire
    deposit( my, -me2you );              // debit
    deposit( your, me2you );             // credit
}                                        // RAII release
```

4.4 C∀ implementation

The C∀ mutex statement takes some ideas from both the Java and C++ features. Like Java, C∀ introduces a new statement rather than building from existing language features, although C∀ has sufficient language features to mimic C++ RAII locking. This syntactic choice makes MX explicit rather than implicit via object declarations. Hence, it is easy for programmers and language tools to identify MX points in a program, *e.g.*, scan for all **mutex** parameters and statements in a body of code, similar to Java's synchronized. Furthermore, concurrent safety is provided across an entire program for the complex operation of acquiring multiple locks in a deadlock-free manner. Unlike Java, C∀'s mutex statement and C++'s scoped_lock both use parametric polymorphism to allow user defined types to work with this feature. In this case, the polymorphism allows a locking mechanism to acquire MX over an object without having to know the object internals or what kind of lock it is using. C∀'s provides and uses this locking trait:

```
forall( L & | sized(L) )
trait is_lock {
    void lock( L & );
    void unlock( L & );
};
```

C++ scoped_lock has this trait implicitly based on functions accessed in a template. scoped_lock also requires try_lock because of its technique for deadlock avoidance (see Section 4.5).

The following shows how the **mutex** statement is used with C∀ streams to eliminate unpredictable results when printing in a concurrent program. For example, if two threads execute:

```
thread1 : sout | "abc" | "def";
thread2 : sout | "uvw" | "xyz";
```

any of the outputs can appear:

```
abc def | abc uvw xyz | uvw abc xyz def | abuvwc dexf | uvw abc def
uvw xyz | def         |                  | yz          | xyz
```

The stream type for sout is defined to satisfy the is_lock trait, so the **mutex** statement can be used to lock an output stream while producing output. From the programmer's perspective, it is sufficient to know an object can be locked and then any necessary MX is easily available via the **mutex** statement. This ability improves safety and programmer productivity since it abstracts away the concurrent details. Hence, a programmer can easily protect cascaded I/O expressions:

```
thread1 : mutex( sout ) sout | "abc" | "def";
thread2 : mutex( sout ) sout | "uvw" | "xyz";
```

constraining the output to two different lines in either order:

```
abc def | uvw xyz
uvw xyz | abc def
```

where this level of safe nondeterministic output is acceptable. Alternatively, multiple I/O statements can be protected using the mutex statement block:

```
mutex( sout ) {    // acquire stream lock for sout for block duration
    sout | "abc";
    sout | "uvw" | "xyz";
    sout | "def";
} // implicitly release sout lock
```

4.5 Deadlock Avoidance

The mutex statement uses the deadlock avoidance technique of lock ordering, where the circular-wait condition of a deadlock cannot occur if all locks are acquired in the same order. The scoped_lock uses a deadlock avoidance algorithm where all locks after the first are acquired using try_lock and if any of the lock attempts fail, all acquired locks are released. This repeats after selecting a new starting point in a cyclic manner until all locks are acquired successfully. This deadlock avoidance algorithm is shown in Figure 4.3. The algorithm is taken directly from the source code of the <**mutex**> header, with some renaming and comments for clarity.

```
int first = 0;  // first lock to attempt to lock
do {
    // locks is the array of locks to acquire
    locks[first].lock();                        // lock first lock
    for ( int i = 1; i < Num_Locks; i += 1 ) {  // iterate over rest of locks
        const int idx = (first + i) % Num_Locks;
        if ( ! locks[idx].try_lock() ) {        // try lock each one
            for ( int j = i; j != 0; j -= 1 )   // release all locks
                locks[(first + j - 1) % Num_Locks].unlock();
            first = idx;                        // rotate which lock to acquire first
            break;
        }
    }
// if first lock is still held then all have been acquired
} while ( ! locks[first].owns_lock() );         // is first lock held?
```

Figure 4.3: C++ scoped_lock deadlock avoidance algorithm

While this algorithm successfully avoids deadlock, there is a livelock scenario. Assume two threads, A and B, create a scoped_lock accessing two locks, $L1$ and $L2$. A livelock can form as follows. Thread A creates a scoped_lock with arguments $L1$, $L2$, and B creates a scoped lock with the lock arguments in the opposite order $L2$, $L1$. Both threads acquire the first lock in their order and then fail the try_lock since the other lock is held. Both threads then reset their starting lock to be their second lock and try again. This time A has order $L2$, $L1$, and B has order $L1$, $L2$, which is identical to the starting setup but with the ordering swapped between threads. If the threads perform this action in lock-step, they cycle indefinitely without entering the CS, *i.e.*, livelock. Hence, to use scoped_lock safely, a programmer must manually construct and maintain a global ordering of lock arguments passed to scoped_lock.

The lock ordering algorithm used in C∀ mutex functions and statements is deadlock and livelock free. The algorithm uses the lock memory addresses as keys, sorts the keys, and then acquires the locks in sorted order. For fewer than 7 locks ($2^3 - 1$), the sort is unrolled performing the minimum number of compare and swaps for the given number of locks; for 7 or more locks, insertion sort is used. It is assumed to be rare to hold more than 6 locks at a time. For 6 or fewer locks the algorithm is fast and executes in $O(1)$ time. Furthermore, lock addresses are unique across program execution, even for dynamically allocated locks, so the algorithm is safe across the entire program execution, as long as lifetimes of objects are appropriately managed. For example, deleting a lock and allocating another one could give the new lock the same address as the deleted one, however deleting a lock in use by another thread is a programming error irrespective of the usage of the **mutex** statement.

The downside to the sorting approach is that it is not fully compatible with manual usages of the same locks outside the **mutex** statement, *i.e.*, the lock are acquired without using the **mutex** statement. The following scenario is a classic deadlock.

```
lock L1, L2; // assume &L1 < &L2
      thread1                          thread2
acquire( L2 );                    mutex( L1, L2 ) {
    acquire( L1 );
       CS                              CS
    release( L1 );
release( L2 );                    }
```

Comparatively, if the scoped_lock is used and the same locks are acquired elsewhere, there is no concern of the scoped_lock deadlocking, due to its avoidance scheme, but it may livelock. The convenience and safety of the **mutex** statement, *i.e.*, guaranteed lock release with exceptions, should encourage programmers to always use it for locking, mitigating most deadlock scenarios versus combining manual locking with the mutex statement. Both C++ and the C∀ do not provide any deadlock guarantees for nested scoped_locks or **mutex** statements. To do so would require solving the nested monitor problem [45], which currently does not have any practical solutions.

4.6 Performance

Given the two multi-acquisition algorithms in C++ and C∀, each with differing advantages and disadvantages, it interesting to compare their performance. Comparison with Java was not conducted, since the synchronized statement only takes a single object and does not provide deadlock avoidance or prevention.

The comparison starts with a baseline that acquires the locks directly without a mutex statement or scoped_lock in a fixed ordering and then releases them. The baseline helps highlight the cost of the deadlock avoidance/prevention algorithms for each implementation.

The benchmark used to evaluate the avoidance algorithms repeatedly acquires a fixed number of locks in a random order and then releases them. The pseudocode for the deadlock avoidance benchmark is shown in Figure 4.4. To ensure the comparison exercises the implementation of each lock avoidance algorithm, an identical spinlock is implemented in each language using a set of builtin atomics available in both C++ and C∀. The benchmarks are run for a fixed duration of 10 seconds and then terminate. The total number of times the group of locks is acquired is returned for each thread. Each variation is run 11 times on 2, 4, 8, 16, 24, 32 cores and with 2, 4, and 8 locks being acquired. The median is calculated and is plotted alongside the 95% confidence intervals for each point. The confidence intervals are calculated using bootstrapping to avoid normality assumptions.

The performance experiments were run on the following multi-core hardware systems to determine differences across platforms:

1. Supermicro AS–1123US–TR4 AMD EPYC 7662 64–core socket, hyper-threading $\times$ 2 sockets (256 processing units), TSO memory model, running Linux v5.8.0–55–generic, gcc–10 compiler
2. Supermicro SYS–6029U–TR4 Intel Xeon Gold 5220R 24–core socket, hyper-threading $\times$ 2 sockets (96 processing units), TSO memory model, running Linux v5.8.0–59–generic, gcc–10 compiler

```
size_t n_locks;                          // number of locks
size_t n_thds;                           // number of threads
size_t n_gens;                           // number of random orderings (default 100)
size_t total = 0;                        // global throughput aggregator
volatile bool done = false;              // termination flag

test_spinlock locks[n_locks];
size_t rands[n_thds][n_locks * n_gens];  // random ordering per thread

void main( worker & w ) with(w) {        // thread main
    size_t count = 0, idx = 0;
    while ( !done ) {
        idx = (count % n_locks * n_gens) * n_locks; // get start of next sequence
        mutex(locks[rands[0]], ..., locks[rands[n_locks - 1]]){} // lock sequence of locks
        count++;
    }
    __atomic_add_fetch(&total, count, __ATOMIC_SEQ_CST); // atomically add to total
}

int main( int argc, char * argv[] ) {
    gen_orders();                        // generate random orderings
    {
        worker w[n_thds];
        sleep( 10`s );
        done = true;
    }
    printf( "%lu\n", total );
}
```

Figure 4.4: Deadlock avoidance benchmark C∀ pseudocode

Figure 4.5 shows the results of the benchmark experiments. The baseline results for both languages are mostly comparable, except for the 8 locks results in 4.5(e) and 4.5(f), where the C∀ baseline is slightly slower. The avoidance result for both languages is significantly different, where C∀'s mutex statement achieves throughput that is magnitudes higher than C++'s scoped_lock. The slowdown for scoped_lock is likely due to its deadlock-avoidance implementation. Since it uses a retry based mechanism, it can take a long time for threads to progress. Additionally the potential for livelock in the algorithm can result in very little throughput under high contention. For example, on the AMD machine with 32 threads and 8 locks, the benchmarks would occasionally livelock indefinitely, with no threads making any progress for 3 hours before the experiment was terminated manually. It is likely that shorter bouts of livelock occurred in many of the experiments, which would explain large confidence intervals for some of the data points in the C++ data. In Figures 4.5(e) and 4.5(f) there is the counter-intuitive result of the **mutex** statement performing better than the baseline. At 7 locks and above the mutex statement switches from a hard coded sort to insertion sort, which should decrease performance. The hard coded sort is branch-free and constant-time and was verified to be faster than insertion sort for 6 or fewer locks. Part of the difference in throughput compared to baseline is due to the delay spent

in the insertion sort, which decreases contention on the locks. This was verified to be part of the difference in throughput by experimenting with varying NCS delays in the baseline; however it only comprises a small portion of difference. It is possible that the baseline is slowed down or the **mutex** is sped up by other factors that are not easily identifiable.

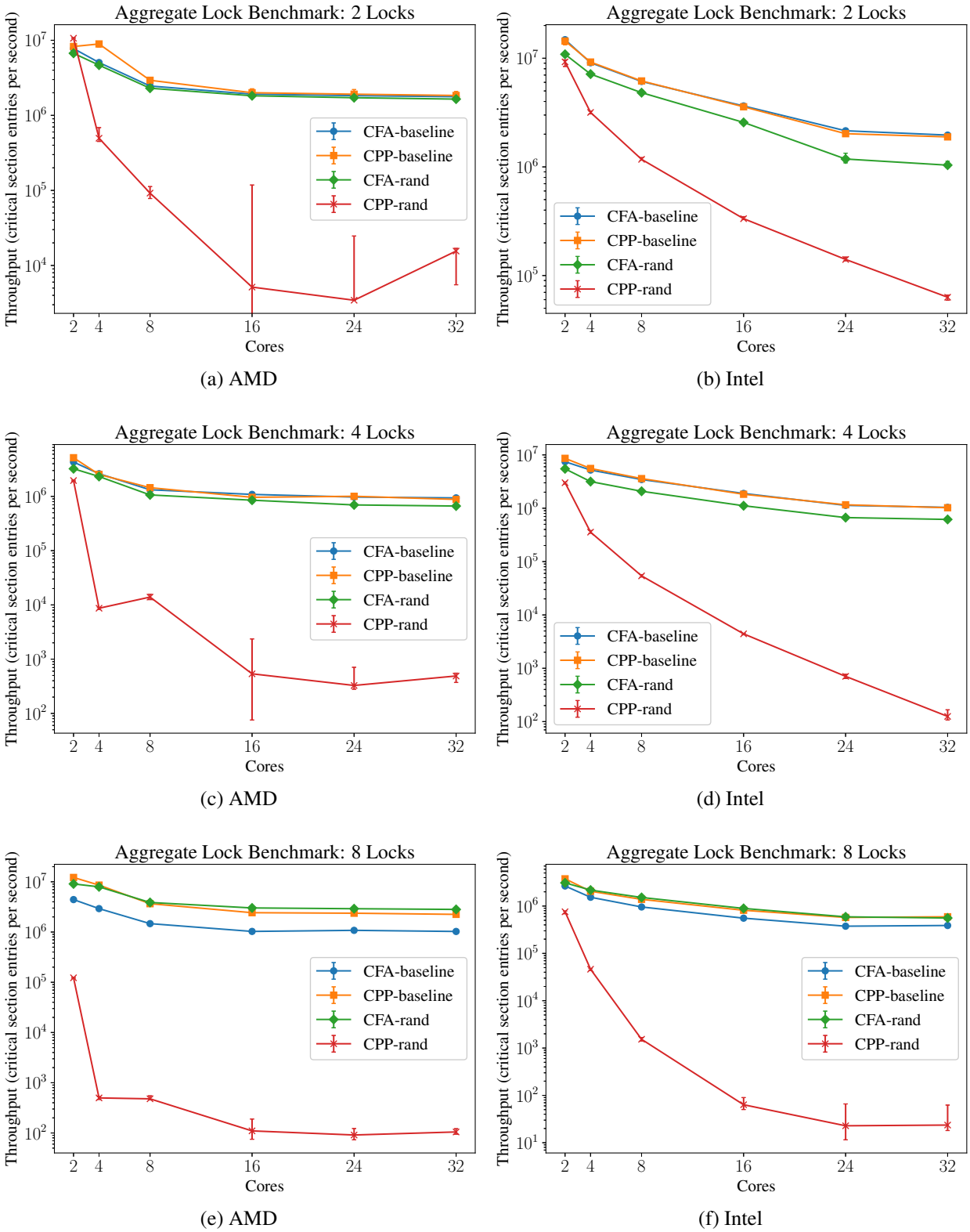


Figure 4.5: The aggregate lock benchmark comparing C++ scoped_lock and C∀ mutex statement throughput (higher is better).

Chapter 5

Channels

Most modern concurrent programming languages do not subscribe to just one style of communication among threads and provide features that support multiple approaches. Channels are a concurrent-language feature used to perform *message-passing concurrency*: a model of concurrency where threads communicate by sending data as messages (mostly nonblocking) and synchronizing by receiving sent messages (blocking). This model is an alternative to shared-memory concurrency, where threads communicate directly by changing shared state.

Channels were first introduced by Kahn [34] and extended by Hoare [28] (CSP). Both papers present a pseudo (unimplemented) concurrent language where processes communicate using input/output channels to send data. Both languages are highly restrictive. Kahn's language restricts a reading process to only wait for data on a single channel at a time and different writing processes cannot send data on the same channel. Hoare's language restricts both the sender and receiver to explicitly name the process that is the destination of a channel send or the source of a channel receive. These channel semantics remove the ability to have an anonymous sender or receiver. Additionally all channel operations in CSP are synchronous (no buffering). Advanced channels as a programming language feature has been popularized in recent years by the language Go [24], which encourages the use of channels as its fundamental concurrent feature. It was the popularity of Go channels that lead to their implementation in C∀. Neither Go nor C∀ channels have the restrictions of the early channel-based concurrent systems.

Other popular languages and libraries that provide channels include C++ Boost [40], Rust [42], Haskell [56], OCaml [50], and Kotlin [21]. Boost channels only support asynchronous (nonblocking) operations, and Rust channels are limited to only having one consumer per channel. Haskell channels are unbounded in size, and OCaml channels are zero-size. These restrictions in Haskell and OCaml are likely due to their functional approach, which results in them both using a list as the underlying data structure for their channel. These languages and libraries are not discussed further, as their channel implementation is not comparable to the bounded-buffer style channels present in Go and C∀. Kotlin channels are comparable to Go and C∀, but unfortunately they were not identified as a comparator until after presentation of this thesis and are omitted due to time constraints.

5.1 Producer-Consumer Problem

A channel is an abstraction for a shared-memory buffer, which turns the implementation of a channel into the producer-consumer problem. The producer-consumer problem, also known as the bounded-buffer problem, was introduced by Dijkstra [19, § 4.1]. In the problem, threads interact with a buffer in two ways: producing threads insert values into the buffer and consuming threads remove values from the buffer. In general, a buffer needs protection to ensure a producer only inserts into a non-full buffer and a consumer only removes from a non-empty buffer (synchronization). As well, a buffer needs protection from concurrent access by multiple producers or consumers attempting to insert or remove simultaneously, which is often provided by MX.

5.2 Channel Size

Channels come in three flavours of buffers:

1. Zero sized implies the communication is synchronous, *i.e.*, the producer must wait for the consumer to arrive or vice versa for a value to be communicated.
2. Fixed sized (bounded) implies the communication is mostly asynchronous, *i.e.*, the producer can proceed up to the buffer size and vice versa for the consumer with respect to removal, at which point the producer/consumer would wait.
3. Infinite sized (unbounded) implies the communication is asymmetrically asynchronous, *i.e.*, the producer never waits but the consumer waits when the buffer is empty.

In general, the order values are processed by the consumer does not affect the correctness of the producer-consumer problem. For example, the buffer can be *last-in first-out* (LIFO), *first-in first-out* (FIFO), or prioritized with respect to insertion and removal. However, like MX, a buffer should ensure every value is eventually removed after some reasonable bounded time (no long-term starvation). The simplest way to prevent starvation is to implement the buffer as a queue, either with a cyclic array or linked nodes. While FIFO is not required for producer-consumer problem correctness, it is a desired property in channels as it provides predictable and often relied upon channel ordering behaviour to users.

5.3 First-Come First-Served

As pointed out, a bounded buffer implementation often provides MX among multiple producers or consumers. This MX should be fair among threads, independent of the FIFO buffer being fair among values. Fairness among threads is called *first-come first-served* (FCFS) and was defined by Lamport [35, p. 454]. FCFS is defined in relation to a doorway [36, p. 330], which is the point at which an ordering among threads can be established. Given this doorway, a CS is said to be FCFS, if threads access the shared resource in the order they proceed through the doorway. A consequence of FCFS execution is the elimination of *barging*, where barging means a thread arrives at a CS with waiting threads, and the MX protecting the CS allows the arriving thread to enter the CS ahead of one or more of the waiting threads.

FCFS is a fairness property that prevents unequal access to the shared resource and prevents starvation, however it comes at a cost. Implementing an algorithm with FCFS can lead to *double blocking*, where arriving threads block outside the doorway waiting for a thread in the lock entry-protocol and inside the doorway waiting for a thread in the CS. An analogue is boarding an airplane: first you wait to get through security to the departure gates (short term), and then wait again at the departure gate for the airplane (long term). As such, algorithms that are not FCFS (barging) can be more performant by skipping the wait for the CS and entering directly; however, this performance gain comes by introducing unfairness with possible starvation for waiting threads.

5.4 Channel Implementation

The programming languages Go, Kotlin, and Erlang provide user-level threading where the primary communication mechanism is channels. These languages have user-level threading and preemptive scheduling, and both use channels for communication. Go and Kotlin provide multiple homogeneous channels; each have a single associated type. Erlang, which is closely related to actor systems, provides one heterogeneous channel per thread (mailbox) with a typed receive pattern. Go and Kotlin encourage users to communicate via channels, but provides them as an optional language feature. On the other hand, Erlang's single heterogeneous channel is a fundamental part of the threading system design; using it is unavoidable. Similar to Go and Kotlin, C$\forall$'s channels are offered as an optional language feature.

While iterating on channel implementation, experiments were conducted that varied the producer-consumer algorithm and lock type used inside the channel. With the exception of non-FCFS or non-FIFO algorithms, no algorithm or lock usage in the channel implementation was found to be consistently more performant that Go's choice of algorithm and lock implementation. Performance of channels can be improved by sharding the underlying buffer [18]. However, the FIFO property is lost, which is undesirable for user-facing channels. Therefore, the low-level channel implementation in C$\forall$ is largely copied from the Go implementation, but adapted to the C$\forall$ type and runtime systems. As such the research contributions added by C$\forall$'s channel implementation lie in the realm of safety and productivity features.

The Go channel implementation utilizes cooperation among threads to achieve good performance [37]. This cooperation only occurs when producers or consumers need to block due to the buffer being full or empty. After a producer blocks it must wait for a consumer to signal it and vice versa. The consumer or producer that signals a blocked thread is called the signalling thread. In these cases, a blocking thread stores their relevant data in a shared location and the signalling thread completes the blocking thread's operation before waking them; *i.e.*, the blocking thread has no work to perform after it unblocks because the signalling threads has done this work. This approach is similar to wait morphing for locks [12, p. 82] and improves performance in a few ways. First, each thread interacting with the channel only acquires and releases the internal channel lock once. As a result, contention on the internal lock is decreased; only entering threads compete for the lock since unblocking threads do not reacquire the lock. The other advantage of Go's wait-morphing approach is that it eliminates the need to wait for signalled threads to run. Note that the property of acquiring/releasing the lock only once can also be achieved with

a different form of cooperation, called *baton passing*. Baton passing occurs when one thread acquires a lock but does not release it, and instead signals a thread inside the critical section, conceptually "passing" the mutual exclusion from the signalling thread to the signalled thread. The baton-passing approach has threads cooperate to pass mutual exclusion without additional lock acquires or releases; the wait-morphing approach has threads cooperate by completing the signalled thread's operation, thus removing a signalled thread's need for mutual exclusion after unblocking. While baton passing is useful in some algorithms, it results in worse channel performance than the Go approach. In the baton-passing approach, all threads need to wait for the signalled thread to unblock and run before other operations on the channel can proceed, since the signalled thread holds mutual exclusion; in the wait-morphing approach, since the operation is completed before the signal, other threads can continue to operate on the channel without waiting for the signalled thread to run.

In this work, all channel sizes (see Sections 5.2) are implemented with bounded buffers. However, only non-zero-sized buffers are analysed because of their complexity and higher usage.

5.5 Safety and Productivity

Channels in C∀ come with safety and productivity features to aid users. The features include the following.

- Toggle-able statistic collection on channel behaviour that count channel and blocking operations. Tracking blocking operations helps illustrate usage for tuning the channel size, where the aim is to reduce blocking.
- Deadlock detection on channel deallocation. If threads are blocked inside a channel when it terminates, this case is detected and the user is informed, as this can cause a deadlock.
- A flush routine that delivers copies of an element to all waiting consumers, flushing the buffer. Programmers use this mechanism to broadcast a sentinel value to multiple consumers. Additionally, the flush routine is more performant than looping around the insert operation since it can deliver the elements without having to reacquire mutual exclusion for each element sent.
- Go-style ?<<? shorthand operator for inserting and removing.

```
channel(int) chan;
int i = 2;
chan << i;                          // insert i into chan
i << chan;                          // remove element from chan into i
```

5.5.1 Toggle-able Statistics

As discussed, a channel is a concurrent layer over a bounded buffer. To achieve efficient buffering, users should aim for as few blocking operations on a channel as possible. Mechanisms to reduce blocking are: change the buffer size, shard a channel into multiple channels, or tweak the number of producer and consumer threads. For users to be able to make informed decisions when tuning channel usage, toggle-able channel statistics are provided. The statistics are toggled on during the C∀ build by defining the CHAN_STATS macro, which guarantees zero cost when not using

this feature. When statistics are turned on, four counters are maintained per channel, two for inserting (producers) and two for removing (consumers). The two counters per type of operation track the number of blocking operations and total operations. In the channel destructor, the counters are printed out aggregated and also per type of operation. An example use case is noting that producer inserts are blocking often while consumer removes do not block often. This information can be used to increase the number of consumers to decrease the blocking producer operations, thus increasing the channel throughput. Whereas, increasing the channel size in this scenario is unlikely to produce a benefit because the consumers can never keep up with the producers.

5.5.2 Deadlock Detection

The deadlock detection in the CV channels is fairly basic but detects a very common channel mistake during termination. That is, it detects the case where threads are blocked on the channel during channel deallocation. This case is guaranteed to deadlock since there are no other threads to supply or consume values needed by the waiting threads. Only if a user maintained a separate reference to the blocked threads and manually unblocks them outside the channel could the deadlock be avoid. However, without special semantics, this unblocking would generate other runtime errors where the unblocked thread attempts to access non-existing channel data or even a deallocated channel. More robust deadlock detection needs to be implemented separate from channels since it requires knowledge about the threading system and other channel/thread state.

5.5.3 Program Shutdown

Terminating concurrent programs is often one of the most difficult parts of writing concurrent code, particularly if graceful termination is needed. Graceful termination can be difficult to achieve with synchronization primitives that need to be handled carefully during shutdown. It is easy to deadlock during termination if threads are left behind on synchronization primitives. Additionally, most synchronization primitives are prone to *time-of-check to time-of-use* (TOCTOU) issues where there is race between one thread checking the state of a concurrent object and another thread changing the state. TOCTOU issues with synchronization primitives often involve a race between one thread checking the primitive for blocked threads and another thread blocking on it. Channels are a particularly hard synchronization primitive to terminate since both sending and receiving to/from a channel can block. Thus, improperly handled TOCTOU issues with channels often result in deadlocks as threads performing the termination may end up unexpectedly blocking in their attempt to help other threads exit the system.

Go Channel Close

Go channels provide a set of tools to help with concurrent shutdown [37] using a close operation in conjunction with the **select** statement. The **select** statement is discussed in 7, where CV's **waituntil** statement is compared with the Go **select** statement.

The close operation on a channel in Go changes the state of the channel. When a channel is closed, sends to the channel panic along with additional calls to close. Receives are handled differently. Receivers (consumers) never block on a closed channel and continue to remove

elements from the channel. Once a channel is empty, receivers can continue to remove elements, but receive the zero-value version of the element type. To avoid unwanted zero-value elements, Go provides the ability to iterate over a closed channel to remove the remaining elements. These Go design choices enforce a specific interaction style with channels during termination: careful thought is needed to ensure additional close calls do not occur and no sends occur after a channel is closed. These design choices fit Go's paradigm of error management, where users are expected to explicitly check for errors, rather than letting errors occur and catching them. If errors need to occur in Go, return codes are used to pass error information up call levels. Note that panics in Go can be caught, but it is not the idiomatic way to write Go programs.

While Go's channel-closing semantics are powerful enough to perform any concurrent termination needed by a program, their lack of ease of use leaves much to be desired. Since both closing and sending panic once a channel is closed, a user often has to synchronize the senders (producers) before the channel can be closed to avoid panics. However, in doing so it renders the close operation nearly useless, as the only utilities it provides are the ability to ensure receivers no longer block on the channel and receive zero-valued elements. This functionality is only useful if the zero-typed element is recognized as a sentinel value, but if another sentinel value is necessary, then close only provides the non-blocking feature. To avoid TOCTOU issues during shutdown, a busy wait with a **select** statement is often used to add or remove elements from a channel. Hence, due to Go's asymmetric approach to channel shutdown, separate synchronization between producers and consumers of a channel has to occur during shutdown.

C∀ Channel Close

C∀ channels have access to an extensive exception handling mechanism [3]. As such C∀ uses an exception-based approach to channel shutdown that is symmetric for both producers and consumers, and supports graceful shutdown.

Exceptions in C∀ support both termination and resumption. *Termination exception*s perform a dynamic call that unwinds the stack preventing the exception handler from returning to the raise point, such as in C++, Python and Java. *Resumption exception*s perform a dynamic call that does not unwind the stack allowing the exception handler to return to the raise point. In C∀, if a resumption exception is not handled, it is reraised as a termination exception. This mechanism is used to create a flexible and robust termination system for channels.

When a channel in C∀ is closed, all subsequent calls to the channel raise a resumption exception at the caller. If the resumption is handled, the caller attempts to complete the channel operation. However, if the channel operation would block, a termination exception is thrown. If the resumption is not handled, the exception is rethrown as a termination. These termination exceptions allow for non-local transfer that is used to great effect to eagerly and gracefully shut down a thread. When a channel is closed, if there are any blocked producers or consumers inside the channel, they are woken up and also have a resumption thrown at them. The resumption exception, channel_closed, has internal fields to aid in handling the exception. The exception contains a pointer to the channel it is thrown from and a pointer to a buffer element. For exceptions thrown from remove, the buffer element pointer is null. For exceptions thrown from insert, the element pointer points to the buffer element that the thread attempted to insert. Utility routines bool is_insert(channel_closed & e); and bool is_remove(channel_closed & e); are

provided for convenient checking of the element pointer. This element pointer allows the handler to know which operation failed and also allows the element to not be lost on a failed insert since it can be moved elsewhere in the handler. Furthermore, due to C∀'s powerful exception system, this data can be used to choose handlers based on which channel and operation failed. For example, exception handlers in C∀ have an optional predicate which can be used to trigger or skip handlers based on the content of the matching exception. It is worth mentioning that using exceptions for termination may incur a larger performance cost than the Go approach. However, this should not be an issue, since termination is rarely on the fast-path of an application. In contrast, ensuring termination can be easily implemented correctly is the aim of the exception approach.

5.6 C∀/ Go channel Examples

To highlight the differences between C∀'s and Go's close semantics, two examples are presented. The first example is a simple shutdown case, where there are producer threads and consumer threads operating on a channel for a fixed duration. Once the duration ends, producers and consumers terminate immediately leaving unprocessed elements in the channel. The second example extends the first by requiring the channel to be empty after shutdown. Both the first and second example are shown in Figure 5.1.

Figure 5.1(a) shows the Go solution. Since some of the elements being passed through the channel are zero-valued, closing the channel in Go does not aid in communicating shutdown. Instead, a different mechanism to communicate with the consumers and producers needs to be used. Flag variables are common in Go-channel shutdown-code to avoid panics on a channel, meaning the channel shutdown has to be communicated with threads before it occurs. Hence, the two flags cons_done and prod_done are used to communicate with the producers and consumers, respectively. Furthermore, producers and consumers need to shutdown separately to ensure that producers terminate before the channel is closed to avoid panicking, and to avoid the case where all the consumers terminate first, which can result in a deadlock for producers if the channel is full. The producer flag is set first; then after all producers terminate, the consumer flag is set and the channel is closed leaving elements in the buffer. To purge the buffer, a loop is added (red) that iterates over the closed channel to process any remaining values.

Figure 5.1(b) shows the C∀ solution. Here, shutdown is communicated directly to both producers and consumers via the close call. A Producer thread knows to stop producing when the insert call on a closed channel raises exception channel_closed. If a Consumer thread ignores the first resumption exception from the close, the exception is reraised as a termination exception and elements are left in the buffer. If a Consumer thread handles the resumptions exceptions (red), control returns to complete the remove. A Consumer thread knows to stop consuming after all elements of a closed channel are removed and the consumer would block, which causes a termination raise of channel_closed. The C∀ semantics allow users to communicate channel shutdown directly through the channel, without having to share extra state between threads. Additionally, when the channel needs to be drained, C∀ provides users with easy options for processing the leftover channel values in the main thread or in the consumer threads.

Figure 5.2 shows a final shutdown example using channels to implement a barrier. A Go and C∀ style solution are presented but both are implemented using C∀ syntax so they can be

```
var channel chan int = make( chan int, 128 )
var prodJoin chan int = make( chan int, 4 )
var consJoin chan int = make( chan int, 4 )
var cons_done, prod_done bool = false, false;
func producer() {
    for {
        if prod_done { break }
        channel <- 5
    }
    prodJoin <- 0 // synch with main thd
}

func consumer() {
    for {
        if cons_done { break }
        <- channel
    }
    consJoin <- 0 // synch with main thd
}


func main() {
    for j := 0; j < 4; j++ { go consumer() }
    for j := 0; j < 4; j++ { go producer() }
    time.Sleep( time.Second * 10 )
    prod_done = true
    for j := 0; j < 4 ; j++ { <- prodJoin }
    cons_done = true
    close(channel) // ensure no cons deadlock
    for elem := range channel {
        // process leftover values
    }
    for j := 0; j < 4; j++ { <- consJoin }
}
```

(a) Go style

```
channel( int ) chan{ 128 };
thread Consumer {};
thread Producer {};

void main( Producer & this ) {
    try {
        for ()
            chan << 5;
    } catch( channel_closed * ) {
        // unhandled resume or full
    }
}
void main( Consumer & this ) {
    int i;
    try {
        for () { i << chan; }
    } catchResume( channel_closed * ) {
        // handled resume ⇒ consume from chan
    } catch( channel_closed * ) {
        // empty or unhandled resume
    }
}
int main() {
    Consumer c[4];
    Producer p[4];
    sleep( 10`s );
    close( chan );
}
```

(b) C∀ style

Figure 5.1: Channel Termination Examples 1 and 2. Code specific to example 2 is highlighted.

easily compared. Implementing a barrier is interesting because threads are both producers and consumers on the barrier-internal channels, entryWait and barWait. The outline for the barrier implementation starts by initially filling the entryWait channel with N tickets in the barrier constructor, allowing N arriving threads to remove these values and enter the barrier. After entryWait is empty, arriving threads block when removing. However, the arriving threads that entered the barrier cannot leave the barrier until N threads have arrived. Hence, the entering threads block on the empty barWait channel until the Nth arriving thread inserts $N-1$ elements into barWait to unblock the $N-1$ threads calling remove. The race between these arriving threads blocking on barWait and the Nth thread inserting values into barWait does not affect correctness;

i.e., an arriving thread may or may not block on channel barWait to get its value. Finally, the last thread to remove from barWait with ticket $N-2$, refills channel entryWait with N values to start the next group into the barrier.

Now, the two channels makes termination synchronization between producers and consumers difficult. Interestingly, the shutdown details for this problem are also applicable to other problems with threads producing and consuming from the same channel. The Go-style solution cannot use the Go close call since all threads are both potentially producers and consumers, causing panics on close to be unavoidable without complex synchronization. As such in Figure 5.2(a), a flush routine is needed to insert a sentinel value, -1, to inform threads waiting in the buffer they need to leave the barrier. This sentinel value has to be checked at two points along the fast-path and sentinel values daisy-chained into the buffers. Furthermore, an additional flag done is needed to communicate to threads once they have left the barrier that they are done. Also note that in the Go version 5.2(a), the size of the barrier channels has to be larger than in the C∀ version to ensure that the main thread does not block when attempting to clear the barrier. For The C∀ solution 5.2(b), the barrier shutdown results in an exception being thrown at threads operating on it, to inform waiting threads they must leave the barrier. This avoids the need to use a separate communication method other than the barrier, and avoids extra conditional checks on the fast path of the barrier implementation.

5.7 Performance

Given that the base implementation of the C∀ channels is very similar to the Go implementation, this section aims to show the performance of the two implementations are comparable. The microbenchmark for the channel comparison is similar to Figure 5.1, where the number of threads and processors is set from the command line. The processors are divided equally between producers and consumers, with one producer or consumer owning each core. The number of cores is varied to measure how throughput scales.

The results of the benchmark are shown in Figure 5.3. The performance of Go and C∀ channels on this microbenchmark is comparable. Note that the performance should decline as the number of cores increases as the channel operations occur in a critical section, so increasing cores results in higher contention with no increase in parallelism.

The performance of C∀ and Go's shutdown mechanisms is not measured, as shutdown is an exceptional case that does not occur frequently in most programs. Additionally, it is difficult to measure channel shutdown performance; threads need to be synchronized between each subsequent shutdown, which is likely more expensive than the shutdown mechanism itself.

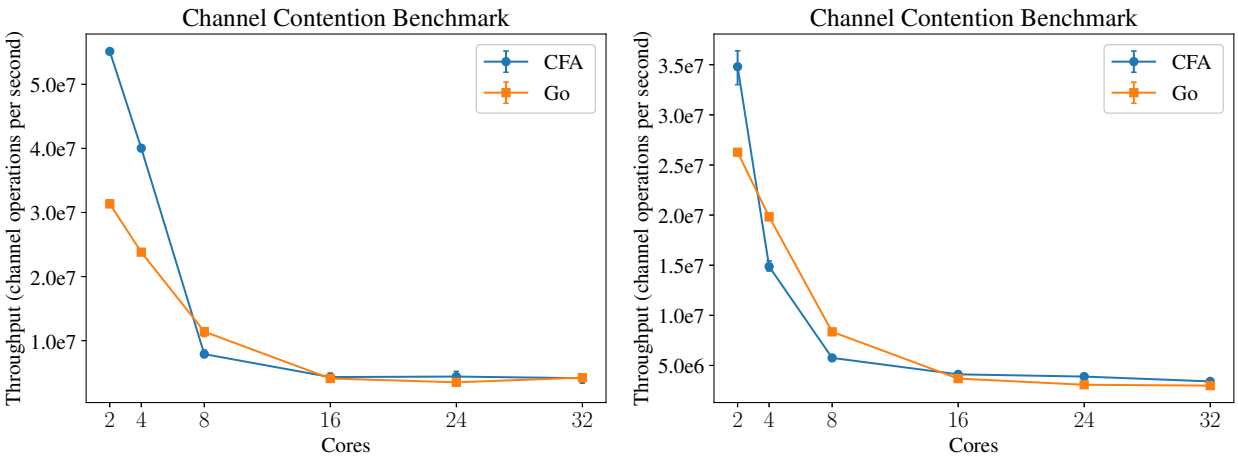
```
struct barrier {
    channel( int ) barWait, entryWait;
    int size;
};
void ?{}( barrier & this, int size ) with(this) {
    barWait{size + 1};   entryWait{size + 1};
    this.size = size;
    for ( i; size )
        insert( entryWait, i );
}
void wait( barrier & this ) with(this) {
    int ticket = remove( entryWait );
    if ( ticket == -1 ) { insert( entryWait, -1 ); return; }
    if ( ticket == size - 1 ) {
        for ( i; size - 1 )
            insert( barWait, i );
        return;
    }
    ticket = remove( barWait );
    if ( ticket == -1 ) { insert( barWait, -1 ); return; }
    if ( size == 1 || ticket == size - 2 ) { // last ?
        for ( i; size )
            insert( entryWait, i );
    }
}
void flush(barrier & this) with(this) {
    insert( entryWait, -1 );   insert( barWait, -1 );
}
enum { Threads = 4 };
barrier b{Threads};
bool done = false;
thread Thread {};
void main( Thread & this ) {
    for () {
      if ( done ) break;
        wait( b );
    }
}
int main() {
    Thread t[Threads];
    sleep(10`s);
    done = true;
    flush( b );
} // wait for threads to terminate
```

(a) Go style

```
struct barrier {
    channel( int ) barWait, entryWait;
    int size;
};
void ?{}( barrier & this, int size ) with(this) {
    barWait{size};   entryWait{size};
    this.size = size;
    for ( i; size )
        insert( entryWait, i );
}
void wait( barrier & this ) with(this) {
    int ticket = remove( entryWait );

    if ( ticket == size - 1 ) {
        for ( i; size - 1 )
            insert( barWait, i );
        return;
    }
    ticket = remove( barWait );

    if ( size == 1 || ticket == size - 2 ) { // last ?
        for ( i; size )
            insert( entryWait, i );
    }
}
void flush(barrier & this) with(this) {
    close( barWait );   close( entryWait );
}
enum { Threads = 4 };
barrier b{Threads};

thread Thread {};
void main( Thread & this ) {
    try {
        for ()
            wait( b );
    } catch ( channel_closed * ) {}
}
int main() {
    Thread t[Threads];
    sleep(10`s);

    flush( b );
} // wait for threads to terminate
```

(b) C∀ style

Figure 5.2: Channel Barrier Termination



(a) AMD Channel Benchmark

(b) Intel Channel Benchmark

Figure 5.3: The channel contention benchmark comparing C∀ and Go channel throughput (higher is better).

Chapter 6

Actors

Actors are an indirect concurrent feature that abstracts threading away from a programmer, and instead provides actors and messages as building blocks for concurrency. Hence, actors are in the realm of *implicit concurrency*, where programmers write concurrent code without dealing with explicit thread creation or interaction. Actor message-passing is similar to channels, but with more abstraction, so there is no shared data to protect, making actors amenable to a distributed environment. Actors are often used for high-performance computing and other data-centric problems, where the ease of use and scalability of an actor system provides an advantage over channels.

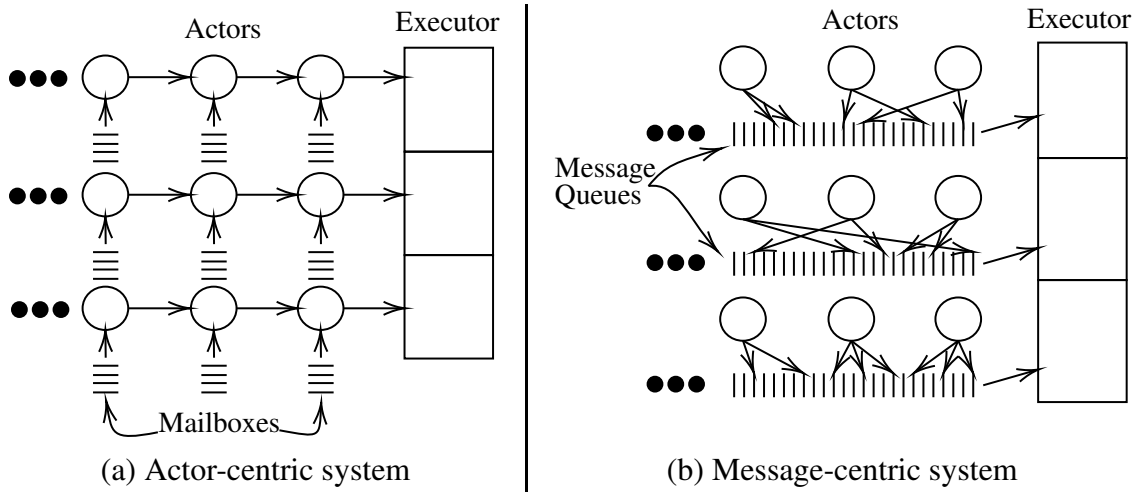
The study of actors can be broken into two concepts, the *actor model*, which describes the model of computation, and the *actor system*, which refers to the implementation of the model. Before discussing C∀'s actor system in detail, it is important to first describe the actor model, and the classic approach to implementing an actor system.

6.1 Actor Model

The *actor model* is a concurrent paradigm where an actor is used as the fundamental building-block for computation, and the data for computation is distributed to actors in the form of messages [26]. An actor is composed of a *mailbox* (message queue) and a set of *behaviours* that receive from the mailbox to perform work. Actors execute asynchronously upon receiving a message and can modify their own state, make decisions, spawn more actors, and send messages to other actors. Conceptually, actor systems can be thought of in terms of channels, where each actor's mailbox is a channel. However, a mailbox behaves like an unbounded channel, which differs from the fixed size channels discussed in the previous chapter. Because the actor model is implicit concurrency, its strength is that it abstracts away many details and concerns needed in other concurrent paradigms. For example, mutual exclusion and locking are rarely relevant concepts in an actor model, as actors typically only operate on local state.

6.1.1 Classic Actor System

An implementation of the actor model with a theatre (group) of actors is called an *actor system*. Actor systems largely follow the actor model, but can differ in some ways.



(a) Actor-centric system (b) Message-centric system

Figure 6.1: Classic and inverted actor implementation approaches with sharded queues.

In an actor system, an actor does not have a thread. An actor is executed by an underlying *executor* (kernel thread-pool) that fairly invokes each actor, where an actor invocation processes one or more messages from its mailbox. The default number of executor threads is often proportional to the number of computer cores to achieve good performance. An executor is often tunable with respect to the number of kernel threads and its scheduling algorithm, which optimize for specific actor applications and workloads (see Section 6.2.3).

While the semantics of message *send* is asynchronous, the implementation may be synchronous or a combination. The default semantics for message *receive* is FIFO, so an actor receives messages from its mailbox in temporal (arrival) order. Some actor systems provide priority-based mailboxes and/or priority-based message-selection within a mailbox, where custom message dispatchers search among or within a mailbox(es) with a predicate for specific kinds of actors and/or messages. Some actor systems provide a shared mailbox where multiple actors receive from a common mailbox [44], which is contrary to the no-sharing design of the basic actor-model (and may require additional locking). For non-FIFO service, some notion of fairness (eventual progress) should exist, otherwise messages have a high latency or starve, *i.e.*, are never received. Another way an actor system varies from the model is allowing access to shared global-state. When this occurs, it complicates the implementation as this breaks any implicit mutual-exclusion guarantees when only accessing local-state.

6.1.2 CA Actor System

Figure 6.1(a) shows an actor system designed as *actor-centric*, where a set of actors are scheduled and run on underlying executor threads [13, 44, 54]. The simplest design has a single global queue of actors accessed by the executor threads, but this approach results in high contention as both ends of the queue by the executor threads. The more common design is to *shard* the single queue among the executor threads, where actors are permanently assigned or can float among the queues. Sharding significantly decreases contention among executor threads adding and removing actors to/from a queue. Finally, each actor has a receive queue of messages (mailbox),

which is a single consumer, multi-producer queue, *i.e.*, only the actor removes from the mailbox but multiple actors add messages. When an actor receives a message in its mailbox, the actor is marked ready and scheduled by a thread to run the actor's current behaviour on the message(s).

Figure 6.1(b) shows an actor system designed as *message-centric*, where a set of messages are scheduled and run on underlying executor threads [10, 52]. This design is *inverted* because actors belong to a message queue, whereas in the classic approach a message queue belongs to each actor. Now a message send must query the actor to know which message queue to post the message to. Again, the simplest design has a single global queue of messages accessed by the executor threads, but this approach has the same contention problem by the executor threads. Therefore, the messages (mailboxes) are sharded and executor threads schedule each message, which points to its corresponding actor. Here, an actor's messages are permanently assigned to one queue to ensure FIFO receiving and/or reduce searching for specific actor/messages. Since multiple actors belong to each message queue, actor messages are interleaved on a queue, but individually in FIFO order. The inverted model can be taken a step further by sharding the message queues for each executor threads, so each executor thread owns a set of queues and cycles through them. Again, this extra level of sharding is to reduce queue contention.

The actor system in C$\forall$ uses a message-centric design, adopts several features from my prior actor work in μC++ [11] but is implemented in C$\forall$. My contributions to the prior actor work include introducing queue gulping, developing an actor benchmark suite, and extending promise support for actors. Furthermore, I improved the design and implementation of the μC++ actor system to greatly increase its performance. As such, the actor system in C$\forall$ started as a copy of the μC++ implementation, which was then refined. This work adds the following new C$\forall$ contributions:

1. Provide insight into the impact of envelope allocation in actor systems (see Section 6.2.2). In all actor systems, dynamic allocation is needed to ensure the lifetime of a unit of work persists from its creation until the unit of work is executed. This allocation is often called an *envelope* as it "packages" the information needed to run the unit of work, alongside any other information needed to send the unit of work, such as an actor's address or link fields. This dynamic allocation occurs once per message sent. Unfortunately, the high rate of message sends in an actor system results in significant contention on the memory allocator. A novel data structure is introduced to consolidate allocations to improve performance by minimizing allocator contention.
2. Improve performance of the inverted actor system using multiple approaches to minimize contention on queues, such as queue gulping and avoiding atomic operations.
3. Introduce work stealing in the inverted actor system. Work stealing in an actor-centric system involves stealing one or more actors among executor threads. In the inverted system, the notion of stealing message queues is introduced. The queue stealing is implemented such that the act of stealing work does not contend with non-stealing executor threads running actors.
4. Introduce and evaluate a timestamp-based work-stealing heuristic with the goal of maintaining non-workstealing performance in work-saturated workloads and improving performance on unbalanced workloads.
5. Provide a suite of safety and productivity features including static-typing, detection of

```
allocation receive( message & msg ) {
    case( msg_type1, msg ) { // discriminate type
        ... msg_d-> ...;    // msg_type1 msg_d
    } else case( msg_type2, msg ) {
        ... msg_d-> ...;    // msg_type2 msg_d
    ...
}
```

(a) dynamic typing

```
allocation receive( msg_type1 & msg ) {
    ... msg ...;
}
allocation receive( msg_type2 & msg ) {
    ... msg ...;
}
...
```

(b) static typing

Figure 6.2: Behaviour Styles

erroneous message sends, statistics tracking, and more.

6.2 C∀ Actor

C∀ is not an object oriented language and it does not have *run-time type information* (RTTI). As such, all message sends and receives among actors can only occur using static type-matching, as in Typed-Akka [43]. Figure 6.2 contrasts dynamic and static type-matching. Figure 6.2(a) shows the dynamic style with a heterogeneous message receive and an indirect dynamic type-discrimination for message processing. Figure 6.2(b) shows the static style with a homogeneous message receive and a direct static type-discrimination for message processing. The static-typing style is safer because of the static check and faster because there is no dynamic type-discrimination. The dynamic-typing style is more flexible because multiple kinds of messages can be handled in a behaviour condensing the processing code.

Figure 6.3 shows a complete C∀ actor example, which is discussed in detail. The actor type my_actor is a **struct** that inherits from the base actor **struct** via the **inline** keyword. This inheritance style is the Plan-9 C-style (see Section 2.7.2). Similarly, the message types str_msg and int_msg are **struct**s that inherits from the base message **struct** via the **inline** keyword. Only str_msg needs a constructor to copy the C string; int_msg is initialized using its C∀ auto-generated constructors. There are two matching receive (behaviour) routines that process the corresponding typed messages. Both receive routines use a **with** clause so message fields are not qualified (see Section 2.4) and return Nodelete indicating the actor is not finished (see Section 6.2.1). Also, all messages are marked with Nodelete as their default allocation state. The program main begins by creating two messages on the stack. Then the executor system is started by calling start_actor_system (see Section 6.2.3). Now an actor is created on the stack and four messages are sent to it using operator ?|? (see Section 2.5). The last message is the builtin finish_msg, which returns Finished to an executor thread, causing it to remove the actor from the actor system (see end of Section 6.2.1). The call to stop_actor_system blocks the program main until all actors are finished and removed from the actor system. The program main ends by deleting the actor and the two messages from the stack. The output for the program is:

```
string message "Hello World"
integer message 42
integer message 42
```

```
// actor
struct my_actor {
    inline actor;                                   // Plan-9 C inheritance
};
// messages
struct str_msg {
    char str[12];
    inline message;                                 // Plan-9 C inheritance
};
void ?{}( str_msg & this, char * str ) { strcpy( this.str, str ); }  // constructor
struct int_msg {
    int i;
    inline message;                                 // Plan-9 C inheritance
};
// behaviours
allocation receive( my_actor &, str_msg & msg ) with(msg) {
    sout | "string message \"" | str | "\"";
    return Nodelete;                                // actor not finished
}
allocation receive( my_actor &, int_msg & msg ) with(msg) {
    sout | "integer message" | i;
    return Nodelete;                                // actor not finished
}
int main() {
    str_msg str_msg{ "Hello World" };               // constructor call
    int_msg int_msg{ 42 };                          // constructor call
    start_actor_system();                           // sets up executor
    my_actor actor;                                 // default constructor call
    actor | str_msg | int_msg;                      // cascade sends
    actor | int_msg;                                // send
    actor | finished_msg;                           // send => terminate actor (builtin Poison-Pill)
    stop_actor_system();                            // waits until actors finish
} // deallocate actor, int_msg, str_msg
```

Figure 6.3: C∀ Actor Syntax

6.2.1 Actor Behaviours

In general, a behaviour for some derived actor and derived message type is defined with the following signature:

```
allocation receive( my_actor & receiver, my_msg & msg )
```

where my_actor and my_msg inherit from types actor and message, respectively. The return value of receive must be a value from enumerated type, allocation:

```
enum allocation { Nodelete, Delete, Destroy, Finished };
```

The values represent a set of actions that dictate what the executor does with an actor or message after a given behaviour returns. For actors, the receive routine returns the allocation status to the executor, which takes the appropriate action. For messages, either the default allocation,

Nodelete, or any changed value in the message is examined by the executor, which takes the appropriate action. Message state is updated via a call to:

```
void set_allocation( message & this, allocation state );
```

In detail, the actions taken by an executor for each of the allocation values are:

Nodelete tells the executor that no action is to be taken with regard to an actor or message. This status is used when an actor continues receiving messages or a message is reused.

Delete tells the executor to call the object's destructor and deallocate (delete) the object. This status is used with dynamically allocated actors and messages when they are not reused.

Destroy tells the executor to call the object's destructor, but not deallocate the object. This status is used with dynamically allocated actors and messages whose storage is reused.

Finished tells the executor to mark the respective actor as finished executing, but not call the object's destructor nor deallocate the object. This status is used when actors or messages are global or stack allocated, or a programmer wants to manage deallocation themselves. Note that for messages there is no difference between allocations Nodelete and Finished because both tell the executor to do nothing to the message. Hence, Finished is implicitly changed to Nodelete in a message constructor, and Nodelete is used internally for message error-checking (see Section 6.5). Therefore, reading a message's allocation status after setting to Finished may be either Nodelete (after construction) or Finished (after explicitly setting using set_allocation).

For the actor system to terminate, all actors must have returned a status other than Nodelete. After an actor is terminated, it is erroneous to send messages to it. Similarly, after a message is terminated, it cannot be sent to an actor. Note that it is safe to construct an actor or message with a status other than Nodelete, since the executor only examines the allocation action *after* a behaviour returns.

6.2.2 Actor Envelopes

As stated, each message, regardless of where it is allocated, can be sent to an arbitrary number of actors, and hence, appear on an arbitrary number of message queues. Because a C program manages message lifetime, messages cannot be copied for each send, otherwise who manages the copies? Therefore, it is up to the actor program to manage message life-time across receives. However, for a message to appear on multiple message queues, it needs an arbitrary number of associated destination behaviours. Hence, there is the concept of an envelope, which is dynamically allocated on each send, that wraps a message with any extra implementation fields needed to persist between send and receive. Managing the envelope is straightforward because it is created at the send and deleted after the receive, *i.e.*, there is 1:1 relationship for an envelope and a many to one relationship for a message.

6.2.3 Actor System

The calls to start_actor_system, and stop_actor_system mark the start and end of a C∀ actor system. The call to start_actor_system sets up an executor and executor threads for the actor system. It is possible to have multiple start/stop scenarios in a program.

start_actor_system has three overloaded signatures that vary the executor's configuration:

void start_actor_system() configures the executor to implicitly use all preallocated kernel-threads (processors), *i.e.*, the processors created by the program main prior to starting the actor system. For example, the program main declares at the start:

```
processor p[3];
```

which provides a total of 4 threads (3 + initial processor) for use by the executor. When the number of processors is greater than 1, each executor's message queue is sharded by a factor of 16 to reduce contention, *i.e.*, for 4 executor threads (processors), there is a total of 4×16 message queues evenly distributed across the executor threads.

void start_actor_system(size_t num_thds) configures the number of executor threads to num_thds, with the same message queue sharding.

void start_actor_system(executor & this) allows the programmer to explicitly create and configure an executor for use by the actor system. Executor configuration options are discussed in Section 6.3.

All actors must be created *after* calling start_actor_system so the executor can keep track of the number of actors that have entered the system but not yet terminated.

6.2.4 Actor Send

All message sends are done using the vertical-bar (bit-or) operator, ?|?, similar to the syntax of the C∀ stream I/O. One way to provide a generic operator is through the C∀ type system:

```
actor & ?|?( actor &, message & ) { // base actor and message types
    // boilerplate to send message to executor mail queue
}
actor | str_msg | int_msg;   // rewritten: ?|?( ?|?( actor, int_msg ), str_msg )
```

In the C∀ type system, calls to this routine work for any pair of parameters that inherit from the actor and message types via Plan-9 inheritance. However, within the body the routine, all type information about the derived actor and message is lost (type erasure), so this approach is unable to find the right receive routine to put in the envelope.

If C∀ had a fully-fledged virtual system, the generic ?|? routine would work, since the virtual system could dynamically select the derived receive routine via virtual dispatch. C∀ does have a preliminary form of virtual routines, but it is not mature enough for use in this work, so a different approach is needed.

Without virtuals, the idiomatic C∀ way to create the generic ?|? routine is using **forall**:

```
// forall types A, M that have a receive that returns allocation
forall( A &, M & | { allocation receive( A &, M & ); } )
A & ?|?( A &, M & ) { // actor and message types
    // boilerplate to send message to executor mail queue
}
```

This approach should work. However, the C∀ type system is still a work in progress, and there is a nontrivial bug where inherited routines are not recognized by **forall**. For example, Figure 6.4

```
struct A {};
struct B { inline A; }
void foo( A & a ) { ... }

// for all types that have a foo routine here is a bar routine
forall( T & | { void foo( T & ); } )
void bar( T & t ) { ... }

int main() {
    B b;
    foo( b ); // B has a foo so it should find a bar via the forall
    bar( b ); // compilation error, no bar found for type B
}
```

Figure 6.4: C∀ Type-System Problem

shows type B has an inherited foo routine through type A and should find the bar routine defined via the **forall**, but does not due the type-system bug.

Users could be expected to write the ?|? routines, but this approach is error prone and creates maintenance issues. As a stopgap until the C∀ type-system matures, a workaround was created using a template-like approach, where the compiler generates a matching ?|? routine for each receive routine it finds with the correct actor/message type-signature. This workaround is outside of the type system, but performs a type-system like action. The workaround requires no annotation or additional code to be written by users; thus, it resolves the maintenance and error problems. It should be possible to seamlessly transition the workaround into any updated version of the C∀ type-system.

Figure 6.5 shows the generated send routine for the int_msg receive in Figure 6.3. Operator ?|? has the same parameter signature as the corresponding receive routine and returns an actor so the operator can be cascaded. The routine sets rec_fn to the matching receive routine using the left-hand type to perform the selection. Then the routine packages the actor and message, along with the receive routine into an envelope. Finally, the envelope is added to the executor queue designated by the actor using the executor routine send.

Figure 6.6 shows three builtin *poison-pill* messages and receive routines used to terminate actors, depending on how an actor is allocated: Delete, Destroy or Finished. Poison-pill messages are common across actor systems, including Akka and ProtoActor [44, 54] to suggest or force actor termination. For example, in Figure 6.3, the builtin finished_msg message and receive are used to terminate the actor because the actor is allocated on the stack, so no deallocation actions are performed by the executor. Note that assignment is used to initialize these messages rather than constructors because the constructor changes the allocation to Nodelete for error checking

6.2.5 Actor Termination

During a message send, the receiving actor and message being sent are stored via pointers in the envelope. These pointers have the base actor and message types, so type information of the

```
// from Figure 6.3
struct my_actor { inline actor; };                          // actor
struct int_msg { inline message; int i; };                  // message
allocation receive( my_actor &, int_msg & msg ) {...}       // receiver

// compiler generated send operator
typedef allocation (*receive_t)( actor &, message & );
actor & ?|?( my_actor & receiver, int_msg & msg ) {
    allocation (*rec_fn)( my_actor &, int_msg & ) = receive; // deduce receive routine
    request req{ (actor *)&receiver, (message *)&msg, (receive_t)rec_fn };
    send( receiver, req );                                  // queue message for execution
    return receiver;
}
```

Figure 6.5: Generated Send Operator

```
message __base_msg_finished @= { .allocation_ : Finished }; // use C initialization
struct delete_msg_t { inline message; } delete_msg = __base_msg_finished;
struct destroy_msg_t { inline message; } destroy_msg = __base_msg_finished;
struct finished_msg_t { inline message; } finished_msg = __base_msg_finished;

allocation receive( actor & this, delete_msg_t & msg ) { return Delete; }
allocation receive( actor & this, destroy_msg_t & msg ) { return Destroy; }
allocation receive( actor & this, finished_msg_t & msg ) { return Finished; }
```

Figure 6.6: Builtin Poison-Pill Messages

derived actor and message is lost and must be recovered later when the typed receive routine is called. After the receive routine is done, the executor must clean up the actor and message according to their allocation status. If the allocation status is Delete or Destroy, the appropriate destructor must be called by the executor. This requirement poses a problem: the derived type of the actor or message is not available to the executor, but it needs to call the derived destructor. This requires downcasting from the base type to the derived type, which requires a virtual system. To accomplish the downcast, a rudimentary destructor-only virtual system was implemented in C∀ as part of this work. This virtual system is used via Plan-9 inheritance of the virtual_dtor type, shown in Figure 6.7. The virtual_dtor type maintains a pointer to the start of the object, and a pointer to the correct destructor. When a type inherits virtual_dtor, the compiler adds code to its destructor to intercepted any destructor calls along this segment of the inheritance tree and restart at the appropriate destructor for that object.

While this virtual destructor system was built for this work, it is general and can be used with any type in C∀. Actors and messages opt into this system by inheriting the virtual_dtor type, which allows the executor to call the right destructor without knowing the derived actor or message type. Again, it should be possible to seamlessly transition this workaround into any updated version of the C∀ type-system.

```
struct base { inline virtual_dtor; };
void ^?{}( base & ) { sout | "^base"; }
struct intermediate { inline base; };
void ^?{}( intermediate & ) { sout | "^intermediate"; }
struct derived { inline intermediate; };
void ^?{}( derived & ) { sout | "^derived"; }

int main() {
    base & b;
    intermediate i;
    derived d1, d2, d3;
    intermediate & ri = d2;
    base & rb = d3;
    // explicit destructor calls
    ^d1{};   sout | nl;
    ^ri{};   sout | nl;
    ^rb{};   sout | nl;
} //^i, ^b
```

(a) Destructor calls

```
^derived
^intermediate
^base

^derived
^intermediate
^base

^derived
^intermediate
^base

^intermediate
^base
```

(b) Output

Figure 6.7: C∀ Virtual Destructor

6.3 C∀ Executor

This section describes the basic architecture of the C∀ executor. An executor of an actor system is the scheduler that organizes where actor behaviours are run and how messages are sent and delivered. In C∀, the executor is message-centric (see Figure 6.1(b)), but extended by over sharding of a message queue (see left side of Figure 6.8), *i.e.*, there are M message queues where M is greater than the number of executor threads N (usually a multiple of N). This approach reduces contention by spreading message delivery among the M queues rather than N, while still maintaining actor FIFO message-delivery semantics. The only extra overhead is each executor cycling (usually round-robin) through its M/N queues. The goal is to achieve better performance and scalability for certain kinds of actor applications by reducing executor locking. Note that lock-free queues do not help because busy waiting on any atomic instruction is the source of the slowdown whether it is a lock or lock-free.

Each executor thread iterates over its own message queues until it finds one with messages. At this point, the executor thread atomically *gulp*s the queue, meaning it moves the contents of message queue to a local queue of the executor thread. Gulping moves the contents of the message queue as a batch rather than removing individual elements. An example of the queue gulping operation is shown in the right side of Figure 6.8, where an executor thread gulps queue 0 and begins to process it locally. This step allows the executor thread to process the local queue without any atomics until the next gulp. Other executor threads can continue adding to the ends of the executor thread's message queues. In detail, an executor thread performs a test-and-gulp, non-atomically checking if a queue is non-empty, before attempting to gulp it. If an executor misses an non-empty queue due to a race, it eventually finds the queue after cycling through its message queues. This approach minimizes costly lock acquisitions.

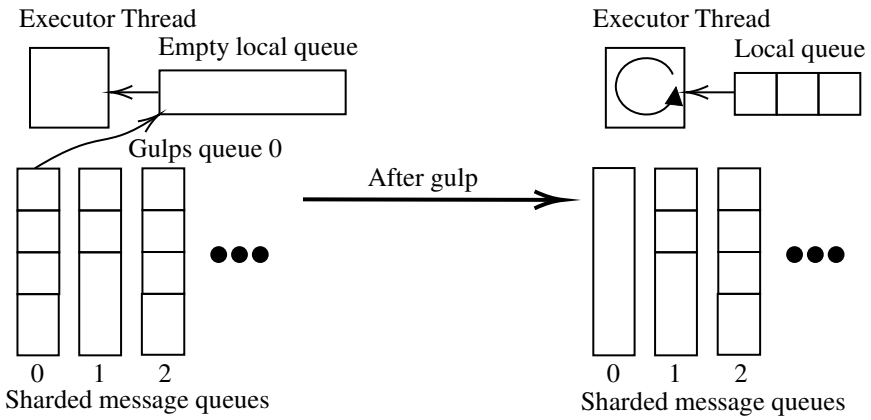


Figure 6.8: Queue Gulping Mechanism

Processing a local queue involves: removing a unit of work from the queue, dereferencing the actor pointed-to by the work unit, running the actor's behaviour on the work-unit message, examining the returned allocation status from the receive routine for the actor and internal status in the delivered message, and taking the appropriate actions. Since all messages to a given actor are in the same queue, this guarantees atomicity across behaviours of that actor since it can only execute on one thread at a time. As each actor is created or terminated by an executor thread, it atomically increments/decrements a global counter. When an executor decrements the counter to zero, it sets a global boolean variable that is checked by each executor thread when it has no work. Once a executor threads sees the flag is set it stops running. After all executors stop, the actor system shutdown is complete.

6.3.1 Copy Queue

Unfortunately, the frequent allocation of envelopes for each send results in heavy contention on the memory allocator. This contention is reduced using a novel data structure, called a *copy queue*. The copy queue is a thin layer over a dynamically sized array that is designed with the envelope use-case in mind. A copy queue supports the typical queue operations of push/pop but in a different way from a typical array-based queue.

The copy queue is designed to take advantage of the gulping pattern, giving an amortized runtime cost for each push/pop operation of $O(1)$. In contrast, a naïve array-based queue often has either push or pop cost $O(n)$ and the other cost $O(1)$ since one of the operations requires shifting the elements of the queue. Since the executor threads gulp a queue to operate on it locally, this creates a usage pattern where all elements are popped from the copy queue without any interleaved pushes. As such, during pop operations there is no need to shift array elements. Instead, an index is stored in the copy-queue data-structure that keeps track of which element to pop next allowing pop to be $O(1)$. Push operations are amortized $O(1)$ since pushes may cause doubling reallocations of the underlying dynamic-sized array (like C++ vector). Note that the copy queue is similar to a circular buffer, but has a key difference. After all elements are popped, the end of the queue is set back to index zero, whereas a standard circular buffer would leave the

end in the same location. Using a standard circular buffer would incur an additional $O(n)$ cost of fixing the buffer after a doubling reallocation.

Since the copy queue is an array, envelopes are allocated first on the stack and then copied into the copy queue to persist until they are no longer needed. For many workloads, the copy queues reallocate and grow in size to facilitate the average number of messages in flight and there are no further dynamic allocations. The downside of this approach is that more storage is allocated than needed, *i.e.*, each copy queue is only partially full. Comparatively, the individual envelope allocations of a list-based queue mean that the actor system always uses the minimum amount of heap space and cleans up eagerly. Additionally, bursty workloads can cause the copy queues to allocate a large amount of space to accommodate the throughput peak, even if most of that storage is not needed for the rest of the workload's execution.

To mitigate memory wastage, a reclamation scheme is introduced. Initially, the memory reclamation naïvely reclaims one index of the array per gulp, if the array size is above a low fixed threshold. However, this approach has a problem. The high memory watermark nearly doubled! The issue is highlighted with an example. Assume a fixed throughput workload, where a queue never has more than 19 messages at a time. If the copy queue starts with a size of 10, it ends up doubling at some point to size 20 to accommodate 19 messages. However, after 2 gulps and subsequent reclamations the array size is 18. The next time 19 messages are enqueued, the array size is doubled to 36! To avoid this issue, a second check is added. Reclamation only occurs if less than half of the array is utilized. This check achieves a lower total storage and overall memory utilization compared to the non-reclamation copy queues. However, the use of copy queues still incurs a higher memory cost than list-based queueing, but the increase in memory usage is reasonable considering the performance gains (see Section 6.6).

6.4 Work Stealing

Work stealing is a scheduling strategy to provide *load balancing*. The goal is to increase resource utilization by having an idle thread steal work from a working thread. While there are multiple parts in a work-stealing scheduler, two important components are the stealing mechanism and victim selection.

6.4.1 Stealing Mechanism

In work stealing, the stealing worker is called the *thief* and the worker being stolen from is called the *victim*. To steal, a thief takes work from a victim's ready queue, so work stealing always results in a potential increase in contention on ready queues between the victim gulping from a queue and the thief stealing the queue. This contention can reduce the victim's throughput. Note that the data structure used for the ready queue is not discussed since the largest cost is the mutual exclusion and its duration for safely performing the queue operations.

The stealing mechanism in this work differs from most work-stealing actor-systems because of the message-centric (inverted) actor-system. Actor systems, such as Akka [44] and CAF [13] using actor-centric systems, steal by dequeuing an actor from a non-empty actor ready-queue and enqueueing to an empty ready-queue. In C∀, the actor work-stealing implementation is

unique because of the message-centric system. With this approach, it is impractical to steal actors because an actor's messages are distributed in temporal order along the message queue. To ensure sequential actor execution and FIFO message delivery in a message-centric system, stealing requires finding and removing *all* of an actor's messages, and inserting them consecutively in another message queue. This operation is $O(N)$ with a non-trivial constant. The only way for work stealing to become practical is to shard each worker's message queue (see Section 6.3), which also reduces contention, and to steal queues to eliminate queue searching.

Given queue stealing, the goal of the presented stealing implementation is to have an essentially zero-contention-cost stealing mechanism. Achieving this goal requires work stealing to have minimal (practically no) effect on the performance of the victim. The implication is that thieves cannot contend with a victim, and that a victim should perform no stealing related work unless it becomes a thief. In theory, this goal is not achievable, but practical results show the goal is nearly achieved.

One important lesson learned while working on μC++ actors [11] and through discussions with fellow student Thierry Delisle, who examined work-stealing for user-threads in his Ph.D. [16], is *not* to aggressively steal. With reasonable workloads, being a thief should be a temporary state, *i.e.*, eventually work appears on the thief's ready-queues and it returns to normal operation. Furthermore, the act of *looking* to find work is invasive, possibly disrupting multiple victims. Therefore, stealing should be done lazily in case work appears for the thief and to minimize disruption of victims. Note that the cost of stealing is not crucial for the thief because it does not have anything else to do except poll or block.

The outline for lazy-stealing by a thief is: select a victim, scan its queues once, and return immediately if a queue is stolen. The thief then assumes its normal operation of scanning over its own queues looking for work, where stolen work is placed at the end of the scan. Hence, only one victim is affected and there is a reasonable delay between stealing events as the thief scans its ready queue looking for its own work before potentially stealing again. This lazy examination by the thief has a low perturbation cost for victims, while still finding work in a moderately loaded system. In all work-stealing algorithms, there is the pathological case where there is too little work and too many workers; this scenario can only be handled by putting workers to sleep or deleting them. This case is not examined in this work.

In more detail, the CFA work-stealing algorithm begins by iterating over its message queues twice without finding any work before it tries to steal a queue from another worker. Stealing a queue is done atomically with a few atomic instructions that only create contention with other stealing workers not the victim. The complexity in the implementation is that victim gulping does not take the mailbox queue; rather it atomically transfers the mailbox nodes to another queue leaving the mailbox empty, as discussed in Section 6.3. Hence, the original mailbox is always available for new message deliveries. However, this transfer logically subdivides the mailbox into two separate queues, and during this period, the mailbox cannot be stolen; otherwise there are two threads simultaneously running messages on an actor in the two parts of the mailbox queue. To solve this problem, an atomic gulp also marks the mailbox queue as subdivided making it ineligible for stealing. Hence, a thief checks if a queue is eligible and non-empty before attempting an atomic steal of a queue.

Figure 6.9 shows the queue architecture and stealing mechanism. Internally, the mailbox

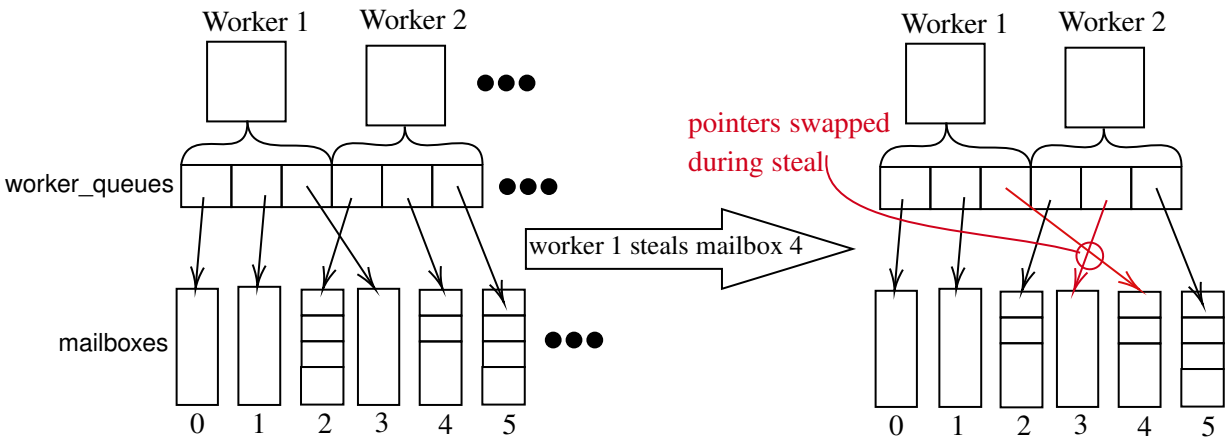


Figure 6.9: Queue Stealing Mechanism

queues are accessed through two arrays of size N, which are shared among all workers. There is an array of mailbox queues, mailboxes, and an array of pointers to the mailboxes, worker_queues:

```
struct work_queue {
    spinlock_t mutex_lock;              // atomicity for queue operations
    copy_queue * owned_queue;           // copy queue
    copy_queue * c_queue;               // current queue
    volatile bool being_processed;      // flag to prevent concurrent processing
};
work_queue * mailboxes;                 // master array of work request queues
work_queue ** worker_queues;            // secondary array of work queues to allow for swapping
```

A send inserts a request at the end of one of mailboxes. A steal swaps two pointers in worker_queues. Conceptually, worker_queues represents the ownership relation between mailboxes and workers. Given M workers and N mailboxes, each worker owns a contiguous M/N block of pointers in worker_queues. When a worker gulps, it dereferences one of the pointers in its section of worker_queues and then gulps the queue from the mailbox it points at. To transfer ownership of a mailbox from one worker to another, a pointer from each of the workers' ranges are swapped. This structure provides near-complete separation of stealing and gulping/send operations. As such, operations can happen on mailboxes independent of stealing, which avoids almost all contention between thief and victim threads.

To steal a queue, a thief does the following:

1. chooses a victim. Victim selection heuristics are independent of the stealing mechanism and discussed in Section 6.4.4.
2. scan the victim's M/N range of worker_queues and non-atomically checks each mailbox to see if it is eligible and non-empty. If a match is found, the thief attempts to steal the mailbox by swapping the appropriate victim worker-queue pointer with an empty thief's pointer, where the pointers come from the victim's and thief's ranges, respectively. This swap can fail if another thief steals the queue, which is discussed further in Section 6.4.3. If no appropriate victim mailbox is found, no swap is attempted. Note that since the

mailbox checks happen non-atomically, the thieves effectively guess which mailbox is ripe for stealing. The thief may read stale data and can end up stealing an ineligible or empty mailbox. This is not a correctness issue and is addressed in Section 6.4.2, but the steal will likely not be productive. These unproductive steals are uncommon, but do occur with some frequency, and are a tradeoff that is made to achieve minimal victim contention.

3. stops searching after a successful mailbox steal, a failed mailbox steal, or all mailboxes in the victim's range are examined. The thief then resumes normal execution and ceases being a thief. Hence, it iterates over its own worker queues because new messages may have arrived during stealing, including ones in the potentially stolen queue. Stealing is only repeated after the worker completes two consecutive iterations over its message queues without finding work.

6.4.2 Stealing Problem

Each queue access (send or gulp) involving any worker (thief or victim) is protected using spinlock mutex_lock. However, to achieve the goal of almost zero contention for the victim, it is necessary that the thief does not acquire any queue spinlocks in the stealing protocol. The victim critical-section is gulping a queue, which involves two steps:

```
temp = worker_queues[x];
// preemption and steal
transfer( local_queue, temp->c_queue );   // atomically sets being_processed
```

where transfer gulps the work from c_queue to the victim's local_queue and leaves c_queue empty, partitioning the mailbox. Specifically,

1. The victim must dereference its current mailbox pointer from worker_queues.
2. The victim calls transfer to gulp from the mailbox.

If the victim is preempted after the dereference, a thief can steal the mailbox pointer before the victim calls transfer. The thief then races ahead, transitions back to a victim, searches its mailboxes, finds the stolen non-empty mailbox, and gulps this queue. The original victim now continues and gulps from the stolen mailbox pointed to by its dereference, even though the thief has logically subdivided this mailbox by gulping it. At this point, the mailbox has been subdivided a second time, and the victim and thief are possibly processing messages sent to the same actor, which violates mutual exclusion and the message-ordering guarantee. Preventing this race requires either a lock acquire or an atomic operation on the victim's fast-path to guarantee the victim's mailbox dereferenced pointer is not stale. However, any form of locking here creates contention between thief and victim.

The alternative to locking is allowing the race and resolving it lazily. To resolve the race, each mailbox header stores a being_processed flag that is atomically set when a queue is transferred. The flag indicates that a mailbox has been gulped (logically subdivided) by a worker and the gulped queue is being processed locally. The being_processed flag is reset once the local processing is finished. If a worker, either victim or thief turned victim, attempts to gulp from a mailbox and finds the being_processed flag set, it does not gulp and moves onto the next mailbox in its range. This resolves the race no matter the winner. If the thief wins the race, it steals the

mailbox and gulps, and the victim sees the flag set and skips gulping from the mailbox. If the victim wins the race, it gulps from the mailbox, and the thief sees the flag set and does not gulp from the mailbox.

There is a final case where the race occurs and is resolved with *both* gulps occurring. Here, the winner of the race finishes processing the queue and resets the being_processed flag. Then the loser unblocks from its preemption and completes its gulp from the same mailbox and atomically sets the being_processed flag. The loser is now processing messages from a temporarily shared mailbox, which is safe because the winner ignores this mailbox, if it attempts another gulp since being_processed is set. The victim never attempts to gulp from the stolen mailbox again because its next cycle sees the swapped mailbox from the thief (which may or may not be empty at this point). This race is now the only source of contention between victim and thief as they both try to acquire a lock on the same queue during a transfer. However, the window for this race is extremely small, making this contention rare. In theory, if this race occurs multiple times consecutively, *i.e.*, a thief steals, dereferences a stolen mailbox pointer, is interrupted, and stolen from, etc., this scenario can cascade across multiple workers all attempting to gulp from one mailbox. The being_processed flag ensures correctness even in this case, and the chance of a cascading scenario across multiple workers is even rarer.

It is straightforward to count the number of missed gulps due to the being_processed flag and this counter is added to all benchmarks presented in Section 6.6. The results show the median count of missed gulps for each experiment is *zero*, except for the repeat benchmark. The repeat benchmark is an example of the pathological case described earlier where there is too little work and too many workers. In the repeat benchmark, one actor has the majority of the workload, and no other actor has a consistent workload, which results in rampant stealing. None of the work-stealing actor-systems examined in this work perform well on the repeat benchmark. Hence, for all non-pathological cases, the claim is made that this stealing mechanism has a (probabilistically) zero-victim-cost in practice. Future work on the work stealing system could include a backoff mechanism after failed steals to further address the pathological cases.

6.4.3 Queue Pointer Swap

To atomically swap the two worker_queues pointers during work stealing, a novel atomic swap-algorithm is needed. The *compare-and-set (swap)* (CAS) is a read-modify-write instruction available on most modern architectures. It atomically compares two memory locations, and if the values are equal, it writes a new value into the first memory location. A sequential specification of CAS is:

```
// assume this routine executes atomically
bool CAS( T * assn, T comp, T new ) {   // T is a basic type
    if ( *assn != comp ) return false;
    *assn = new;
    return true;
}
```

However, this instruction does *not* swap assn and new, which is why compare-and-swap is a misnomer. If T can be a double-wide address-type (128 bits on a 64-bit machine), called a

double-wide (width) compare-and-set (swap) (DWCAS), then it is possible to swap two values, if and only if the two addresses are juxtaposed in memory.

```
union Pair {
    struct { void * ptr1, * ptr2; };   // 64–bit pointers
    __int128 atom;
};
Pair pair1 = { addr1, addr2 }, pair2 = { addr2, addr1 };
Pair top = pair1;
DWCAS( top.atom, pair1.atom, pair2.atom );
```

However, this approach does not apply because the mailbox pointers are seldom juxtaposed.

Only a few architectures provide a *double compare-and-set (swap)* (DCAS), which extends CAS to two memory locations [22].

```
// assume this routine executes atomically
bool DCAS( T * assn1, T * assn2, T comp1, T comp2, T new1, T new2 ) {
    if ( *assn1 == comp1 && *assn2 == comp2 ) {
        *assn1 = new1;
        *assn2 = new2;
        return true;
    }
    return false;
}
```

DCAS can be used to swap two values; for this use case the comparisons are superfluous.

```
DCAS( x, y, x, y, y, x );
```

A restrictive form of DCAS can be simulated using *load linked* (LL)/*store conditional* (SC) [7] or more expensive transactional memory with the same progress property problems as LL/SC.

Similarly, very few architectures have a true memory/memory swap instruction (Motorola M68K, SPARC 32-bit). The x86 XCHG instruction (and most other architectures with a similar instruction) only works between a register and memory location. In this case, there is a race between loading the register and performing the swap (discussed shortly).

Either a true memory/memory swap instruction or a DCAS would provide the ability to atomically swap two memory locations, but unfortunately neither of these instructions are supported on the architectures used in this work. There are lock-free implemetions of DCAS, or more generally K-word CAS (also known as MCAS or CASN) [25] and LLX/SCX [8] that can be used to provide the desired atomic swap capability. However, these lock-free implementations were not used as it is advantageous in the work stealing case to let an attempted atomic swap fail instead of retrying. Hence, a novel atomic swap specific to the actor use case is simulated, called *queue pointer compare-and-set (swap)* (QPCAS). Note that this swap is *not* lock-free. The QPCAS is effectively a DCAS special cased in a few ways:

1. It works on two separate memory locations, and hence, is logically the same as.

    ```
    bool QPCAS( T * dst, T * src ) {
        return DCAS( dest, src, *dest, *src, *src, *dest );
    }
    ```

2. The values swapped are never null pointers, so a null pointer can be used as an intermediate value during the swap. As such, null is effectively used as a lock for the swap.

Figure 6.10 shows the C∀ pseudocode for the QPCAS. In detail, a thief performs the following steps to swap two pointers:

1. Stores local copies of the two pointers to be swapped.
2. Verifies the stored copy of the victim queue pointer, vic_queue, is valid. If vic_queue is null, then the victim queue is part of another swap so the operation fails. No state has changed at this point so the thief just returns. Note that thieves only set their own queues pointers to null when stealing, and queue pointers are not set to null anywhere else. As such, it is impossible for my_queue to be null since each worker owns a disjoint range of the queue array. Hence, only vic_queue is checked for null.
3. Attempts to atomically set the thief's queue pointer to null. The CAS only fails if the thief's queue pointer is no longer equal to my_queue, which implies this thief has become a victim and its queue has been stolen. At this point, the thief-turned-victim fails, and since it has not changed any state, it just returns false. If the CAS succeeds, the thief's queue pointer is now null. Only thieves look at other worker's queue ranges, and whenever thieves need to dereference a queue pointer, it is checked for null. A thief can only see the null queue pointer when looking for queues to steal or attempting a queue swap. If looking for queues, the thief will skip the null pointer, thus only the queue swap case needs to be considered for correctness.
4. Attempts to atomically set the victim's queue pointer to my_queue. If the CAS succeeds, the victim's queue pointer has been set and the swap can no longer fail. If the CAS fails, the thief's queue pointer must be restored to its previous value before returning.
5. Sets the thief's queue pointer to vic_queue completing the swap.

Theorem 1 *QPCAS is correct in both the success and failure cases.*

To verify sequential correctness, Figure 6.11 shows a simplified QPCAS. Step 2 is missing in the sequential example since it only matters in the concurrent context. By inspection, the sequential swap copies each pointer being swapped, and then the original values of each pointer are reset using the copy of the other pointer.

The concurrent proof of correctness is shown through the existence of an invariant. The invariant states when a queue pointer is set to 0p by a thief, then the next write to the pointer can only be performed by the same thief. This is effictively a mutual exclusion condition for the later write. To show that this invariant holds, it is shown that it is true at each step of the swap.

- Step 1 and 2 do not write, and as such, they cannot invalidate the invariant of any other thieves.
- In step 3, a thief attempts to write 0p to one of their queue pointers. This queue pointer cannot be 0p. As stated above, my_queue is never equal to 0p since thieves only write 0p to queue pointers from their own queue range and all worker's queue ranges are disjoint. As such, step 3 upholds the invariant, since in a failure case no write occurs, and in the success case, the value of the queue pointer is guaranteed to not be 0p.

```
bool try_swap_queues( worker & this, uint victim_idx, uint my_idx ) with(this) {
    // Step 1: mailboxes is the shared array of all sharded queues
    work_queue * my_queue = mailboxes[my_idx];
    work_queue * vic_queue = mailboxes[victim_idx];

    // Step 2 If the victim queue is 0p then they are in the process of stealing so return false
    // 0p is Cforall's equivalent of C++'s nullptr
    if ( vic_queue == 0p ) return false;

    // Step 3 Try to set our own (thief's) queue ptr to be 0p.
    // If this CAS fails someone stole our (thief's) queue so return false
    if ( !CAS( &mailboxes[my_idx], &my_queue, 0p ) )
        return false;

    // Step 4: Try to set victim queue ptr to be our (thief's) queue ptr.
    // If it fails someone stole the other queue, so fix up then return false
    if ( !CAS( &mailboxes[victim_idx], &vic_queue, my_queue ) ) {
        mailboxes[my_idx] = my_queue; // reset queue ptr back to prev val
        return false;
    }
    // Step 5: Successfully swapped.
    // Thief's ptr is 0p so no one touches it
    // Write back without CAS is safe
    mailboxes[my_idx] = vic_queue;
    return true;
}
```

Figure 6.10: QPCAS Concurrent

```
void swap( uint victim_idx, uint my_idx ) {
    // Step 1:
    work_queue * my_queue = mailboxes[my_idx];
    work_queue * vic_queue = mailboxes[victim_idx];
    // Step 3:
    mailboxes[my_idx] = 0p;
    // Step 4:
    mailboxes[victim_idx] = my_queue;
    // Step 5:
    mailboxes[my_idx] = vic_queue;
}
```

Figure 6.11: QPCAS Sequential

- In step 4, the thief attempts to write my_queue to the victim's queue pointer. If the current value of the victim's queue pointer is 0p, then the CAS fails since vic_queue cannot be equal to 0p because of the check in step 2. Therefore, when the CAS succeeds, the value of the victim's queue pointer must not be 0p. As such, the write never overwrites a value of 0p, hence the invariant is held in the CAS of step 4.
- The write back to the thief's queue pointer that happens in the failure case of step 4 and in step 5 hold the invariant since they are the subsequent write to a 0p queue pointer and are being set by the same thief that set the pointer to 0p.

Given this informal proof of invariance it can be shown that the successful swap is correct. Once a thief atomically sets their queue pointer to be 0p in step 3, the invariant guarantees that that pointer does not change. In the success case of step 4, it is known the value of the victim's queue-pointer, which is not overwritten, must be vic_queue due to the use of CAS. Given that the pointers all have unique memory locations (a pointer is never swapped with itself), this first write of the successful swap is correct since it can only occur when the pointer has not changed. By the invariant, the write back in the successful case is correct since no other worker can write to the 0p pointer. In the failed case of step 4, the outcome is correct in steps 2 and 3 since no writes have occurred so the program state is unchanged. Therefore, the program state is safely restored to the state it had prior to the 0p write in step 3, because the invariant makes the write back to the 0p pointer safe. Note that the pointers having unique memory locations prevents the ABA problem.

6.4.4 Victim Selection

In any work stealing algorithm, thieves use a heuristic to determine which victim to choose. Choosing this algorithm is difficult and can have implications on performance. There is no one selection heuristic known to be best for all workloads. Recent work focuses on locality aware scheduling in actor systems [2, 62]. However, while locality-aware scheduling provides good performance on some workloads, randomized selection performs better on other workloads [2]. Since locality aware scheduling has been explored recently, this work introduces a heuristic called *longest victim* and compares it to randomized work stealing.

The longest-victim heuristic maintains a timestamp per executor thread that is updated every time a worker attempts to steal work. The timestamps are generated using rdtsc [32] and are stored in a shared array, with one index per worker. Thieves then attempt to steal from the worker with the oldest timestamp, which is found by performing a linear search across the array of timestamps. The intuition behind this heuristic is that the slowest worker receives help via work stealing until it becomes a thief, which indicates that it has caught up to the pace of the rest of the workers. This heuristic should ideally result in lowered latency for message sends to victim workers that are overloaded with work. It must be acknowledged that this linear search could cause a lot of cache coherence traffic. Future work on this heuristic could include introducing a search that has less impact on caching. A negative side-effect of this heuristic is that if multiple thieves steal at the same time, they likely steal from the same victim, which increases the chance of contention. However, given that workers have multiple queues, often in the tens or hundreds of queues, it is rare for two thieves to attempt stealing from the same queue. This

approach may seem counter-intuitive, but in cases with not enough work to steal, the contention among thieves can result in less stealing, due to failed swaps. This can be beneficial when there is not enough work for all the stealing to be productive. This heuristic does not boast performance over randomized victim selection, but it is comparable (see Section 6.6.4). However, it constitutes an interesting contribution as it shows that adding some complexity to the heuristic of the stealing fast-path does not affect mainline performance, paving the way for more involved victim selection heuristics.

6.5 Safety and Productivity

C∀'s actor system comes with a suite of safety and productivity features. Most of these features are only present in C∀'s debug mode, and hence, have zero-cost in no-debug mode. The suite of features include the following.

- Static-typed message sends: If an actor does not support receiving a given message type, the receive call is rejected at compile time, preventing unsupported messages from being sent to an actor.
- Detection of message sends to Finished/Destroyed/Deleted actors: All actors receive a ticket from the executor at creation that assigns them to a specific mailbox queue of a worker. The maximum integer value of the ticket is reserved to indicate an actor is terminated, and assigned to an actor's ticket at termination. Any subsequent message sends to this terminated actor results in an error.
- Actors cannot be created before the executor starts: Since the executor distributes mailbox tickets, correctness implies it must be created *before* any actors so it can give out the tickets.
- When an executor is configured, $M >= N$. That is, each worker must receive at least one mailbox queue, since otherwise a worker cannot receive any work without a queue pull messages from.
- Detection of unsent messages: At program termination, a warning is printed for all deallocated messages that are not sent. Since the Finished allocation status is unused for messages, it is used internally to detect if a message has been sent. Deallocating a message without sending it could indicate problems in the program design.
- Detection of messages sent but not received: As discussed in Section 6.3, once all actors have terminated, shutdown is communicated to the executor threads via a status flag. During termination of the executor threads, each worker checks its mailbox queues for any messages. If so, an error is reported. Messages being sent but not received means their allocation action has not occur and their payload is not delivered. Missed deallocations can lead to memory leaks and unreceived payloads can mean logic problems.

In addition to these features, the C∀'s actor system comes with a suite of statistics that can be toggled on and off when C∀ is built. These statistics have minimal impact on the actor system's performance since they are counted independently by each worker thread. During shutdown of the actor system, these counters are aggregated sequentially. The statistics measured are as follows.

Actors Created Includes both actors made in the program main and ones made by other actors.

Messages Sent and Received Includes termination messages send to the executor threads.

Gulps Gulps across all worker threads.

Average Gulp Size Average number of messages in a gulped queue.

Missed gulps Missed gulps due to the current queue being processed by another worker.

Steal attempts All worker thread attempts to steal work.

Steal failures (no candidates) Work stealing failures due to selected victim not having any non-empty or non-being-processed queues.

Steal failures (failed swaps) Work stealing failures due to the two-stage atomic-swap failing.

Messages stolen Aggregate number of messages in stolen queues.

Average steal size Average number of messages across stolen queues.

These statistics enable a user to make informed choices about how to configure the executor or how to structure the actor program. For example, if there are a lot of messages being stolen relative to the number of messages sent, it indicates that the workload is heavily imbalanced across executor threads. Another example is if the average gulp size is very high, it indicates the executor needs more queue sharding, *i.e.*, increase M.

Finally, the poison-pill messages and receive routines, shown earlier in Figure 6.6, are a convenience for programmers and can be overloaded to have specific behaviour for derived actor types.

6.6 Performance

The performance of the C∀'s actor system is tested using a suite of microbenchmarks, and compared with other actor systems. Most of the benchmarks are the same as those presented in [11], with a few additions. This work compares with the following actor systems: C∀ 1.0, μC++ 7.0.0, Akka Typed 2.7.0, CAF 0.18.6, and ProtoActor-Go v0.0.0-20220528090104-f567b547ea07. Akka Classic is omitted as Akka Typed is their newest version and seems to be the direction they are headed. The experiments are run on two popular architectures:

1. Supermicro SYS–6029U–TR4 Intel Xeon Gold 5220R 24–core socket, hyper-threading $\times$ 2 sockets (96 processing units), running Linux v5.8.0–59–generic
2. Supermicro AS–1123US–TR4 AMD EPYC 7662 64–core socket, hyper-threading $\times$ 2 sockets (256 processing units), running Linux v5.8.0–55–generic

The benchmarks are run on 1–48 cores. On the Intel, with 24 core sockets, there is the choice to either hop sockets or use hyperthreads on the same socket. Either choice causes a blip in performance, which is seen in the subsequent performance graphs. The choice in this work is to use hyperthreading instead of hopping sockets for experiments with more than 24 cores.

All benchmarks are run 5 times and the median is taken. Error bars showing the 95% confidence intervals appear on each point in the graphs. The confidence intervals are calculated using bootstrapping to avoid normality assumptions. If the confidence bars are small enough, they may

Table 6.1: Static Actor/Message Performance: message send, program memory (lower is better)

	C∀ (100M)	μC++ (100M)	CAF (10M)	Akka (100M)	ProtoActor (100M)
AMD	23ns	42ns	2780ns	67ns	68ns
Intel	25ns	40ns	1699ns	67ns	90ns

Table 6.2: Dynamic Actor/Message Performance: message send, program memory (lower is better)

	C∀ (20M)	μC++ (20M)	CAF (2M)	Akka (2M)	ProtoActor (2M)
AMD	46ns	74ns	9519ns	8409ns	5513ns
Intel	59ns	79ns	8491ns	5451ns	4167ns

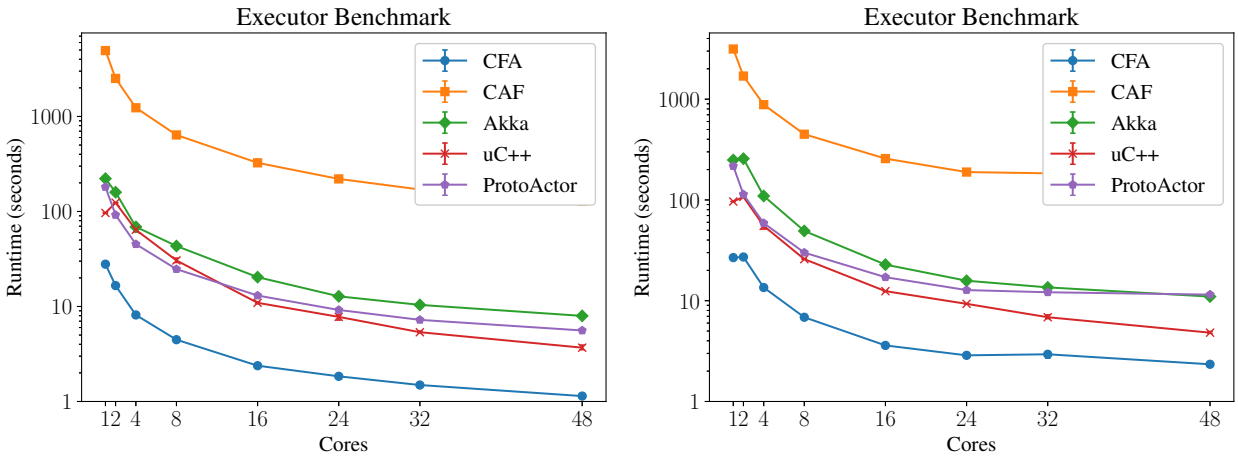
be obscured by the data point. In this section, μC++ is compared to C∀ frequently, as the actor system in C∀ is heavily based off of μC++'s actor system. As such, the performance differences that arise are largely due to the contributions of this work. Future work is to port some of the new C∀ work back to μC++.

6.6.1 Message Sends

Message sending is the key component of actor communication. As such, latency of a single message send is the fundamental unit of fast-path performance for an actor system. The static and dynamic send-benchmarks evaluate the average latency for a static actor/message send and a dynamic actor/message send. In the static-send benchmark, a message and actor are allocated once and then the message is sent to the same actor 100 million (100M) times. The average latency per message send is then calculated by dividing the duration by the number of sends. This benchmark evaluates the cost of message sends in the actor use case where all actors and messages are allocated ahead of time and do not need to be created dynamically during execution. The CAF static-send benchmark only sends a message 10M times to avoid extensively long run times.

In the dynamic-send benchmark, the same experiment is used, but for each send, a new actor and message is allocated. This benchmark evaluates the cost of message sends in the other common actor pattern where actors and messages are created on the fly as the actor program tackles a workload of variable or unknown size. Since dynamic sends are more expensive, this benchmark repeats the actor/message creation and send 20M times (μC++, C∀), or 2M times (Akka, CAF, ProtoActor), to give an appropriate benchmark duration.

The results from the static/dynamic-send benchmarks are shown in Tables 6.1 and 6.2, respectively. C∀ has the best results in both benchmarks, largely due to the copy queue removing the majority of the envelope allocations. Additionally, the receive of all messages sent in C∀ is statically known and is determined via a function pointer cast, which incurs no runtime cost. All the other systems use virtual dispatch to find the correct behaviour at message send. This operation actually requires two virtual dispatches, which is an additional runtime send cost. Note that Akka also statically checks message sends, but still uses the Java virtual system. In the static-



(a) AMD Executor Benchmark

(b) Intel Executor Benchmark

Figure 6.12: Executor benchmark comparing actor systems (lower is better).

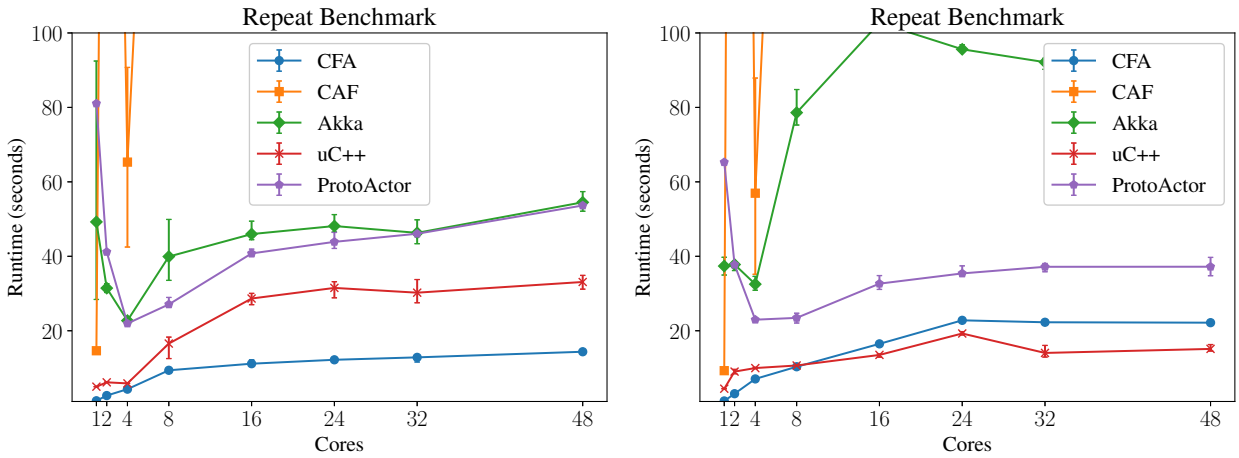
send benchmark, all systems except CAF have static send costs that are in the same ballpark, only varying by 70ns. In the dynamic-send benchmark, all systems experience slower message sends, due to the memory allocations. However, Akka and ProtoActor, slow down by two-orders of magnitude. This difference is likely a result of Akka and ProtoActor's garbage collection, which results in performance delays for allocation-heavy workloads, whereas μC++ and C$\forall$ have explicit allocation/deallocation. Tuning the garage collection might reduce garbage-collection cost, but this exercise is beyond the scope of this work.

6.6.2 Executor

The benchmarks in this section are designed to stress the executor. The executor is the scheduler of an actor system and is responsible for organizing the interaction of executor threads to service the needs of an actor workload. Three benchmarks are run: executor, repeat, and high-memory watermark.

The executor benchmark creates 40,000 actors, organizes the actors into adjacent groups of 100, where an actor sends a message to each group member, including itself, in round-robin order, and repeats the sending cycle 400 times. This microbenchmark is designed to flood the executor with a large number of messages flowing among actors. Given there is no work associated with each message, other than sending more messages, the intended bottleneck of this experiment is the executor message send process.

Figures 6.12(b) and 6.12(a) show the results of the AMD and Intel executor benchmark. There are three groupings of results, and the difference between AMD and Intel is small. CAF is significantly slower than the other actor systems; followed by a tight grouping of μC++, ProroActor, and Akka; and finally C$\forall$ with the lowest runtime relative to its peers. The difference in runtime between μC++ and C$\forall$ is largely due to the copy queue described in Section 6.3.1. The copy queue both reduces and consolidates allocations, heavily reducing contention on the memory allocator. Additionally, due to the static typing in C$\forall$'s actor system, there is no expensive dynamic (RTTI) casts that occur in μC++ to discriminate messages types. Note that while



(a) AMD Repeat Benchmark

(b) Intel Repeat Benchmark

Figure 6.13: The repeat benchmark comparing actor systems (lower is better).

dynamic cast is relatively inexpensive, the remaining send cost in both μC++ and C$\forall$ is small; hence, the relative cost for the RTTI in μC++ is significant.

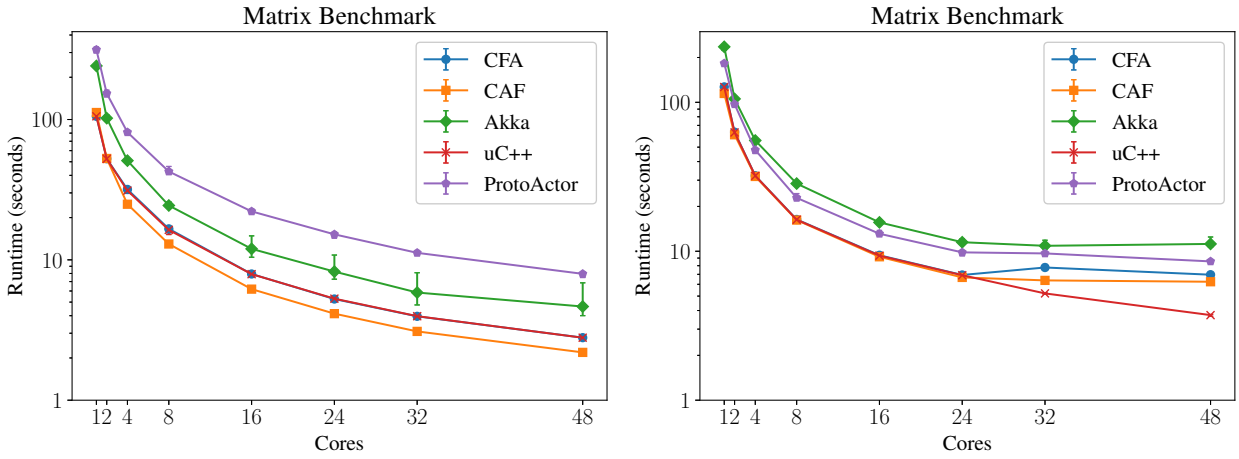
The repeat benchmark also evaluates the executor. It stresses the executor's ability to withstand contention on queues. The repeat benchmark repeatedly fans out messages from a single client to 100,000 servers who then respond back to the client. The scatter and gather repeats 200 times. The messages from the servers to the client all come to the same mailbox queue associated with the client, resulting in high contention among servers. As such, this benchmark does not scale with the number of processors, since more processors result in higher contention on the single mailbox queue.

Figures 6.13(a) and 6.13(b) show the results of the AMD and Intel repeat benchmark. The results are spread out more, and there is a difference between AMD and Intel. Again, CAF is significantly slower than the other actor systems. To keep the graphs readable, the y-axis was cut at 100 seconds; as the core count increases from 8-32, CAF ranges around 200 seconds on AMD and between 300-1000 seconds on the Intel. On the AMD there is a tight grouping of uC++, ProtoActor, and Akka; on the Intel, uC++, ProtoActor, and Akka are spread out. Finally, C$\forall$ runs consistently on both of the AMD and Intel, and is faster than μC++ on the AMD, but slightly slower on the Intel. Here, gains from using the copy queue are much less apparent.

Table 6.3 shows the high memory watermark of the actor systems when running the executor benchmark on 48 cores measured using the `time` command. C$\forall$'s high watermark is slightly higher than the other non-garbage collected systems μC++ and CAF. This increase is from the over-allocation in the copy-queue data-structure with lazy deallocation. Whereas, the per envelope allocations of μC++ and CFA allocate exactly the amount of storage needed and eagerly deallocate. The extra storage is the standard tradeoff of time versus space, where C$\forall$ shows better performance. As future work, tuning parameters can be provided to adjust the frequency and/or size of the copy-queue expansion.

Table 6.3: Executor Program Memory High Watermark

	C∀	μC++	CAF	Akka	ProtoActor
AMD	298MB	185MB	165MB	8046MB	563MB
Intel	331MB	116MB	139MB	7714MB	549MB



(a) AMD Matrix Benchmark

(b) Intel Matrix Benchmark

Figure 6.14: The matrix benchmark comparing actor systems (lower is better).

6.6.3 Matrix Multiply

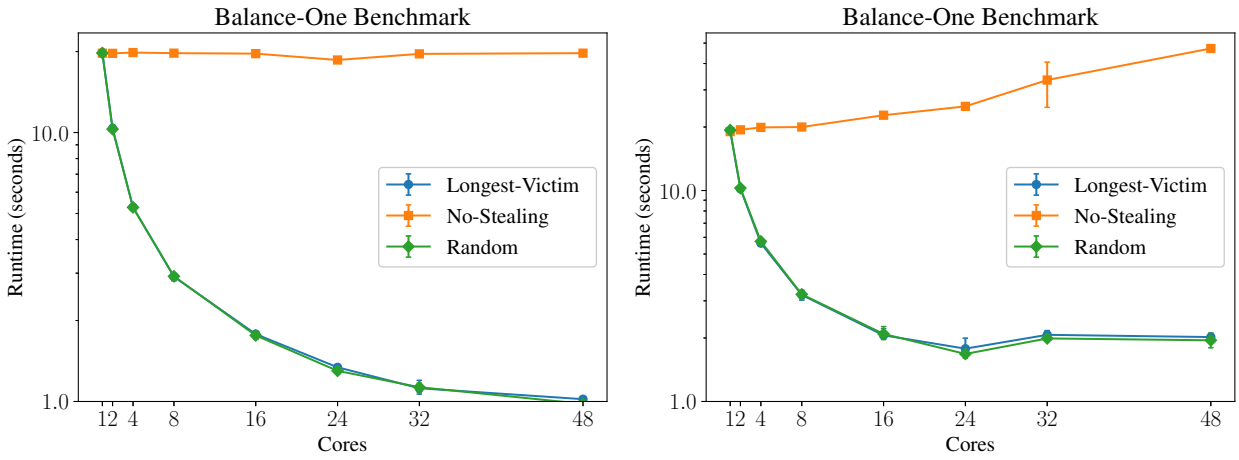
The matrix-multiply benchmark evaluates the actor systems in a practical application, where actors concurrently multiply two matrices. In detail, given $Z_{m,r} = X_{m,n} \cdot Y_{n,r}$, the matrix multiply is defined as:

$$X_{i,j} \cdot Y_{j,k} = \left(\sum_{c=1}^{j} X_{row,c} Y_{c,column} \right)_{i,k}$$

The majority of the computation in this benchmark involves computing the final matrix, so this benchmark stresses the actor systems' ability to have actors run work, rather than stressing the message sending system, and might trigger some work stealing if a worker finishes early.

The matrix-multiply benchmark has input matrices X and Y, which are both 3072 by 3072 in size. An actor is made for each row of X and sent a message indicating the row of X and the column of Y to calculate a row of the result matrix Z. Because Z is contiguous in memory, there can be small cache write-contention at the row boundaries.

Figures 6.14(a) and 6.14(b) show the matrix-multiply results. There are two groupings with Akka and ProtoActor being slightly slower than μC++, C∀, and CAF. On the Intel, there is an unexplained divergence between μC++ and C∀/CAF at 24 cores. Given that the bottleneck of this benchmark is the computation of the result matrix, all executors perform well on this embarrassingly parallel application. Hence, the results are tightly clustered across all actor systems. This result also suggests CAF has a good executor but poor message passing, which results in its poor performance in the other message-passing benchmarks.



(a) AMD C∀ Balance-One Benchmark

(b) Intel C∀ Balance-One Benchmark

Figure 6.15: The balance-one benchmark comparing stealing heuristics (lower is better).

6.6.4 Work Stealing

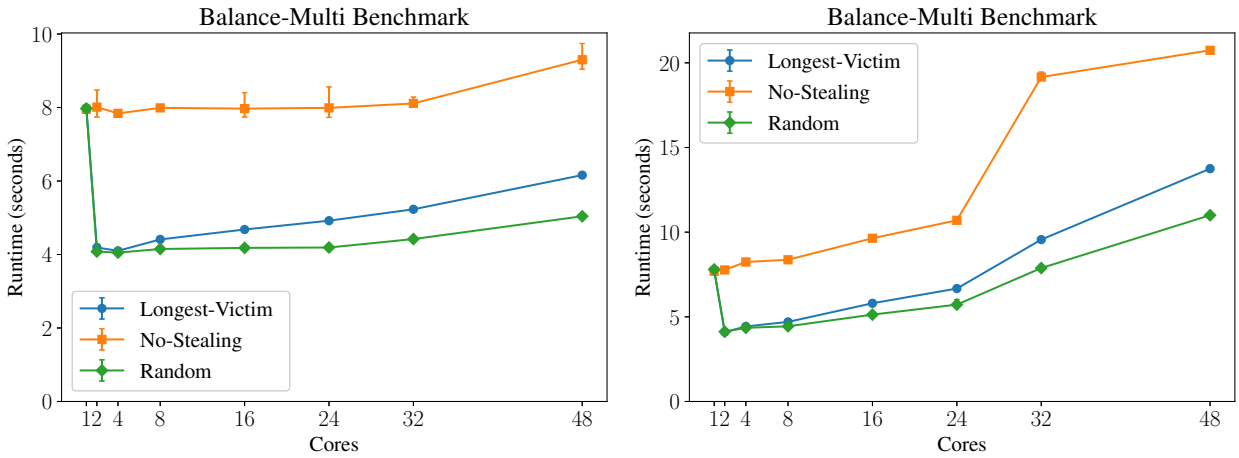
C∀'s work stealing mechanism uses the longest-victim heuristic, introduced in Section 6.4.4. In this performance section, C∀'s approach is first tested in isolation on a pathological unbalanced benchmark, then with other actor systems on general benchmarks.

Two pathological unbalanced cases are created, and compared using vanilla and randomized work stealing in C∀. These benchmarks adversarially take advantage of the round-robin assignment of actors to workers by loading actors only on specific cores (there is one worker per core). The workload on the loaded cores is the same as the executor benchmark described in 6.6.2, but with fewer rounds.

The balance-one benchmark loads all the work on a single core, whereas the balance-multi loads all the work on half the cores (every other core). Given this layout, the ideal speedup of work stealing in the balance-one case should be $N/N - 1$ where N is the number of threads; in the balance-multi case, the ideal speedup is 0.5. Note that in the balance-one benchmark, the workload is fixed so decreasing runtime is expected; in the balance-multi experiment, the workload increases with the number of cores so an increasing or constant runtime is expected.

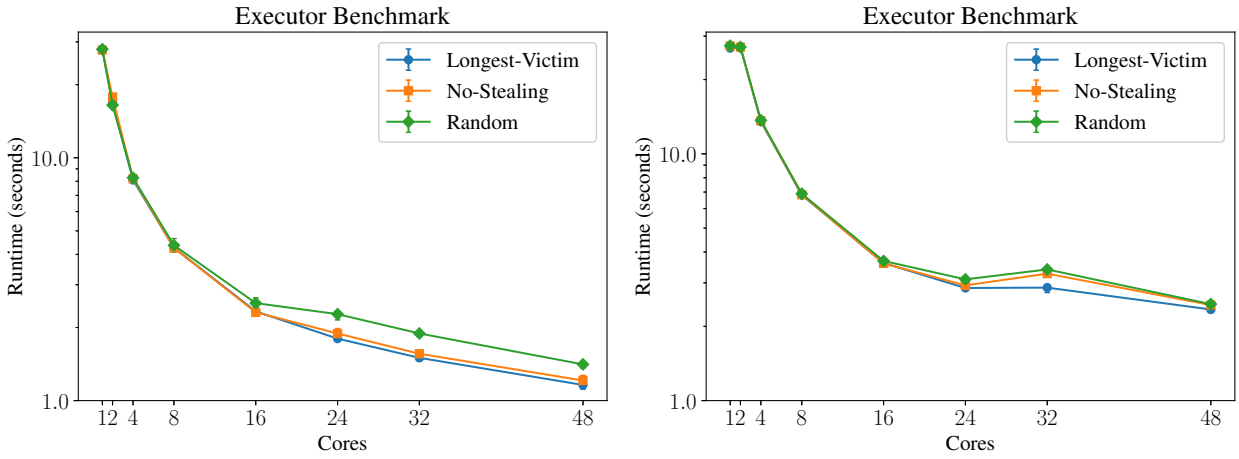
For the balance-one benchmark on AMD in Figure 6.15(a), the performance bottoms out at 32 cores onwards likely due to the amount of work becoming less than the cost to steal it and move it across cores and cache. On Intel in Figure 6.15(b), above 32 cores the performance gets worse for all variants due to hyperthreading. Here, the longest-victim and random heuristic are the same. Note that the non-stealing variation of balance-one slows down slightly (no decrease in graph) as the cores increase, since a few *dummy* actors are created for each of the extra cores beyond the first to adversarially layout all loaded actors on the first core.

For the balance-multi benchmark in Figures 6.16(a) and 6.16(b), the random heuristic outperforms the longest victim. The reason is that the longest-victim heuristic has a higher stealing cost as it needs to maintain timestamps and look at all timestamps before stealing. Additionally, a performance cost on the Intel is observed when hyperthreading kicks in after 24 cores in



(a) AMD C∀ Balance-Multi Benchmark (b) Intel C∀ Balance-Multi Benchmark

Figure 6.16: The balance-multi benchmark comparing stealing heuristics (lower is better).



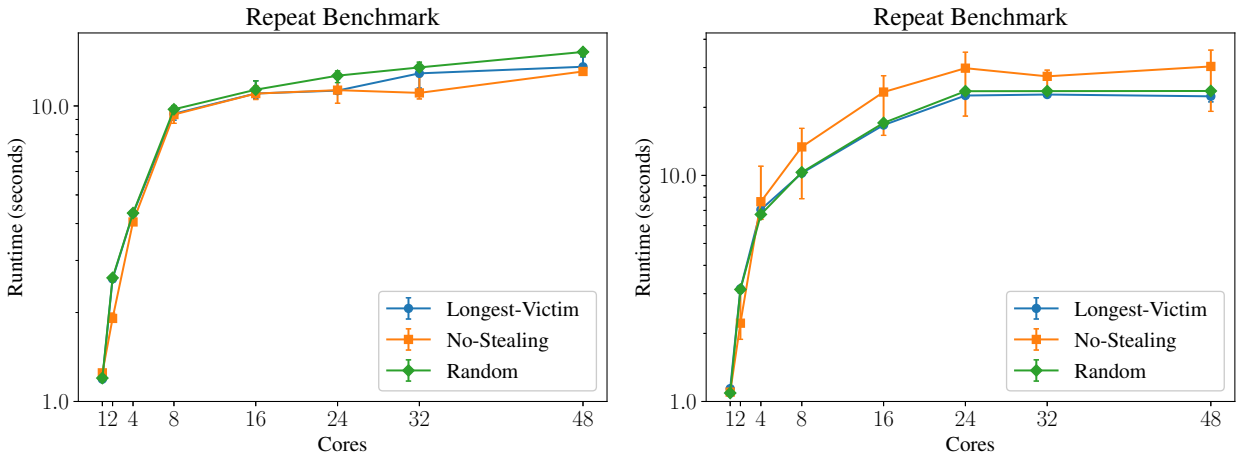
(a) AMD C∀ Executor Benchmark (b) Intel C∀ Executor Benchmark

Figure 6.17: Executor benchmark comparing C∀ stealing heuristics (lower is better).

Figure 6.16(b).

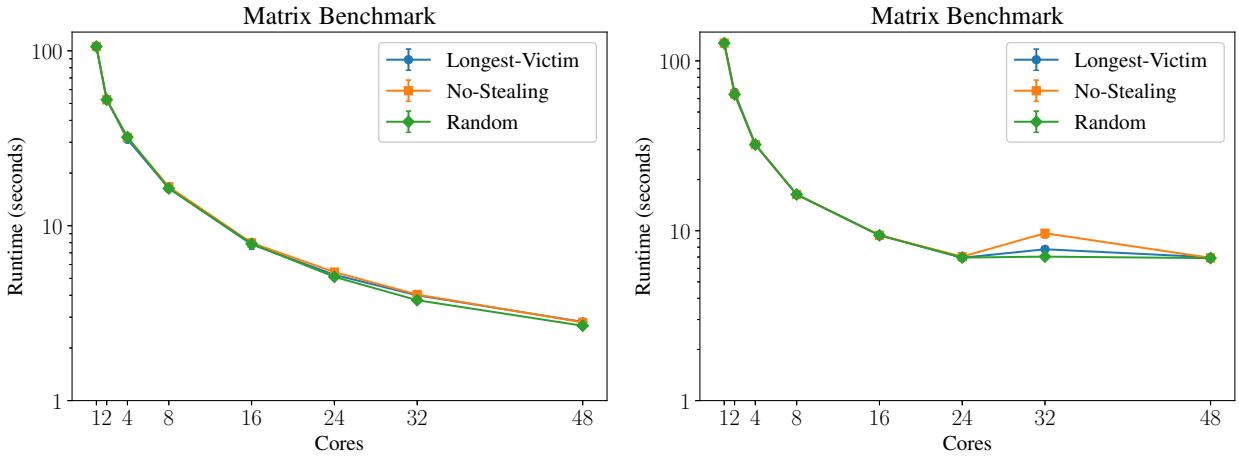
Figures 6.17(a) and 6.17(b) show the effects of the stealing heuristics for the executor benchmark. For the AMD, in Figure 6.17(a), the random heuristic falls slightly behind the other two, but for the Intel, in Figure 6.17(b), the runtime of all heuristics are nearly identical to each other, except after crossing the 24-core boundary.

Figures 6.18(a) and 6.18(b) show the effects of the stealing heuristics for the repeat benchmark. This benchmark is a pathological case for work stealing actor systems, as the majority of work is being performed by the single actor conducting the scatter/gather. The single actor (the client) of this experiment is long running and maintains a lot of state, as it needs to know the handles of all the servers. When stealing the client or its respective queue (in C∀'s inverted model), moving the client incurs a high cost due to cache invalidation. This worst-case steal is likely to happen since there is no other work in the system between scatter/gather rounds.



(a) AMD C∀ Repeat Benchmark (b) Intel C∀ Repeat Benchmark

Figure 6.18: The repeat benchmark comparing C∀ stealing heuristics (lower is better).



(a) AMD C∀ Matrix Benchmark (b) Intel C∀ Matrix Benchmark

Figure 6.19: The matrix benchmark comparing C∀ stealing heuristics (lower is better).

However, all heuristics are comparable in performance on the repeat benchmark. This result is surprising especially for the No-Stealing variant, which should have better performance than the stealing variants. However, stealing happens lazily and fails fast, hence the queue containing the long-running client actor is rarely stolen.

Finally, Figures 6.19(a) and 6.19(b) show the effects of the stealing heuristics for the matrix-multiply benchmark. Here, there is negligible performance difference across stealing heuristics, because of the long-running workload of each message.

In theory, work stealing might improve performance marginally for the matrix-multiply benchmark. Since all row actors cannot be created simultaneously at startup, they correspondingly do not shutdown simultaneously. Hence, there is a small window at the start and end with idle workers so work stealing might improve performance. For example, in 6.14(a), CAF is slightly better than μC++ and C∀, but not on the Intel. Hence, it is difficult to attribute the AMD gain to

the aggressive work stealing in CAF.

Chapter 7

Waituntil

Consider the following motivating problem. There are N stalls (resources) in a bathroom and there are M people (threads) using the bathroom. Each stall has its own lock since only one person may occupy a stall at a time. Humans solve this problem in the following way. They check if all of the stalls are occupied. If not, they enter and claim an available stall. If they are all occupied, people queue and watch the stalls until one is free, and then enter and lock the stall. This solution can be implemented on a computer easily, if all threads are waiting on all stalls and agree to queue.

Now the problem is extended. Some stalls are wheelchair accessible and some stalls have gender identification. Each person (thread) may be limited to only one kind of stall or may choose among different kinds of stalls that match their criteria. Immediately, the problem becomes more difficult. A single queue no longer solves the problem. What happens when there is a stall available that the person at the front of the queue cannot choose? The naïve solution has each thread spin indefinitely continually checking every matching kind of stall(s) until a suitable one is free. This approach is insufficient since it wastes cycles and results in unfairness among waiting threads as a thread can acquire the first matching stall without regard to the waiting time of other threads. Waiting for the first appropriate stall (resource) that becomes available without spinning is an example of *synchronous multiplexing*: the ability to wait synchronously for one or more resources based on some selection criteria.

7.1 History of Synchronous Multiplexing

There is a history of tools that provide synchronous multiplexing. Some well known synchronous multiplexing tools include Unix system utilities: select [49], poll [48], and epoll [46], and the select statement provided by Go [58], Ada [1, § 9.7], and μC++ [10, § 3.3.1]. The concept and theory surrounding synchronous multiplexing was introduced by Hoare in his 1985 book, Communicating Sequential Processes (CSP) [29],

> A communication is an event that is described by a pair $c.v$ where c is the name of the channel on which the communication takes place and v is the value of the message which passes. [29, p. 113]

The ideas in CSP were implemented by Roscoe and Hoare in the language Occam [57].

```
when( G1 )
    waituntil( R1 ) S1
or when( G2 )
    waituntil( R2 ) S2
or when( G3 )
    waituntil( R3 ) S3
```

(a) Guards

```
if ( G1 )
    if ( G2 )
        if ( G3 ) waituntil( R1 ) S1 or waituntil( R2 ) S2 or waituntil( R3 ) S3
        else waituntil( R1 ) S1 or waituntil( R2 ) S2
    else
        if ( G3 ) waituntil( R1 ) S1 or waituntil( R3 ) S3
        else waituntil( R1 ) S1
else
    if ( G2 )
        if ( G3 ) waituntil( R2 ) S2 or waituntil( R3 ) S3
        else waituntil( R2 ) S2
    else
        if ( G3 ) waituntil( R3 ) S3
```

(b) Simulated Guards

Figure 7.1: C∀ guard simulated with **if** statement.

Both CSP and Occam include the ability to wait for a *choice* among receiver channels and *guards* to toggle which receives are valid. For example,

(G1(x) → P | G2(y) → Q)

waits for either channel x or y to have a value, if and only if guards G1 and G2 are true; if only one guard is true, only one channel receives, and if both guards are false, no receive occurs. In detail, waiting for one resource out of a set of resources can be thought of as a logical exclusive-or over the set of resources. Guards are a conditional operator similar to an **if**, except they apply to the resource being waited on. If a guard is false, then the resource it guards is not in the set of resources being waited on. If all guards are false, the ALT, Occam's synchronous multiplexing statement, does nothing and the thread continues. Guards can be simulated using **if** statements as shown in [57, rule 2.4, p 183]

ALT(b & g P, G) = IF (b ALT(g P, G), ¬ b ALT(G)) (boolean guard elim).

but require $2^N - 1$ **if** statements, where N is the number of guards. The exponential blowup comes from applying rule 2.4 repeatedly, since it works on one guard at a time. Figure 7.1 shows in C∀ an example of applying rule 2.4 for three guards. Also, notice the additional code duplication for statements S1, S2, and S3.

When discussing synchronous multiplexing implementations, the resource being multiplexed is important. While CSP waits on channels, the earliest known implementation of synchronous multiplexing is Unix's select [49], multiplexing over file descriptors, which conceptually differ from channels in name only. The select system-call is passed three sets of file descriptors (read, write, exceptional) to wait on and an optional timeout. select blocks until either some subset of file descriptors are available or the timeout expires. All file descriptors that are ready are returned by modifying the argument sets to only contain the ready descriptors.

This early implementation differs from the theory presented in CSP: when the call from select returns it may provide more than one ready file descriptor. As such, select has logical-or multiplexing semantics, whereas the theory described exclusive-or semantics. It is possible to achieve exclusive-or semantics with select by arbitrarily operating on only one of the returned

```
task type buffer is -- thread type
    ... -- buffer declarations
    count : integer := 0;
begin -- thread starts here
    loop
        select
            when count < Size => -- guard
            accept insert( elem : in ElemType ) do  -- method
                ... -- add to buffer
                count := count + 1;
            end;
            -- executed if this accept called
        or
            when count > 0 => -- guard
            accept remove( elem : out ElemType ) do  -- method
                ... --remove and return from buffer via parameter
                count := count - 1;
            end;
            -- executed if this accept called
        or delay 10.0;  -- unblock after 10 seconds without call
        or else -- do not block, cannot appear with delay
        end select;
    end loop;
end buffer;
var buf : buffer; -- create task object and start thread in task body
```

Figure 7.2: Ada Bounded Buffer

descriptors. select passes the interest set of file descriptors between application and kernel in the form of a worst-case sized bit-mask, where the worst-case is the largest numbered file descriptor. poll reduces the size of the interest sets changing from a bit mask to a linked data structure, independent of the file-descriptor values. epoll further reduces the data passed per call by keeping the interest set in the kernel, rather than supplying it on every call.

These early synchronous multiplexing tools interact directly with the operating system and others are used to communicate among processes. Later, synchronous multiplexing started to appear in applications, via programming languages, to support fast multiplexed concurrent communication among threads. An early example of synchronous multiplexing is the select statement in Ada [31, § 9.7]. This select allows a task object, with their own threads, to multiplex over a subset of asynchronous calls to its methods. The Ada select has the same exclusive-or semantics and guards as Occam ALT; however, it multiplexes over methods rather than channels.

Figure 7.2 shows the outline of a bounded buffer implemented with an Ada task. Note that a task method is associated with the **accept** clause of the **select** statement, rather than being a separate routine. The thread executing the loop in the task body blocks at the **select** until a call occurs to insert or remove. Then the appropriate **accept** method is run with the called arguments. Hence, the **select** statement provides rendezvous points for threads, rather than providing channels with message passing. The **select** statement also provides a timeout and **else** (nonblocking), which

changes synchronous multiplexing to asynchronous. Now the thread polls rather than blocks.

Another example of programming-language synchronous multiplexing is Go using a select statement with channels [58]. Figure 7.3(a) shows the outline of a bounded buffer administrator implemented with a Go routine. Here two channels are used for inserting and removing by client producers and consumers, respectively. (The done channel is used to synchronize with the program main.) Go's select has the same exclusive-or semantics as the ALT primitive from Occam and associated code blocks for each clause like ALT and Ada. However, unlike Ada and ALT, Go does not provide guards for the **case** clauses of the **select**. As such, the exponential blowup can be seen comparing Go and μC++ in Figure 7.3. Go also provides a timeout via a channel and a **default** clause like Ada **else** for asynchronous multiplexing.

Finally, μC++ provides synchronous multiplexing with Ada-style select over monitor and task methods with the _Accept statement [10, § 2.9.2.1], and over futures with the _Select statement [10, § 3.3.1]. The _Select statement extends the ALT/Go select by offering both **and** and **or** semantics, which can be used together in the same statement. Both _Accept and _Select statements provide guards for multiplexing clauses, as well as, timeout, and **else** clauses. Figure 7.3(b) shows the outline of a bounded buffer administrator implemented with μC++ _Accept statements.

There are other languages that provide synchronous multiplexing, including Rust's select! over futures [41], Java's allof/anyof over futures [14], OCaml's select over channels [50], and C++14's when_any over futures [55]. Note that while C++14 and Rust provide synchronous multiplexing, the implementations leave much to be desired as both rely on polling to wait on multiple resources.

7.2 Other Approaches to Synchronous Multiplexing

To avoid the need for synchronous multiplexing, all communication among threads/processes must come from a single source. For example, in Erlang each process has a single heterogeneous mailbox that is the sole source of concurrent communication, removing the need for synchronous multiplexing as there is only one place to wait on resources. Similar, actor systems circumvent the synchronous multiplexing problem as actors only block when waiting for the next message never in a behaviour. While these approaches solve the synchronous multiplexing problem, they introduce other issues. Consider the case where a thread has a single source of communication and it wants a set of N resources. It must sequentially request the N resources and wait for each response. During the receives for the N resources, it can receive other communication, and has to save and postpone these communications, or discard them.

7.3 C∀'s Waituntil Statement

The new C∀ synchronous multiplexing utility introduced in this work is the **waituntil** statement. There already exists a **waitfor** statement in C∀ that supports Ada-style synchronous multiplexing over monitor methods [17], so this **waituntil** focuses on synchronizing over other resources. All of the synchronous multiplexing features mentioned so far are monomorphic, only waiting on one kind of resource: Unix select supports file descriptors, Go's select supports channel operations,

```
insert := make( chan int )
remove := make( chan * int )
buffer := make( chan int Size )
done := make( chan int )
count := 0
func in_buf( int val ) {
    buffer <— val
    count++
}
func out_buf( int * ptr ) {
    *ptr := <—buffer
    count——
}
func BoundedBuffer {
    L: for {
        if ( count < Size && count > 0 ) {
            select { // wait for message
                case i := <— insert: in_buf( i )
                case p := <— remove: out_buf( p )
                case <— done: break L
            }
        } else if ( count < Size ) {
            select { // wait for message
                case i := <— insert: in_buf( i )
                case <— done: break L
            }
        } else ( count > 0 ) {
            select { // wait for message
                case p := <— remove: out_buf( p )
                case <— done: break L;
            }
        }
    }
    done <— 0
}
func main() {
    go BoundedBuffer() // start administrator
}
```

(a) Go

```
_Task BoundedBuffer {
    int * buffer;
    int front = back = count = 0;
  public:
    // ... constructor implementation
    void insert( int elem ) {
        buffer[front] = elem;
        front = ( front + 1 ) % Size;
        count += 1;
    }
    int remove() {
        int ret = buffer[back];
        back = ( back + 1 ) % Size;
        count -= 1;
        return ret;
    }
  private:
    void main() {
        for ( ;; ) {
            _Accept( ~buffer ) break;
            or _When ( count < Size ) _Accept( insert );
            or _When ( count > 0 ) _Accept( remove );
        }
    }
};
buffer buf; // start thread in main method
```

(b) μC++

Figure 7.3: Bounded Buffer

```
forall(T & | sized(T))
trait is_selectable {
    // For registering a waituntil stmt on a selectable type
    bool register_select( T &, select_node & );

    // For unregistering a waituntil stmt from a selectable type
    bool unregister_select( T &, select_node & );

    // on_selected is run on the selecting thread prior to executing
    // the statement associated with the select_node
    bool on_selected( T &, select_node & );
};
```

Figure 7.4: Trait for types that can be passed into C∀'s **waituntil** statement.

μC++'s select supports futures, and Ada's select supports monitor method calls. The C∀ **waituntil** is polymorphic and provides synchronous multiplexing over any objects that satisfy the trait in Figure 7.4. No other language provides a synchronous multiplexing tool polymorphic over resources like C∀'s **waituntil**.

Currently locks, channels, futures and timeouts are supported by the **waituntil** statement, and this set can be expanded through the is_selectable trait as other use-cases arise. The **waituntil** statement supports guard clauses, both **or** and **and** semantics, and timeout and **else** for asynchronous multiplexing. Figure 7.5 shows a C∀ **waituntil** usage, which is waiting for either Lock to be available *or* for a value to be read from Channel into i *and* for Future to be fulfilled *or* a timeout of one second. Note that the expression inside a **waituntil** clause is evaluated once at the start of the **waituntil** algorithm.

7.4 Waituntil Semantics

The **waituntil** semantics has two parts: the semantics of the statement itself, *i.e.*, **and**, **or**, **when** guards, and **else** semantics, and the semantics of how the **waituntil** interacts with types like locks, channels, and futures.

7.4.1 Statement Semantics

The **or** semantics are the most straightforward and nearly match those laid out in the ALT statement from Occam. The clauses have an exclusive-or relationship where the first available one is run and only one clause is run. C∀'s **or** semantics differ from ALT semantics: instead of randomly picking a clause when multiple are available, the first clause in the **waituntil** that is available is executed. For example, in the following example, if foo and bar are both available, foo is always selected since it comes first in the order of **waituntil** clauses.

```
future(int) bar, foo;
waituntil( foo ) { ... } or waituntil( bar ) { ... } // prioritize foo
```

```
future(int) Future;
channel(int) Channel;
owner_lock Lock;
int i = 0;

waituntil( Lock ) { ... }
or when( i == 0 ) waituntil( i << Channel ) { ... }
and waituntil( Future ) { ... }
or waituntil( timeout( 1`s ) ) { ... }
// else { ... }
```

Figure 7.5: Example of C∀'s waituntil statement

The reason for these semantics is that prioritizing resources can be useful in certain problems, such as shutdown. In the rare case where there is a starvation problem with the ordering, it possible to follow a **waituntil** with its reverse form, alternating which resource has the highest priority:

```
waituntil( foo ) { ... } or waituntil( bar ) { ... } // prioritize foo
waituntil( bar ) { ... } or waituntil( foo ) { ... } // prioritize bar
```

While this approach is not general for many resources, it handles many basic cases.

The C∀ **and** semantics match the **and** semantics of μC++ **_Select**. When multiple clauses are joined by **and**, the **waituntil** makes a thread wait for all to be available, but still runs the corresponding code blocks *as they become available*. When an **and** clause becomes available, the waiting thread unblocks and runs that clause's code-block, and then the thread waits again for the next available clause or the **waituntil** statement is now satisfied. This semantics allows work to be done in parallel while synchronizing over a set of resources, and furthermore, gives a good reason to use the **and** operator. If the **and** operator waited for all clauses to be available before running, it is the same as just acquiring those resources consecutively by a sequence of **waituntil** statements.

As with normal C expressions, the **and** operator binds more tightly than the **or**. To give an **or** operator higher precedence, parenthesis are used. For example, the following **waituntil** unconditionally waits for C and one of either A or B, since the **or** is given higher precedence via parenthesis.

```
( waituntil( A ) { ... }      // bind tightly to or
or waituntil( B ) { ... } )
and waituntil( C ) { ... }
```

The guards in the **waituntil** statement are called **when** clauses. Each boolean expression inside a **when** is evaluated *once* before the **waituntil** statement is run. Like Occam's ALT, the guards toggle clauses on and off, where a **waituntil** clause is only evaluated and waited on if the corresponding guard is true. In addition, the **waituntil** guards require some nuance since both **and** and **or** operators are supported (see Section 7.5.4). When a guard is false and a clause is removed, it can be thought of as removing that clause and its preceding operation from the statement. For example, in the following, the two **waituntil** statements are semantically equivalent.

```
when( true ) waituntil( A ) { ... }                waituntil( A ) { ... }
or when( false ) waituntil( B ) { ... }    ≡       and waituntil( C ) { ... }
and waituntil( C ) { ... }
```

The **else** clause on the **waituntil** has identical semantics to the **else** clause in Ada. If all resources are not immediately available and there is an **else** clause, the **else** clause is run and the thread continues.

7.4.2 Type Semantics

As mentioned, to support interaction with the **waituntil** statement a type must support the trait in Figure 7.4. The **waituntil** statement expects types to register and unregister themselves via calls to register_select and unregister_select, respectively. When a resource becomes available, on_selected is run, and if it returns false, the corresponding code block is not run. Many types do not need on_selected, but it is provided if a type needs to perform work or checks before the resource can be accessed in the code block. The register/unregister routines in the trait also return booleans. The return value of register_select is true, if the resource is immediately available and false otherwise. The return value of unregister_select is true, if the corresponding code block should be run after unregistration and false otherwise. The routine on_selected and the return value of unregister_select are needed to support channels as a resource. More detail on channels and their interaction with **waituntil** appear in Section 7.5.3.

The trait can be used directly by having a blocking object support the is_selectable trait, or it can be used indirectly through routines that take the object as an argument. When used indirectly, the object's routine returns a type that supports the is_selectable trait. This feature leverages C∀'s ability to overload on return type to select the correct overloaded routine for the **waituntil** context. Indirect support through routines is needed for types that want to support multiple operations such as channels that allow both reading and writing.

7.5 waituntil Implementation

The **waituntil** statement is not inherently complex, and Figure 7.6 shows the basic outline of the **waituntil** algorithm. Complexity comes from the consideration of race conditions and synchronization needed when supporting various primitives. The following sections use examples to fill in complexity details missing in Figure 7.6. After which, the full pseudocode for the **waituntil** algorithm is presented in Figure 7.9.

The basic steps of the algorithm are:

1. The **waituntil** statement declares N select_nodes, one per resource that is being waited on, which stores any **waituntil** data pertaining to that resource.
2. Each select_node is then registered with the corresponding resource.
3. The thread executing the **waituntil** then loops until the statement's predicate is satisfied. In each iteration, if the predicate is unsatisfied, the **waituntil** thread blocks. When another thread satisfies a resource clause (*e.g.*, sends to a channel), it unblocks the **waituntil** thread. This thread checks all clauses for completion, and any completed clauses have their code

```
select_nodes s[N];                                  // declare N select nodes
for ( node in s )                                   // register nodes
    register_select( resource, node );
while ( statement predicate not satisfied ) {       // check predicate
    // block until clause(s) satisfied
    for ( resource in waituntil statement ) {       // run true code blocks
        if ( resource is avail ) run code block
        if ( statement predicate is satisfied ) break;
    }
}
for ( node in s )                                   // deregister nodes
    if ( unregister_select( resource, node ) ) run code block
```

Figure 7.6: **waituntil** Implementation

blocks run. While checking clause completion, if enough clauses have been run such that the statement predicate is satisfied, the loop exits early.

4. Once the thread escapes the loop, the select_nodes are unregistered from the resources.

These steps give a basic overview of how the statement works. The following sections shed light on the specific changes and provide more implementation detail.

7.5.1 Locks

The C∀ runtime supports a number of spinning and blocking locks, *e.g.*, semaphore, MCS, futex, Go mutex, spinlock, owner, *etc*. Many of these locks satisfy the is_selectable trait, and hence, are resources supported by the **waituntil** statement. For example, the following waits until the thread has acquired lock l1 or locks l2 and l3.

```
owner_lock l1, l2, l3;
waituntil ( l1 ) { ... }
or waituntil( l2 ) { ... }
and waituntil( l3 ) { ... }
```

Implicitly, the **waituntil** is calling the lock acquire for each of these locks to establish a position in the lock's queue of waiting threads. When the lock schedules this thread, it unblocks and runs the code block associated with the lock and then releases the lock.

In detail, when a thread waits on multiple locks via a **waituntil**, it enqueues a select_node in each of the lock's waiting queues. When a select_node reaches the front of the lock's queue and gains ownership, the thread blocked on the **waituntil** is unblocked. Now, the lock is held by the **waituntil** thread until the code block is executed, and then the node is unregistered, during which the lock is released. Immediately releasing the lock after the code block prevents the waiting thread from holding multiple locks and potentially introducing a deadlock. As such, the only unregistered nodes associated with locks are the ones that have not run.

7.5.2 Timeouts

A timeout for the **waituntil** statement is a duration passed to routine timeout[1], *e.g.*:

```
Duration D1{ 1`ms }, D2{ 2`ms }, D3{ 3`ms };
```

waituntil(i << C1) {}	**waituntil**(i << C1) {}
or waituntil(i << C2) {}	**or waituntil**(i << C2) {}
or waituntil(i << C3) {}	**or waituntil**(i << C3) {}
or waituntil(timeout(D1)) {}	**or waituntil**(timeout(min(D1, D2, D3))) {}
or waituntil(timeout(D2)) {}	
or waituntil(timeout(D3)) {}	

These two examples are basically equivalent. Here, the multiple timeouts are useful because the durations can change during execution and the separate clauses provide different code blocks if a timeout triggers. Multiple timeouts can also be used with **and** to provide a minimal delay before proceeding. In following example, either channel C1 or C2 must be satisfied but nothing can be done for at least 1 or 3 seconds after the channel read, respectively.

```
waituntil( i << C1 ){} and waituntil( timeout( 1`s ) ){}
or waituntil( i << C2 ){} and waituntil( timeout( 3`s ) ){}
```

If only C2 is satisfied, *both* timeout code-blocks trigger because 1 second occurs before 3 seconds. Note that the C∀ **waitfor** statement only provides a single **timeout** clause because it only supports **or** semantics.

The timeout routine is different from UNIX sleep, which blocks for the specified duration and returns the amount of time elapsed since the call started. Instead, timeout returns a type that supports the is_selectable trait, allowing the type system to select the correct overloaded routine for this context. For the **waituntil**, it is more idiomatic for the timeout to use the same syntax as other blocking operations instead of having a special language clause.

7.5.3 Channels

Channels require more complexity to allow synchronous multiplexing. For locks, when an outside thread releases a lock and unblocks the **waituntil** thread (WUT), the lock's MX property is passed to the WUT (no spinning locks). For futures, the outside thread deliveries a value to the future and unblocks any waiting threads, including WUTs. In either case, after the WUT unblocks, it is safe to execute its corresponding code block knowing access to the resource is protected by the lock or the read-only state of the future. Similarly, for channels, when an outside thread inserts a value into a channel, it must unblock the WUT. However, for channels, there is a race condition that poses an issue. If the outside thread inserts into the channel and unblocks the WUT, there is a race where another thread can remove the channel data, so after the WUT unblocks and attempts to remove from the buffer, it fails, and the WUT must reblock (busy waiting). This scenario is a TOCTOU race that needs to be consolidated. To close the race, the outside thread must detect this case and insert directly into the left-hand side of the channel expression (i << chan) rather than into the channel, and then unblock the WUT. Now the

[1] **timeout** is a quasi-keyword in C∀, allowing it to be used an identifier.

unblocked WUT is guaranteed to have a satisfied resource and its code block can safely executed. The insertion circumvents the channel buffer via the wait-morphing in the C∀ channel implementation (see Section 5.4), allowing **waituntil** channel unblocking to not be special-cased. Note that all channel operations are fair and no preference is given between **waituntil** and direct channel operations when unblocking.

Furthermore, if both **and** and **or** operators are used, the **or** operations stop behaving like exclusive-or due to the race among channel operations, *e.g.*:

```
int i;
waituntil( i << A ) {} and waituntil( i << B ) {}
or waituntil( i << C ) {} and waituntil( i << D ) {}
```

If exclusive-or semantics are followed, only the code blocks for A and B are run, or the code blocks for C and D. However, four outside threads inserting into each channel can simultaneously put values into i and attempt to unblock the WUT to run the four code-blocks. This case introduces a race with complexity that increases with the size of the **waituntil** statement. However, due to TOCTOU issues, it is impossible to know if all resources are available without acquiring all the internal locks of channels in the subtree of the **waituntil** clauses. This approach is a poor solution for two reasons. It is possible that once all the locks are acquired, the subtree is not satisfied and the locks must be released. This work incurs a high cost for signalling threads and heavily increase contention on internal channel locks. Furthermore, the **waituntil** statement is polymorphic and can support resources that do not have internal locks, which also makes this approach infeasible. As such, the exclusive-or semantics are lost when using both **and** and **or** operators since it cannot be supported without significant complexity and significantly affects **waituntil** performance. Therefore, the example of reusing variable i by multiple output channels is considered a user error without exclusive-or semantics. Given aliasing in C, it is impossible to even warn of the potential race. In the future, it would be interesting to support Go-like syntax, **case** i := ← ..., defining a new scoped i variable for each clause.

It was deemed important that exclusive-or semantics are maintained when only **or** operators are used, so this situation has been special-cased, and is handled by having all clauses race to set a value *before* operating on the channel. Consider the following example where thread 1 is reading and threads 2 and 3 are writing to channels A and B concurrently.

```
channel(int) A, B; // zero size channels
```

thread 1	thread 2	thread 3
waituntil(i << A) {} **or waituntil**(i << B) {}	A << 1;	B << 2;

For thread 1 to have exclusive-or semantics, it must only consume from exactly one of A or B. As such, thread 2 and 3 must race to establish the winning clause of the **waituntil** in thread 1. This race is consolidated by thread 2 and 3 each attempting to set a pointer to the winning clause's select_node address using CAS. The winner bypasses the channel and inserts into the WUT's left-hand, and signals thread 1. The loser continues checking if there is space in the channel, and if so performs the channel insert operation with a possible signal of a waiting remove thread; otherwise, if there is no space, the loser blocks. It is important the race occurs *before* operating on the channel, because channel actions are different with respect to each thread. If the race was consolidated after the operation, both thread 2 and 3 could potentially write into i concurrently.

Channels introduce another interesting implementation issue. Supporting both reading and writing to a channel in a **waituntil** means that one **waituntil** clause may be the notifier of another **waituntil** clause. This conjunction poses a problem when dealing with the special-cased **or** where the clauses need to win a race to operate on a channel. Consider the following example, alongside a described ordering of events to highlight the race.

```
channel(int) A, B; // zero size channels
thread 1                          | thread 2
waituntil( i << A ) {}            | waituntil( B << 2 ) {}
or waituntil( i << B ) {}         | or waituntil( A << 1 ) {}
```

Assume thread 1 executes first, registers with channel A and proceeds in the **waituntil**. Since A is empty, thread 1 cannot remove, and then thread 1 is interrupted before registering with B. Thread 2 similarly registers with channel B, and proceeds in the **waituntil**. Since B is zero size there is no space to insert, and then thread 2 is interrupted before registering with A. At this point, thread 1 and 2 resume execution. Remember from above, each exclusive-or **waituntil** holds a race to set the winning clause of the statement. The issue that arises is that these two **waituntil** statements must have matching winning clauses (both A clauses or both B clauses) to preserve the exclusive-or semantics, since a zero-sized channel needs an insert/remove pair for an operation to occur. If threads 1 and 2 race to set a winner only in their own **waituntil**, thread 1 can think it successfully removed from B, and thread 2 can think it successfully inserted into A, which is an error. Hence, there is a race on two fronts. If thread 1 wins the race and sees that B has a waiting insertion, then thread 2 must execute the first clause of its **waituntil** and thread 1 execute its second. Correspondingly, if thread 2 wins the race and sees that A has a waiting removal, then thread 1 must execute the first clause of its **waituntil** and thread 2 execute its second. Any other execution scenario is incorrect for exclusive-or semantics. Note that priority execution of multiple satisfied **waituntil** causes (*i.e.*, top to bottom) is not violated because, in this scenario, there is only one satisfied clause for either thread.

The Go select solves this problem by acquiring all the internal locks of the channels before registering the select on the channels. This approach eliminates the race shown above since thread 1 and 2 cannot both be registering at the same time. However, this approach cannot be used in C∀, since the **waituntil** is polymorphic. Not all types in a **waituntil** have an internal lock, and when using non-channel types, acquiring all the locks incurs extra unneeded overhead. Instead, this race is consolidated in C∀ in two phases by having an intermediate pending status value for the race. This race case is detectable, and if detected, each thread first races to set its own **waituntil** race pointer to be pending. If it succeeds, it then attempts to set the other thread's **waituntil** race pointer to its success value. If either thread successfully sets the the other thread's **waituntil** race pointer, then the operation can proceed, if not the signalling threads set its own race pointer back to the initial value and repeats. This retry mechanism can potentially introduce a livelock, but in practice a livelock here is highly unlikely. Furthermore, the likelihood of a livelock here is zero unless the program is in the niche case of having two or more exclusive-or **waituntil**s with two or more clauses in reverse order of priority. This livelock case can be fully eliminated using locks like Go, or if a DCAS instruction is available. If any other threads attempt to set a WUT's race pointer and see a pending value, they wait until the value changes before proceeding to ensure that, in the case the WUT fails, the signal is not lost. This protocol ensures that signals cannot be

```
bool pending_set_other( select_node & other, select_node & mine ) {
    unsigned long int cmp_status = UNSAT;

    // Try to set other status, if we succeed break and return true
    while( ! CAS( other.clause_status, &cmp_status, SAT ) ) {
        if ( cmp_status == SAT )
            return false; // If other status is SAT we lost so return false

        // Toggle own status flag to allow other thread to potentially win
        mine.status = UNSAT;

        // Reset compare flag
        cmp_status = UNSAT;

        // Attempt to set own status flag back to PENDING to retry
        if ( ! CAS( mine.clause_status, &cmp_status, PENDING ) )
            return false; // If we fail then we lost so return false

        // Reset compare flag
        cmp_status = UNSAT;
    }
    return true;
}
```

Figure 7.7: Exclusive-or **waituntil** channel deadlock avoidance protocol

lost and that the two races can be resolved in a safe manner. The implementation of this protocol is shown in Figure 7.7.

Channels in C∀ have exception-based shutdown mechanisms that the **waituntil** statement needs to support. These exception mechanisms are supported through the on_selected routine. This routine is needed by channels to detect if they are closed after unblocking in a **waituntil** statement, to ensure the appropriate behaviour is taken and an exception is thrown. Hence, the channel close-down mechanism is handled correctly.

7.5.4 Guards and Statement Predicate

It is trivial to check when a synchronous multiplexing utility is done for the or/xor relationship, since any resource becoming available means that the blocked thread can proceed and the **waituntil** statement is finished. In μC++ and C∀, the synchronous multiplexing mechanism have both an and/or relationship, which along with guards, make the problem of checking for completion of the statement difficult. Consider the following **waituntil**.

```
when( GA ) waituntil( A ) {}
and when( GB ) waituntil( B ) {}
or when( GC ) waituntil( C ) {}
```

When the **waituntil** thread wakes up, the following predicate represents satisfaction:

```
A && B || C || ! GA && B || ! GB && A || ! GA && ! GB && ! GC
```

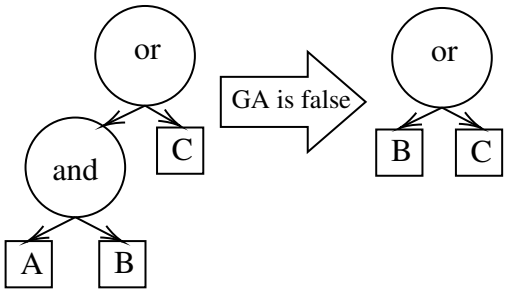


Figure 7.8: μC++ select tree modification

which can be simplified to:

```
( A || ! GA ) && ( B || ! GB ) || C || ! GA && ! GB && ! GC
```

Checking the complete predicate on each iteration of the pending **waituntil** is expensive so μC++ and C∀ both take steps to simplify checking for statement completion.

In the μC++ _Select statement, this problem is solved by constructing a tree of the resources, where the internal nodes are operators and the leaves are booleans storing the state of each resource. A diagram of the tree for the complete predicate above is shown in Figure 7.8, alongside the modification of the tree that occurs when GA is false. Each internal node stores the statuses of the two subtrees beneath it. When resources become available, their corresponding leaf node status is modified, which percolates up the tree to update the state of the statement. Once the root of the tree has both subtrees marked as true then the statement is complete. As an optimization, when the internal nodes are updated, the subtrees marked as true are pruned and not examined again. To support statement guards in μC++, the tree is modified to remove an internal node if a guard is false to maintain the appropriate predicate representation.

The C∀ **waituntil** statement blocks a thread until a set of resources have become available that satisfy the complete predicate of a **waituntil**. The waiting condition of the **waituntil** statement is implemented as the complete predicate over the resources, joined by the **waituntil** operators, where a resource is true if it is available, and false otherwise. This complete predicate is used as the mechanism to check if a thread is done waiting on a **waituntil**. Leveraging the compiler, a predicate routine is generated per **waituntil**. This predicate routine accepts the statuses of the resources being waited on as arguments. A resource status is an integer that indicates whether a resource is either not available, available, or has already run its associated code block. The predicate routine returns true when the **waituntil** is done, *i.e.*, enough resources have run their associated code blocks to satisfy the **waituntil**'s predicate, and false otherwise. To support guards on the C∀ **waituntil** statement, the status of a resource disabled by a guard is set to a boolean value that ensures that the predicate function behaves as if that resource is no longer part of the predicate. The generated code allows the predicate that is checked with each iteration to be simplified to not check guard values. For example, the following is generated for the complete predicate above:

```
// statement completion predicate
bool check_completion( select_node * nodes ) {
    return nodes[0].status && nodes[1].status || nodes[2].status;
```

```
}
if ( GA || GB || GC ) {                    // skip statement if all guards false
    select_node nodes[3];
    nodes[0].status = ! GA && GB;          // A's status
    nodes[1].status = ! GB && GA;          // B's status
    nodes[2].status = ! GC;                // C's status
    // ... rest of waituntil codegen ...
}
```

μC++'s _Select, supports operators both inside and outside of the **_Select** clauses. In the following example, the code blocks run once their corresponding predicate inside the round braces is satisfied.

```
Future_ISM<int> A, B, C, D;
_Select( A || B && C ) { ... }
and _Select( D && E ) { ... }
```

This feature is more expressive than the **waituntil** statement in C∀, allowing the code block for && to only run after *both* resources are available.

In C∀, since the **waituntil** statement supports more resources than just futures, implementing operators inside clauses is avoided for a few reasons. As a motivating example, suppose C∀ supported operators inside clauses as in:

```
owner_lock A, B, C, D;
waituntil( A && B ) { ... }
or waituntil( C && D ) { ... }
```

If the **waituntil** acquires each lock as it becomes available, there is a possible deadlock since it is in a hold-and-wait situation. Other semantics are needed to ensure this operation is safe. One possibility is to use C++'s scoped_lock approach described in Section 4.5; however, that opens the potential for livelock. Another possibility is to use resource ordering similar to C∀'s **mutex** statement, but that alone is insufficient, if the resource ordering is not used universally. One other way this could be implemented is to wait until all resources for a given clause are available before proceeding to acquire them, but this also quickly becomes a poor approach. This approach does not work due to TOCTOU issues, *i.e.*, it is impossible to ensure that the full set of resources are available without holding them all first. Operators inside clauses in C∀ could potentially be implemented with careful circumvention of the problems. Finally, the problem of operators inside clauses is also a difficult to handle when supporting channels. It would require some way to ensure channels used with internal operators are modified, if and only if, the corresponding code block is run. However, that is not feasible due to reasons described in the exclusive-or portion of Section 7.5.3. Ultimately, this feature was not considered crucial after taking into account the complexity and runtime cost.

7.5.5 The full waituntil picture

Given all the complex details handled by the **waituntil**, its full pseudocode is presented in Figure 7.9. Some things to note are as follows. The **finally** blocks provide exception-safe RAII unregistering of nodes, and in particular, the **finally** inside the innermost loop performs the immediate unregistering required for deadlock-freedom mentioned in Section 7.5.1. The when_conditions

```
bool when_conditions[N];
for ( node in nodes )                                   // evaluate guards
    if ( node has guard )
        when_conditions[node] = node_guard;
    else
        when_conditions[node] = true;

if ( any when_conditions[node] are true ) {
    select_nodes nodes[N];                              // declare N select nodes
    try {
        // ... set statuses for nodes with when_conditions[node] == false ...

        for ( node in nodes )                           // register nodes
            if ( when_conditions[node] )
                register_select( resource, node );

        while ( !check_completion( nodes ) ) {          // check predicate
            // block
            for ( resource in waituntil statement ) {   // run true code blocks
                if ( check_completion( nodes ) ) break;
                if ( resource is avail )
                    try {
                        if( on_selected( resource ) )   // conditionally run block
                            run code block
                    } finally                           // for exception safety
                        unregister_select( resource, node );  // immediate unregister
            }
        }
    } finally {                                         // for exception safety
        for ( registered nodes in nodes )               // deregister nodes
            if ( when_conditions[node] && unregister_select( resource, node )
                    && on_selected( resource ) )
                run code block                          // run code block upon unregister
    }
}
```

Figure 7.9: Full **waituntil** Pseudocode Implementation

array is used to store the boolean result of evaluating each guard at the beginning of the **waituntil**, and it is used to conditionally omit operations on resources with false guards. As discussed in Section 7.5.3, this pseudocode includes conditional code-block execution based on the result of both on_selected and unregister_select, which allows the channel implementation to ensure all available channel resources have their corresponding code block run.

7.6 waituntil Performance

Similar facilities to **waituntil** are discussed in Section 7.1 covering C, Ada, Rust, C++, and OCaml. However, these facilities are either not meaningful or feasible to benchmark against. The UNIX select and related utilities are not comparable since they are system calls that go into the kernel and operate on file descriptors, whereas the **waituntil** exists solely in user space. Ada's **select** and μC++'s **_Accept** only operate on method calls, which is done in C$\forall$ via the **waitfor** statement. Rust and C++ only offer a busy-wait approach, which is not comparable to a blocking approach. OCaml's select waits on channels that are not comparable with C$\forall$ and Go channels, so OCaml select is not benchmarked against Go's select and C$\forall$'s **waituntil**.

The two synchronous multiplexing utilities that are in the realm of comparability with the C$\forall$ **waituntil** statement are the Go **select** statement and the μC++ **_Select** statement. As such, two microbenchmarks are presented, one for Go and one for μC++ to contrast this feature. Given the differences in features, polymorphism, and expressibility between **waituntil** and **select**, and μC++ **_Select**, the aim of the microbenchmarking in this chapter is to show that these implementations lie in the same realm of performance, not to pick a winner.

7.6.1 Channel Benchmark

The channel multiplexing benchmarks compare C$\forall$'s **waituntil** and Go's **select**, where the resource being waited on is a set of channels. Although Unix's select, poll and epoll multiplex over file descriptors, which are effectively channels, these utilities are not included in the benchmarks. Reading or writing to a file descriptor requires a system call which is much more expensive than operating on a user-space channel. As such, it is infeasible to compare Unix's select, poll and epoll with C$\forall$'s **waituntil** and Go's **select**. The basic structure of the benchmark has the number of cores split evenly between producer and consumer threads, *i.e.*, with 8 cores there are 4 producer and 4 consumer threads. The number of resource clauses C is also varied across 2, 4, and 8 clauses, where each clause has a different channel that it waits on. Each producer and consumer repeatedly waits to either produce or consume from one of the C clauses and respective channels. For example, in C$\forall$ syntax, the work loop in the consumer main with $C = 4$ clauses is:

```
for ()
    waituntil( val << chans[0] ); or waituntil( val << chans[1] );
    or waituntil( val << chans[2] ); or waituntil( val << chans[3] );
```

A successful consumption is counted as a channel operation, and the throughput of these operations is measured over 10 seconds. The first benchmark measures throughput of the producers and consumer synchronously waiting on the channels and the second has the threads asynchronously wait on the channels using the Go **default** and C$\forall$ **else** clause. The results are shown in Figures 7.10 and 7.11 respectively.

Both Figures 7.10 and 7.11 have similar results when comparing **select** and **waituntil**. In the AMD benchmarks (left column), the performance is very similar as the number of cores scale. The AMD machine has a high-caching contention cost because of its *chiplet* L3 cache (*i.e.*, many L3 caches servicing a small number of cores), which creates a bottleneck on the channel locks and dominates the shape of the performance curve for both C$\forall$ and Go. Hence,

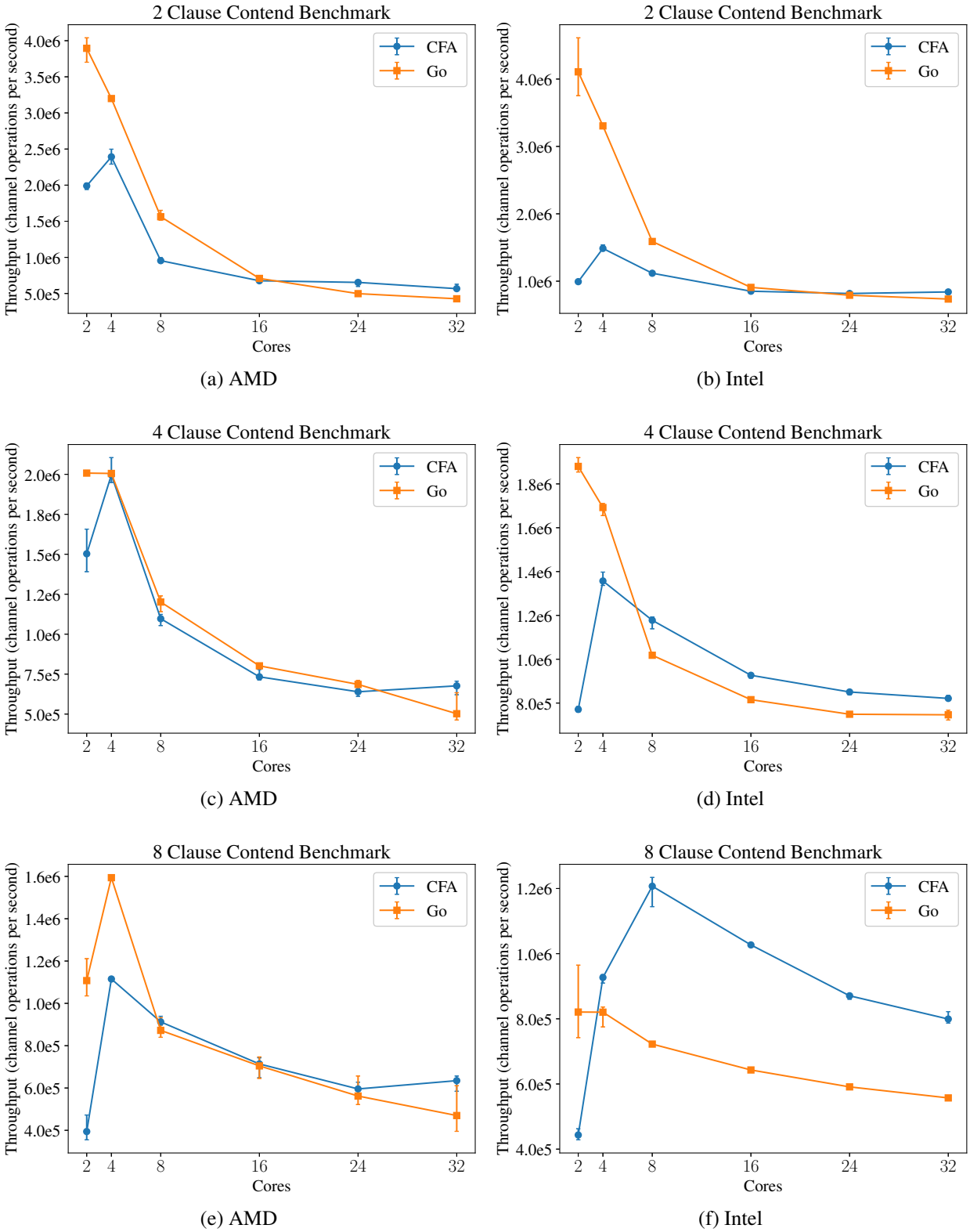


Figure 7.10: The channel synchronous multiplexing benchmark comparing Go select and C∀ **waituntil** statement throughput (higher is better).

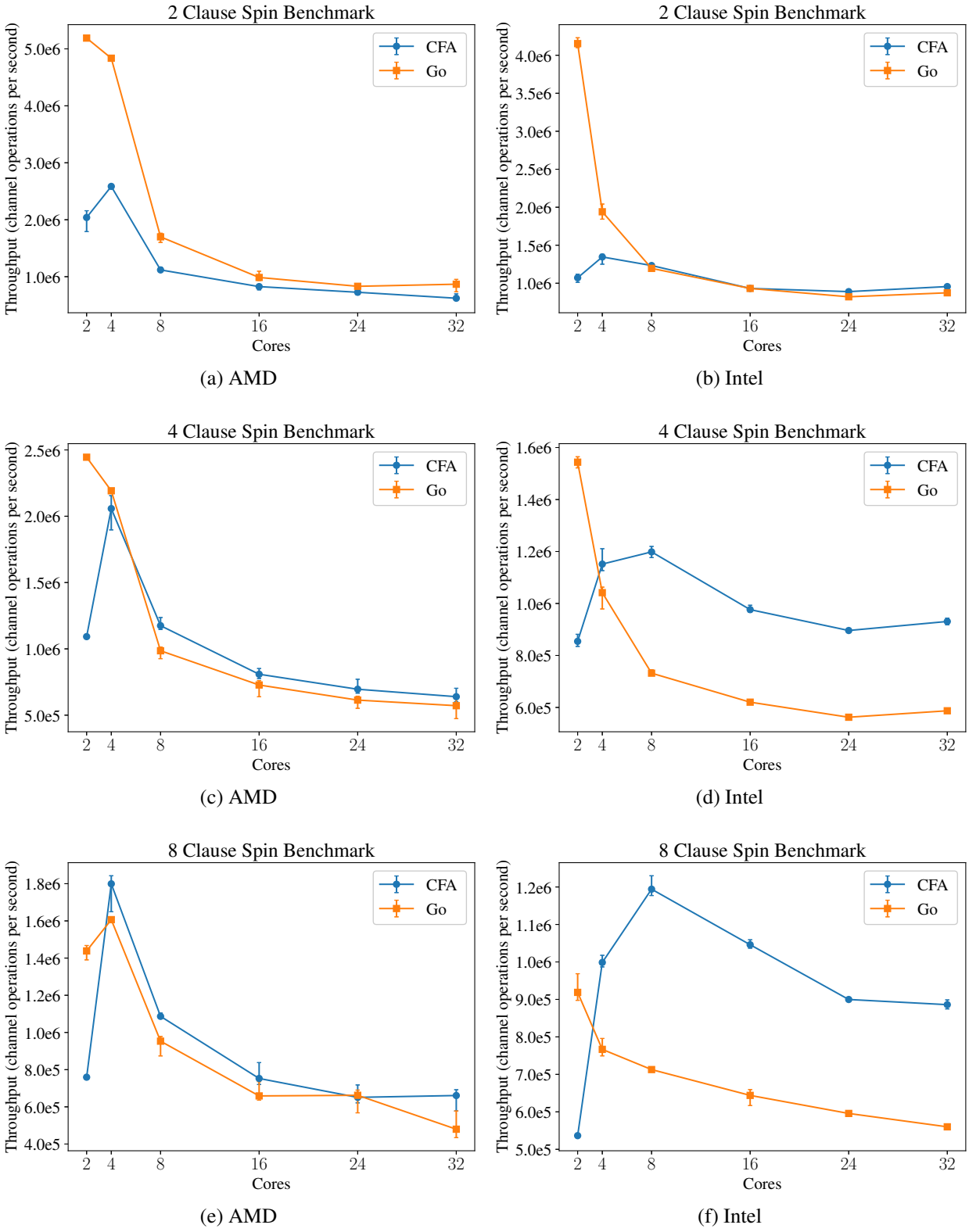


Figure 7.11: The asynchronous multiplexing channel benchmark comparing Go select and C∀ **waituntil** statement throughput (higher is better).

Table 7.1: Throughput (channel operations per second) of C∀ and Go in a pathological case for contention in Go's select implementation

	C∀	Go
AMD	15844020	6904002
Intel	6836856	6045435

it is difficult to detect differences in the synchronous multiplexing, except at low cores, where Go has significantly better performance, due to an optimization in its scheduler. Go heavily optimizes thread handoffs on the local run-queue, which can result in very good performance for low numbers of threads parking/unparking each other [38]. In the Intel benchmarks (right column), C∀ performs better than Go as the number of cores scales past 2/4 and as the number of clauses increase. This difference is due to Go's acquiring all channel locks when registering and unregistering channels on a **select**. Go then is holding a lock for every channel, resulting in worse performance as the number of channels increase. In C∀, since races are consolidated without holding all locks, it scales much better both with cores and clauses since more work can occur in parallel. This scalability difference is more significant on the Intel machine than the AMD machine since the Intel has lower cache-contention costs.

The Go approach of holding all internal channel-locks in the **select** has additional drawbacks. There are pathological cases where Go's throughput has significant jitter. Consider a producer and consumer thread, P1 and C1, selecting from both channels A and B.

```
P1                                          C1
waituntil( A << i ); or waituntil( B << i );    waituntil( val << A ); or waituntil( val << B );
```

Additionally, there is another producer and consumer thread, P2 and C2, operating solely on B.

```
P2          C2
B << val;   val << B;
```

In Go, this setup results in significantly worse performance since P2 and C2 cannot operate in parallel with P1 and C1 due to all locks being acquired. Interestingly, this case may not be as pathological as it seems. If the set of channels belonging to a **select** have channels that overlap with the set of another **select**, these statements lose the ability to operate in parallel. The implementation in C∀ only holds a single lock at a time, resulting in better locking granularity, and hence, more parallelism. Comparison of this pathological case is shown in Table 7.1. The AMD results highlight the worst-case scenario for Go since contention is more costly on this machine than the Intel machine.

Another difference between Go and C∀ is the order of clause selection when multiple clauses are available. Go *randomly* selects a clause [39], but C∀ chooses in the order clauses are listed. This C∀ design decision allows users to set implicit priorities, which can result in more predictable behaviour and even better performance. In the previous example, threads P1 and C1 prioritize channel A in the **waituntil**, which can reduce contention for threads P2 and C2 accessing channel B. If C∀ did not have priorities, the performance difference in Table 7.1 would be significant less due to extra contention on channel B.

7.6.2 Future Benchmark

The future benchmark compares CFA's **waituntil** with μC++'s **_Select**, with both utilities waiting on futures. While both statements have very similar semantics, supporting **and** and **or** operators, **_Select** can only wait on futures, whereas the **waituntil** is polymorphic. As such, the underlying implementation of the operators differs between **waituntil** and **_Select**. The **waituntil** statement checks for statement completion using a predicate function, whereas the **_Select** statement maintains a tree that represents the state of the internal predicate.

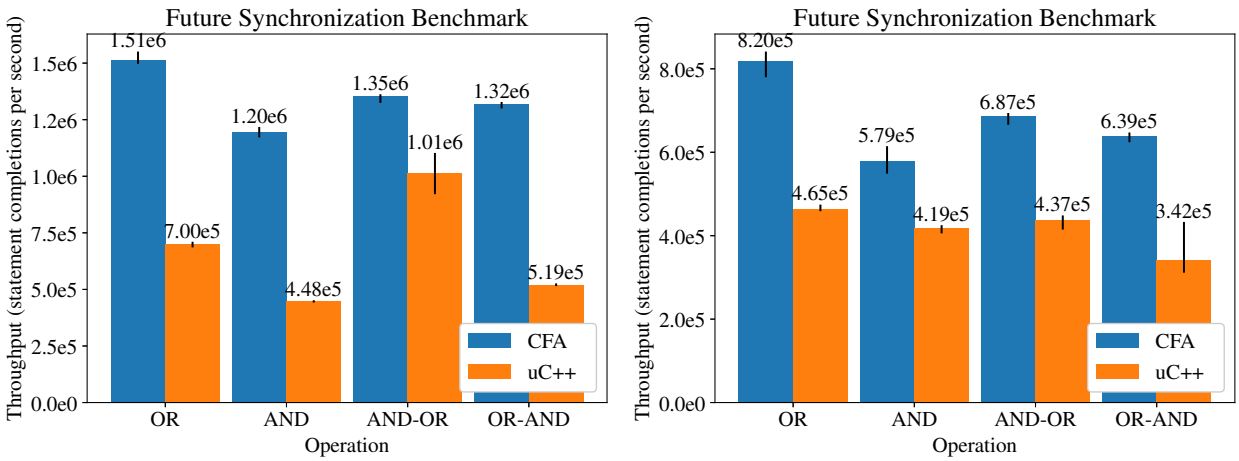
This benchmark aims to indirectly measure the impact of various predicates on the performance of the **waituntil** and **_Select** statements. The benchmark is indirect since the performance of futures in CFA and μC++ differ by a significant margin. The experiment has a server, which cycles fulfilling three futures, A, B, and C, and a client, which waits for these futures to be fulfilled using four different kinds of predicates given in CFA:

OR	AND
waituntil(A) { get(A); } **or waituntil**(B) { get(B); } **or waituntil**(C) { get(C); }	**waituntil**(A) { get(A); } **and waituntil**(B) { get(B); } **and waituntil**(C) { get(C); }

AND-OR	OR-AND
waituntil(A) { get(A); } **and waituntil**(B) { get(B); } **or waituntil**(C) { get(C); }	(**waituntil**(A) { get(A); } **or waituntil**(B) { get(B); }) **and waituntil**(C) { get(C); }

The server and client use a low cost synchronize after each fulfillment, so the server does not race ahead of the client.

Results of this benchmark are shown in Figure 7.12. Each pair of bars is marked with the predicate name for that experiment and the value at the top of each bar is the standard deviation. In detail, μC++ results are lower in all cases due to the performance difference between futures and the more complex synchronous multiplexing implementation. However, the bars for both systems have similar height patterns across the experiments. The OR column for CFA is more performant than the other CFA predicates, due to the special-casing of **waituntil** statements with only **or** operators. For both μC++ and CFA, the AND experiment is the least performant, which is expected since all three futures need to be fulfilled for each statement completion. Interestingly, CFA has lower variation across predicates on the AMD (excluding the special OR case), whereas μC++ has lower variation on the Intel. Given the differences in semantics and implementation between μC++ and CFA, this test only illustrates the overall costs among the different kinds of predicates.



(a) AMD Future Synchronization Benchmark (b) Intel Future Synchronization Benchmark

Figure 7.12: C∀ **waituntil** and μC++ _Select statement throughput synchronizing on a set of futures with varying wait predicates (higher is better).

Chapter 8

Conclusion

The goal of this thesis is to expand concurrent support in C∀ to fill in gaps and increase support for writing safe and efficient concurrent programs. The presented features achieve this goal and provide users with the means to write scalable concurrent programs in C∀ through multiple avenues. Additionally, the tools presented provide safety and productivity features including: detection of deadlock and other common concurrency errors, easy concurrent shutdown, and toggleable performance statistics.

For locking, the mutex statement provides a safe and easy-to-use interface for mutual exclusion. If programmers prefer the message-passing paradigm, C∀ now supports it in the form of channels and actors. The **waituntil** statement simplifies writing concurrent programs in both the message-passing and shared-memory paradigms of concurrency. Finally, no other programming language provides a synchronous multiplexing tool that is polymorphic over resources like C∀'s **waituntil**. This work successfully provides users with familiar concurrent-language support, but with additional value added over similar utilities in other popular languages.

On overview of the contributions made in this thesis include the following:

1. The mutex statement, which provides performant and deadlock-free multi-lock acquisition.
2. Channels with comparable performance to Go, which have safety and productivity features including deadlock detection and an easy-to-use exception-based channel close routine.
3. An in-memory actor system, which achieves the lowest latency message send of systems tested due to the novel copy-queue data structure.
4. As well, the actor system has built-in detection of six common actor errors, with excellent performance compared to other systems across all benchmarks presented in this thesis.
5. A **waituntil** statement, which tackles the hard problem of allowing a thread wait synchronously for an arbitrary set of concurrent resources.

The added features are now commonly used to solve concurrent problems in C∀. The **mutex** statement sees use across almost all concurrent code in C∀, as it is the simplest mechanism for providing thread-safe input and output. The channels and the **waituntil** statement see use in programs where a thread operates as a server or administrator, which accepts and distributes work among channels based on some shared state. When implemented, the polymorphic support of the **waituntil** statement will see use with the actor system to enable user threads outside the actor system to wait for work to be done or for actors to finish. Finally, the new features are often

combined, *e.g.*, channels pass pointers to shared memory that may still need mutual exclusion, requiring the **mutex** statement to be used.

From the novel copy-queue data structure in the actor system and the plethora of user-supporting safety features, all these utilities build upon existing concurrent tooling with value added. Performance results verify that each new feature is comparable or better than similar features in other programming languages. All in all, this suite of concurrent tools expands a C∀ programmer's ability to easily write safe and performant multi-threaded programs.

8.1 Future Work

8.1.1 Further Implicit Concurrency

This thesis only scratches the surface of implicit concurrency by providing an actor system. There is room for more implicit concurrency tools in C∀. User-defined implicit concurrency in the form of annotated loops or recursive concurrent functions exists in other languages and libraries [10, 53]. Similar implicit concurrency mechanisms could be implemented and expanded on in C∀. Additionally, the problem of automatic parallelism of sequential programs via the compiler is an interesting research space that other languages have approached [61, 60] and could be explored in C∀.

8.1.2 Advanced Actor Stealing Heuristics

In this thesis, two basic victim-selection heuristics are chosen when implementing the work-stealing actor-system. Good victim selection is an active area of work-stealing research, especially when taking into account NUMA effects and cache locality [2, 62]. The actor system in C∀ is modular and exploration of other victim-selection heuristics for queue stealing in C∀ could be provided by pluggable modules. Another question in work stealing is: when should a worker thread steal? Work-stealing systems can often be too aggressive when stealing, causing the cost of the steal to be have a negative rather positive effect on performance. Given that thief threads often have cycles to spare, there is room for a more nuanced approaches when stealing. Finally, there is the very difficult problem of blocking and unblocking idle threads for workloads with extreme oscillations in CPU needs.

8.1.3 Synchronously Multiplexing System Calls

There are many tools that try to synchronously wait for or asynchronously check I/O. Improvements in this area pay dividends in many areas of I/O based programming [49, 48, 46, 47]. Research on improving user-space tools to synchronize over I/O and other system calls is an interesting area to explore in the world of concurrent tooling. Specifically, incorporating I/O into the **waituntil** to allow a network server to work with multiple kinds of asynchronous I/O interconnects without using tradition event loops.

8.1.4 Better Atomic Operations

When writing low-level concurrent programs, especially lock/wait-free programs, low-level atomic instructions need to be used. In C, the gcc-builtin atomics [59] are commonly used, but leave much to be desired. Some of the problems include the following. Archaic and opaque macros often have to be used to ensure that atomic assembly is generated instead of locks. The builtins are polymorphic, but not type safe since they use void pointers. The semantics and safety of these builtins require careful navigation since they require the user to have a deep understanding of concurrent memory-ordering models. Furthermore, these atomics also often require a user to understand how to fence appropriately to ensure correctness. All these problems and more would benefit from language support in C∀. Adding good language support for atomics is a difficult problem, which if solved well, would allow for easier and safer writing of low-level concurrent code.

References

[1] Ada16. *Ada Reference Manual ISO/IEC 8652:2012(E) with COR.1:2016*. AXE Consultants, Madison WI, USA, 3rd with technical corrigendum 1 for ada 2012 edition, 2016. https://docs.adacore.com/live/wave/arm12/pdf/arm12/arm-12.pdf.

[2] Saman Barghi. *Improving the Performance of User-level Runtime Systems for Concurrent Applications*. PhD thesis, School of Computer Science, University of Waterloo, September 2018. https://uwspace.uwaterloo.ca/handle/10012/13935.

[3] Andrew James Beach. Exception handling in C∀. Master's thesis, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, 2021. http://hdl.handle.net/10012/17617.

[4] Andrew Birrell, Mark R. Brown, Luca Cardelli, Jim Donahue, Lucille Glassman, John Gutag, Jim Harning, Bill Kalsow, Roy Levin, and Greg Nelson. *Systems Programming with Modula-3*. Prentice-Hall Series in Innovative Technology. Prentice-Hall, Englewood Cliffs, 1991.

[5] Per Brinch Hansen. *Operating System Principles*. Prentice-Hall, Englewood Cliffs, 1973.

[6] Per Brinch Hansen. The programming language concurrent pascal. *IEEE Trans. Softw. Eng.*, SE-1(2):199–207, June 1975.

[7] Trevor Brown, Faith Ellen, and Eric Ruppert. Pragmatic primitives for non-blocking data structures. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, PODC '13, pages 13–22, New York, NY, USA, 2013. ACM.

[8] Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 329–342, New York, NY, USA, 2014. ACM.

[9] P. A. Buhr, Glen Ditchfield, R. A. Stroobosscher, B. M. Younger, and C. R. Zarnke. μC++: Concurrency in the object-oriented language C++. *Softw. Pract. Exper.*, 22(2):137–172, February 1992.

[10] Peter A. Buhr. *μC++ Annotated Reference Manual, Version 7.0.0*. University of Waterloo, Waterloo Ontario, Canada, September 2020. https://plg.uwaterloo.ca/~usystem/pub/uSystem/uC++.pdf.

[11] Peter A. Buhr, Colby A. Parsons, Thierry Delisle, and He Nan Li. High-performance extended actors. *Softw. Pract. Exper.*, 2022. submitted July 2022.

[12] David R. Butenhof. *Programming with POSIX Threads*. Professional Computing. Addison-Wesley, Boston, 1997.

[13] Dominik Charousset, Raphael Hiesgen, and Thomas C. Schmidt. CAF – the C++ actor framework for scalable and resource-efficient applications. AGERE'14, pages 15–28, New York, NY, USA, 2014. Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control, ACM.

[14] Java Util Concurrent. Class completablefuture. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html. [Online; accessed 23-May-2023].

[15] Thierry Delisle. Concurrency in C∀. Master's thesis, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, 2018. http://hdl.handle.net/10012/12888.

[16] Thierry Delisle. *The C∀ Scheduler*. PhD thesis, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, 2022. http://hdl.handle.net/10012/18941.

[17] Thierry Delisle and Peter A. Buhr. Advanced control-flow and concurrency in C∀. *Softw. Pract. Exper.*, 51(5):1005–1042, May 2021.

[18] David Dice and Oleksandr Otenko. Brief announcement: Multilane-a concurrent blocking multiset. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 313–314, 2011.

[19] Edsger W. Dijkstra. Cooperating sequential processes. Technical report, Technological University, Eindhoven, Neth., 1965. https://pure.tue.nl/ws/files/4279816/344354178746665.pdf.

[20] Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965.

[21] Kotlin Documentation. Channel. https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.channels/-channel/. [Online; accessed 11-September-2023].

[22] Simon Doherty, David L Detlefs, Lindsay Groves, Christine H Flood, Victor Luchangco, Paul A Martin, Mark Moir, Nir Shavit, and Guy L Steele Jr. Dcas is not a silver bullet for nonblocking algorithm design. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 216–224, 2004.

[23] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, Reading, 2nd edition, 2000.

[24] Robert Griesemer, Rob Pike, and Ken Thompson. *Go Programming Language*. Google, Mountain View, CA, USA, 2009. http://golang.org/ref/spec.

[25] Timothy L Harris, Keir Fraser, and Ian A Pratt. A practical multi-word compare-and-swap operation. In *Distributed Computing: 16th International Conference, DISC 2002 Toulouse, France, October 28–30, 2002 Proceedings 16*, pages 265–279. Springer, 2002.

[26] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. IJCAI'73, pages 235–245, Standford, California, U.S.A., August 1973. Proceedings of the 3rd International Joint Conference on Artificial Intelligence.

[27] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

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

[29] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Upper Saddle River, NJ, USA, 1985. http://www.usingcsp.com/cspbook.pdf.

[30] R. C. Holt and J. R. Cordy. The turing programming language. *Communications of the ACM*, 31(12):1410–1423, December 1988.

[31] Jean D Ichbiah. Preliminary ada reference manual. *ACM Sigplan Notices*, 14(6a):1–145, 1979.

[32] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, March 2023.

[33] Kathleen Jensen and Niklaus Wirth. *Pascal User Manual and Report, ISO Pascal Standard*. Springer–Verlag, 4th edition, 1991. Revised by Andrew B. Mickel and James F. Miner.

[34] Gilles Kahn. The semantics of a simple language for parallel programming. IFIP Congress, 1974.

[35] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[36] Leslie Lamport. The mutual exclusion problem: Part II–statement and solutions. *Journal of the ACM*, 33(2):327–348, April 1986.

[37] The Go Programming Language. src/runtime/chan.go. https://go.dev/src/runtime/chan.go. [Online; accessed 23-May-2023].

[38] The Go Programming Language. src/runtime/proc.go. https://go.dev/src/runtime/proc.go. [Online; accessed 23-May-2023].

[39] The Go Programming Language. src/runtime/select.go. https://go.dev/src/runtime/select.go. [Online; accessed 23-May-2023].

[40] Boost C++ Libraries. experimental::basic_concurrent_channel. https://www.boost.org/doc/libs/master/doc/html/boost_asio/reference/experimental__basic_concurrent_channel.html. [Online; accessed 23-May-2023].

[41] The Rust Standard Library. Macro futures::select. https://docs.rs/futures/latest/futures/macro.select.html. [Online; accessed 23-May-2023].

[42] The Rust Standard Library. std::sync::mpsc::sync_channel. https://doc.rust-lang.org/std/sync/mpsc/fn.sync_channel.html. [Online; accessed 23-May-2023].

[43] Lightbend. Akka typed actors. https://doc.akka.io/docs/akka/2.5/typed-actors.html, 2022.

[44] Lightbend Inc. *Akka Scala Documentation, Release 2.4.11*, September 2016. http://doc.akka.io/docs/akka/2.4/AkkaScala.pdf.

[45] Andrew Lister. The problem of nested monitor calls. *Operating Systems Review*, 11(3):5–7, July 1977.

[46] Linux man pages. epoll(7) - linux manual page. https://man7.org/linux/man-pages/man7/epoll.7.html. [Online; accessed 23-May-2023].

[47] Linux man pages. io_uring(7) - linux manual page. https://man7.org/linux/man-pages/man7/io_uring.7.html. [Online; accessed 23-May-2023].

[48] Linux man pages. poll(2) - linux manual page. https://man7.org/linux/man-pages/man2/poll.2.html. [Online; accessed 23-May-2023].

[49] Linux man pages. select(2) - linux manual page. https://man7.org/linux/man-pages/man2/select.2.html. [Online; accessed 23-May-2023].

[50] The OCaml Manual. Ocaml library : Event. https://v2.ocaml.org/api/Event.html. [Online; accessed 23-May-2023].

[51] James G. Mitchell, William Maybury, and Richard Sweet. Mesa language manual. Technical Report CSL–79–3, Xerox Palo Alto Research Center, Palo Alto, California, U.S.A., April 1979.

[52] Libero Nigro. Parallel theatre: An actor framework in Java for high performance computing. *Simulation Modelling Practice and Theory*, 106(102189), 2021.

[53] *OpenMP Application Program Interface, Version 4.5*, November 2015. https://-www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

[54] proto.actor. Asynkron ab. https://proto.actor, 2023.

[55] C++ reference. std::experimental::when_any. https://en.cppreference.com/w/cpp/experimental/when_any. [Online; accessed 23-May-2023].

[56] The Haskell Package Repository. Control.concurrent.chan. https://hackage.haskell.org/package/base-4.18.0.0/docs/Control-Concurrent-Chan.html. [Online; accessed 23-May-2023].

[57] Andrew William Roscoe and Charles Antony Richard Hoare. The laws of OCCAM programming. *Theoretical Computer Science*, 60(2):177–229, 1988.

[58] The Go Programming Language Specification. Select statements. https://go.dev/ref/spec#Select_statements. [Online; accessed 23-May-2023].

[59] GCC team. Built-in functions for memory model aware atomic operations. https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html. [Online; accessed 23-May-2023].

[60] Haskell Wiki. Parallel haskell. https://wiki.haskell.org/Parallel. [Online; accessed 23-May-2023].

[61] Robert Wilson, Robert French, Christopher Wilson, Saman Amarasinghe, Jennifer Anderson, Steve Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary Hall, Monica Lam, et al. The SUIF compiler system: a parallelizing and optimizing research compiler. Technical report, Stanford University Technical Report No. CSL-TR-94-620, 1994.

[62] Sebastian Wölke, Raphael Hiesgen, Dominik Charousset, and Thomas C Schmidt. Locality-guided scheduling in CAF. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, pages 11–20, 2017.

Glossary

actor A basic unit of an actor system that can store local state and send messages to other actors. 33

actor model A concurrent computation model, where tasks are broken into units of work that are distributed to actors in the form of messages. 33

actor system An implementation of the actor model. 33

gulp Move the contents of message queue to a local queue of the executor thread using a single atomic instruction. 42–44

implicit concurrency A class of concurrency features that abstract away explicit thread synchronization and mutual exclusion. 33

synchronous multiplexing synchronization waiting for some subset of a set of resources. 63–66, 68, 75, 79, 82, 83