An Asynchronous Messaging Library for C#

Georgio Chrysanthakopoulos
Microsoft
One Microsoft Way
Redmond, WA 98052, USA
+1 425 703 5715
georgioc@microsoft.com

Satnam Singh
Microsoft
One Microsoft Way
Redmond WA 98052, USA
+1 425 705 8208
satnams@microsoft.com

ABSTRACT

We present the design of a small number of application level communication coordination constructs which can be hierarchically combined to allow the expression of sophisticated and flexible communication patterns. We are able to express join patterns found in languages like Comega by providing a library based implementation rather than making language extensions. We go beyond the joins in Comega to show how dynamically constructed joins can be implemented. We also demonstrate other communication coordination patterns for expressing choice and single writer and multiple reader scenarios. We believe that this experimental framework provides a useful workbench for investigating concurrency idioms which might be candidates for language level constructs.

1. INTRODUCTION

The use of operating system level threads seems appealing for the implementation of concurrency in applications as well as a method for exploiting parallel execution on multiple processors or cores. However, Ousterhout [14] argues that threads are too heavyweight for many applications and that events are more appropriate. Programming in an explicitly event based model is also very difficult. In this paper we propose an approach for message-passing based concurrent programming which tries to make event-based programming easier. An event in our library corresponds to the arrival of messages at one or more ports which triggers the execution of some code to handle the event (the "continuation"). Our objective is to design a high level asynchronous messaging library for an existing object-oriented language which can realize concurrent activities competitively compared to explicit threads without paying the cost of allocating large amounts of stack space. The design of our library is based on a small number of *arbiters* which can be composed to implement higher level synchronization constructs like join patterns.

High level communication and coordination constructs like join patterns have enjoyed considerable interest recently because they provide a higher level abstraction for performing concurrent programming [1][5][7][8][11][14]. Join patterns have typically been implemented as language extensions. We show how join patterns and other message-passing concurrency constructs can be added as a library to C# without requiring any language extensions. Furthermore, we demonstrate how it is possible to implement a join pattern library which can support a large number of asynchronous work items. This is also implemented as a regular C# library. This paper reviews an existing implementation of join patterns and then goes on to describe the architecture of a join pattern library in C#.

Join patterns offer some advantages compared to programming explicitly with threads and locks [3]. For example, a join pattern may result in more optimistic locking since all required resources are atomically acquired.

We present an implementation of an asynchronous messaging library that provides dynamic join patterns and other port-based communication and coordination constructs. Our library, called the Concurrency and Coordination Runtime (CCR) is designed to run on an existing commercial operating system and we want to support a programming style that heavily relies on messaging as a software structuring mechanism and as a way to achieve isolation between computations. This has led us to develop our own scheduler to support continuation passing style code [1] instead of using heavyweight operating system threads and processes. Although programming directly in continuation passing style is arduous we show how constructs like anonymous delegates and iterators in C# can help to write blocking-style code which is actually CPS-transformed by the C# compiler. Our CCR library is also supported with deep integration in a commercial debugger using causality checking.

We first briefly introduce and critique join patterns in Comega (Polyphonic-C#) which are representative of a higher level concurrency coordination mechanism which may be appropriate for application level development. We then present our design constraints and give a detailed description of how ports are represented in our system and how continuation code is asynchronously executed in response to messages arriving at ports. We briefly contrast asynchronous choice expressions in our language with the synchronous choice that can be implemented with an Ada rendezvous. We also identify similar concurrency constructs in the Java JSR-166 library [13]. We then outline some more sophisticated communication coordination routines to express static and dynamic join patterns and interleaving of single writers and multiple readers over a shared resource. Many of the patterns we describe go beyond what can be directly represented in a language like Comega which has only static join pattern declarations. We then outline some disadvantages of a more dynamic join pattern approach including the greater difficulty introduced into static analyses of join-based programs [4] and the compiler optimizations advantages that a statically analyzable set of join declarations can enjoy [1] which are not available in a library based approach. Some performance characteristics of our system are presented followed by a discussion of related work.

2. JOIN PATTERNS IN COMEGA

The polyphonic extensions to C# comprise just two new concepts: (i) *asynchronous methods* which return control to the caller immediately and execute the body of the method concurrently; and (ii) *chords* (also known as 'synchronization patterns' or 'join patterns') which are methods whose execution is predicated by the prior invocation of some null-bodied asynchronous methods.

2.1 ASYNCHRONOUS METHODS

The code below is a complete Comega program that demonstrates an asynchronous method.

```
using System ;

public class MainProgram
{ public class ArraySummer
  { public async sumArray (int[] intArray)
    { int sum = 0 ;
      foreach (int value in intArray)
        sum += value ;
      Console.WriteLine ("Sum  = " + sum) ;
    }
  }

  static void Main()
  { Summer = new ArraySummer () ;
    Summer.sumArray (new int[] {1, 0, 6, 3, 5}) ;
    Summer.sumArray (new int[] {3, 1, 4, 1, 2}) ;
    Console.WriteLine ("Main method done.") ;
  }
}
```

Comega introduces the **async** keyword to identify an asynchronous method. Calls to an asynchronous method return immediately and asynchronous methods do not have a return type (they behave as if their return type is **void**). The `sumArray` asynchronous method captures an array from the caller and its body is run concurrently with respect to the caller's context. The compiler may choose a variety of schemes for implementing the concurrency. For example, a separate thread could be created for the body of the asynchronous method or a work item could be created for a thread pool or, on a multi-processor or multi-core machine, the body may execute in parallel with the calling context. The second call to the `sumArray` does not need to wait until the body of the `sumArray` method finishes executing from the first call to `sumArray`.

In this program the two calls to the `sumArray` method of the `Summer` object behave as if the body of `sumArray` was forked off as a separate thread and control returns immediately to the main program. When this program is compiled and run it will in general write out the results of the two summations and the `Main method done` text in arbitrary orders. The Comega compiler can be downloaded from: http://research.microsoft.com/Comega/

2.2 CHORDS

The code below is a complete Comega program that demonstrates how a chord can be used to make a buffer.

```
using System ;
public class MainProgram
{ public class Buffer
  { public async Put (int value) ;
    public int Get () & Put(int value)
    { return value ; }
  }

  static void Main()
  { buf = new Buffer () ;
    buf.Put (42) ;
    buf.Put (66) ;
    Console.WriteLine (buf.Get() + " " +
                       buf.Get()) ;
  }
}
```

The `&` operator groups together methods that form a join pattern in Comega. A join pattern that contains only asynchronous methods will concurrently execute its body when all of the constituent methods have been called. A join pattern may have one (but not more) synchronous method which is identified by a return type other than **async**. The body for a synchronous join pattern fires when all the constituent methods (including the synchronous method) are called. The body is executed in the caller's context (thread). The Comega join pattern behaves like a join operation over a collection of ports (e.g. in JoCaml) with the methods taking on a role similar to ports. The calls to the `Put` method are similar in spirit to performing an asynchronous message send (or post) to a port. In this case the port is identified by a method name (i.e. `Put`). Although the asynchronous posts to the `Put` 'port' occur in series in the main body the values will arrive in the `Put` 'port' in an arbitrary order. Consequently the program shown above will have a non-deterministic output writing either "`42 66'`" or "`66 42`".

2.3 CRITIQUE OF COMEGA JOIN PATTERNS

Comega style join patterns offer many advantages over programming explicitly with locks. For example, one can model the availability of a resource by joining on it in the join pattern. For example, a synchronous Comega program that needs to use a printer and a scanner can write a join pattern like:

```
 public void doJob (doc d) & printer() &
              scanner()
{ // scan and print the job
  // release resources
  scanner() ;
  printer() ;
}
```

and the availability of a printer and a scanner is indicated by calling the asynchronous methods for the printer and the scanner. When the method is finished with the resources it frees them up by calling the asynchronous methods for each resource. A key feature of join patterns is that values are *atomically* read from all the channels in the pattern. This allows several threads to try and use this method which needs exclusive access to two shared

resources. One does not need to lock one resource while waiting for the other. Explicit locking would require the locks to be acquired in a specified (alphabetic) order but this is a difficult property to verify and it is easy to write buggy programs that do not observe the lock acquisition constraint. The join pattern can mention the resources in any order. Furthermore, the resources are only tied up by this method when all of them become available and the acquisition occurs atomically. One can still easily write deadlocking programs e.g. by forgetting to release a resource. However, the atomic nature of join patterns helps to avoid an important class of concurrent programming errors.

Join patterns fit in conveniently into the class system of an object-oriented language like C#. One thing to note about join patterns in C# is their static nature since join patterns are *declarations*. This means that for a given join pattern the same code is always executed when the join pattern can fire. This scheme permits a very efficient implementation of join patterns and it allows the compiler to perform powerful static analyses of Comega programs to help perform optimizations. The static nature of join patterns allows the Comega compiler to build bitmasks to help rapidly compute when a join pattern can fire.

Although Comega join patterns can effectively capture many kinds of concurrent calculations there are several idioms that do not work out so well. There is an appealing analogy between Comega asynchronous methods and port-based programming (i.e. writing to a port). Indeed, that is how one imagines the Comega compiler implements asynchronous methods. However, in port-based asynchronous programming one often sends a reply port to some asynchronous calculation. Although this can be modeled in Comega one has to use the rather indirect approach of specifying a delegate (a reference to method that corresponds to the reply port). Another constraint arises from the rather static nature of join patterns. Sometimes one wishes to dynamically adapt the code that is executed in response to a join pattern firing. The declarative nature of Comega joins means that the same code is always executed in response to a join pattern firing. We propose that join patterns should really be implemented as *statements*. This allows join patterns to be dynamically constructed. A basic kind of join would also only issue once. When it is done it can activate another 'handler' to establish a different join pattern or it can just re-issue itself.

3. PORT-BASED PROGRAMMING IN CCR

This section describes a library called the Control and Concurrency Runtime (CCR) to support port-based programming in C#.

3.1 DESIGN CONSTRAINTS

We propose an alternative method for implementing join patterns out of basic building blocks which can be realized as a library in C#. These building blocks also let us express other useful port-based concurrent programming idioms. First, we specify some constraints that we adopt for the design of our asynchronous port-based programming library:

1. We wish to constraint ourselves to introducing join-based programming constructs as a library rather than by modifying an existing language or inventing a new language. We agree that eventually the best way to add concurrency constructs may be through language level support. However, we believe that it is too early to freeze join pattern based features into a language because we do not yet have enough experience of their use. For the purpose of experimentation a library based approach seems more prudent.
2. We wish to support port-based programming where message passing is used not just at the interface of the program to communicate with external entities but as a software engineering mechanism to help structure our programs. Our desire is to use message passing to help provide isolation between different components for reliability and parallelism.
3. We wish to support scenarios in which there is a very large amount of message-passing (e.g. web services). Constraining ourselves to mainstream operating systems, this means that we can not directly use heavyweight operating system threads to implement blocking reads. We propose an alternative work-item scheme which does not use operating system threads which works by implementing a continuation passing scheme for port-based programming. We show how language features in C# can help the programmer avoid having to explicitly write continuation passing style code.

3.2 PORTS, ARBITERS AND RECEIVERS

The CCR library defines a port class which is generically parameterized on the types of values that it may carry. For example:

```
Port<float> pF = new Port<float>() ;
```

defines a port `pF` which can only carry values of type float The `Port` type is generically instantiated by giving a type name in angle brackets. One can declare a port that can carry different kinds of values:

```
Port<int,string> pIS = new Port<int,string>();
```

Here the port `pIS` can carry types of `int` or `string`. One can instantiate a `Port` with up to 16 different types. A port that carries multiple values works by maintaining unrelated sub-ports for each type. Reading an `int` value from `pIS` works by reading from the sub-port that contains `ints` and this read operation is unaffected by the state of the `string` sub-port.

Ports comprise a linked list data-structure to hold the values in a port and a list of continuations which execute when a message of a particular type arrives. A continuation is represented by a C# delegate.

A message is sent to a port by using the `post` method:

```
Pot<int> pi = new Port<int>() ;
pi.post (42) ;
```

We provide a method called `test` to atomically remove a value from a port (if the port is not empty). The `test` method returns a boolean result: false if the port was empty (a default value is returned) and true otherwise.

```
int iv ;
if (pi.test (out iv))
  Console.WriteLine ("Read " + iv) ;
else
  Console.WriteLine ("Port empty.") ;
```

One typically does not use the `test` method directly. Instead, this low level function is used to define arbiters which implement concurrency constructs like join patterns and interleaved calculations.

An example of an arbiter is a one time single item receiver. This arbiter identifies the code to execute when a value becomes available on a port. The code to be run is specified as a named delegate in C# or as an anonymous delegate (which is rather like a lambda expression in functional languages). An example is shown below.

```
activate (p.with(delegate (int i)
                  { Console.WriteLine(i) ; }
                 )
         ) ;
```

The `with` method associates a receiver function to call when a value of type `int` is available at port `p`. The call to `activate` causes the listener to come to life by watching for `int` values on port `p` and then when one arrives it will remove it from the port, bind the integer value to `i` in the formal arguments of the delegate and execute the body of the delegate. In this case the handler code just writes out the value that was read from the port. The single item receiver case source code for the `with` method is:

```
public override SingleItemReceiver with
                (Handler<T0> H)
{ return new SingleItemReceiver
    (_t0, new Task<T0>(H));
}
```

This is a one-shot activation. After one message has been dealt with this activation will not respond to further messages that arrive on port `p`.

The `!` operator takes a single item receiver and converts it into a recurring receiver by re-activating the handler after processing each message:

```
activate (!p.with(delegate (int i)
                   { Console.WriteLine(i) ; }
                  )
         ) ;
```

A single item receiver is a special case of a 'receiver thunk'. Receivers are executed under arbiters which control how values flow from ports to receivers. When a continuation has been nominated for execution in response to the arrival of messages on a port it is scheduled for future execution with a dispatcher. The dispatcher maintains a pool of tasks which are dispatched to worker operating system threads. Typically the number of threads is set to the number of physical processors.

Arbiters provide a flexible, lock free framework for adding new coordination primitives. Arbiters take a `ThunkHandler` routine which can be invoked in response to a message being posted. These handler routines must be non side-effecting and execute in a short time without blocking or attempting any asynchronous operations. The job of an arbiter is to decide if a message should be consumed. One typically has a hierarchy of arbiters (due to composite coordination expressions) and each arbiter is responsible for calling its parent arbiters before it tries to consume messages.

We have developed several kinds of arbiters including:

1. single item receivers for specifying handlers on a single port;
2. a choice arbiter which will chose which of two receivers to fire based on the availability of messages;
3. an interleave arbiter (explained later);
4. a join arbiter for static join expressions;
5. `joinvariable` and `joinarray` arbiters for specifying dynamic joins.

The arbiter scheme is designed to allow one to conveniently compose arbiters into composite expressions (e.g. a choice over dynamic joins).

The single item receiver uses a dummy arbiter which is always willing to consume a value on a port and then it produces a `Task()` wrapper for the continuation (handler) which contains the value which was read from the port. The receiver code is invoked in the context of the call of the `post` method. In the implementation of the post operation the insertion of messages and removal of receivers is performed under a lock since posts can be concurrent.

The activation of a receiver is an asynchronous operation. One may post messages to a port with no receivers. When a receiver is actually attached it is presented with an opportunity to process the messages stored in the port. This helps to avoid deadlocks or message leaks.

Receivers in our system inherit from the C# `ITask` interface which is designed to represent work items for thread pools. We provide a dispatcher which accepts receiver work items and schedules their execution in some worker thread. The dispatcher behaves like a port and work is sent to it by posting receivers. The code for `post` extracts the work item dispatcher (`pDispatcher`) to be used from the nodes that contain information about receivers. If the system discovers an asynchronous task that needs to be scheduled for execution (`asyncTask`) it performs scheduling by posting it to dispatcher's port..

The single item receiver calls of `activate` using a handler shown earlier translate into calls to a dispatcher with some handler:

```
void activate(SingleItemReceiver Receiver)
{ _pDispatcher.post(Receiver);
}
```

Note that the code:

```
p.post(5) ;
activate(p.with(H));
```

is equivalent to:

```
activate(p.with(h);
// wait here arbitrary time to ensure
// SingleITemReceiver is attached
p.post(5);
```

One can also 'spawn' a continuation with no receive constraint:

```
spawn<T,….>(Handler<T,…> H);
```

This is defined by posting work to the dispatcher:

```
_pDispatcher.post(new Task<T,..>(Handler<T,..>));
```

4. USING THREADS AND LOCKS

If one just wants to read from a single port at a time then why not just use regular operating system threads and implement a channel

using a FIFO? We give a sample implementation for a FIFO that uses locks below based on code in [3]:

```
public class QueueElem<t>
{   public t v;
    public QueueElem<t> next = null;
    public QueueElem(t v)
    { this.v = v; }
}

public class Queue<t>
{   QueueElem<t> head = null;
    QueueElem<t> tail = null;

    public void Enqueue(t v)
    { lock (this)
      { QueueElem<t> e = new QueueElem<t>(v);
        if (head == null)
            {   head = e;
                Monitor.PulseAll(this);
            }
            else
                tail.next = e;
            tail = e;
      }
    }

    public t Dequeue()
    {   t result ;
        lock (this)
        {   while (head == null)
                Monitor.Wait(this);
            result = head.v;
            head = head.next;
        }
        return result ;
    }
}
```

Through profiling we have found that this code works well for hundreds of threads even under high contention. However, as the number of threads approach 2000 the performance of the system deteriorates dramatically. This is due to the large stack space requirement for each thread on many commercial operating systems. The performance section of this paper presents experimental results which compare the performance of a queue-oriented application against a port orientated implementation in the CCR.

Although the code for reading from a single port does not seem too difficult to write, the implementation of an operation that atomically reads from multiple ports is significantly more challenging to get right and perform efficiently.

5. CHOICE

The choice arbiter allows one to specify that one of two (or more) branches should fire and the others should be discarded. For example:

```
activate(p.with(MyIntHandler)
         |
         p.with(MyStringHandler);
```

will run either `MyIntHandler` or `MyStringHandler` but not both. The choice will be determined by the availability of `int` or `string` message on the `p` port and if both types of messages are available then a non-deterministic choice is made. In Ada one might write a non-deterministic select like:

```
select
  accept intPort (iv : in integer) do
    // int handler code
or
  accept stringPort (sp : in string) do
      // string handler code
end select ;
```

However, this code is different in several respects from the CCR choice expression. The call to `activate` does not block: it just schedules the choice expression over two ports and specifies the code to be asynchronously run when either branch is taken. The Ada **`select`** statement is a blocking construct. The task will wait at the **`select`** statement until one of the branches can accept a rendezvous. The Ada code for handling the arrival of a message (modeled here with a rendezvous) can make use of any variables that are in the enclosing scope. The handler code for the CCR case must be either explicitly passed all the variables it needs or one can use an anonymous delegate to capture variables from the enclosing context. Implementations of thread-safe queues are also part of the JSR-166 specification e.g. `LinkedBlockingQueue`.

Often one wants to repeatedly process certain kinds of messages until a shutdown notification is received or a timeout occurs. This kind of operation can be expressed as:

```
activate(!p.with(MyIntHandler)
         |
         !p.with(MyStringHandler)
         |
          pShutdown(Shutdown)
         );
```

This will repeatedly process messages of type `int` and `string` on port `p` until a shutdown message is received on the `pShutdown` port. Once the shutdown message is processed then the whole activation is torn down and re-issued `int` and `string` handlers are terminated.

6. JOIN PATTERNS

The join arbiter allows one to atomically consume messages from multiple channels. For example:

```
join<int,string>(p1,p2).with(Handler2);
```

will schedule the execution of `Handler2` when values can be atomically read on ports `p1` and `p2`. `Handler2` will be passed the values that were read. One can join over several ports of different types.

As usual the handler can be specified with an anonymous delegate which makes it clearer what happens when the join pattern fires:

```
activate(!join<int, int>(balance, deposit)
         .with(delegate(int b, int d)
               { balance.post(b + d); }));
```

Optimistic and two phase joins allow for concurrent progress for other overlapping joins, roll backs and retries. A common use for joins is for updating shared state as illustrated by the program below.

```
class SampleService : CcrServiceBase
{ Port<string> pState = new Port<string>();
  ServicePort pMain = new ServicePort();
  …..
  ServicePort InitStatefull()
  { pState.post("Hello");
    activate(
        !join<string,msgUpdate>(pState,pMain).with
                        (StatefullUpdate)
        |
        pMain.with(Shutdown)
        );
    return pMain;
}
```

This program can respond to many concurrent posts to the state but each statefull update occurs with exclusive access to the state (which in this case is just a string).

7. DYNAMIC JOIN EXAMPLE

We now present an example of a dynamic join operation which is not easily expressed using Comega style static join declarations. The code fragment below takes a list of files. The objective is to perform a scatter/gather style operation to find a given pattern in each file. The `foreach` loop posts messages which trigger concurrent work items which search for patterns in files. This corresponds to the scattering phase. The `joinvariable` method will create a join over a number of ports which is dynamically specified by the second argument (`filelist.Length`). This corresponds to the gathering stage.

```
if (filelist.Length > 0)
{ Port<msgFileSearchResult> pFileResult
    = new Port<msgFileSearchResult>();

  // Scattter.
  foreach (string f in filelist)
  { msgTextFileSearch msg = new
      msgTextFileSearch(f, msgSearch.Pattern,
                        pFileResult);
    _pMain.post(msg);
  }

 // Gather.
 activate(joinvariable<msgFileSearchResult>
           (pFileResult, filelist.Length,
  delegate(msgFileSearchResult[] complete)
  { msgFileSearchResult sum =
      msgFileSearchResult.Accumalte(complete);
    sum.DirectoriesSearched = 1;
    pResult.post(sum);
  })
  );
}
```

Our experimental results show that this code scales up well in a multi-processor system and we present results in the performance section later in the paper.

8. ITERATORS

Our library expresses concurrent programs in explicit continuation passing style which is often quite an arduous way for a human to express concurrent calculations. CLU iterators [12] provide a mechanism which allows execution to resume from a yield point. We can exploit CLU-style iterators in the C# 2.0 language to help write code which appears closer to what one would write with blocking receives using threads but which is in effect CPS-transformed by the C# compiler.

Inheriting from the `IEnumerator` interface allows one to specify "yield" points in a method at which control flow returns to the caller. When a `MoveNext()` method is called on such a method, control flow resumes from the yield point: not from the start of the method.

The coordination primitives in CCR all inherit from the `ITask` interface and this is used to help identify the iteration variable for use with the `IEnumerator` interface.

We illustrate the use of yield in the code below which defines a method which will execute a choice expression a given number of times (specified by `num`). Each time it will try to execute a choice which either tries to read a float from a port (`_pTest`) or it tries to fire a join pattern that joins on an `int` and `string` value (on the same port). The result of either choice is captured by the `Result` string variable which is written out after each invocation of the choice expression. The `spawniterator` method takes a method written using this yield approach and 'drives' the iterations.

```
class MyClass : CcrServiceBase
{ public static void Main()
  { DispatcherPort dispatcher =
      Dispatcher.Create(-1, 10, "CCR Sample");
    new MyClass(dispatcher).Start();
  }

  public void Start()
  { spawniterator<int>(num,
                       IteratorOnChoiceAndJoin);
  }

  IEnumerator<ITask> IteratorOnChoiceAndJoin
                      (int num)
  { _pTest = new Port<string, int, float>();
    for (int i = 0; i < num; i++)
     { string Result = null;
       _pTest.post(i);
```

```
        _pTest.post(i.ToString());
        yield return
          (_pTest.with(
            delegate (float f)
            { Console.WriteLine("Got a float");
              Result = f.ToString() ;
            })
          |
            join<int, string>
               (_pTest, _pTest).with(
             delegate(int j, string s)
             { Console.WriteLine
                 ("Got and int and string");
                Result = j.ToString() + " " + s;
             }));
      // We can use Result below because this
      // line of code will execute *after*
      // either branch above.
      Console.WriteLine(Result);
  }
}
```

Note that the `Result` stack variable is captured appropriately and it is set inside the continuations, yet it can be accessed correctly from a context outside of the continuation.

Although this is not as direct as writing blocking receives it is a significant improvement over writing explicit continuation passing style code and we get the advantage of being able to express concurrent programs without using expensive operating system threads to represent blocking behavior.

9. INTERLEAVE

The interleave arbiter allows one to express concurrent calculations which could have been described using join patterns but our experience has found that the join encodings were error prone so a special arbiter was created for capturing reader writer locks in the style of `ReadWriteLock` in the JSR-166 library.

The interleave arbiter is shown at work in the code below where a writer (`DoneHandler`) is specified as 'exclusive' and the state readers (`GetState`) are specified as 'concurrent'. The interleave (^) ensures that these operations are scheduled safely and concurrently.

```
activate(exclusive(p.with(DoneHandler),
         !p.with(UpdateState))
         ^
         concurrent(!p.with(GetState),
         !p.with(QueryState))
);
```

The continuations under an interleave arbiter may be iterators. The interleave arbiter will provide atomicity guarantees until the sequential state machine has completed. The implementation uses interlocked routines and ports to queue readers and writers and once again there is no thread blocking. This idiom has found widespread use in our own distributed systems and concurrent programs.

10. PERFORMANCE

The chart in Figure 1 illustrates the comparative performance of three approaches for implementing a producer/consumer style benchmark application. The application is structured as a collection of identical pipeline stages which each perform a heavyweight arithmetic calculation. The result of one pipeline stage is communicated to the following pipeline stage using the queue implementation presented earlier. Once the pipeline is full each stage can potentially execute in parallel with every other pipeline stage. The plain C# version of this benchmark application accomplishes this parallelism by creating a thread for each pipeline stage. The CCR version uses ports to communicate values between processing stages. The pipeline stages are scheduled as work items which can be sent to one of several dispatchers each of which is associated with an operating system thread. The study the behavior of the various implementations for different sizes of work items we vary the duration of the work item.

The number of pipeline stages connected by queues is represented by the vertical axis. The *x* axis corresponds to the duration of the work item processed at each node of the pipelined calculation. The three lines record the results for (i) an implementation performed using C# threads and locks to implement a queue (using exactly the code shown earlier); (ii) a sequential implementation of the pipelined calculation; (iii) and a CCR version. The results were obtained using an eight processor 64-bit Opteron workstation.

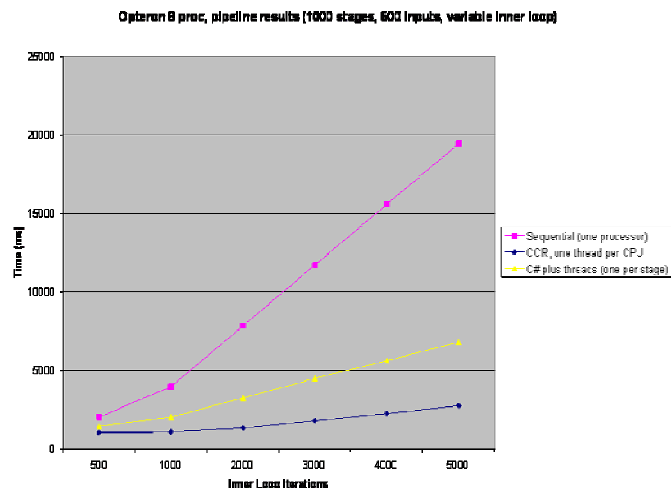

Figure 1: Pipeline Performance Measurements

The CCR measurements give the best results showing an almost linear speedup (7.1x/8) for a work item contaminating 5000 iterations of a loop performing a double precision floating point multiple and accumulate. As the number of pipeline stages approach 1500 the threads and locks version diverges as the large number of threads allocate stack space which exhausts the available virtual memory.

In an experiment with 1000 pipeline stages, the explicitly threaded version allocates a huge working set: 30MB. This has a significant impact on performance. In comparison, the corresponding working set for the CCR version is just 20MB.

Rather than explicitly allocating a thread to each stage of the pipeline we repeated the experiment on a 32 processor machine using a thread pool with 64 threads (to match the 32 hyperthreaded processors). Our experiments showed that the CCR version (also using 64 dispatcher threads) performed around 100 times faster than the thread pool version. We also experimented with using 640 and 2000 dispatcher threads with the CCR version (this may be useful for supporting blocking constructs). The additional threads had little impact on the throughout of the system. Note that for a system that explicitly uses a thread pool one has to find some way to manage the coordination between work items whereas the CCR provides a mechanism for both sharing threads and for coordination.

The scatter gather file search example shown previously was run over 44,400 files in 1,111 directories on an eight processor machine. We also produced a sequential version of the file search application. The sequential version runs at the same speed as the single thread CCR version indicating a low overhead for the CCR concurrency overhead for this type of scenario. Figure 2 shows that we get a four times speedup with 8 processors.

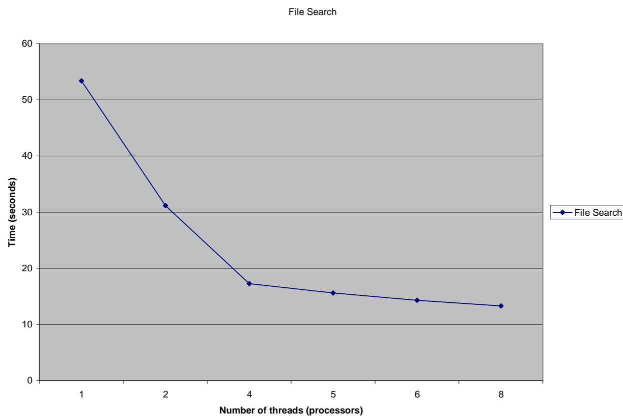

Figure 2: File Search on an 8 processor machine

11. DEBUGGING

Moving from a stack-oriented model to a message-oriented model has many repercussions including changing the way in which programs have to be debugged. We propose a causality-based debugger to help understand the state of a message-passing program. Our debugger is integrated as a component into a commercially available debugger.

Even with a debugger there is still a need to find as many errors as possible at compile time and we are currently trying to apply to work of Kobayashi [10] to help analyze abstractions of our message passing programs.

12. RELATED AND FUTURE WORK

The Comega system has some similarity with our system since it is also an object-orientated language that implements join patterns albeit in a rather static manner. We believe it should be possible to adapt the join pattern declarations to become join pattern statements. We have experimented with writing a translator that maps Comega programs to C# programs with use our CCR library. A port is created for each asynchronous method and a call of an asynchronous method corresponds to posting a message to its corresponding port. A call to a synchronous method can be modeling using a reply port to carry the result of the invocation. Translating the use of synchronous join patterns requires a CPS-transformation on the code that calls uses such methods in chords although we can once again exploit C# iterators to make the compiler effectively perform these transformations for us.

Another candidate for an application level concurrency construct is the notion of a future. This technique allows a call to a function or method to occur asynchronously. Futures are an appealing and accessible model for programmers and they are supported in the Java JSR-166 library. One could imagine compiling futures using the mechanisms described in this paper.

Exceptional behavior in CCR is handled by sending messages to an error port. The `synchronized` approach for locking objects in C# and Java provides an unsatisfactory method for locking especially in the presence of errors. JSR-166 provides an alterative locking approach which can be used with `try/finally` to ensure a lock is given up even after an exception.

13. CONCLUSIONS

We have presented an alternative scheme for implementing concurrency coordination constructs including dynamic joins and choice and interleaves without adding any new language features to an existing language. The constructs that we present are more expressive than those found in Comega which also has join patterns. In particular, we present a small number of arbiters which can be hierarchically composed to express quite sophisticated and flexible concurrent coordination constructs.

Our library interface is designed to support large amounts of fine grained concurrency. This comes from our attempt to turn almost everything into communication. We also leverage latency as an opportunity for concurrency.

We believe that ultimately the best way of supporting concurrency constructs may be through language level support. This allows a compiler to perform more sophisticated static analyses to help find bugs and it also permits the compiler to perform more powerful optimizations. However, we argue that many concurrency constructs recently proposed for application level programming are not mature enough yet. We have presented a flexible framework for experimenting with concurrency constructs for join patterns and interleaved calculations. As we gain experience with experiments that use these idioms we hope to get insight into what might be good candidates for inclusion as language level concurrency constructs.

14. REFERENCES

[1] Appel, A. Compiling with Continuations. Cambridge University Press. 1992

[2] Benton, N., Cardelli, L., Fournet, C. Modern Concurrency Abstractions for C#. ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 26, Issue 5, 2004.

[3] Birrell, A. D. An Introduction to Programming with Threads. Research Report 35, DEC. 1989.

[4] Chaki, S., Rajamani, S. K., and Rehof, J. Types as models: Model Checking Message Passing Programs. In Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM, 2002.

[5] Conchon, S., Le Fessant, F. JoCaml: Mobile agents for Objective-Caml. In First International Symposium on Agent

Systems and Applications. (ASA'99)/Third International Symposium onMobile Agents (MA'99). IEEE Computer Society, 1999.

[6] Daume III, H. Yet Another Haskell Tutorial. 2004. Available at http://www.isi.edu/~hdaume/htut/ or via http://www.haskell.org/.

[7] Fournet, C., Gonthier, G. The reflexive chemical abstract machine and the join calculus. In Proceedings of the 23rd ACM-SIGACT Symposium on Principles of Programming Languages. ACM, 1996.

[8] Fournet, C., Gonthier, G. The join calculus: a language for distributed mobile programming. In Proceedings of the Applied Semantics Summer School (APPSEM), Caminha, Sept. 2000, G. Barthe, P. Dybjer, , L. Pinto, and J. Saraiva, Eds. Lecture Notes in Computer Science, vol. 2395. Springer-Verlag, 2000.

[9] Harris, T., Marlow, S., Jones, S. P., Herlihy, M. Composable Memory Transactions. Submitted to PPoPP 2005.

[10] Igarashi, A., Kobayashi, K. Resource Usage Analysis. ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 27 Issue 2, 2005.

[11] Itzstein, G. S, Kearney, D. Join Java: An alternative concurrency semantics for Java. Tech. Rep. ACRC-01-001, University of South Australia, 2001.

[12] Liskov, Barbara. A History of CLU. ACM SIGPLAN Notices, 28:3, 1993.

[13] Lea, D. The java.util.concurrent Synchronizer Framework. PODC CSJP Workshop. 2004.

[14] Ousterhout, J. Why Threads Are A Bad Idea (for most purposes). Presentation at USENIX Technical Conference. 1996

[15] Odersky, M. Functional nets. In Proceedings of the European Symposium on Programming. Lecture Notes in Computer Science, vol. 1782. Springer-Verlag, 2000.